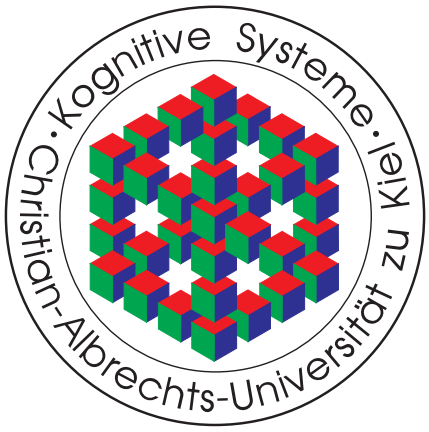
Medical Image Segmentation With the 2-Dimensional Analytic Signal

Diplomarbeit



vorgelegt von

Felix Thomsen

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl Kognitive Systeme

Betreuer:
Dipl. Inf. Lennart Wietzke
Dipl. Inf. Oliver Fleischmann
Prof. Dr. Gerald Sommer

Kiel, den 30. April 2010

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.
Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Kiel, den 30. April 2010

Felix Thomsen

Danksagung

Nach dem Abschluss der Diplomarbeit möchte ich allen Personen danken, die mich dabei unterstützt haben.

Besonderer Dank gilt Herrn Prof. Dr. Sommer für das Thema und die Räumlichkeiten, die er mir für die Diplomarbeit zur Verfügung gestellt hat. In diesem Abschnitt meines Studiums durfte ich sehr tief in das Thema der elektronischen Bildverarbeitung eintauchen und habe zudem viel über das wissenschaftliche Arbeiten lernen dürfen.

Meinen Betreuern Lennart Wietzke und Oliver Fleischmann danke ich für die Anregungen, die Unterstützung und die Korrekturvorschläge.

Dem Universitätsklinikum Kiel, stellvertretend Dr. med. Ole Kayser, danke ich für die freundliche Bereitstellung der verwendeten Wirbelsäulen- und Leberfotografien.

Andrea Menge, Caprice Sturm, Hanjo Hamer und Andreas Stadler danke ich für das Korrekturlesen.

Meiner Familie danke ich für die finanzielle und moralische Unterstützung.

Contents

1. Introduction

In this thesis we consider the development and application of a novel segmentation filter, called the **S**cale **S**pace **S**egmentation **F**ilter. The SSSF is a phase- and amplitude-based segmentation filter, which uses the 2-dimensional analytic signal, described in [40].
An image segmentation filter is in our case a functionality which maps the image in a representation having ones at positions of the region of interest and zeros at positions of the background.
Segmentation of grey value images is a main task of computer vision. Our approach is rotationally invariant, global, and independent from special image classes: It is applicable to any class of grey value images or objects, regardless of the profile, the luminance, rotation, scale or other attributes of the region of interest. The implementation of our filter is based on an application of the analytic signal transform for n different scale-space intervals. It maps the signal into a set of n phase- and amplitude-images. Based on this set we apply a special threshold operator, before we reconstruct the image in a pixel based manner.
The filter is designed to be executed on parallel processors, therefore we are able to utilise fully a given GPU[1]. Furthermore the calculation time is constant for a given filter and independent of the design of the input image.
The filter is specialised and tested on medical image data of the human spine and on image-stacks of the human liver.
This thesis is organised as follows:
We start with the introduction of the 2-dimensional analytic signal. In this chapter we optimise the given implementation in relation to speed. Further we improve the main-scale detection[2].
The next chapter is about the SSSF. We give examples for the training and application of the filter. Finally we present the results of the SSSF in case of medical image data. We apply it to the spine detection and the liver volume estimation, followed by a summary of the properties and advantages of the SSSF.

[1] **G**raphics **P**rocessing **U**nit
[2] The main-scale detection or extrema-detection returns the scale-space interval which maximises the filter response. This is the scale interval which maps best to the local image region.

In the third chapter we implement a spine curve detector which can be used to find the degree of a scoliosis[3].
The results of our approach show a significant improvement compared to the state-of-the-art methods[4] with the advantages of higher computational speed and higher segmentation accuracy.
We assume that the reader of this thesis is familiar with the basics of general image processing and computer vision, which can be found in [2], [20], [36], and [37].
Future works will deal with an expansion of this filter for not only 2-dimensional signals, but 3-dimensional signals like the image-stack of the human liver. They will be based on the application of 3 scale space segmentation filters spanning the transverse, the coronal, and the sagittal plane.

This thesis contains a DVD with the programme code and the image data. The text is written in British English.

[3]The scoliosis is a medical condition in which a person's spine is curved from side to side.

2. The 2-Dimensional Analytic Signal

2.1. Preliminaries

Signal

A (n-dimensional) signal is a mapping $\mathbb{Z}^n \to \mathbb{R}^l$. We will consider 2-dimensional signals $\mathbb{Z}^2 \to \mathbb{R}^l$ with $l \in \{1, 2, 4\}$. The set of signals $\mathbb{Z}^n \to \mathbb{R}^l$ will be called $\mathbb{R}^{l^{\mathbb{Z}^n}}$.

Image

A discrete image is a signal $f \in \mathbb{N}^{1^{\mathbb{Z}^2}}$, where in general $f \in \{f | f : \mathbb{Z}^2 \to \{0, \ldots, 255\}\}$.

Local image region

A (local) image region is a set of adjacent pixels inside an image. Generally we consider circular or rectangular image regions. One pixel or the complete image are special cases of image regions.

Floor, ceiling and rounding function

We define the floor function $\lfloor x \rfloor$ and the ceiling function $\lceil x \rceil$.

$$\lfloor x \rfloor := \{z \in \mathbb{Z} | z \leq x < z + 1\} \tag{2.1}$$

$$\lceil x \rceil := \{z \in \mathbb{Z} | z - 1 < x \leq z\} \tag{2.2}$$

The rounding is defined by

$$\text{round}(x) := \begin{cases} \lfloor x + 0.5 \rfloor : & x \geq 0 \\ \lceil x - 0.5 \rceil : & x < 0 \end{cases} . \tag{2.3}$$

Rotation

A rotation $r_{c,\alpha}$ for a constant rotation centre point $c = (x_c, y_c)$ and rotation angle α is a mapping $r : \mathbb{Z}^2 \to \mathbb{Z}^2$ with

$$\begin{aligned} r_{c,\alpha}(x,y) = \quad & [\text{round}(x_c + (x - x_c)\cos\alpha - (y - y_c)\sin\alpha), \\ & \text{round}(y_c + (y - y_c)\cos\alpha + (x - x_c)\sin\alpha)] \end{aligned} \tag{2.4}$$

and $(x, y) \in \mathbb{Z}^2$.

Rotation of a signal

Let $f \in \mathbb{R}^{n^{\mathbb{Z}^2}}$. A rotation of f in c is defined by $r_{f,c,\alpha}(x,y) := f(r_{c,\alpha}(x,y))$.

Single-point mapping

Let $f : \mathbb{R}^{l^{\mathbb{Z}^k}} \to \mathbb{R}^{m^{\mathbb{Z}^k}}$ be a mapping between two signals. f is a single-point mapping if

$$\forall g \in \mathbb{R}^{l^{\mathbb{Z}^k}}, h = f(g), x, y \in \mathbb{Z}^k : g(x) = g(y) \Rightarrow h(x) = h(y). \tag{2.5}$$

In case of mappings between images, the colour of the output image at a certain position is determined by only the colour of the input image at the same position.

Rotation invariant/variant mapping

Let $f : \mathbb{R}^{l^{\mathbb{Z}^2}} \to \mathbb{R}^{m^{\mathbb{Z}^2}}$, $s \in \mathbb{R}^{l^{\mathbb{Z}^2}}$, $g = f(s)$ and $g_{r,p,\alpha} = f(r_{s,p,\alpha})$. f is rotation invariant in $p \in \mathbb{Z}^2$ if $\forall s, \alpha : g(p) = g_{r,p,\alpha}(p)$, otherwise f is rotation variant in p.
Note: The rotation centre equals the evaluation point p, hence if f is a single-point it follows that f is rotation invariant.

Intrinsic dimension

Let f be a function $\Omega \subseteq \mathbb{R}^l \to \mathbb{R}^m$. The intrinsic dimension of f is

$$\min\{n | \exists\, g : \mathbb{R}^l \to \mathbb{R}^n, h : \mathbb{R}^n \to \mathbb{R}^m, \forall \mathbf{x} \in \Omega : f(\mathbf{x}) = (h \circ g)(\mathbf{x})\}. \tag{2.6}$$

The value n depicts the dimension of f we need to describe f without any loss.
An image has the intrinsic dimension 0 ($i0D$) if $\forall (x,y) \in \Omega f(x,y) = c$. We will also consider $i1D$ functions $f(x,y) = (g \circ h)(x,y)$ for example $f(x,y) = sin(x+y)$). The maximal intrinsic dimension for a signal $f \in \mathbb{R}^{m^{\mathbb{Z}^k}}$ is ikD or for images $i2D$, respectively.

Intrinsic dimension in a local image region

As real images are in general $i2D$, we do not consider the whole image but make statements about the intrinsic dimension in a local image region. Local $i1D$ regions are lines or edges, whereas local $i0D$ regions are plateaus, $i2D$ regions are generally superpositions of two or more lines. In general noise is of intrinsic dimension 2 for 2-dimensional images.[1]

2.2. The Analytic Signal

The development of the monogenic signal [10] which is a special case of the 2-dimensional analytic signal [40] is motivated by the one-dimensional analytic signal or simply called analytic signal.
Let $f \in \mathbb{R} \to \mathbb{R}$ be square integrable

$$f \in L_2(\mathbb{R}, \mathbb{R}) \Leftrightarrow \int_{-\infty}^{\infty} |f(x)|^2 dx < \infty. \tag{2.7}$$

The Fourier series of f is given by

$$f(x) = \sum_{\nu} a_\nu \cos(2\pi\nu x + \phi_\nu) \tag{2.8}$$

with ν as the frequency. Every 1D function can be locally decomposed to an unique set of basis-frequencies ν with different amplitudes a_ν and phases ϕ_ν. For the introduction into the analytic signal we follow [40] and [13]: The local 1D signal model for the analytic signal at origin 0 of the applied signal reads

$$g^e := a\cos(\phi) := a_s \cos(\phi_s) := \mathcal{P}_s\{g\}(0) \tag{2.9}$$

with $s \geq 1$ as the scale parameter of the filter operator $\mathcal{P}_s\{\cdot\}$ of the analytic signal[2].
Since the local signal model is an even function, hence $\forall x \in \mathbb{R} : \cos(x) = \cos(-x)$, we call g^e the even part of the analytic signal. The odd part can be calculated by using the convolution of the filtered original signal with the classical 1D Hilbert transform

[1] Statistically the occurrences of local ikD regions decrease for increasing k: $i0D$ regions appear most frequently whereas in case of images $i2D$ regions are most rare.
[2] The scale parameter s is the equivalent to the frequency ν, but with the difference that high values of s correspond with low values of ν.

kernel $h(\tau) := \frac{1}{\pi\tau}$ and

$$g^o := a\sin(\phi) := a_s\sin(\phi_s) = (h * \mathcal{P}_s\{g\})(0) \tag{2.10}$$

with $*$ as the convolution operator and

$$(h * \mathcal{P}_s\{g\})(x) = \frac{1}{\pi}\text{P.V.}\int_{\tau\in\mathbb{R}} \frac{\mathcal{P}_s\{g\}(x-\tau)}{\tau}d\tau \tag{2.11}$$

as the Hilbert transform of the signal g in scale space and P.V. as the Cauchy principal value [22].
The phase ϕ and the amplitude a can be determined by

$$\phi = \arctan\frac{g^o}{g^e} \tag{2.12}$$

$$a = \sqrt{(g^o)^2 + (g^e)^2}. \tag{2.13}$$

The complex extension $g^e + i\ g^o$ of a scalar valued one-dimensional signal g is called the 'analytic signal'. The analytic signal now enables the identification of amplitudes and phases for all scale parameters.

2.3. The 2-Dimensional Analytic Signal

The 2-dimensional analytic signal expands the input function. It is defined for $g \in L_2(\mathbb{R}^2, \mathbb{R})$. With this expansion a new feature arises, the orientation θ. It depicts the orientation of amplitude and phase. In the case of the simple monogenic signal we take the orientation which maximises the amplitude. The benefit of the monogenic signal is its isotropy [9], hence for the calculation of the (main) orientation we only need one application of the operator[3].
That approach does not yield to a true extension from the first to the second dimension, as the phase and amplitude are only calculated for one line defined by the orientation. A better method for the calculation of at least two orientations is the 2-dimensional analytic signal [40]. With the expansion from the monogenic to the 2-dimensional analytic signal, there comes up a second orientation which generates a further rotation invariant property, the apex angle between the first and the second orientation.
We continue with the formulas of the four different scalar values of the 2-dimensional an-

[3] We do not need any steerable operators [14] to calculate the main orientation.

alytic signal. To generate the 2-dimensional analytic signal we need the Poisson kernel k_s^p and the second order Hilbert transform kernel k_s^h for a given scale s

$$k_s^p(x,y) = \frac{1}{2\pi(s^2+x^2+y^2)^{\frac{3}{2}}} \tag{2.14}$$

$$k_s^h(x,y) = \frac{s(2s^2+3(x^2+y^2))-2(s^2+x^2+y^2)^{\frac{3}{2}}}{2\pi(x^2+y^2)^2(s^2+x^2+y^2)^{\frac{3}{2}}}. \tag{2.15}$$

We calculate the 2-dimensional analytic signal for a given scale interval which includes the scale of interest. We take two scale values s_f and s_c the fine- and the coarse-scale with $0 < s_f < s_c$. For these scale values we calculate the following six masks

$$q^p_{s_f,s_c}(x,y) := s_f k^p_{s_f}(x,y) - s_c k^p_{s_c}(x,y) \tag{2.16}$$

$$\begin{bmatrix} q^x_{s_f,s_c} \\ q^y_{s_f,s_c} \end{bmatrix}(x,y) := \begin{bmatrix} x \\ y \end{bmatrix}(k^p_{s_f}(x,y) - k^p_{s_c}(x,y)) \tag{2.17}$$

$$\begin{bmatrix} q^{xx}_{s_f,s_c} \\ q^{xy}_{s_f,s_c} \\ q^{yy}_{s_f,s_c} \end{bmatrix}(x,y) := \begin{bmatrix} x^2 \\ xy \\ y^2 \end{bmatrix}(k^h_{s_f}(x,y) - k^h_{s_c}(x,y)) \tag{2.18}$$

which lead to the signals

$$f_\xi := q^\xi_{s_f,s_c} * f, \text{ for } \xi \in \{p, x, y, xx, xy, yy\} \tag{2.19}$$

with f as the input signal and $*$ as the 2-dimensional convolution.
For the signal calculation we introduce the abbreviations

$$f_s := \frac{f_p}{2} \tag{2.20}$$

$$f_{+-} := \frac{f_{xx} - f_{yy}}{2} \tag{2.21}$$

$$f_e := \frac{\sqrt{f_{+-}^2 + f_{xy}^2}}{|f_s|} \tag{2.22}$$

$$f_q := \frac{2(f_x^2 + f_y^2)}{1 + f_e}. \tag{2.23}$$

For the final signal formulas, we define

$$\text{atan2}(x,y) := \begin{cases} \arctan\frac{y}{x}: & x>0 \\ \frac{\pi}{2}: & x=0, y>0 \\ -\frac{\pi}{2}: & x=0, y<0 \\ \pi+\arctan\frac{y}{x}: & x<0, y\geq 0 \\ -\pi+\arctan\frac{y}{x}: & x<0, y<0 \end{cases}. \tag{2.24}$$

Note: $\text{atan2}(0,0)$ is undefined. The formulas for the phase and amplitude read

$$\phi = \text{atan2}(\sqrt{f_q}, f_p) \tag{2.25}$$

$$a = 0.5\sqrt{(f_p^2+f_q)}. \tag{2.26}$$

The formulas for the main orientation θ_{main} and the apex angle α are given by

$$\theta_{\text{main}} = \text{atan2}(f_y, f_x) \tag{2.27}$$

$$\alpha = \arccos\left(\frac{\sqrt{f_y^2+f_x^2}}{|f_s|}\right). \tag{2.28}$$

The apex angle depicts the angle between the two calculated orientations[4]

$$\alpha = |\theta_1 - \theta_2| \tag{2.29}$$

with

$$\theta_{\text{mean}} = \frac{\theta_1+\theta_2}{2} = \frac{1}{2}\arctan\frac{f_{xy}}{f_{+-}} \tag{2.30}$$

and

$$\theta_{1,2} = \theta_{\text{mean}} \pm \frac{\alpha}{2}. \tag{2.31}$$

Note that the domain of the main orientation θ_{main} is not equal to the domain of the mean orientation θ_{mean}. The mean orientation is the mid-orientation between θ_1 and θ_2 whereas the main orientation is the one with the highest gradient. In the following we substitute θ_{mean} with θ.

The authors of the monogenic signal and the 2-dimensional analytic signal released many publications concerning this topic: Felsberg introduced the monogenic signal in [9],

[4]Note: The formulas for the orientation and the phase are formulated in such a way, that the orientation is not only in range $[0,\pi]$ but in $[-\pi,\pi]$. It depicts not only the orientation but the direction. This reduces the range of the phase from $[-\pi,\pi]$ to $[0,\pi]$ in this implementation.

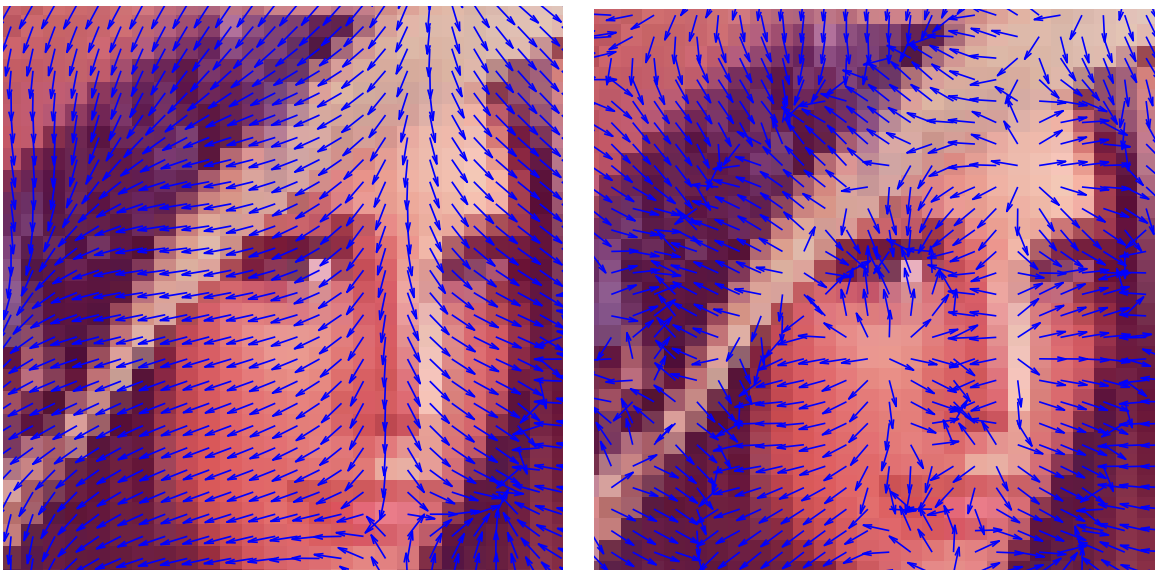


Figure 2.1.: Calculation of the mean orientation on a 'Lena'-image. From the left to the right: coarser and finer scale interval. The resolution is one pixel per arrow.

and [10]. The features and the potential of this low level image processing tool are described in [8], [11], and [12]. Sedlazeck [34] gave solutions for the local feature detection on images using two similar methods. Wietzke et al. published a large set of researches concerning the 2-dimensional signal and its applications [41], [43], [39], and [44]. Recent researches [42] deal with the extension of the 2-dimensional analytic signal to the calculation of any number of orientations which is the most general formulation.

The analytic signal is a tool for low-level image processing. It delivers four different values based on the local neighbourhood of a location $p \in \mathbb{R}^n$: $(\phi, a, \theta, \alpha)$. The values ϕ and a have the same meaning as in the 1-dimensional case whereas the orientation θ depicts the flow of the grey values inside the scaled neighbourhood. Considering the signal as a Monge patch[5], θ is the deepest descent at p. A bowl at p would start to roll in direction θ. Hence, the orientation is the direction of the gradient. Continuing with this illustration the size of a bowl (which equals the scale) influences the direction. We illustrate that relationship by applying the orientation on the 'Lena'-image for coarser and finer scales.
At coarse scale the arrows only satisfy the coarsest structures. For finer scales there arise new orientations at positions where the orientations for the coarse scale do not change.

[5] A Monge patch is a patch $f : \mathbb{R}^2 \to \mathbb{R}$ of the form $f(x, y) = (x, y, g(x, y))$. It is named after Gaspard Monge (1746-1818).

The 2-dimensional analytic signal is able to determine this orientation for a given scale s (see Figure 2.1).
If we consider the neighbourhood of p, we focus on the adjacent points of p within radius r. In the case of the analytic signal we express the observation of the neighbourhood of p by convolving f at p with the six different convolution masks (2.16), (2.17), and (2.18). The size of the convolution masks is basically influenced by the scale s. Small values of r satisfy finer scales, coarser scales require larger values of r. To understand the exact relationship between r and s we have to take into account the shape of the masks.

2.4. Application of the Analytic Signal

From now on we follow and expand the implementation of the analytic signal given in [40].

2.4.1. Calculation of the Mask Size

The convolution mask size $n \in \mathbb{N}$ of a considered neighbourhood is related to the scale s. The basic low pass filter of the analytic signal is the scaled Poisson kernel

$$p_s := s\ k_s^p \tag{2.32}$$

where k^p is defined in Equation (2.14). This is combined by using two different scale parameters[6], a coarser s_c and a finer one s_f. The construction is given in Equation (2.16). We get a band pass filter which is known as Difference-of-Poisson (DoP) filter[7].
The parameters s_c and s_f define the parameters of the low pass and the high pass filters respectively. The size n of the DoP-filter defines the radius of the considered neighbourhood. Since it is impossible to convolve each pixel position with a mask with an infinity radius we have to find an optimal n depending on the scale-values. Therefore, we cut all elements on the border which have no significant effect on the result. Hence,

[6] The parameters must be positive and the coarse scale must be bigger than the fine scale. Felsberg actually postulates in his PhD thesis [8] $1 \leq s_f$ since a scale parameter less than '1' would be meaningless for discrete signals.

[7] The DoP filter is the difference of two Poisson-filters, similar to the Difference-of-Gaussian (DoG) filter for Gauss-filters.

we assign a static error e with $0 < e < 1$ and in the continuous case

$$e \int\limits_{\mathbb{R}^2} p_{s_c}(x,y)dx\ dy = \int\limits_{-n}^{n}\int\limits_{-n}^{n} p_{s_c}(x,y)dx\ dy. \tag{2.33}$$

The maximal mask size only depends on the coarse scale, as this is the parameter which is responsible for the coarser structures. Because of the attribute of the Poisson kernel, being normalised

$$1 = \int\limits_{\mathbb{R}^2} p_s(x,y)dx\ dy \tag{2.34}$$

it follows that

$$e \int\limits_{\mathbb{R}^2} p_{s_c}(x,y)dx\ dy = e = \int\limits_{-n}^{n}\int\limits_{-n}^{n} p_{s_c}(x,y)dx\ dy. \tag{2.35}$$

Hence (still for the continuous case)

$$\begin{aligned}
e &= \int\limits_{-n}^{n}\int\limits_{-n}^{n} p_{s_c}(x,y)dx\ dy \\
&= \frac{s_c}{2\pi}\int\limits_{-n}^{n}\int\limits_{-n}^{n} \frac{1}{(s_c^2+x^2+y^2)^{\frac{3}{2}}}dx\ dy \\
&= \frac{2}{\pi}\arctan\left(\frac{xy}{s_c\sqrt{s_c^2+x^2+y^2}}\right) \\
&\overset{n:=x=y}{=} \frac{2}{\pi}\arctan\left(\frac{n^2}{s_c\sqrt{s^2+2n^2}}\right) \qquad (2.36)\\
\Rightarrow n &= \underbrace{s_c\sqrt{\tan\left(\frac{e\pi}{2}\right)\left(\tan\left(\frac{e\pi}{2}\right)+\sqrt{1+\tan\left(\frac{e\pi}{2}\right)}\right)}}_{=c_{error}}. \qquad (2.37)
\end{aligned}$$

The value c_{error} only depends on the constant e. The parameter n depends only linearly on s_c. It is reasonable to keep the error independent from s_c. In the discrete case, it follows that

$$n = \lceil s_c c_{error} \rceil \tag{2.38}$$

with $c_{error} \in [0, \infty[$. Felsberg uses a value $c_{error} = 1.66$. In his implementation [7] and [6] he does not solve the integral, but accumulates the sum, until he covers the value of e. This approach needs $\mathcal{O}(n^2)$ calculations.

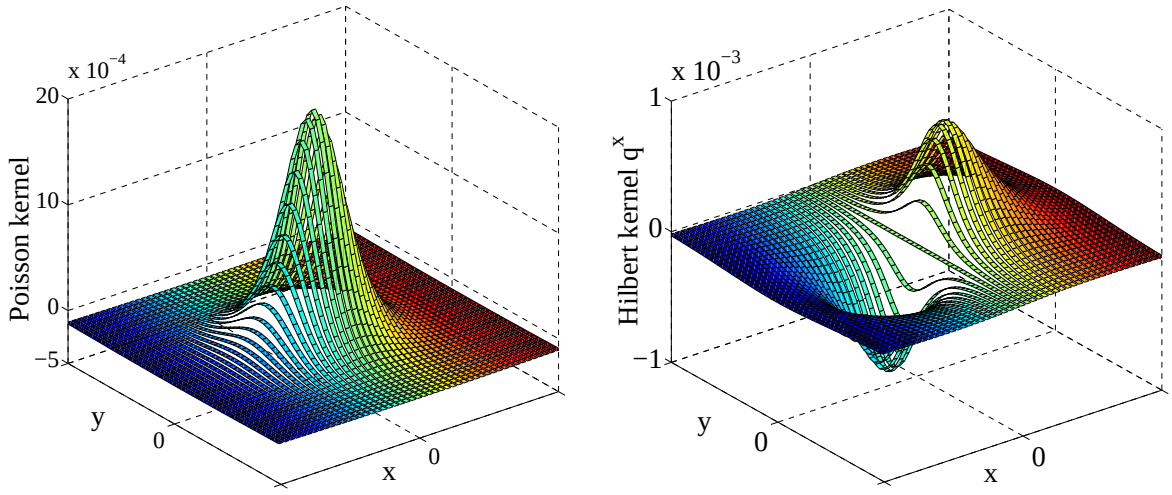


Figure 2.2.: Convolution kernels with $c_{error} = 1.66$ and $n = 24$. From left to right: $\hat{q}^p_{14,15}$ and $\hat{q}^x_{14,15}$

In the discrete case Equation (2.33) equals

$$e \sum_{x,y=-\infty}^{\infty} p_{s_c}(x,y) = \sum_{x,y=-n}^{n} p_{s_c}(x,y) \tag{2.39}$$

whereby

$$\sum_{x,y=-\infty}^{\infty} p_s(x,y) > \int_{\mathbb{R}^2} p_s(x,y)dx\ dy \tag{2.40}$$

for $s < \infty$. The Equation (2.38) does not change in the discrete case, i.e. n still depends linearly on s_c. However, for c_{error} another term follows compared to (2.37), because of the inequality (2.40).

2.4.2. Calculation of the Offset

As the unlimited integral $p_s(x,y)$ is always 1, it follows

$$q^p_{s_f,s_c}(x,y) = 1 - 1 = 0. \tag{2.41}$$

Hence, a convolution with constant signal f always generates the result '0'. This postulation is important for the analytic signal. When calculating in the discrete case this attribute disappears, as $p_s(x,y)$ is restricted to a limited sum. In the discrete case for limited n it always follows that $q^p_{s_f,s_c}(x,y) \neq 0$. We need to post-process the signal

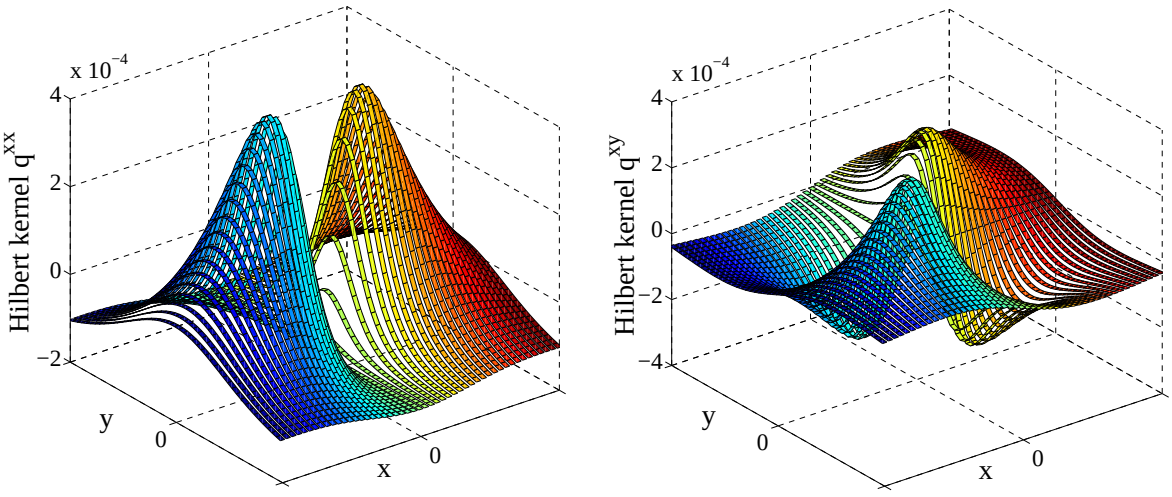


Figure 2.3.: Kernels $\hat{q}^{xx}_{14,15}$ and $\hat{q}^{xy}_{14,15}$

response by subtracting the mean elements-value. This substraction is only required for three of the six masks

$$\hat{q}^{\xi}_{s_f,s_c}(x,y) = q^{\xi}_{s_f,s_c}(x,y) - \sum_{\tilde{x},\tilde{y}=-n}^{n} \frac{q^{\xi}_{s_f,s_c}(\tilde{x},\tilde{y})}{(2n+1)^2},\ \xi \in \{p, xx, yy\}. \tag{2.42}$$

The formulas for the other masks do not change.

$$\hat{q}^{\xi}_{s_f,s_c}(x,y) = q^{\xi}_{s_f,s_c}(x,y),\ \xi \in \{x, y, xy\} \tag{2.43}$$

Figure 2.2 and 2.3 depict the convolution kernels $\hat{q}^{\xi}_{14,15}$ with $\xi \in \{p, x, xx, xy\}$ and $c_{error} = 1.66$. The radius is $n = 24$ and the diameter is $(2n + 1) = 49$. The design of the kernels is determined by the quotient $\frac{s_c}{s_f}$, the resolution is induced by s_c. These four kernels and the two transpositions of $\hat{q}^{x}_{s_f,s_c}$ and $\hat{q}^{xx}_{s_f,s_c}$ are used in the implementation of the 2-dimensional signal.

2.4.3. The Basics of the Analytic Signal

As already mentioned, the analytic signal is not only used to calculate the local phase and amplitude of the neighbourhood, but also for the mean orientation and the apex angle. The fine- and coarse-scale values supply the boundaries in which the considered structure is located. In this subsection we are focussing on the class which is the signal model of the analytic signal.
Figure 2.4 depicts an example for the usage of the analytic signal on a pure non-axis-

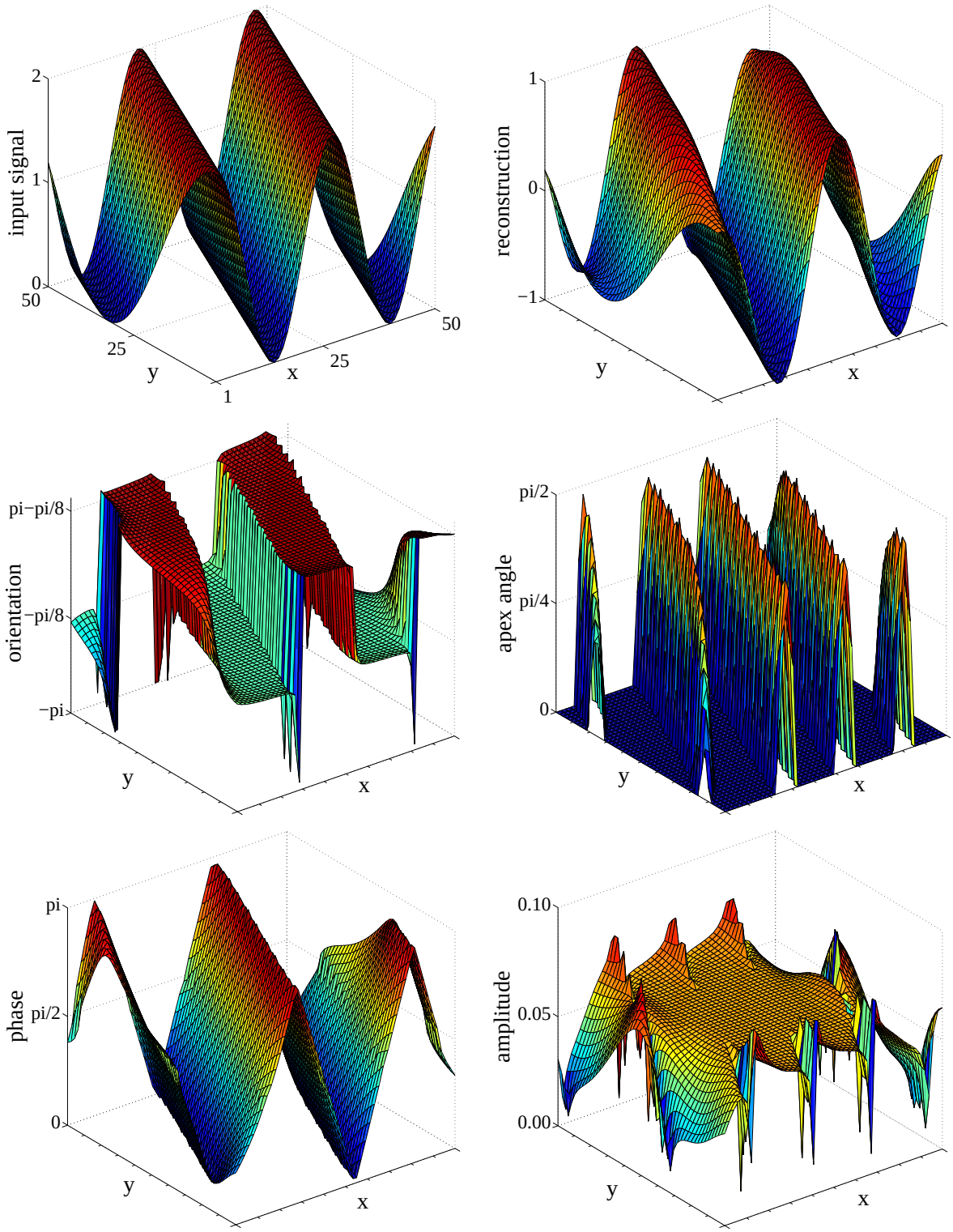


Figure 2.4.: Application of the analytic signal on a synthetic image with $a = 1$, $\theta = \pi/8$ and $s = 6$. The values of the analytic signal are $s_f = 5$ and $s_c = 7$. From the left to the right and the top to the bottom: input signal, reconstructed and normalised signal, θ, α, ϕ and a.

parallel sinus-oscillation. The distance of two maxima in the input signal is 24 pixels, π is represented by 12 pixels. The relation between the scale and the pixels is

$$s = \frac{\hat{\pi}}{2 \text{ pixel}} \tag{2.44}$$

with $\hat{\pi}$ as the size in pixels used to describe π. For this example it follows

$$s = \frac{12 \text{ pixel}}{2 \text{ pixel}} = 6.$$

The sine wave is exactly detected in a scale interval which includes scale $s = 6$. In our example, we have chosen the interval $[5, 7]$. The lower four plots depict the results of the analytic signal. The first picture is the mean orientation. It only adopts the values $-\frac{\pi}{8}$ and $\frac{7\pi}{8}$. This is exactly the rotation of the sine wave concerning the x-axis[8]. Only the crossover areas and the borders introduce small errors. Tracking the input signal from the left to the right, the mean orientation displays whether the input signal is ascending or descending - the higher value represents a locally ascending wave.
The next plot - the apex angle - is the angle between the two orientations in one point. In most cases the apex angle is '0', any calculated orientations are positioned in the same direction, it follows that the intrinsic dimension at this point is '1'. But there are also five ridges of values near to $\frac{\pi}{2}$. These ridges are located at extreme values of the input signal which are the points of changing the orientation. These positions are crossovers from one to another state[9].
The plot for the local phase ϕ represents the structure of the input signal. At $\phi = 0$ the input signal is reaching its maximum value, at $\phi = \pi$ the input signal becomes minimal. The linear development of ϕ between minimum and maximum value is only visible if the scale interval is chosen in a correct way.
The last plot shows the amplitude a. In the ideal case one expects a constant value. This cannot be matched at every point, because there are problematic fringe effects on the border of the input signal. The input signal f can be reconstructed using a and ϕ with

$$\hat{f} = a \cos(\phi). \tag{2.45}$$

[8] On an axis-parallel wave, the orientations would equal 0 and π.
[9] The width of these crossovers is induced by $w = s_c - s_f$. For $w \to 0$, also the width of the crossovers becomes '0'.

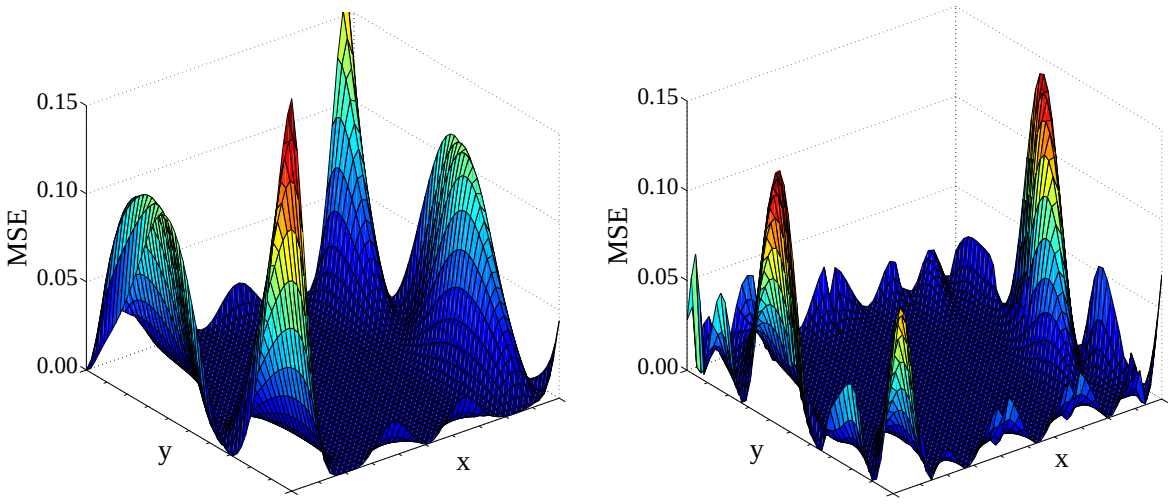


Figure 2.5.: MSE of the reconstructions with $\hat{f} = a\cos(\phi)$ and $\hat{f}' = \cos(\phi)$.

This reconstruction is depicted next to the input signal. Since the amplitude is constant[10], Zang [44] suggests an only phase-based reconstruction

$$\hat{f}' = \cos(\phi). \tag{2.46}$$

This reconstruction can be helpful in some cases, especially if the amplitude is not of any interest. The mean square error (MSE) between the two reconstructions and the input signal is depicted in 2.5. The following equation is used to compare both MSE's.

$$\frac{\text{mse}(\hat{f}) > \text{mse}(\hat{f}')}{X\ Y} = -0.08 \tag{2.47}$$

with $X\ Y$ as the number of pixels in the test image, mse is the mean square error and $a > b$ the function

$$a > b := \begin{cases} 1: & a > b \\ 0: & a = b \\ -1: & a < b \end{cases}. \tag{2.48}$$

The quality of both reconstruction methods is very similar[11].

[10] This makes sense only for this example.

[11] Equation (2.47) means: 46% of the pixels of $\hat{f}$ are afflicted with greater MSE than the same pixels of $\hat{f}'$.

2.4.4. Expansion to Multiple Scale Intervals

The practical application of the analytic signal consists of the comparison of the results for different scale intervals. For n scale intervals $\{[s_{f_1}, s_{c_1}], \ldots, [s_{f_n}, s_{c_n}]\}$ we calculate the analytic signal and compare the different results. An useful requirement is the disjunctive covering of the scale intervals

$$s_{f_1} < s_{c_1} = s_{f_2} < s_{c_2} \ldots s_{c_n}. \tag{2.49}$$

A simple requirement concerning a static shape of the convolution masks leads to[12]

$$q = \frac{s_c}{s_f}. \tag{2.50}$$

This can be realised by

$$s_i = 2^{(\frac{i}{d})} \tag{2.51}$$

with $s_{f_i} = s_{i-1}$ and $s_{c_i} = s_i$. An octave[13] is decomposed into d different parts.
A well known application is the analysis of the attenuation $\mathfrak{att}$ which is identical with the amplitude. For this certain partition of the scale space it reads

$$\mathfrak{att}_i^a = a_i \tag{2.52}$$

We take an one-pixel-stripe inside the input image of Figure 2.4 and apply the analytic signal n times. In our example, the scale interval borders run from 1 to 24, using 40 different intervals. The attenuation is maximal at the scale interval which includes the induced scale. Figure 2.6 depicts the results. The maximum ridge is positioned in the middle, for too fine or too coarse scales the responses are less intensive. The maxima only differ in the fore- and background, but these inaccuracies are the result of convolution. The induced scale can be detected by isolating the scale with highest attenuation.
The first plot in Figure 2.7 depicts the scale interval in which the attenuation reaches its maximum for every point. This procedure is proposed in [11], but the result is inaccurate: The located scales vary inside seven different intervals between $[4.18, 7.28]$; the main scale is located at 5.73. The results are only applicable in the areas with a maximum in the input image, in other areas the scale becomes too small, which means it is not ascertainable by the main scale detection. The maximum grey value in the input

[12] This requirement is not necessary but yields to a simple equidistant scale space where each octave has the same number of intervals, in other words $\forall i, j : (\log s_{c_i} - \log s_{f_i}) = (\log s_{c_j} - \log s_{f_j})$.
[13] Octave has the same meaning as known from music: For scale s the octave is given by $2s$.

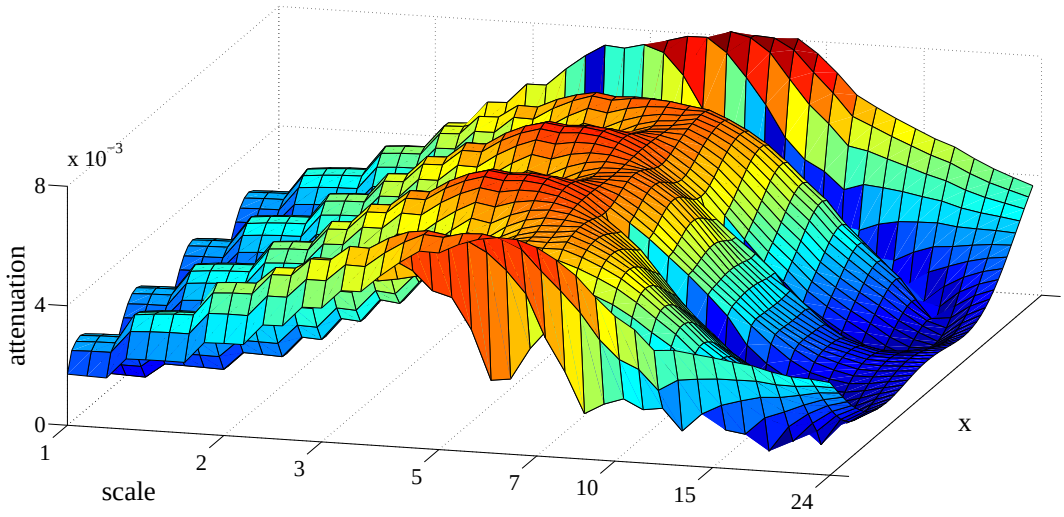


Figure 2.6.: 40 different attenuations for an image stripe in the input image of 2.4 at $y = 25$ with scales running from 1 to 24.

signal is indicated by $\phi \to 0$ or $\phi \to \pi$. In other words there is a relationship between the inaccuracy of the main scale detection operator (*error*) and the phase

$$error = 1 - |\cos(\phi)| \in [0, 1]. \tag{2.53}$$

This leads to the use of the phase to improve the main scale detection.
We introduce another measurement which replaces the attenuation $\mathfrak{att}^a$. We call this the new measure $\mathfrak{att}^b$ with

$$\mathfrak{att}_i^b = a_i |\cos(\phi_i)|. \tag{2.54}$$

The application of the maxima detection with $\mathfrak{att}^b$ is a bit more accurate, however it does not lead to an absolute elimination of the weak points. It obtains to high responses at points where the first method is too low. A benefit of this approach is the appearance of a large uniform plane between the error-wedges. The plot for this method is depicted in 2.7.
The combination of both methods lead to the final main scale detection:

1. calculate n attenuations based on $\mathfrak{att}_i^a$
2. exclude the $(n - d)$ scale intervals where $d > 0$ responses are below a certain threshold
3. take highest attenuation $\mathfrak{att}_i^b$ inside the d remaining intervals.

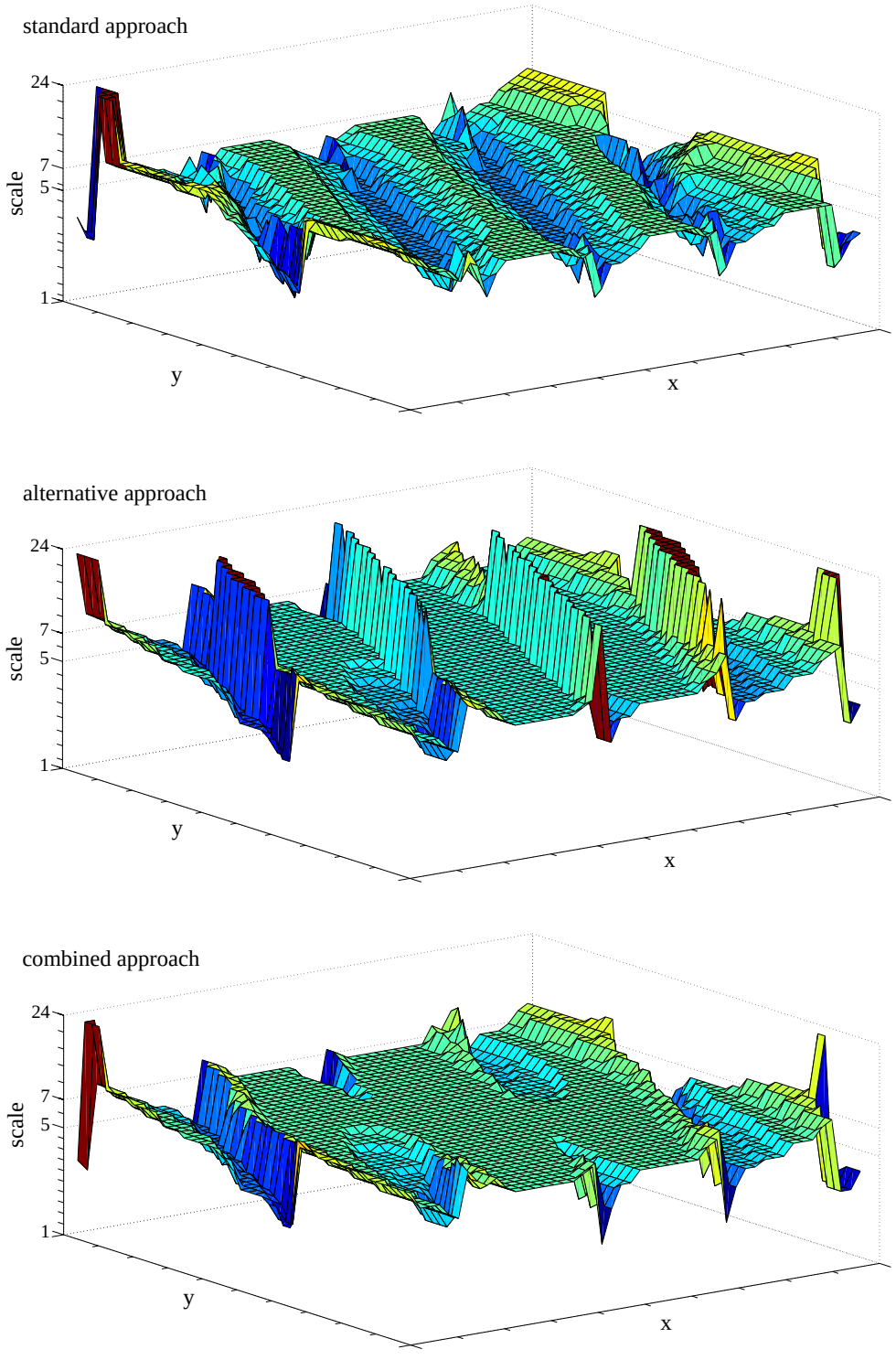


Figure 2.7.: Detection of the main scale.

One interesting point of this routine is how to find the threshold. There are two possible ways:

1. choose d of the highest responses for constant $d > 0$.

2. choose all responses higher than a given threshold, e.g. $threshold = factor \max(a)$ with $0 < factor \leq 1$.

In both cases we find a set of scales $\{s_{i_1}, \ldots, s_{i_d}\}$. An appropriate choice in the first approach is $d = \frac{n}{2}$. In the second approach $factor = 0.8$ leads to reasonable results. The last plot in Figure 2.7 shows the second approach with $factor = 0.8$.
This method is superior to the other two approaches. The advantages of the alternative method are combined with an outlier-elimination using the normal method.

So far, the considered attenuations were only based on a or a and ϕ, respectively. We will now introduce an approach only based on the phase congruence. This approach uses a property of exactness of a local scale in p. It finds the scale in p, having the minimum difference to other adjacent scales concerning the phase response. In our definition the method operates on the modulus of differences[14]

$$\mathfrak{att}_i^c = 1 - \frac{|\cos(\phi_i) - \cos(\phi_{i-1})| + |\cos(\phi_i) - \cos(\phi_{i+1})|}{4} \tag{2.55}$$

with $\mathfrak{att}_i^c \in [0, 1]$ having a good value close to '1'. The best scale includes the highest possible variance of a band pass width at lowest variance concerning the detected structure. The detected scale will also be called the main scale.
The detected structures differ from those, detected by the other methods, because the amplitudes are ignored and phase and amplitude are in general linear independent. There is no quantitative information about the intensities concerning signal response, rather than qualitative information about separateness of the scale, hence the reconstruction using phase congruence differs from other methods in a fundamental way.

2.4.5. Scale Detection on Multiple Waves

As in real images, there is always a local superposition of many arbitrary waves, we will consider the behaviour of the scale detection on two arbitrary waves. Figure 2.8 shows

[14] For the following definition it is important to assume a comparability between different scale intervals. In a simple approach using the same number of intervals per octave it is already guaranteed.

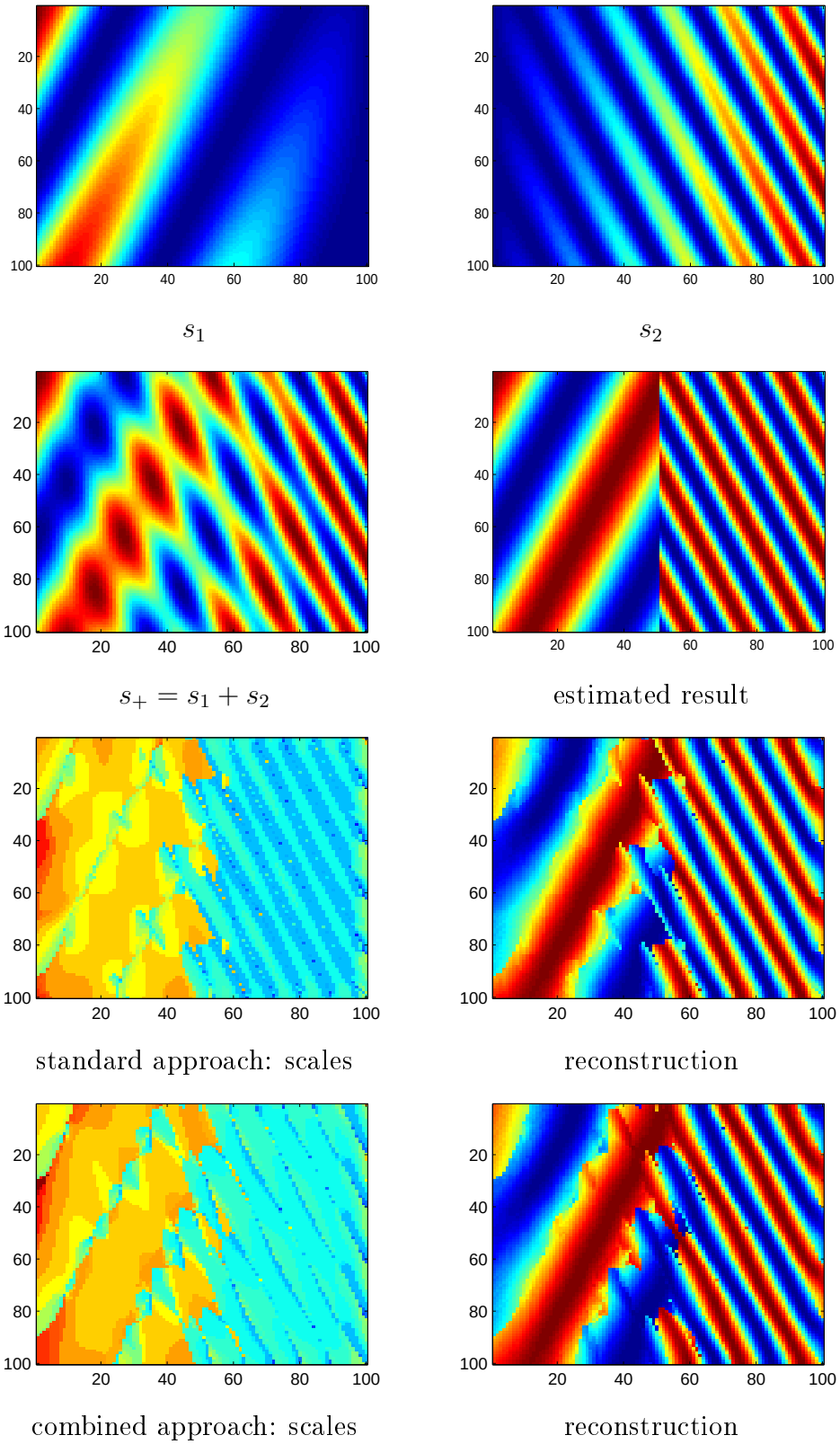
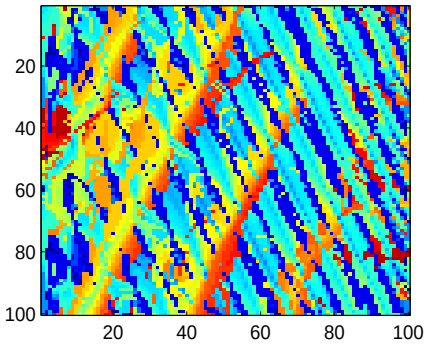
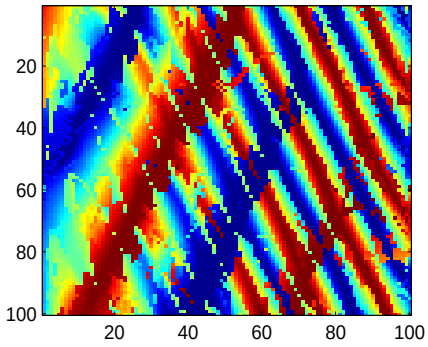


Figure 2.8.: Main scale detection on the signal s_+ using the standard approach and the combined approach.

the construction of our input signal. The signal is built by two sine waves with different scales and orientations. The first wave is three times larger than the second. The intensities are distributed in a way, that the first wave descends linearly from the left to the right and the second behaves oppositely. The crossover of the intensities is located in the middle of the picture. The third picture displays the overlay of the signals, which is the input signal for the scale detection of the three proposed methods. The fourth figure is the estimated result for the standard method and the combined method.
The next four plots show the results of the two amplitude-based main scale detectors. The left images depict the detected scales. The right image is the phase-based reconstruction $\hat{f}'$ using only these scales.
The standard approach is also known as 'maxima detection'. The combined method is characterised by a higher concentration of a detected scale in a certain part.
The first two plots in 2.9 depict the application of the phase congruence. The next figure is the histogram of the detected main scales for the three methods. The phase-congruence based method induces another kind of signal than the amplitude based approaches: The correct scales are only detected at extreme points concerning a certain scale which are points with $\phi = 0$ or $\phi = \pi$. The amplitude-intensity is ignored.
The histogram depicts the differences of the three methods. The estimated result consists of two peaks at $s = 4$ and $s = 12$. The peaks of the three methods are marked by a square.
The standard approach detects two peaks for the fine wave at $s = 3.3$ and $s = 4.5$. For the coarse wave it detects two peaks at $s = 10$ and $s = 12$.
The combined method performs better. It just detects the two estimated peaks at $s = 4.5$ and $s = 12$. In this sense, it is superior to the standard approach.
The phase congruence generates the worst results as it detects a huge false positive rate at $s = 1.5$. The other two local maxima are given by $s = 5$ and $s = 12$, whereas the number of occurrences of these two maxima is only a bit higher than the number of the adjacent occurrences.
The computer code for the main-scale detection is given in listing A.5, the code for the function which generates the synthetic image is given in listing A.6.



phase congruence: scales



reconstruction

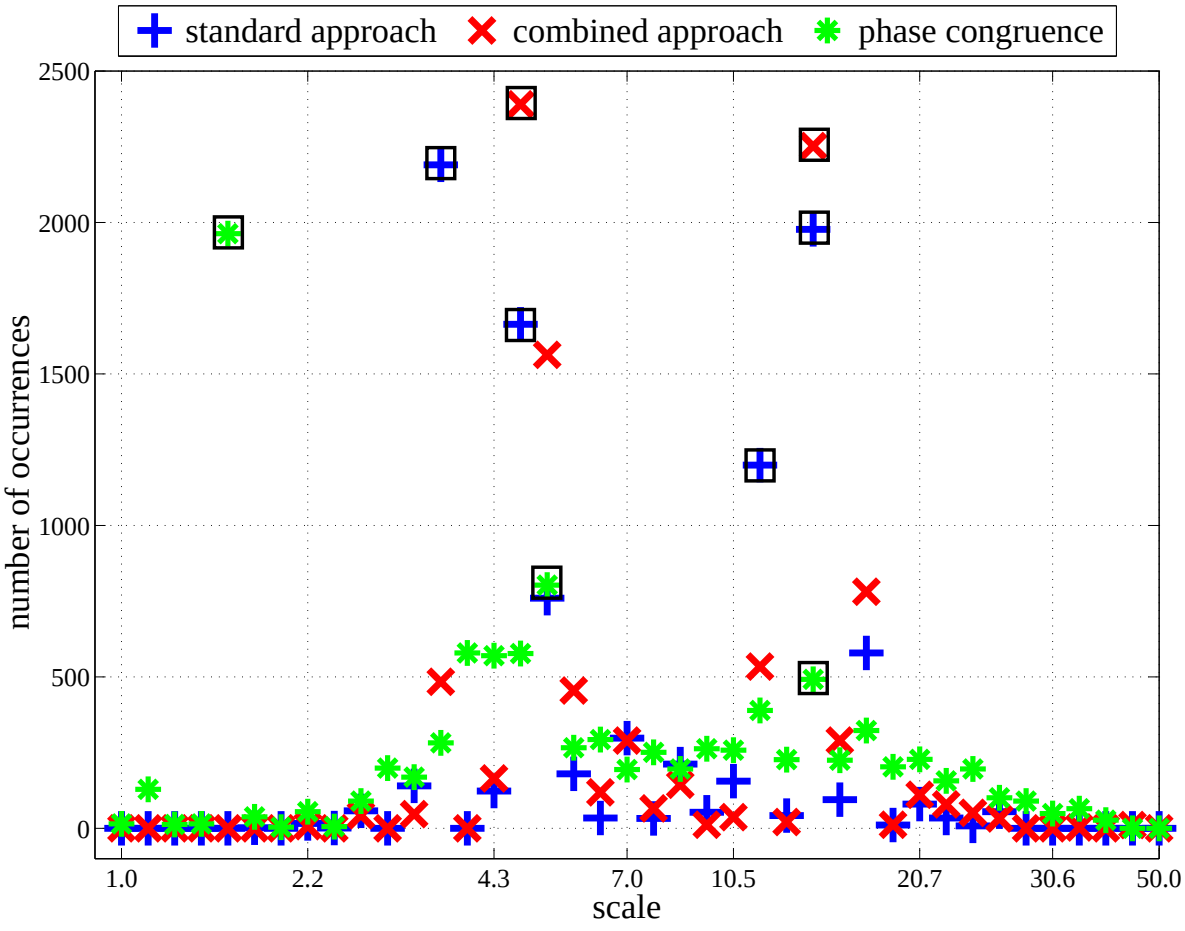


Figure 2.9.: Top: Application of s_+ with the phase congruence approach. Bottom: Histogram of the detected scales. The squares mark the peaks of each approach.

2.4.6. General Definition of the Attenuation

If we ignore the constraint, that we need the same number of intervals per octave, the general definition of the first attenuation reads

$$\mathfrak{att}_i^a = \frac{a_i}{\log(s_{c_i}) - \log(s_{f_i})}. \tag{2.56}$$

For the second attenuation it follows

$$\mathfrak{att}_i^b = \frac{a_i|\cos(\phi_i)|}{\log(s_{c_i}) - \log(s_{f_i})}. \tag{2.57}$$

The phase congruence is defined by

$$\mathfrak{att}_i^c = 1 - \frac{|\cos(\phi_i) - \cos(\phi_{i-1})|w_r + |\cos(\phi_i) - \cos(\phi_{i+1})|w_l}{2(w_l + w_r)} \tag{2.58}$$

with $w_r = \log(s_{c_{i+1}}) - \log(s_{f_i})$ and $w_l = \log(s_{c_i}) - \log(s_{f_{i-1}})$.
These definitions allow the usage of every partition of the scale space and not only the one given in Equation (2.51). Now it is possible to separate the scale space by

$$[s_{f_i}, s_{c_i}] = [(i-1)c_I + c_O, ic_I + c_O], \text{ with } c_I, c_O \in \mathbb{N}^+ \tag{2.59}$$

with c_I the interval size and c_O the offset. It also satisfies the constraint of disjunctive covering.
The code for the calculation of the three attenuation-types can be found in A.5.

2.4.7. Modified Image Reconstruction

If the sum of all scale intervals covers each important wave, the image can be reconstructed by

$$\hat{f}(x, y) = \sum_{i=1}^{n} a_i \cos(\phi_i). \tag{2.60}$$

Using Equation (2.60) we introduce the modified reconstruction by introducing an array of factors b

$$\hat{f}(x, y) = \sum_{i=1}^{n} b_i a_i \cos(\phi_i), b_i \in [0, 1]. \tag{2.61}$$

With the support of the modified reconstruction we can design a large set of different phase- and amplitude-based filters. It allows us to suspend or emphasise some scale

responses according to the information given by a local histogram or previous knowledge. In Section 2.4.4, the reconstruction satisfies the definition

$$b_i := \begin{cases} 1: & s_i = \text{main scale} \\ 0: & \text{otherwise} \end{cases} . \tag{2.62}$$

This method is also applicable to noise reduction using the analysis of the respective attenuation. Low values of attenuation can be considered as noise, whereas too high attenuations can be determined as scratches on the surface of a scanned photography.
After noise reduction the resulting signal $\hat{f}$ can be considered as a new input image for further computations.
We come back to this definition in the main chapter of this thesis, as it will be the basis of the SSSF.

2.4.8. A Fast Implementation of the Analytic Signal

We finish the chapter about the analytic signal with a description of a fast performing implementation. We introduce a convolution pyramid and an instruction for the implementation on the GPU.
As already mentioned, an introduction to the implementation in C code is given in [40]. From the programmers viewpoint, the convolution draws most of the attention. The analysis of a neighbourhood at a point p with radius n takes $(2n+1)^2$ image- and filter-mask-accesses[15] combined with the same number of multiplications and sums. This complexity of $\mathcal{O}(n^2)$ leads to a non-acceptable draw-back for the application especially for big convolution sizes. The only way to reduce the number of calculations is to restrict the considered (image-) points by only paying attention to every dx-th pixel[16]. This approach is also known as convolution pyramid [1]. The mask-calculation does not need to be changed using the relationship

$$\hat{s} = \frac{s}{dx}. \tag{2.63}$$

The shortened convolution mask has the same design and size as the convolution mask created by shortened scale parameters:
Consider a convolution mask m with parameters s_c and s_f and the size $kn \times kn$. Computing the sum for every $k \times k$-square results in an $n \times n$ mask $\hat{m}$. This new convolution

[15] The main problem is generally the speed of the image accesses.
[16] The value dx is the offset between two considered points on the signal.

mask has the same design and sum as the first one, but less entries. Because the design is defined by the $q = \frac{s_c}{s_f}$ and the size is linear in s_c it follows that $\hat{m}$ equals a convolution mask created by $\hat{s_c} = \frac{s_c}{dx}$ and $\hat{s_f} = \frac{s_f}{dx}$.
The application of the offset dx on the first kernel q^p is depicted in Figure 2.10. The height and the design is not touched, whereas the resolution is minimised.
The opposite part of the convolution - the input image - must be preprocessed by a mean value convolution with the same mask size as dx. A good approach is the limitation of the maximum convolution mask size to n_{max}. For a scale interval with convolution size n, the value dx is computed by

$$dx = \left\lceil \frac{n}{n_{max}} \right\rceil . \tag{2.64}$$

This decreases the complexity of the convolution to a maximum of $(2n_{max} + 1)^2$ steps which is independent from n. Although the calculation for n_{max}^2 takes time, this part has an effort of $\mathcal{O}(1)$ concerning n.
The calculation of the mean value can be decomposed by an iterative approach:
We need two different convolution masks, the one to have an odd number n of elements for calculating the mean values reads

$$M_{odd}^n = \frac{1}{n^2} \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} . \tag{2.65}$$

For an even number of n we need an $(n+1) \times (n+1)$ matrix

$$M_{even}^n = \frac{1}{n^2} \begin{bmatrix} 0.25 & 0.5 & \dots & 0.5 & 0.25 \\ 0.5 & 1 & \dots & 1 & 0.5 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0.5 & 1 & \dots & 1 & 0.5 \\ 0.25 & 0.5 & \dots & 0.5 & 0.25 \end{bmatrix} . \tag{2.66}$$

When calculating different mean values $f_{odd}^i := f * M_{odd}^i$ and $f_{even}^i := f * M_{even}^i$ it is reasonable to use the results iteratively. The origin equals a 2×2 convolution mask

$$f_{odd}^2 = f * \begin{bmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \end{bmatrix} . \tag{2.67}$$

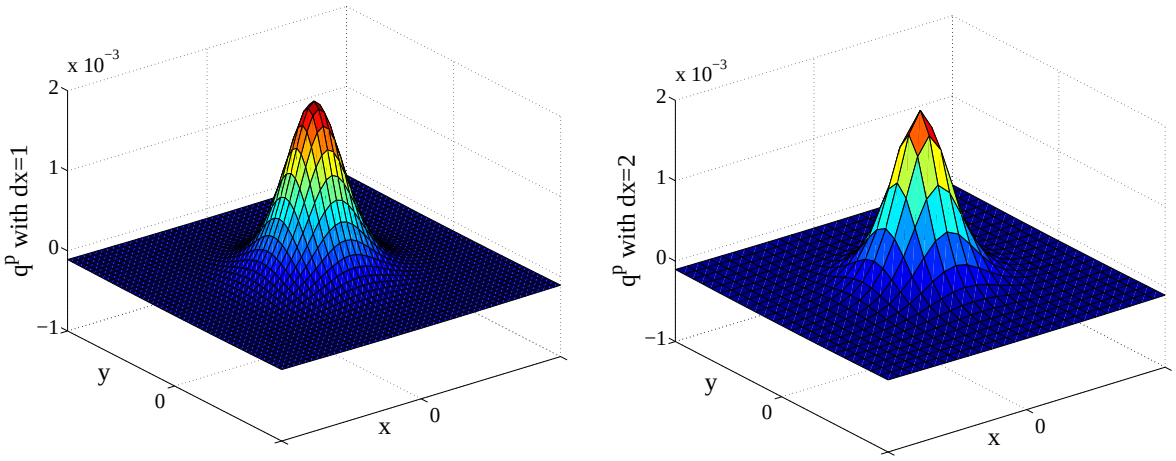


Figure 2.10.: Convolution mask q^p as in 2.2 and its resulting equivalent with $dx = 2$.

The iteration starting at $i = 3$ reads

1. calculate f^i_{even} using f^{i-1}_{odd}
2. calculate f^i_{odd} using f^i_{even},f^{i-2}_{odd} and f.
3. increase i.

It is possible to store the required information in the four slots of the graphic-card, hence $p[r, g, b, \alpha] = p[f_{cur}, f^j_{odd}, f^{j-1}_{odd}, f]$ with $p[r, g, b, \alpha]$ being the four channels of one image pixel.
Initially $p[f^2_{odd}, f^1_{odd}(= f), f^2_{odd}, f]$ becomes starting at $i = 3$

- $p[f^i_{even}, f^{i-2}_{odd}, f^{i-1}_{odd}, f]$
- $p[f^i_{odd}, f^{i-1}_{odd}, f^i_{odd}, f]$.

The start-equations for the mean value calculation steps read

$$f^1_{odd}(x, y) = f(x, y) \tag{2.68}$$

$$f^2_{odd}(x, y) = \frac{1}{4} \sum_{x',y'=0}^{1} f(x + x', y + y'). \tag{2.69}$$

The further equations are given by

$$f^i_{even}(x,y) = \frac{1}{4}\sum_{x',y'=0}^{1} f^{i-1}_{odd}(x-x', y-y') \tag{2.70}$$

$$\begin{aligned} f^i_{odd}(x,y) = \frac{1}{i^2}\Bigg[& 2(i-1)^2 f^i_{even}(x,y) - (i-2)^2 f^{i-2}_{odd}(x,y) \\ & + \frac{1}{2}\sum_{x',y'=0}^{1} f\left(x + \frac{i-1}{2}(-1)^{x'} + \frac{i-1}{2} - \left\lfloor \frac{i-1}{2} \right\rfloor \right. \\ & \left. , y + \frac{i-1}{2}(-1)^{y'} + \frac{i-1}{2} - \left\lfloor \frac{i-1}{2} \right\rfloor \right) \Bigg]. \end{aligned} \tag{2.71}$$

The used values of f are depicted in Figure 2.11. The terms f^i with $i \in \{2,4,6,\ldots\}$ are

dx	1	2	3	4	5	...
$\overline{f}$	f^1_{odd}	f^3_{even}	f^3_{odd}	f^5_{even}	f^5_{odd}	...

Figure 2.11.: Masks for the calculation of the mean value

not used for the calculation of the mean value.
From one step to the next one, we need only 5 or 13 image accesses[17], independent from dx. This leads to a time complexity of $\mathcal{O}(1)$. The time complexity of the convolution and the calculation of the mean value on an $n \times m$ pixel image having s different scale intervals is in $\mathcal{O}(n \times m \times s)$.
Obviously we are going to lose some information by applying this pyramid, but the mean square errors between two following values of dx are still reasonably small. The application on the sine wave - as in Figure 2.4 with $dx = 1$ and $dx = 2$ - is depicted in Figure 2.12. The implementation with this special convolution pyramid is given in the Appendix A.1.
On a graphic processor unit there are some continuative possibilities for acceleration. The analytic signal is defined on grey scale images, we only need one channel for the image information. When using GLSL[18] there are always four different channels per pixel[19]: three channels for red, green and blue and one channel for the alpha value. The

[17] For f^i_{odd} we need 4 pixel accesses on the border edges of f and one pixel access on f^i_{even} and f^{i-2}_{odd} which are stored in the certain pixel; for f^i_{even} with odd value of i we need to calculate f^{i-1}_{even} with 4, f^{i-1}_{odd} with 5 and finally f^i_{even} with 4 pixel accesses.
[18] Open **G**raphics **L**ibrary **S**hading **L**anguage is the language, which is used for the implementation on the GPU.
[19] a GPU register consists of a vector having 4 floats. It follows that the processor always calculates 4 floats per clock. However one does not have to use these 4 channels.

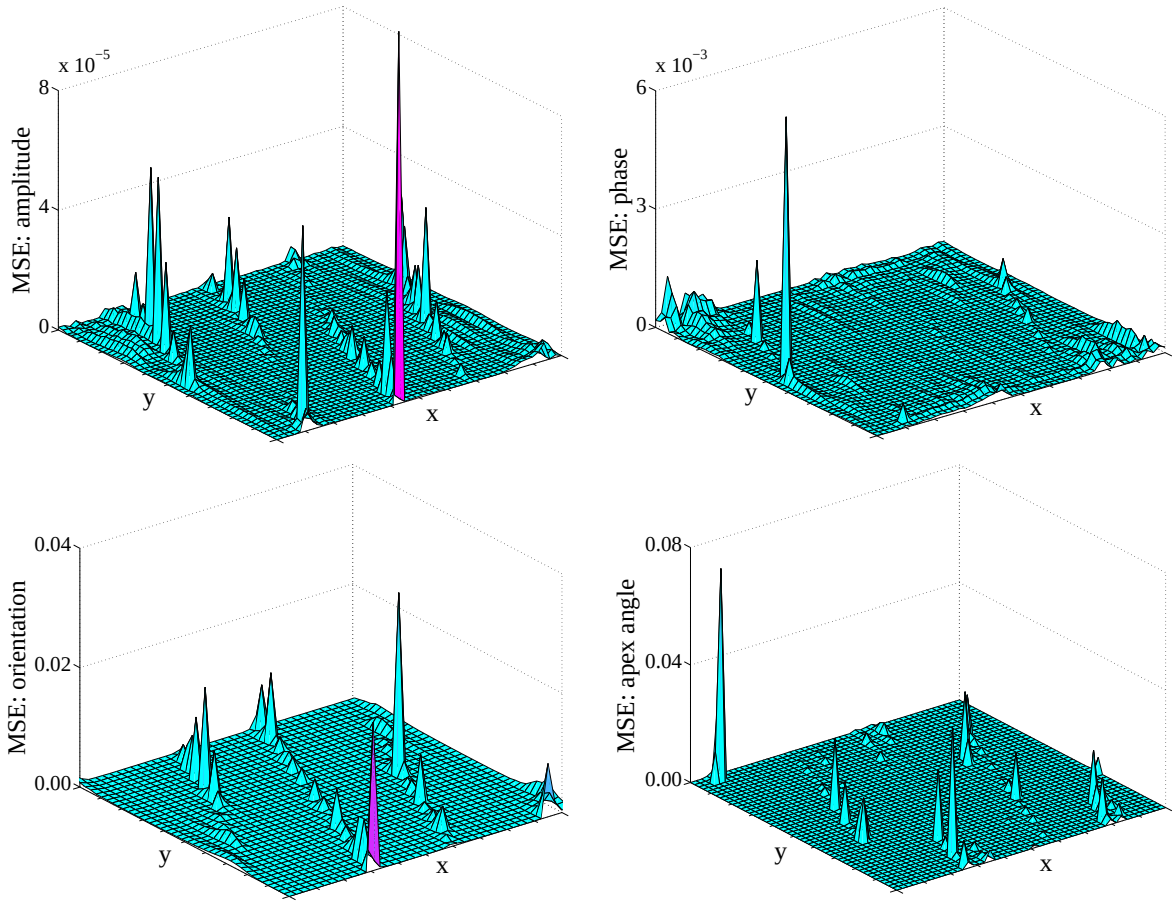


Figure 2.12.: MSE's for the the application as in 2.4 between $dx = 1$ and $dx = 2$. First row: a and ϕ. Second row: θ and α.

access to a pixel always uses these four channels. A common way to use this for grey scale images is to put the adjacent image pixels into the three remaining free channels, this results for a certain dx in

$$p[r, g, b, \alpha]_{(x,y;dx)} = [\overline{f}_{(x,y)}, \overline{f}_{(x+dx,y)}, \overline{f}_{(x,y+dx)}, \overline{f}_{(x+dx,y+dx)}] \tag{2.72}$$

with $\overline{f}$ being the input signal after mean valuing as in 2.11. It reduces the image accesses by factor 4. To accelerate the mask acesses[20] we only submit the non-redundant parts because the fast memory on a GPU is quite small. When calculating the complete masks outside the GPU there are two different types of masks. As shown in Figure 2.2 for the Poisson kernel it is enough to transmit an eighth slice. For the other kernels we need at

[20]It is unavoidable to calculate the mask before pushing into GPU, because it is static for all pixels and the calculation (for each pixel) is much more complex than an storage access.

least a quarter slice for the kernel reconstruction. Furthermore the kernels q^x and q^y, respectively q^{xx} and q^{yy} are identical except for rotation. This results in a minimum requirement of memory

$$place_{\min} = \underbrace{\frac{(n+1)n}{2}}_{q^p} + \underbrace{2(n+1)n}_{q^x \text{ and } q^{xx}} + \underbrace{n^2}_{q^{xy}} \tag{2.73}$$

for a given maximal kernel size of $(2n+1)^2$. This is a reduction to about $\frac{7}{48}$ of the complete number of the mask entries. However depending on the specific GPU it can be more useful to transmit some redundant information to reduce the jumps concerning the array access. The optimal way with respect to the speed and the image representation is an uniform transmission of a quarter per mask without the values $x = 0$ or $y = 0$ plus the stripe having $x = 0$, respectively $y = 0$. This results in a memory requirement of

$$place_{\text{opt}} = \underbrace{4n^2}_{q^p, q^x, q^{xx}, q^{xy}} + \underbrace{3n}_{\text{additional for } q^p, q^x \text{ and } q^{xx}} + \underbrace{1}_{\text{middle of } q^p} \tag{2.74}$$

which is still a reduction to about $\frac{1}{6}$ and accelerates the convolution-time. We implemented the calculation of the analytic signal for $place_{\min}$ and $place_{\text{opt}}$ in C++, OpenGL[21] and GLSL. We used an NVidia GeForce 9600GS GPU[22]. The faster method ($place_{opt}$) is restricted to a convolution of 33^2 pixels, the second one allows up to 65^2 pixels per mask. A test on a 0.25 mega pixel image and another 1 mega pixel image gave the in 2.13 depicted frame rates. The calculations are stopped if the frame rate becomes lower than 5 frames per second.

For $place_{opt}$, $n = 16$ and the smaller image, we achieved 45 frames per second, for $place_{\text{opt}}$, $n = 16$ and the larger image, we achieved 19 frames per second.

The calculations with the $place_{\min}$ approach takes much more time. For the larger image, it effects an abortion of the calculation at $n \geq 20$. As the convolution is the most time consuming part of the calculation, we are able to compute the analytic signals for 19 scale intervals with $n_{\max} = 16$ in less than one second for the larger 1024×1024 pixel image or to compute one frame in real-time[23].

The calculation of the mean value takes not longer than the calculation of the analytic signal with $n = 1$ and $place_{\text{opt}}$, which is 727 frames per second for the smaller image and 448 frames per second for the larger one.

[21] **Open G**raphics **L**ibrary

[22] The complete specification of our our GPU is given in the Appendix B.

[23] In this case, real-time is defined as a frequency of 20 Hz.

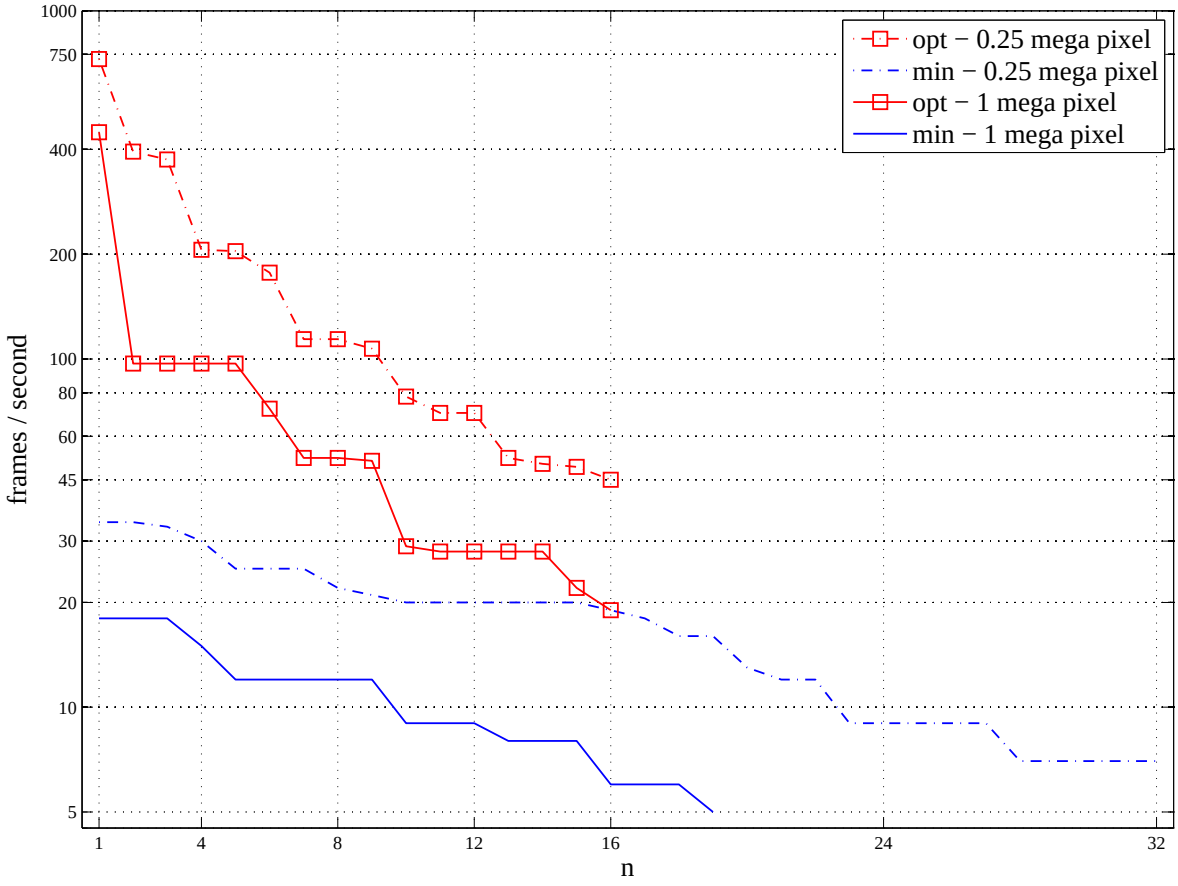


Figure 2.13.: Frame rates for the analytic signal calculation on two 'Lena' - images with 512×512 and 1024×1024 pixels.

The code of the fragment shader which is the significant part of the calculation on the GPU is listed in the Appendix B.

3. The Scale Space Segmentation Filter

In this chapter we develop a new segmentation filter which operates on the 2D analytic signal transform. We call it **S**cale **S**pace **S**egmentation **F**ilter[1]. The main scope of application is the detection whether a pixel is part of a special texture or not[2]. The input is a grey scale image as in the case of the analytic signal. The output is also a grey scale image providing the likelihood of each pixel being part of the texture. The range of this filter is neither region based nor edge based, but pixel based in a special way. Hence, the results are generally not connected like normal region- or edge-based segmentation outputs nor does the segmented pixel set grow or shrink from one iteration to the next. Another important attribute is the rotation invariance which arises with the inherited rotation invariance of the analytic signal which is the only not single-point operation in this filter.

As texture segmentation is a central task for computer vision, we can find a lot of different publications concerning segmentation or detection routines. We will list some works which are related with this thesis. There are publications about the segmentation in general and about segmentation on medical images. There are also publications, which describe the phase-based approach.

Brandt [4] gives a programme for the detection of the spine in his doctoral thesis. He uses template matching and simulated annealing on the grey value input images, with the limitation not being luminance, rotation, size or design invariant. He reaches segmentation results with a true positive value between 0.75 and 0.85.

The authors of [24] propose a lung segmentation, which is based on thresholds and histograms on grey scale images, the authors of [35] make a similar approach for the liver segmentation. Kaminsky et al. publish an interactive tool[3] for a 3-dimensional segmentation of the spine [21], which is basically a region growing in three dimensions.

There are also some approaches using the watershed transform. The authors of [25] expose a segmentation approach, combining the watershed transform with the region

[1] In the following, we use the abbreviation SSSF .

[2] In our case, the region of interest (ROI) is given by a texture.

[3] An interactive segmentation tool normally needs a pre-allocation of the segmentation by the user, before it starts to calculate the final segmentation.

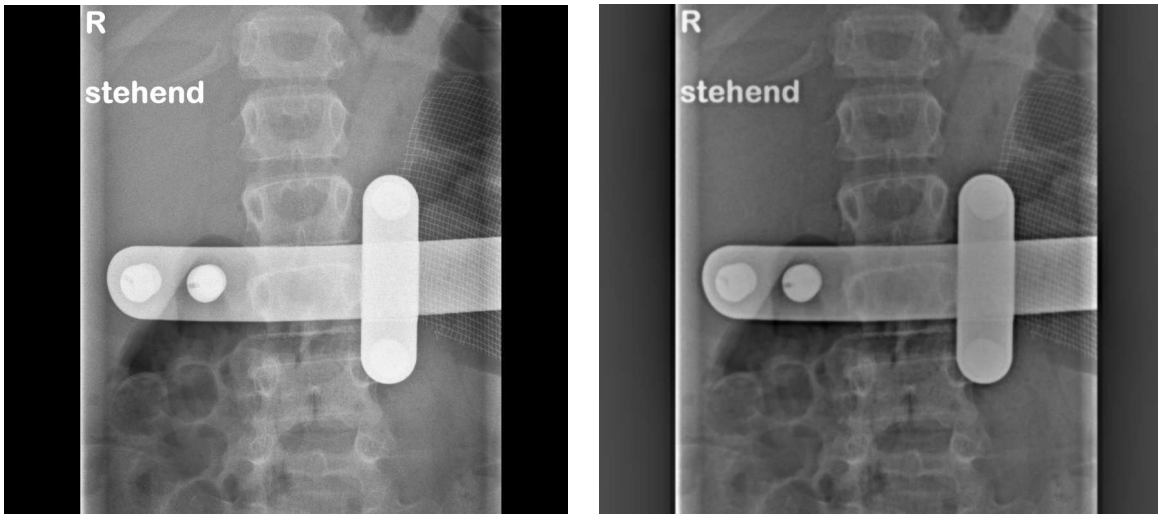


Figure 3.1.: Input image and reconstructed image

growing, similar to [28], [17] or [5]. Also the watershed transform was motivated by this medical task. A spine segmentation is given in [38]. A segmentation of the brain with the magnetic resonance tomography is given in [3], [15] and [29]. Kindler et. al [23] use a two scale framework with an energy-driven operator and an adaption of a positioned shape on the structure.
The field of the phase based approaches is dominated by edge-detector-based or feature-detection-based segmentations. The authors of [19] give a phase-based edge detector. The author of [34] presents a feature detector based on the monogenic signal, the publications [31], [16] and [18] give solutions for phase based feature detectors using ultrasound photographs.
This chapter is split in a short introduction and the description of the fundamental components. Then we discuss the different possible designs of the components and their different properties. At the end we will show some results and give an outlook.

3.1. Motivation: Exclusion of Surrounding Area

The development of the SSSF is motivated by a spine detection task. Consider a computer tomography (CT) photography of a human's upper part of the body on which we want to locate the spine. Generally the spine and other bone structures are lighter than softer parts. But also other structures, like synthetic objects as implants, lettering and image boundaries can reach the same grey scale value as the bones. In a first step we want to exclude these blockages to obtain a plainer view on the spine.

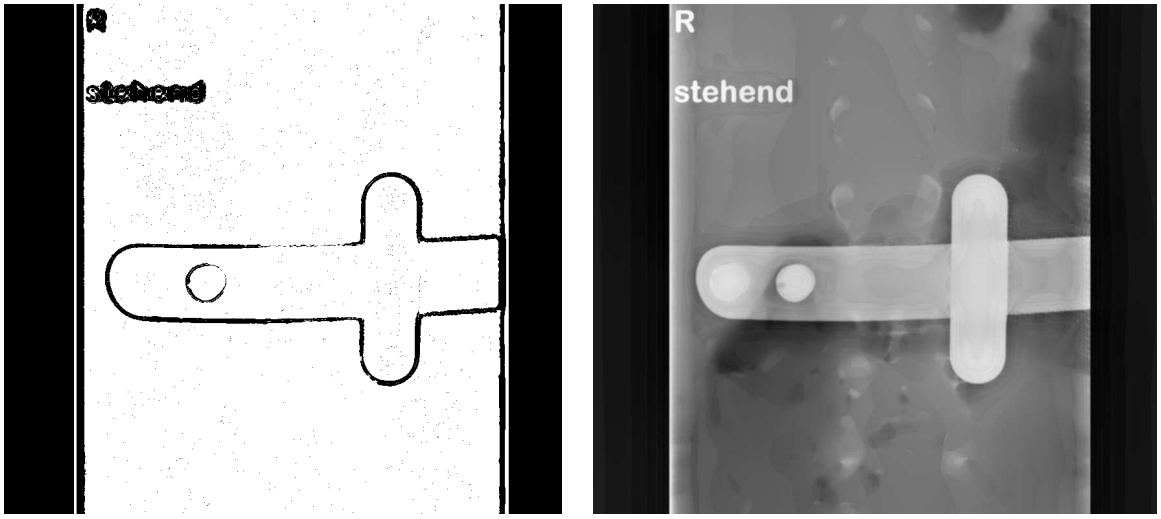


Figure 3.2.: First image: Band pass example for $\mathfrak{att}_1^a$. White: $\mathfrak{att}_1^a$ inside the band pass. Second image: Modified reconstruction for a low pass filter.

In Section 2.4.7 we introduced the reconstruction of the image using the phase ϕ and the amplitude a. We also considered a modification using a signal $b \in [0,1]$ which allows us to filter noise or scratches of a photograph in dependence of the attenuation $\mathfrak{att}$.
Figure 3.1 depicts a normal reconstruction of an image like in (2.60). The reconstructed image is not totally equivalent with the source image, in particular fine details get lost. Furthermore the image looks washed-out, as the black or white regions are discoloured[4].
Figure 3.2 illustrates the power of constructing a band pass filter inside the attenuation. In the first image in 3.2 we use a simple band pass filter

$$b(x,y) := \begin{cases} 1 & : \mathfrak{att}^a(x,y) \in (t_{low}, t_{high}] \\ 0 & : \text{otherwise} \end{cases} . \tag{3.1}$$

This filter maps the pixels within a certain range of attenuation to one, and these with higher or lower attenuation to zero. The considered scales are pretty small[5].
The resulting image b is an inverted edge detector for the high pass - as especially strong edges get high amplitudes. For low amplitudes b equals a plateau detector, as low amplitudes go with low grey scale changes in the input image[6]. Hence the attenuation $\mathfrak{att}^a$ induces a simple edge detector.

[4] As we can only apply a finite set of different scales, we accumulate every detailed information inside one scale interval.
[5] In this example we used is $s_f = 1$ and $s_c = 2$.
[6] The amplitude has similar issues as the modulus of the derivation

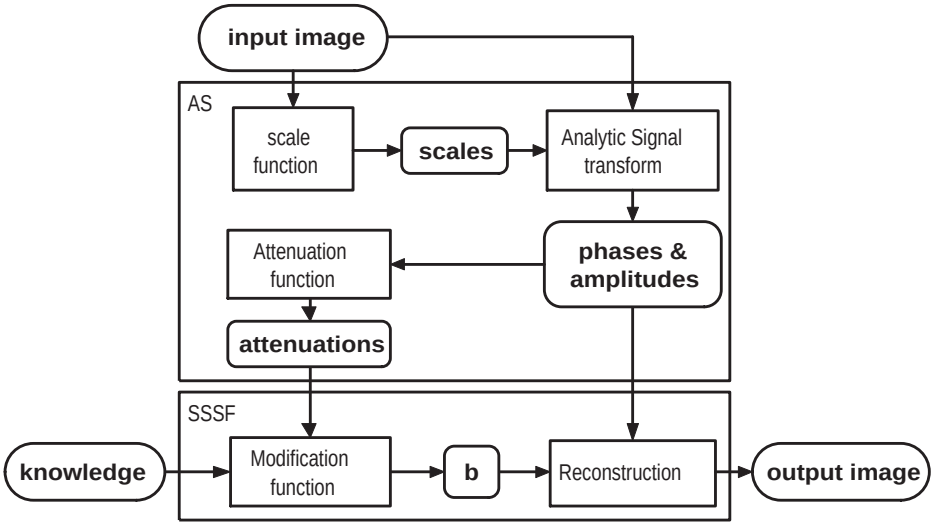


Figure 3.3.: Principal procedure for filtering inside the analytic signal scale space.

A noise detector can also be reached with (3.1) but with values $t_{low} = 0$ and $t_{high} - t_{low} = \epsilon$ with ϵ being small. Noise is characterised by attenuations just greater than zero but very small.
The second image in 3.2 depicts the modified reconstruction as in (2.61). The mapping for b_i is a high pass

$$b_i(x,y) := \begin{cases} 1 & : \mathfrak{att}_i^a(x,y) > t_{high} \\ 0 & : \text{otherwise} \end{cases} . \tag{3.2}$$

The resulting reconstruction contains almost exclusively synthetic objects like the implant and lettering, whereas the spine and the soft parts of the photograph are excluded. We ask if there is a band pass based approach which is not only able to detect edges or objects with extreme grey values, but every well defined texture. As we will see, this question can be answered in a positive manner.

3.2. Components of the SSSF

The SSSF is based on band passes of the attenuation as in (3.1) but with more degrees of freedom to adapt the desired texture.
The flowchart in Figure 3.3 shows the main concept of this filter. The input is a grey scale image and some knowledge about the texture information. The output image is the modified reconstruction similar to the second image in 3.2 which contains a highlighting

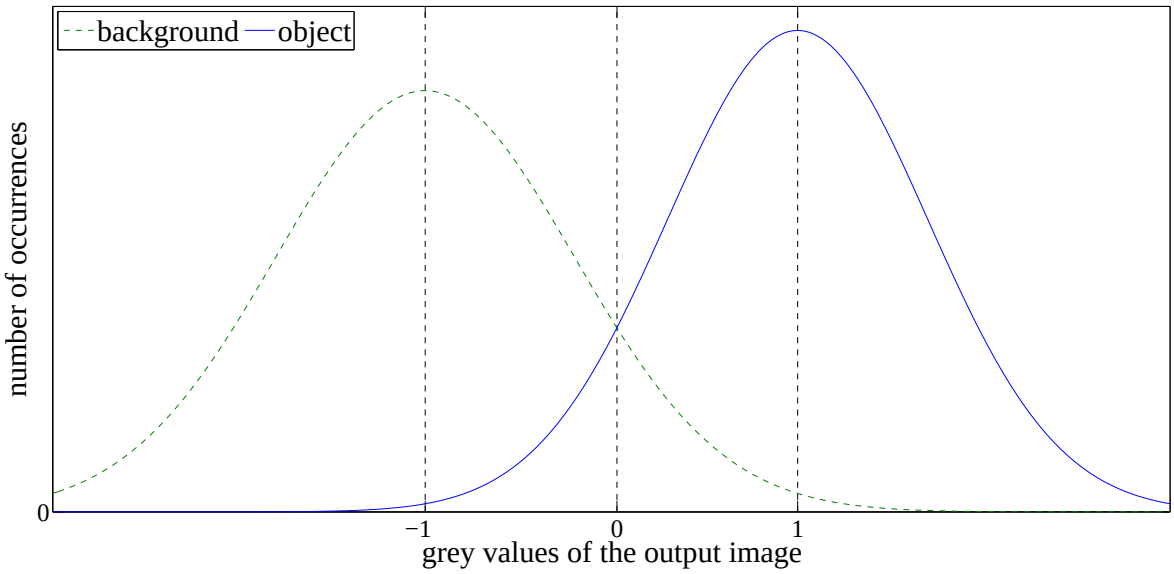


Figure 3.4.: Histogram of the output image with same number of object pixels and background pixels.

of the searched texture.

In the first step the amplitudes, phases and attenuations are calculated. The scale function calculates the size of the input image and returns a number of scale intervals `scales`. The scales and the input image are transformed with the analytic signal. It returns phases and amplitudes which are mapped to a set of attenuations by the `Attenuation function`.

The SSSF computes a set of `b`-signals similar to the ones in 3.2, the band pass parameters are stored in `knowledge`. Finally the information in b, containing the texture information, and the phase and amplitude- signals are merged in the reconstruction function which returns the output image.

This output image is not yet the segmented image but it is a colouration for each pixel, which depicts the likelihood of being part of the texture or part of the background. It is possible to stretch and move the results of the output image to obtain the scale like in Figure 3.4. A simple segmentation can be reached by applying a threshold on the output image.

In the next sections we will explain the low-level and the higher-level components. The low-level components are the scale function, the attenuation function and the reconstruction. The higher-level components are the different modification functions.

3.3. Low-Level Components

3.3.1. Scale Function

The scale function computes the scale intervals for the analytic signal transform. The input images which contain the same texture class may not have the same resolution, but if the input images have same resolution the scale function returns constant scale intervals, independent of the input image.
The scale intervals are given by

$$s_i = \exp\left(i\frac{\log(s_{\max})}{n}\right), i \in \{0, \ldots, n\} \tag{3.3}$$

with $s_{\max}$ the highest coarse scale. Let h be the height of the input image and w its width. The maximal scale $s_{\max}$ can be estimated by

$$s_{max} = f(y\ h, x\ w), f\ \ \in \{\min, \max, +\} \text{ and } a, b \in \mathbb{R}^{\geq 0}. \tag{3.4}$$

For the spine data we use values $y = 0.092, x = 0$ and $f = +$. We only consider the image height[7].
The resulting value for $s_{\max}$ must barely cover the object diameter of the largest pattern in the texture. In our case, this is the spine body which is not exceeding $s_{\max} = 0.092 \cdot h$.

3.3.2. Attenuation Function

The attenuation function maps the phases and amplitudes of each scale interval into three different attenuations. In our case, the attenuation is the signal, on which the band passes are computed. We use the attenuations $\mathfrak{att}^a$, $\mathfrak{att}^b$ and $\mathfrak{att}^c$ which are described in detail in (2.56), (2.57), and (2.58).

3.3.3. Reconstruction

We will consider four different types for the modified reconstruction. The original modified reconstruction is (3.5). The other reconstruction types are actually derivatives of the first one. Only the first equation leads to accurate reconstructions like in Figure 3.1 by setting $b = \mathbf{1}$. The others are exclusive for the modified reconstruction as in Figure 3.2.

[7] We consider medical images of the spine, these images only share being photographs of the upper part of the body. Hence the height is the only constant information of these photographs.

The considered reconstruction types read

$$rec_1(x,y;b) = \sum_{i=1}^{S}(b_i a_i \cos(\phi_i))(x,y) \tag{3.5}$$

$$rec_2(x,y;b) = \sum_{i=1}^{S}(b_i \cos(\phi_i))(x,y) \tag{3.6}$$

$$rec_3(x,y;b) = \sum_{i=1}^{S}(b_i a_i |\cos(\phi_i)|)(x,y) \tag{3.7}$$

$$rec_4(x,y;b) = \sum_{i=1}^{S}(b_i |\cos(\phi_i)|)(x,y). \tag{3.8}$$

3.4. Modification Function

In this section we study different types of the modification function starting with a trend from special ones to more abstract formulations. The modification function introduces the knowledge into the SSSF, it is the main component of the filter.
We study the band pass filter, the fuzzy band pass filter, the polynomial filter, the general filter and the comprehensive filter. Each filter makes use of the scale function, the three attenuation functions and the four reconstructions.

3.4.1. Band Pass Filter

The filtering via thresholds is the first and most simple filter method as it needs a really small set of knowledge. It can be applied to each of the three attenuations and four reconstructions. However in this first description we only consider the first reconstruction method and the first two attenuation types.
By modified reconstruction as in Equation (3.5) we can filter at least the kind of artifacts which are depicted in Figure 3.1 and marked in 3.2. The modification function for band pass thresholds is similar to Equation (3.2), but needs S high passes $t_{high,i}$ and S low passes $t_{low,i}$ for $i \in \{1,\ldots,S\}$.
For $i \in \{1,\ldots,S\}$ the function reads

$$b_i(x,y) := \begin{cases} 1 & : \mathfrak{att}_i(x,y) \in (t_{low,i}, t_{high,i}] \\ 0 & : \text{otherwise} \end{cases}. \tag{3.9}$$

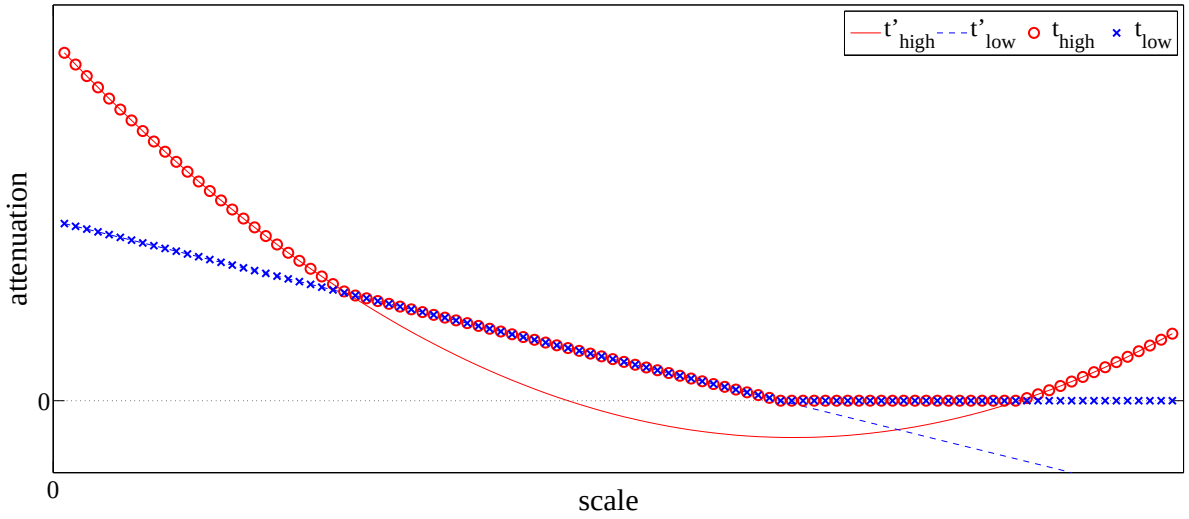


Figure 3.5.: Polynomial band pass. t'_{high} and t'_{low}: High pass and low pass as (3.10). t'_{high} and t'_{low}: High pass and low pass as in (3.12)

By using the first two attenuations we can filter scratches, borders and lettering. Also implants can be filtered out up to a certain accuracy.
For every $i \in \{1, \ldots, S\}$ with S different scale intervals the property b_i depends on t_{low,s_i} and t_{high,s_i}. These $2S$ thresholds define the filter characteristics.

3.4.2. Polynomial Filter

The polynomial filter equals the band pass filter but replaces the properties $t_{low,1}, \ldots, t_{low,S}$ and $t_{high,1} - t_{low,1}, \ldots, t_{high,S} - t_{low,S}$ by two polynomial functions. The degrees of the polynomials are $k_1 < S$ and $k_2 < S$ with

$$t_{low,i} := \sum_{j=0}^{k_1} \check{a}_j \underbrace{\left(\frac{log(s_{c_i}) + log(s_{f_i})}{log(s_{c_n}) + log(s_{f_n})} \right)}_{=x}^{j}$$
$$t_{high,i} - t_{low_i} := \sum_{j=0}^{k_2} \hat{a}_j \left(\frac{log(s_{c_i}) + log(s_{f_i})}{log(s_{c_n}) + log(s_{f_n})} \right)^{j}. \tag{3.10}$$

The reduction can be useful, as the calculated thresholds often have strong coherence and the knowledge description shrinks down to $k_1 + k_2 + 2$ entries[8]. The induced thresholds

[8]Note: This specification delivers a compact range of $x \in [0, 1]$.

must satisfy the constraint

$$\begin{aligned} & 0 \leq \quad t_{low,i} \leq t_{high,i} \\ \Rightarrow \quad & 0 \leq \quad t_{low,i} \\ \wedge \quad & 0 \leq \quad (t_{high,i} - t_{low,i}) \end{aligned} \tag{3.11}$$

because $\mathfrak{att} \in [0, \infty)$.
To satisfy this constraint there are two possible ways: The first one uses a simple non-linear mapping between the results of (3.10) $\hat{t}_{\cdot,i}$ and the requirement of (3.11)

$$t_{\cdot,i} := \max(0, \hat{t}_{\cdot,i}). \tag{3.12}$$

Figure 3.5 illustrates this mapping.
The other approach is included directly in the formula of the polynomials

$$\hat{t}_{\cdot,i} = t_{\cdot,i} \Leftrightarrow \forall x \in [0,1] : \sum_{i=0}^{k_1} a_i x^i \geq 0. \tag{3.13}$$

This can be solved for a small number of dimensions by

$$k = 0 \quad \rightarrow \quad a_0 \geq 0 \tag{3.14}$$

$$k = 1 \quad \rightarrow \quad a_0 \geq 0, a_1 > -a_0 \tag{3.15}$$

$$\begin{aligned} k = 2 \quad \rightarrow \quad & (a_2 > 0, a_0 = 0, a_1 \geq 0) \vee (a_2 > 0, 0 < a_0 < a_2, a_1 > -2\sqrt{a_0 a_2}) \\ \vee \quad & (a_2 > 0, a_0 \geq a_2, a_1 > -(a_0 + a_2)) \\ \vee \quad & (a_2 \leq 0, a_0 \geq 0, a_1 > -(a_0 + a_2)). \end{aligned} \tag{3.16}$$

The implementation of this approach demonstrates, that the method is more of theoretical interest as the optimisation of the polynomials is much harder than the optimisation of the other proposed filters. But we get a really small set of knowledge which can be used to estimate the intrinsic dimension or complexity of the segmentation task.
However, after this step the threshold vector can be described by

$$T_{pol} =< a_{0,1}, \ldots, a_{k_1,1}, a_{0,2}, \ldots, a_{k_2,2} > \tag{3.17}$$

which is only of size $k_1 + k_2 + 2$.

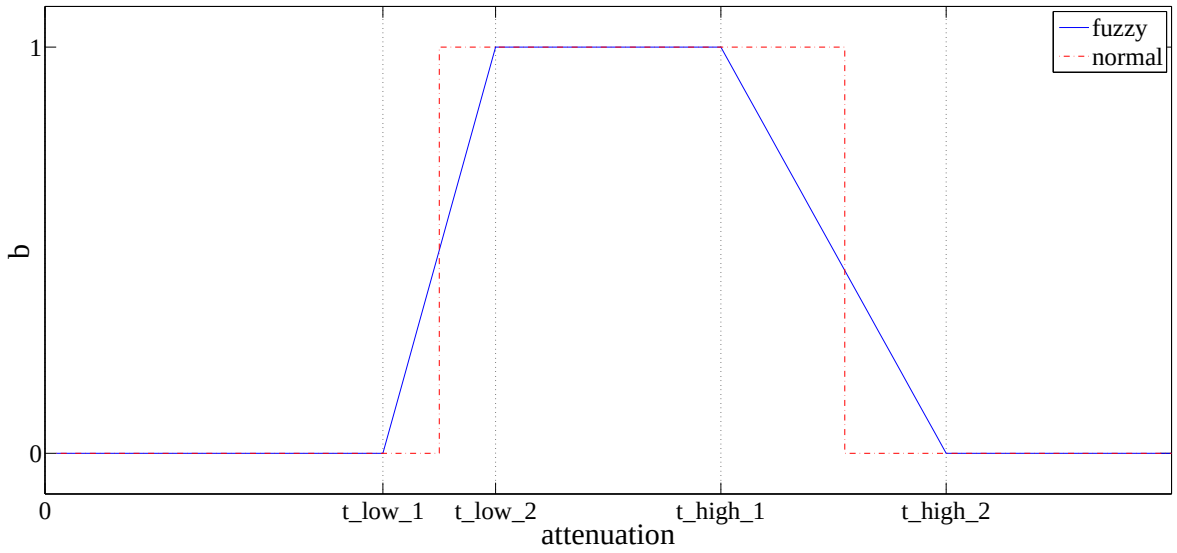


Figure 3.6.: Blue line: fuzzy interval as in (3.18), red dotted line: same as normal interval.

This reduction is independent from S, it only depends on the estimated polynomial[9]. The simplest choice is $k_1 = k_2 = 0$ which means $t_{low,i} = t_{low}$ and $t_{high,i} = t_{high}$. This mapping is similar to (3.1). For a vector of size $k_1 + k_2$ there are $k_1 + k_2 - 1$ different possible combinations.

3.4.3. Fuzzy Band Pass Filter

We return to the band pass filter and expand it by using fuzzy intervals. Fuzzy intervals are intervals with fuzzy boundaries, hence the boundaries are intervals between 0 and 1. The most simple approach reads

$$b_i(x,y) := \begin{cases} \frac{\mathfrak{att}_i(x,y) - t_{low_1,i}}{t_{low_2,i} - t_{low_1,i}} & : \mathfrak{att}_i(x,y) \in (t_{low_1,i}, t_{low_2,i}) \\ 1 & : \mathfrak{att}_i(x,y) \in [t_{low_2,i}, t_{high_1,i}] \\ \frac{t_{high_2,i} - \mathfrak{att}_i(x,y)}{t_{high_2,i} - t_{high_1,i}} & : \mathfrak{att}_i(x,y) \in (t_{high_1,i}, t_{high_2,i}) \\ 0 & : \mathfrak{att}_i(x,y) \in [0, t_{low_1,i}] \cup [t_{high_2,i}, \infty) \end{cases} \tag{3.18}$$

with edges $(t_{low_1,i}, t_{low_2,i})$ and $(t_{high_1,i}, t_{high_2,i})$.
Figure 3.6 depicts the fuzzy band pass filter (as in 3.18) and the normal band pass filter.

[9]Polynomials of size $k_1 = S - 1$ and $k_2 = S - 1$ obviously lead to the same vector length as the unrestricted vector. Each unrestricted vector does not have more intrinsic dimensions, that means there is a mapping between both, in this sense this degree is no more a true restriction.

Since the fuzzy band pass is a superset of the normal band pass we get the normal band pass by setting

$$t_{low_1,i} = t_{low_2,i} = t_{low,i} \tag{3.19}$$

$$t_{high_1,i} = t_{high_2,i} = t_{high,i} \tag{3.20}$$

which satisfies the definition (3.9) for the normal band pass.

3.4.4. General Filter

In the last sections we described filters with minimal parameter sets. The reconstruction have been restricted by only using one band pass per scale interval and $b \in [0, 1]$. There is for example no possible filtering for the dual object[10].
The first improvement of the general filter is the expansion of the range of b to $\mathbb{R}$.
Furthermore the proposed methods only allow to filter connected attenuations at each scale interval, as they use only one band pass. Consequently the second improvement is the introduction of more than one band pass per scale interval.
A general scale space filter is just a discrete mapping from scale intervals s and attenuations to the real numbers:

$$b_s(x, y) : \mathfrak{att}_s(x, y) \to \mathbb{R} \tag{3.21}$$

For every considered pair $(s, \mathfrak{att})$ there is one specified value in $\mathbb{R}$ respectively in $[-1, 1]$ for the mapping.
This filter cannot be described with reasonable effort as the second value $\mathfrak{att}$ has a potentially large number of different values. In other words it has uncountable infinite elements. To receive a discrete mapping, we must approximate the function by segmenting the attenuations in $\mathfrak{I}$ disjunctive intervals, representing equivalence classes for actual attenuations $\{[\mathfrak{att}_1], \ldots, [\mathfrak{att}_\mathfrak{I}]\}$

$$b_s(x, y) : [\mathfrak{att}_j]_s(x, y) \to \mathbb{R}. \tag{3.22}$$

The scale interval s is already discrete. Figure 3.7 shows such a segmentation of the attenuation in 16 equivalence classes.
The practical approximation of the general filter highly depends on $\mathfrak{I}$. If the equivalence

[10] Let f_1 be a filter for object Γ_1 on background Γ_2, such that Γ_1 is highlighted compared to Γ_2 and $b_{f_1}(x, y; s) \geq 0$. The dual object of Γ_1 is Γ_2 and vice versa, hence the filter f_2 for background Γ_2 is that which satisfies $b_{f_2}(x, y; s) = -b_{f_1}(x, y; s) \leq 0$, hence $b_{f_2}(x, y; s) \in [-1, 0] \notin [0, 1]$.

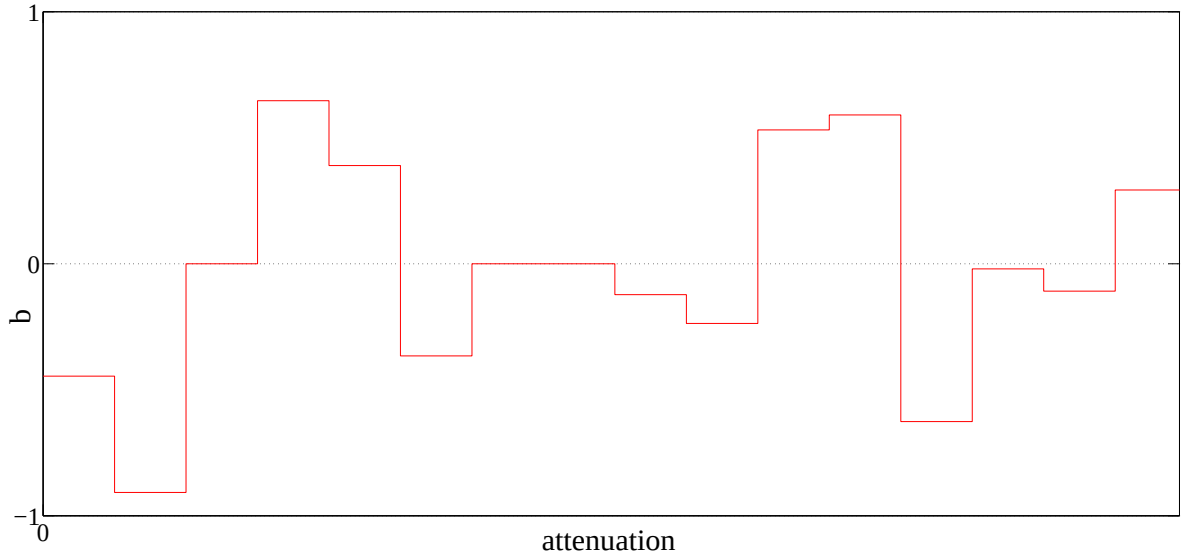


Figure 3.7.: red line: mapping of a general filter having 16 attenuation-equivalence-classes

classes are as small as the resolution of $\mathfrak{att}$, every attenuation value gets its own equivalence class - it is not an approximation anymore.
This new filter is theoretically able to extract or accentuate every object like anatomical regions, patterns and textures. The benefit for this filter is the fast applicability because it yields to a single-point operator in scale space. The disadvantage is the big size of the knowledge and the high costs for training the function.

3.4.5. Relationship Between Band Pass and General Filter

The approaches of the general and band pass filter can be combined by introducing a maximum band pass amplitude m into the filter

$$b_{i,m}(x,y) = b_i(x,y) \cdot m \tag{3.23}$$

with $b_i(x,y)$ as in (3.9).
The value m specifies the mapping for the general filter whereas the boundaries of the band pass t_{low} and t_{high} depict the boundaries of one certain equivalence class.
The general filter with $\mathfrak{I}$ equivalence classes can be substituted by $\mathfrak{I}$ band pass filters with thresholds $\tilde{t}_1, \ldots, \tilde{t}_{\mathfrak{I}+1}$. For band pass filter i it follows $t_{low} = \tilde{t}_i$ and $t_{high} = \tilde{t}_{i+1}$.

3.4.6. The Comprehensive Filter

Each of the considered filters works so far on one specific attenuation type and one reconstruction function. The client has to choose the ones which have the best results on the considered task.
There are twelve different combinations of three attenuations and four reconstruction types. To combine these, there are several possibilities:

- Take the attenuation- and reconstruction-type of the best one - or
- calculate a segmentation based on the 12 different responses - or
- calculate a segmentation based on the 12 different attenuation- and reconstruction-types.

The first choice returns only one combination, whereas the last two approaches combine in fact all options. The difference between the last two approaches is: The former approach makes twelve different separations which are particular as good as possible. The other approach does not reveal the separation character until all mappings are done. Certainly the performance of these approaches gets better from the first to the last one as the second one is a superset of the first and the third is a superset of the second[11].
The second approach reads

$$rec_{comp,2}(x,y;b) = \left(\sum_{r=1}^{4} \sum_{k \in \{a,b,c\}} \mathfrak{a}(r,k) rec_r(x,y;b_{r,k}) \right) - \Delta. \tag{3.24}$$

The first approach can be produced by setting $\Delta = 0$, $\mathfrak{a}(r,k) = 0$ and only $\mathfrak{a}(\hat{r},\hat{k}) = 1$ with $< \hat{r},\hat{k} >$ as the chosen combination between reconstruction type and attenuation type. The third approach reads

$$rec_{comp,3}(x,y;b) = \left(\sum_{r=1}^{4} \sum_{k \in \{a,b,c\}} rec_r(x,y;b_{r,k}) \right) - \Delta. \tag{3.25}$$

[11] The second and the third approach are equivalent except the third approach is less restricted during optimisation.

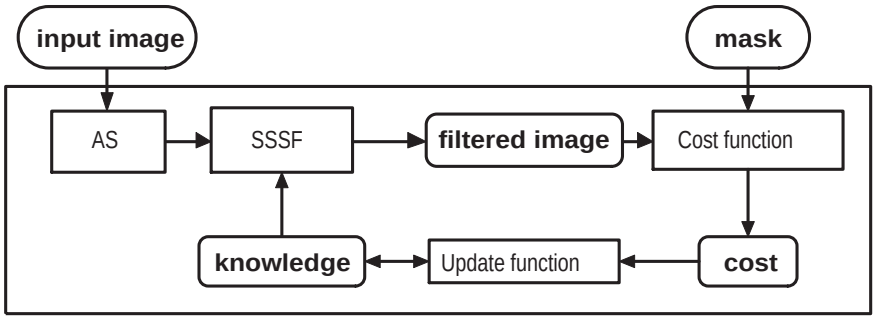


Figure 3.8.: Simplified training loop to optimise knowledge Θ.

This can be transformed into the second approach as the factor $\mathfrak{a}(r,k)$ is contained in $b_{r,k}$

$$\begin{aligned} rec_{comp,3}(x,y;b) &= \left(\sum_{r=1}^{4}\sum_{k\in\{a,b,c\}} rec_r(x,y;b_{r,k})\right) - \Delta \\ &= \left(\sum_{r=1}^{4}\sum_{k\in\{a,b,c\}} \mathfrak{a}(r,k) rec_r(x,y;\underbrace{b_{r,k}\mathfrak{a}(r,k)^{-1}}_{=\hat{b}_{r,k}})\right) - \Delta \\ &= rec_{comp,2}(x,y;\hat{b}). \end{aligned} \tag{3.26}$$

In this thesis we implicitly always use the third approach $rec_{comp,3}$ as it returns better results than the other two approaches.

3.5. Training of the SSSF

After we discussed the different kinds of filters and the principal functionality of this approach we consider the development of a knowledge set Θ. As every image class needs its own knowledge set, we must train it accordingly to the image class before applying it. Therefore we use a recursive optimisation process:

1. Guess solution Θ_i
2. Compute the cost for solution Θ_i
3. Quit or repeat process controlled by cost

This training loop is depicted in 3.8. The process initially starts with an empty knowledge $\Theta_1 = \emptyset$. The filtered image on this initial knowledge becomes $\mathbf{0}$.

The cost function compares the filter results with a target mask containing a colouration in object and non-object. The output of the cost function is the scalar: cost $\in [0, 1]$. Finally the update function changes the knowledge to minimise the cost of the filtered image of the next iteration. Globally the training loop must be steered by the use of a cost-based exit condition.
In the next sections we will evaluate different 1- and n-dimensional cost functions. Afterwards we look at the update functions of the filters.

3.5.1. 1-Dimensional Cost Functions

We start with the derivation of simple cost functions, based on the difference between filtered image and object-mask. Later we will expand it to more dimensions which improves the results of the update function.
We start with the discussion of 'first order' separation-based functions. These functions return a value which depicts the separation of object and non-object in the filtered image.
When searching the knowledge for one specific filter method, we must evaluate the filtered image by comparing it to some target image. This cost function normally returns the difference between filtered and target image, which must be minimal for the best filter. Positive points are object points and negative points are non-object points.
The proper choice of the separation sep between object and non-object obviously decides about the cost, a change of sep influences the ratio between true positive and negative points. Figure 3.9 illustrates this relationship: The index sep-X marks the boundary between negative and positive test points.
The target image consists of zeros for non-object points and ones for object points. The set of positive points is $\{pos\}$, the negative set is $\{neg\}$. In this section $rec(p)$ simply describes the grey value of the reconstructed image at point p, but not the reconstruction function itself. To assure the speed of the evaluation function we set

$$sep = M(M(rec\{pos\}), M(rec\{neg\})) \tag{3.27}$$

with $M(a, b)$ as the mean value of a and b.
For each positive pixel $pos_1, \ldots, pos_{|\{pos\}|}$ and each negative pixel $neg_1, \ldots, neg_{|\{neg\}|}$ the

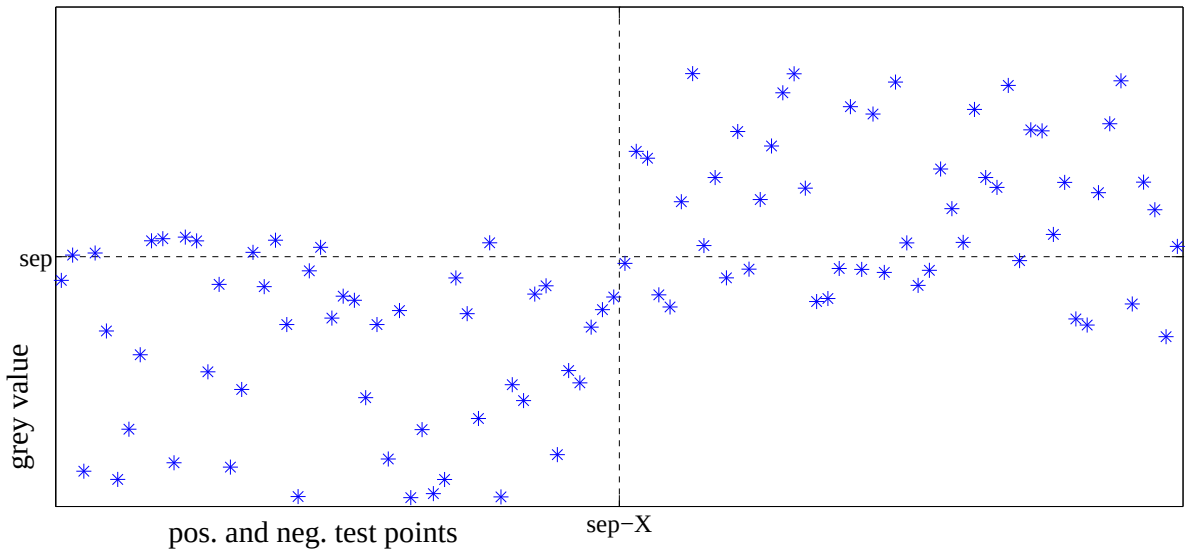


Figure 3.9.: Exemplary filtered signal. Lower left quadrant: true negative, upper right: true positive, upper left: false positive, lower right: false negative.

cost function reads

$$\text{cost}_p(rec, pos_i) = \begin{cases} 0 & : rec(pos_i) > sep \\ 1 & : \text{otherwise} \end{cases} \tag{3.28}$$

$$\text{cost}_n(rec, neg_i) = \begin{cases} 0 & : rec(neg_i) \leq sep \\ 1 & : \text{otherwise} \end{cases} . \tag{3.29}$$

The cost function for the complete set is

$$\text{cost}_P(rec, \{pos\}) = \sum_{i=1}^{|\{pos\}|} \frac{\text{cost}_p(rec, pos_i)}{|\{pos\}|} \hat{=} \text{true positive value} \tag{3.30}$$

$$\text{cost}_N(rec, \{neg\}) = \sum_{i=1}^{|\{neg\}|} \frac{\text{cost}_n(rec, neg_i)}{|\{neg\}|} \hat{=} \text{true negative value.} \tag{3.31}$$

The costs cost_P and cost_N are in the interval $[0, 1]$. The value cost $= 0.5$ depicts no separation, 0 is the best separation for the considered object and 1 is the best separation for the dual of the object[12], but the worst for the object itself. The compound cost is the mapping

$$(\text{cost}_P(rec, \{pos\}), \text{cost}_N(rec, \{neg\})) \rightarrow [0, 1]. \tag{3.32}$$

[12]The dual is the inversion between object and non-object. The dual of the object is the non-object.

We use the cost functions

$$\text{cost}_{\text{mean}}(rec, \{pos\}, \{neg\}) = 0.5(\text{cost}_P(rec, \{pos\}) + \text{cost}_N(rec, \{neg\})) \tag{3.33}$$

which is a special case of

$$\begin{aligned} \text{cost}_{p,\epsilon}(rec, \{pos\}, \{neg\}) &= \text{cost}_P(rec, \{pos\})p + \text{cost}_N(rec, \{neg\})(1-p) \\ &- \epsilon|\text{cost}_P(rec, \{pos\})p - \text{cost}_N(rec, \{neg\})(1-p)| \\ &\text{with } \epsilon, p \in [0,1]. \end{aligned} \tag{3.34}$$

Another simple cost function is given by

$$\text{cost}_{\text{max}}(rec, \{pos\}, \{neg\}) = \max(\text{cost}_P(rec, \{pos\}), \text{cost}_N(rec, \{neg\})). \tag{3.35}$$

The first most simple evaluation mapping is Equation (3.33). This has the drawback to optimise only the easier side, the difference between cost_P and cost_N can become large. Equation (3.34) is an expansion of (3.33) by introducing $\epsilon \in [0,1]$ as a measure of disparity between $\text{cost}_P(\{pos\})$ and $\text{cost}_N(\{neg\})$, and $p \in [0,1]$ to control an one-sided optimisation. This cost function can be useful when aiming for a low false-negative or low false-positive value. A better approach, when aiming for an equivalent optimisation is the cost function cost_{max}. With this function both values grow uniformly as the optimisation only considers the worse value.

3.5.2. Multi-Dimensional Cost Functions

To combine n different cost functions we define the multi-dimensional cost function. An n-dimensional cost function is a mapping from n different 1-dimensional cost functions: $\text{cost}_1, \ldots, \text{cost}_n$ to the scalar cost $\in [0,1]$ with

$$\text{cost} = \frac{\Delta_1}{1+\Delta_1}\left(\frac{\text{cost}_1}{\Delta_1} + \frac{\Delta_2}{1+\Delta_2}\left(\frac{\text{cost}_2}{\Delta_2} + \ldots(\text{cost}_n)\ldots\right)\right). \tag{3.36}$$

The value cost_1 has the highest relevance, whereas cost_n is least significant. The value Δ_i is the smallest distance between two different evaluations of cost_i, this difference is the grid size of cost_i, which is only available if cost_i is a discrete mapping.
To simplify the construction of multi-dimensional cost functions, we will restrict the test points to a subset with $|\{pos\}| = |\{neg\}|$.

Calculation of Δ for the 1-Dimensional Cost Functions

The cost functions introduced in Section 3.5.1 are already discrete:
Let rec_1 and rec_2 be two different reconstructions of the same input image with

$$\underbrace{\text{cost}(rec_1, \{pos\}, \{neg\})}_{=\text{cost}(rec_1)} > \underbrace{\text{cost}(rec_2, \{pos\}, \{neg\})}_{=\text{cost}(rec_2)}.$$

At least one value of $\text{cost}_P(rec_i, \{pos\})$ or $\text{cost}_N(rec_i, \{neg\})$ must improve from the first to the second reconstruction rec_i. Let without loss of generality

$$\underbrace{\text{cost}_P(rec_1, \{pos\})}_{=\text{cost}_P(rec_1)} > \underbrace{\text{cost}_P(rec_2, \{pos\})}_{=\text{cost}_P(rec_2)}$$

and

$$\begin{aligned} \text{cost}_p(rec_1) - \text{cost}_p(rec_2) &= \sum_{i=1}^{|\{pos\}|} \frac{\text{cost}_p(rec_1, pos_i)}{|\{pos\}|} - \frac{\text{cost}_p(rec_2, pos_i)}{|\{pos\}|} \\ &\geq \frac{1}{|\{pos\}|} = \Delta_{pos}. \end{aligned} \tag{3.37}$$

With $N = |\{neg\}| = |\{pos\}|$ and

$$\text{cost}_N(rec_i) := \text{cost}(rec_i, \{neg\}) \tag{3.38}$$

it follows

$$\text{cost}_N(rec_1) > \text{cost}_N(rec_2) \Rightarrow \Delta_{neg} = \frac{1}{|\{neg\}|} = \frac{1}{N} = \Delta_{pos}. \tag{3.39}$$

The smallest difference Δ for a cost function $\text{cost}(\cdot) \in \{\text{cost}_P(\cdot), \text{cost}_N(\cdot)\}$ is given by

$$\begin{aligned} \Delta_{\text{cost}} &= \min\{\Upsilon_{\text{cost}} | \Upsilon_{\text{cost}} > 0 \wedge \Upsilon_{\text{cost}} = \text{cost}(rec1) - \text{cost}(rec_2)\} \\ \Rightarrow \quad & [\text{cost}(rec_1) > \text{cost}(rec_2) \Rightarrow \underbrace{n}_{\in \mathbb{N}} \Delta_{\text{cost}} = \text{cost}(rec_1) - \text{cost}(rec_2)]. \end{aligned} \tag{3.40}$$

Now it is possible to compute Δ for $\text{cost}_{\text{mean}}$ and $\text{cost}_{\max}$. The equation for $\text{cost}_{\max}$ reads

$$\begin{aligned} \Upsilon_{\text{cost}_{\max}} &= \max(\text{cost}_P(rec_1), \text{cost}_N(rec_1)) - \max(\text{cost}_P(rec_2), \text{cost}_N(rec_2)) \\ &= \sum_{i=1}^{N} \frac{\text{cost}_{\{P \vee N\}}(rec_1, \cdot)}{N} - \frac{\text{cost}_{\{P \vee N\}}(rec_2, \cdot)}{N} \geq \frac{1}{N} \end{aligned} \tag{3.41}$$

$$\Delta_{\text{cost}_{\max}} = \min(\Upsilon_{\text{cost}_{\max}}) = \frac{1}{N}. \tag{3.42}$$

Let $\epsilon = 0$, hence $\text{cost}_{p,\epsilon}$ can be simplified by reducing it to the first term of (3.34). $\Delta_{\text{cost}_{p,0}}$ reads

$$\begin{aligned}
\Upsilon_{\text{cost}_{p,0}} &= (\text{cost}_P(rec_1)p + \text{cost}_N(rec_1)(1-p)) \\
&\quad -(\text{cost}_P(rec_2)p + \text{cost}_N(rec_2)(1-p)) \\
&= p[\text{cost}_P(rec_1) - \text{cost}_P(rec_2)] + (1-p)[\text{cost}_N(rec_1) - \text{cost}_N(rec_2)] \\
&= \frac{p\ j}{N} + \frac{(1-p)\ k}{N} = \frac{p(j-k)+k}{N}, \text{ with } j,k \in \mathbb{Z} \\
\Delta_{\text{cost}_{p,0}} &= \min(\Upsilon_{\text{cost}_{p,0}}) = \begin{cases} \frac{\gcd(m,n)}{nN} & : \frac{m}{n} = p \quad \in \mathbb{Q} \\ \emptyset & : p \qquad\quad \notin \mathbb{Q} \end{cases}
\end{aligned} \tag{3.43}$$

with $\gcd(m,n)$ as greatest common divisor of m and n. Obviously the discretisation of $\text{cost}_{p,0}$ is only possible for $p \in \mathbb{Q}$.
Let $p = 0.5$, it follows

$$\Delta_{\text{cost}_{0.5,0}} = \frac{1}{2N} = \Delta_{\text{cost}_{\text{mean}}}. \tag{3.44}$$

As we solved Δ for $\text{cost}_{\max}$ and $\text{cost}_{\text{mean}}$, we can combine it with the 1-dimensional second order cost functions.

Second Order Cost Functions

We will specify five different second order cost functions which can be used to accelerate the optimisation process inside the update function. The first two cost functions are based on the band pass design of the knowledge. Let $|m|$ be the modulus of the maximum amplitude of the band pass interval, $w = t_{high} - t_{low} \geq 0$, $|m_{\max}|$ a maximum value of $|m|$ and $w_{\max}$ a maximum value of w. The first two cost functions read

$$\text{cost}^{\Theta}_{\min} = \frac{|w\ m|}{w_{\max} m_{\max}} \tag{3.45}$$

$$\text{cost}^{\Theta}_{\max} = 1 - \frac{|w\ m|}{w_{\max} m_{\max}}. \tag{3.46}$$

The function $\text{cost}^{\Theta}_{\min}$ returns lower costs if the surface area of the band pass contracts, whereas $\text{cost}^{\Theta}_{\max}$ delivers lower costs if the surface area grows.
The other three cost function are based on the filtered image similar to the 1-dimensional cost functions in Section 3.5.1. These functions are based on the absolute distance

between object and non-object. With $d = |M(\{pos\}) - M(\{neg\})|$ the functions read

$$\text{cost}^d_{\text{min}} = |\frac{1}{\exp(d)} - 1| \tag{3.47}$$

$$\text{cost}^d_{\text{max}} = \frac{1}{\exp(d)} \tag{3.48}$$

$$\text{cost}^d_{\text{opt}} = |\frac{1}{\exp(|d-2|)} - 1|. \tag{3.49}$$

The function $\text{cost}^d_{\text{min}}$ returns lower costs for $d \to 0$, $\text{cost}^d_{\text{max}}$ optimises $d \to \infty$ and $\text{cost}^d_{\text{opt}}$ returns best results for $d = 2$.

As the cost functions are not discrete, we have to discretise them, since we want to combine them with the other lower cost functions.

Let

$$\text{cost}_f \in \{\text{cost}^\Theta_{\text{min}}, \text{cost}^\Theta_{\text{max}}, \text{cost}^d_{\text{min}}, \text{cost}^d_{\text{max}}, \text{cost}^d_{\text{opt}}\}. \tag{3.50}$$

The discretisation of cost_f reads

$$\frac{1}{\Delta} = n \in \mathbb{N} \tag{3.51}$$

$$\widehat{\text{cost}_f} = \Delta \lfloor \frac{\text{cost}_f}{\Delta} \rfloor. \tag{3.52}$$

The value $\widehat{\text{cost}_f}$ is the discrete equivalent of cost_f. With $f = \widehat{\text{cost}_f}$, cost_f as in (3.50) and $g \in \{\text{cost}_{\text{mean}}, \text{cost}_{\text{max}}\}$ the resulting 2-dimensional cost function is

$$\text{cost}_{f,g} = \frac{\Delta_g}{1 + \Delta_g}(\frac{\text{cost}_g}{\Delta_g} + \text{cost}_f). \tag{3.53}$$

3.5.3. Update Function: Adaption of the Knowledge

The update function is responsible for finding the optimal adaption of the knowledge Θ. In our case:

$$\underset{\Theta}{\operatorname{argmin}} \operatorname{cost}(rec_\Theta, \{pos\}, \{neg\}) \tag{3.54}$$

with rec_Θ as the application of knowledge Θ on the image. The optimisation itself is done by the algorithm described below.

Nelder-Mead Method

The Nelder-Mead method[13] needs a start vector v_0 and tries to calculate the global optimum vector. If the vector range is of dimension d, the Nelder-Mead method reads:

- Take d linear independent seeds $v_1, \ldots, v_d$; Rename it according to the costs $\operatorname{cost}(v_i)$ such that v_0 has the maximum cost; begin loop:
  - Calculate centre of gravity $\hat{v}$ for all points except v_0.
  - Set $\alpha = \beta = 1$; begin loop:
    - $v' = v_0 + \beta\alpha(\hat{v} - v_0)$
    - if $\operatorname{cost}(v') < \operatorname{cost}(v_d)$ then increase α
    - elseif $\operatorname{cost}(v') > \operatorname{cost}(v_0)$ then decrease α, $\beta = -1$
    - else decrease α
    - if $\operatorname{cost}(v')$ does not change anymore, quit loop.
  - replace v_0 with v' and rename vectors according to their costs.
  - if $\operatorname{cost}(v') = \operatorname{cost}(v_0)$, quit loop.
- return v_d

The steps in the inner loop are called the reflection for $\alpha = \beta = 1$, expansion for $\alpha > 1$ and reduction for $0 < \alpha < 1$. The method is published in [27]. Nelder-Mead is very general, as it belongs to the class of derivative free and nonlinear optimisation methods. As we do not know anything about the structure of our input signal[14] we take this general optimisation method. In our case v is a vector-like subset of knowledge Θ, to calculate $\operatorname{cost}(v)$ we apply the knowledge concerning the vector Θ_v to generate an output image I^{Θ_v}, hence $\operatorname{cost}(v) = \operatorname{cost}(I^{\Theta_v})$.

[13] also known as Downhill simplex method or Amoeba method

[14] The only restriction for the input signal f reads $f \in L_2$.

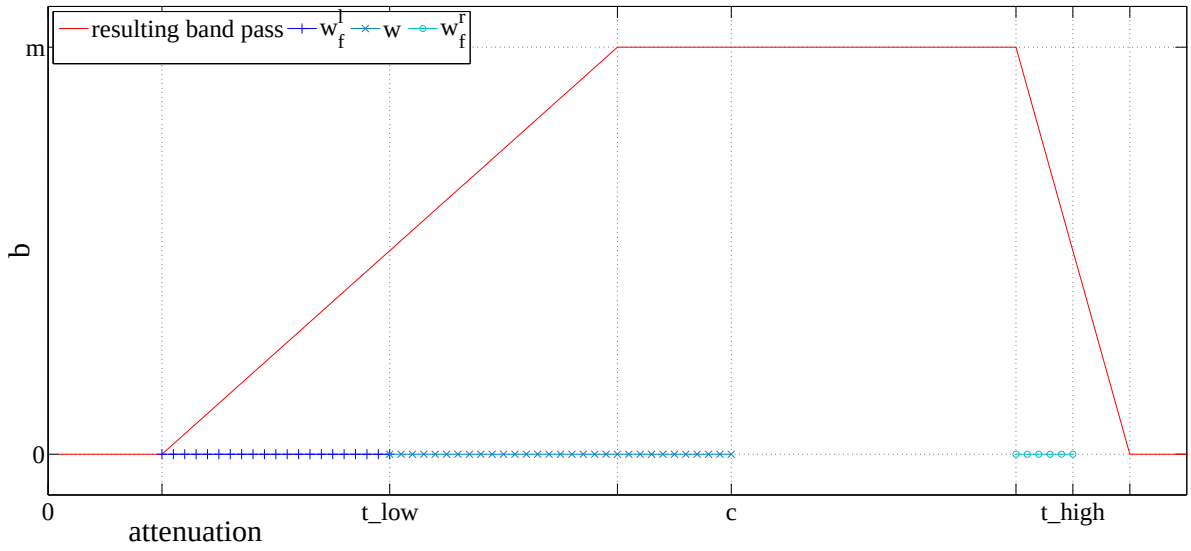


Figure 3.10.: Interpretation of one fuzzy band pass filter: $< m, c, w, w_f^l, w_f^r >$.

Description of the Knowledge

The design of the knowledge set must satisfy the restrictions of

- a small parameter set Θ_v to optimise simultaneously - and
- a fast evaluation of one choice of the parameter set Θ_v.

The smallest entity of knowledge for a given scale interval s is in general the following vector:

$$v =< c, w, w_f^l, w_f^r, m > \tag{3.55}$$

with c and $w \geq 0$ as centre and width of one band pass, $w_f^l \geq 0$ and $w_f^r \geq 0$ as size of left and right fuzzy edge and m as the maximum value of the band pass. This set (see Figure 3.10) has the same dimension as the parameter set in Figure 3.6 but offers better optimisation by the Nelder-Mead method. For each reconstruction r, attenuation type a and scale interval s we need at least one knowledge entity $v_{r,a,s}$. In other words, the number of parameters we get for one knowledge set is given by

$$|\Theta_{\min}| = |r| \times |a| \times |s| \times |v|. \tag{3.56}$$

In our case it follows $|\Theta_{\min}| = 4 \times 3 \times 16 \times 5 = 960$. Additionally we combine n band passes per scale interval, reconstruction type and attenuation type. So the upper boundary of

the size of the knowledge set reads $|\Theta_{\max}| = n|\Theta_{\min}|$. Obviously $\Theta \in \mathbb{R}^{|\Theta_{\max}|}$ is too high dimensional for a simultaneous optimisation which leads to an optimal solution.
We bypass the large optimisation range by optimising only one knowledge entity v concurrently. Each knowledge entity is separable.
Let

$$\text{replace}(v, m) =< v(1), v(2), v(3), v(4), m > \tag{3.57}$$

be the function which replaces the fifth entry of v with m and let Θ_p be the knowledge entity at position

$$p =< r, a, s, i > \tag{3.58}$$

with r as a reconstruction type, a an attenuation type, s a scale interval and i the i-th band pass filter. Let furthermore

$$\Omega(\Theta, p, m) = \begin{cases} \Theta_{pos} & : pos \neq p \\ \Theta_{\text{replace}(\Theta_p, m)} & : pos = p \end{cases} \tag{3.59}$$

be the replacement of the knowledge entity at position p and let

$$\Psi(\Theta, p, m) = \begin{cases} \mathbf{0} & : pos \neq p \\ \Theta_{\text{replace}(\Theta_p, m)} & : pos = p \end{cases} \tag{3.60}$$

be the replacement of the knowledge entity at position p and the suppression of the other knowledge entities.
Let $rec(\Theta)$ be the reconstructed image induced by knowledge Θ, and let $\Theta \equiv \Omega(\Theta, p, m)$. Hence separability means

$$\begin{aligned} rec(\Theta) &= rec(\Omega(\Theta, p, 0)) + rec(\Psi(\Theta, p, m)) & (3.61) \\ rec(\Omega(\Theta, p, 0)) &= rec(\Theta) - rec(\Psi(\Theta, p, m)) \\ &= rec(\Theta) + rec(\Psi(\Theta, p, -m)). & (3.62) \end{aligned}$$

Due to this characteristic we can reduce the simultaneous optimisation to only one entity $v \in \mathbb{R}^5$. This equation accelerates the evaluation for one guessed knowledge set, as only the values of one reconstruction have to be considered.
In one step, the optimisation method guesses a vector $\hat{v} \in \mathbb{R}^5$. Since a knowledge entity v is restricted to $(R, R^{\geq 0}, R^{\geq 0}, R^{\geq 0}, R)$ we need a mapping between $\hat{v}$ and v.

Let $\hat{v}$ be the guessed vector

$$\hat{v} =< c, \hat{w}, \hat{w}_f^l, \hat{w}_f^r, m > . \tag{3.63}$$

The mapping reads

$$v =< c, |\hat{w}|, \min(|\hat{w}_f^l|, |\hat{w}|), \min(|\hat{w}_f^r|, |\hat{w}|), m > . \tag{3.64}$$

This general knowledge description enables the configuration of every proposed filter.

Band Pass Filter

The band pass filter has no fuzzy edges and a constant value m. It follows that only the first two parameters of the knowledge have to be trained:

$$k_1 =< c, w, 0, 0, 1 > \tag{3.65}$$

Hence we need a 2-dimensional optimisation function as in Listing 3.1.

```
1  width := 0.5;
2  for i:=1 to S do begin
3       c      :=(max(att)+min(att))/2;
4       w      :=(max(att)-min(att)) * width;
5       [c,w]  := Opt1([c,w],...);
6  end;
```

Listing 3.1: Optimisation scheme for the band pass filter

Fuzzy Band Pass Filter

The fuzzy band pass filter must also train the fuzzy edges

$$k_2 =< c, w, w_f^l, w_f^r, 1 > . \tag{3.66}$$

This can be done by a 4-dimensional minimisation function as in Listing 3.2, but also by two different 2-dimensional ones, which are depicted in 3.3.

General Filter

The function for the general filter is much more complex than those for the band pass filter. We need much more values to optimise: When having $\mathfrak{I}$ equivalence classes, we

```
1  width := 0.5;
2  fuzzyWidth := 0;
3  for i:=1 to S do begin
4      c      :=(max(att)+min(att))/2;
5      w      :=(max(att)-min(att))*width;
6      [c,c,w_f_l,w_f_r] := Opt2([c,w,fuzzyWidth,fuzzyWidth],...);
7  end;
```

Listing 3.2: First optimisation scheme for the fuzzy band pass filter

```
1  width := 0.5;
2  fuzzyWidth := 0;
3  for i:=1 to S do begin
4      c      :=(max(att)+min(att))/2;
5      w      :=(max(att)-min(att)) * width;
6      [c,w] := Opt1([c,w],...);
7      [w_f_l,w_f_r] := Opt3([fuzzyWidth,fuzzyWidth],c,w,...);
8  end;
```

Listing 3.3: Second optimisation scheme for the fuzzy band pass filter

get at least $\mathfrak{I}|\Theta_{min}|$ knowledge entries which is 15360 for $\Theta_{min} = 960$ and $\mathfrak{I} = 16$. Fortunately we must only find one parameter per knowledge entity, as the centre c_{const} and width w_{const} are given by the equivalence class

$$k_3 =< c_{const}, w_{const}, 0, 0, m > . \tag{3.67}$$

The optimisation is one-dimensional. It follows a programme as in Listing 3.4 To accelerate the application and to get more robust results, we can create an iterative process starting with only one equivalence class. This scheme is illustrated in Figure 3.11. At each iteration, the size of one equivalence class shrinks, whereas its number grows. The initialisation $m = 0$ is skipped starting with the second iteration and replaced by the value of the last iteration.

Advanced Band Pass Filter

With the combination of the fuzzy band pass filter and the general filter, we achieve the best results. The knowledge entities read

$$k_4 = < c, w, 0, 0, m > \text{ and} \tag{3.68}$$

$$k_5 = < c, w, w_f^l, w_f^r, m > . \tag{3.69}$$

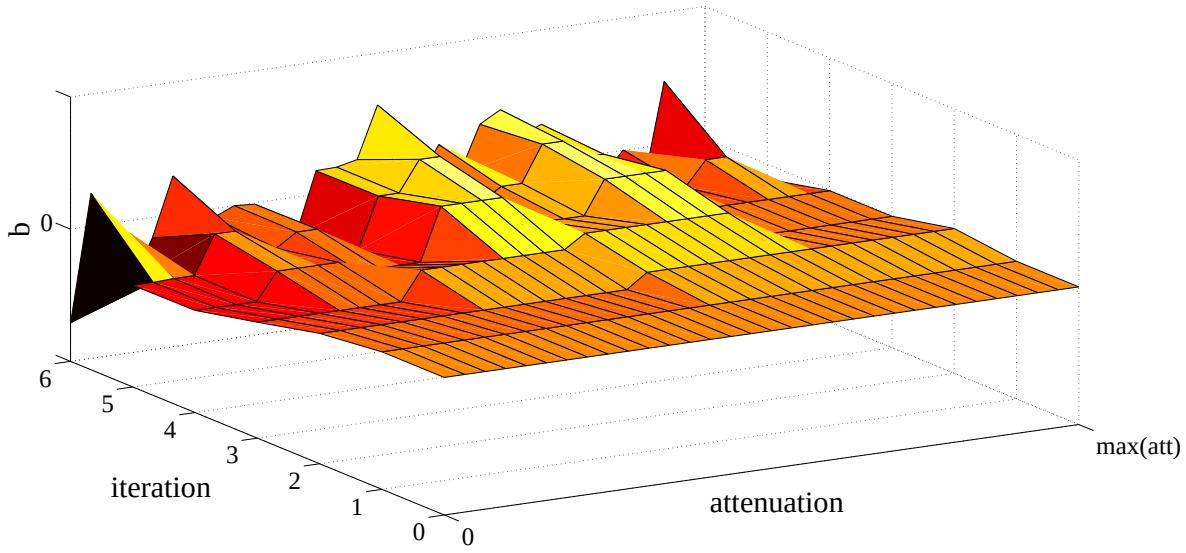


Figure 3.11.: Iteration cycle to get optimal knowledge for b

```
1       m := 0;
2       w:= (max(att)-min(att))/(2*I);
3       for i=0 to I-1
4           c  := min(att) + w + I*2*w;
5           m  := Rec4([m],c,w,...);
6       end;
7   end;
```

Listing 3.4: Optimisation scheme for the general filter

The search space becomes three-, respectively five-dimensional. The superposition of n advanced band pass filters is superior to any other proposed filter - even for n being a small number.

Polynomial Filter

The polynomial filter represents an exception, as the knowledge set differs from the proposed one. We are searching the vector with size $|v| = k_1+k_2+2$ for one reconstruction type r and attenuation type a

$$v = T_{pol} =< a_{0,1}, \ldots, a_{k_1,1}, a_{0,2}, \ldots, a_{k_2,2} > . \tag{3.70}$$

For the second degree polynomials it follows $|\Theta| = 4 \times 3 \times 6 = 72$. For one reconstruction type and attenuation type, the functionality is listed in 3.5. At first we optimise the high

```
1  % start values for low pass polynomial
2  a1[0,...,k_1] := 0;
3  % start values for high pass polynomial
4  a2[0,...,k_2] := 0;
5  for i=0:k_2
6      % optimise (i+1)-th value in polynomial a2
7      a2 := Opt(a2,i+1,...);
8  end;
9  for i=0:k_1
10     % optimise (i+1)-th value in polynomial a1
11     a1 := Opt(a1,i+1,...);
12 end;
```

Listing 3.5: Optimisation scheme for the polynomial filter

passes and then the low passes. In each optimisation process, starting with a constant polynomial value, we let the degree grow from 0 to k_1 or k_2 respectively. In the next loop we use the estimated vector part to obtain new start values. Let $\tilde{a}_i$ be the factors of the new polynomial $\tilde{f}$ and a_i the factors of the last polynomial f. The mean value of f must equal the mean value of $\tilde{f}$ for a set of given intervals

$$\int_{x_1}^{x_2} \sum_{i=0}^{k-1} a_i x^i dx = \int_{x_1}^{x_2} \sum_{i=0}^{k} \tilde{a}_i x^i dx. \tag{3.71}$$

The integral boundaries induce a partition of $[0,1]$ into k intervals with simplest partitioning $\{[0,\frac{1}{k}],\ldots,[\frac{k-1}{k},1]\}$. There are k constraints for searching $k+1$ free parameters, in other words, there is only one free parameter left. Another equivalent choice is $\{[0,\frac{1}{k}],[0,\frac{2}{k}],\ldots,[0,1]\}$, which leads to simpler constraints. For $l \in \{1,\ldots,k\}$ the constraints read

$$\sum_{i=0}^{k-1} \left(\frac{l}{k}\right)^{i+1} \frac{a_i}{i+1} = \int_0^{l/k} \sum_{i=0}^{k-1} a_i x^i dx = \int_0^{l/k} \sum_{i=0}^{k} \tilde{a}_i x^i dx = \sum_{i=0}^{k} \left(\frac{l}{k}\right)^{i+1} \frac{\tilde{a}_i}{i+1}. \tag{3.72}$$

The steps are the following: Set $k=0$, find a_0 using the optimisation method. Get seeds for $k=1$ using Equation 3.72

$$a_0 = \int_0^1 a_0 \, dx = \int_0^1 \tilde{a}_0 + \tilde{a}_1 x \, dx = \tilde{a}_0 + \tilde{a}_1/2 \tag{3.73}$$

$$\Longrightarrow \tilde{a}_1 = 2(a_0 - \tilde{a}_0). \tag{3.74}$$

Use the optimisation method with new seeds to get optimal parameters a_0 and a_1, continue with the same procedure by incrementing k.
For $k = 2$, we get

$$\tilde{a}_1 = 6(a_0 - \tilde{a}_0) + a_1 \tag{3.75}$$
$$\tilde{a}_2 = -6(a_0 - \tilde{a}_0). \tag{3.76}$$

In the same manner for $k = 3$ we get

$$\tilde{a}_1 = 11(a_0 - \tilde{a}_0) + a_1 \tag{3.77}$$
$$\tilde{a}_2 = -27(a_0 - \tilde{a}_0) + a_2 \tag{3.78}$$
$$\tilde{a}_3 = 18(a_0 - \tilde{a}_0). \tag{3.79}$$

Let $\breve{a} = a_0 - \tilde{a}_0$. We get the seeds

$$k = 0 \; : \; a =< a_0 > \tag{3.80}$$
$$k = 1 \; : \; a =< a_0, 2\breve{a} > \tag{3.81}$$
$$k = 2 \; : \; a =< a_0, 6\breve{a} + a_1, -6\breve{a} > \tag{3.82}$$
$$k = 3 \; : \; a =< a_0, 11\breve{a} + a_1, -27\breve{a} + a_2, 18\breve{a} > . \tag{3.83}$$

The results obtained by this approach are very unstable and unpredictable. To improve the behaviour we may vary the start values for the lower and higher polynomials and select the best results. However, this method cannot gain the other proposed ones in speed and accuracy[15].

[15] Although this approach leads to short knowledge descriptions, there is a significant drawback. The application of the filter cannot be accelerated and the training of this filter up to same quality as the band pass filter will take much longer.

Limitations of the Update Function

When optimising the band passes separately, a band pass change is only accepted if it decreases the cost. Hence being at position p (defined as in Equation 3.58), the reconstruction including the current knowledge entity $rec(\Theta)$ and the reconstruction without the knowledge entity $rec(\Omega(\Theta, p, 0))$ feature the relationship

$$\text{cost}(rec(\Theta)) = \text{cost}(rec(\Omega(\Theta, p, 0))) - \Delta,\ \Delta > 0 \tag{3.84}$$

since a band pass is only accepted if it produces a cost decrease.
A good separation is indicated by a cost value near '0', whereas no separation is indicated by a value of '0.5'. After the training of the knowledge via for instance the band pass method, we get n knowledge vectors for specified scale intervals, attenuation- and reconstruction types.
For testing which values are most significant and which can be suppressed we consider all n values for $\Delta : \Delta_1, \ldots, \Delta_n$.
The sum of the single cost decreases differs from the global cost decrease

$$\sum_n \Delta_i \neq \text{cost}(rec) - 0.5. \tag{3.85}$$

The sum of the individual cost decreases is only about 25% of the complete cost decrease. In other words, the complete cost decrease is influenced by cross correlations between two or more band passes to about 75%.
It follows that the optimum knowledge set of band pass filters must be trained in parallel. Since two band pass filter have a cost decrease in combination, one of them must not uncover a cost decrease.
The dimension of one band pass filter is within the bounds of possibility to be calculated. In our case an optimisation of all band passes in one step is too large as it produces a search space of $2 \times 3 \times 4 \times 16 = 384$ dimensions[16].
One the one hand, the presented approach has the benefit to be programmable and being relatively fast, but on the other hand it has the drawback not being absolutely correct. A change of knowledge entity v_p requires the re-adaption of potentially all other knowledge entities. Conversely the adaption of the other knowledge entities require a re-adaption of v_p. With a growing set of already optimised knowledge entities, the value

[16] The dimensions are calculated by 2 parameters per band pass, 3 different attenuation types, 4 reconstruction types and 16 different scale intervals.

of each entity converges against its optimum value. It follows that also the presented approach yields to similar results as the optimum high dimensional parallel approach. In other words, for growing iteration cycles in the optimisation process, the estimated knowledge converges against the optimal one.

3.5.4. Postprocessing of the Knowledge

As already mentioned in Section 3.2, we normalise Θ to fulfill the constraints which are depicted in Figure 3.4.
Let $d = \frac{M(\{pos\})-M(\{neg\})}{2}$ be half of the difference between the mean value of the object and the background points. Let $\Theta_{a,r,s,i}$ be the i-th knowledge entry for $\mathfrak{att} = a$, reconstruction type r and scale interval s. The normalised knowledge $\hat{\Theta}$ reads

$$\hat{\Theta}_{a,r,s,i} = < \Theta_{a,r,s,i}(1), \ldots, \Theta_{a,r,s,i}(4), \frac{\Theta_{a,r,s,i}(5)}{d} > \tag{3.86}$$

$$\Delta = \frac{M(\{pos\})}{d} - 1. \tag{3.87}$$

This normalisation allows the application of the final segmentation with a constant threshold t. Let I^{out} be the normalised filtered image. The segmentation Ξ for a threshold t is a binary image with

$$\Xi(x,y) = \begin{cases} 1 & : I^{out}(x,y) > t \\ -1 & : otherwise \end{cases} . \tag{3.88}$$

The absolute values of t satisfy the constancy relationship where $t = 0$ always yields to a segmentation with same false positive and false negative ratio, $t = 1$ depicts the centre of gravity concerning the object points and $t = -1$ the centre of gravity of the background points. For $t \neq 0$, the relative ration between false positive and false negative values depends on the variances of the two curves in Figure 3.4.

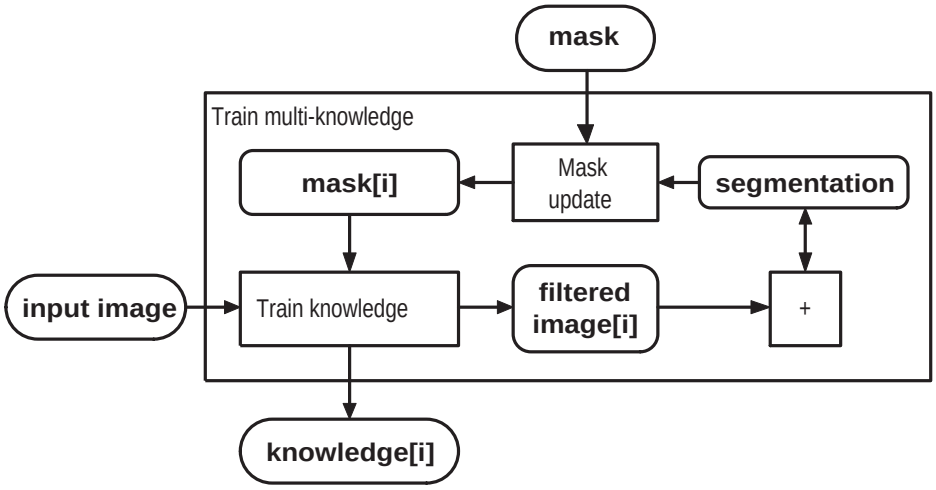


Figure 3.12.: Training scheme of the multi-filter.

3.6. The Multi-Filter

The results of one SSSF is a colouration into object pixels represented by '1', and the background pixels represented by '-1'. The training mask induces a specific error which perturbs the colouration.

In contrast to the simple SSSF , the scale space multi-segmentation filter uses $n > 1$ different knowledge sets $\Theta_1, \ldots, \Theta_n$. In the first step we develop Θ_1 for the initial mask, which already results in a segmentation Ξ_1. In step i we design a new mask based on the segmentation result of the sum of all knowledge sets $\Xi^{\Sigma} = \sum_{j=1}^{i} \Xi_i$ and the initial mask to train only those points which are badly separated. This iteration is executed n times. The n knowledge matrices are able to filter more information than only one knowledge set. In particular, it obtains the filtering of difficult image regions like implants and lettering. The advantages are better segmentations, whereas the optimisation process takes n times longer. This process is depicted in Figure 3.12. The sample implementation is shown in Listing 3.6.

The values of `maskSum` depict the colouration *col* of the relationship between positive and negative filtering with $col = |pos| - |neg|$, $|pos|$ the number of positive segmentations and $|neg|$ the number of negative segmentations. The threshold `t` depicts the maximal number of true segmentations at a single position which are integrated into the training set of the new mask. In this example we take $t = 2$. Pixel p is removed from the current mask if there are 2 more adequate knowledge sets than inadequate knowledge sets for p.

```
1  function MultiKnowledge(amplitude,phase,mask):array of knowledge
2  % initialise variables
3  N := 20;
4  t := 2;
5  % calculate first knowledge
6  knowledge[1] := TrainKnowledge(amplitude,phase,maskOld);
7  maskSum := ((SSSF(amplitude,phase,knowledge[1])>0)-0.5)*2;
8  % make N-1 iterations
9  for (i:=2; i<=N; i++)
10     % compute new mask containing 1 for new object points, 0 for new
           background points and NaN for points to ignore
11     maskNew := NaN(zeros(size(mask)));
12     maskNew((maskSum<t)&(mask==1)) := 1;
13     maskNew((maskSum>-t)&(mask==0)):= 0;
14     % compute knowledge i
15     knowledge[i]:=TrainKnowledge(amplitude,phase,maskNew);
16     % Calculate separation:
17     maskSum:=maskSum+(((SSSF(amplitude,phase,knowledge[i])>0)-0.5)
          *2);
18 end;
19 return knowledge;
```

Listing 3.6: Computing the multi-knowledge

In general the first five iterations do not improve the segmentation, but the cost always shrinks down with growing iterations. An absolute saturation, in the sense that n_{max} iterations lead to the best results, is not observable[17], but the lower boundary of costs is 0. Hence

$$\forall \delta > 0 : \exists n : cost(rec(\Omega_n)) < \delta. \tag{3.89}$$

In other words, this filter can always return the optimum segmentation for finite images if execution time does not matter.

3.6.1. The Discrete and the Linear Multi-Filter

The result of the training function is not a single knowledge set Θ but an array $[\Theta_1, \ldots, \Theta_n]$. These n knowledge sets induce n different filtered images $I_1^{out}, \ldots, I_n^{out}$, which again induce the n segmentation results $\Xi_1, \ldots, \Xi_n \in \{-1, 1\}$.

[17] There is in fact a saturation which is the minimal number of iterations we need to create a satisfying filter with $cost = 0$. For a training mask with $N = |\{pos\}| + |\{neg\}|$ pixels the upper boundary is given by N. This saturation is however not useful, as the information received from this specific training image is at most a subset of the information induced by the texture. This 'illusional' saturation can easily be generated by setting $n \ll N$.

To summarise these signals, there is a discrete approach $\hat{\Xi}^{\Sigma}_{dis}$ and a linear approach $\hat{\Xi}^{\Sigma}_{lin}$ with

$$\hat{\Xi}^{\Sigma}_{dis} = \sum_{i=1}^{n} \Xi_i \tag{3.90}$$

$$\hat{\Xi}^{\Sigma}_{lin} = \frac{1}{N} \sum_{i=1}^{n} I_i^{out}. \tag{3.91}$$

To receive the final segmentation Ξ^{Σ}, we map $\hat{\Xi}^{\Sigma}$ in $\{-1, 1\}$ by

$$\Xi^{\Sigma}(x,y) = \begin{cases} 1 & : \hat{\Xi}^{\Sigma}(x,y) > 0 \\ -1 & : otherwise \end{cases}. \tag{3.92}$$

The Equation (3.91) yields to a linear mapping, the filtered images are simply summarised, we can also obtain $\hat{\Xi}^{\Sigma}_{lin}$ by summarising the knowledge entities.
Let $b(\Theta_i)$ be the reconstruction factor matrix for Θ_i and $rec(x, y; b(\Theta_i))$ the reconstruction for Θ_i. $\hat{\Xi}^{\Sigma}_{lin}$ satisfies the relationship

$$\begin{aligned}
\hat{\Xi}^{\Sigma}_{lin} &= \frac{1}{N} \sum_{i=1}^{N} rec(x, y, b(\Theta_i)) \\
&= \frac{1}{N} \left(\sum_{i=1}^{N} \left[\sum_{r=1}^{4} \sum_{k \in \{a,b,c\}} rec_r(x, y; b_{r,k}(\Theta_i)) \right] - \Delta_i \right) \\
&= \frac{1}{N} \left(\sum_{i=1}^{N} \sum_{r=1}^{4} \sum_{k \in \{a,b,c\}} rec_r(x, y; b_{r,k}(\Theta_i)) \right) - \underbrace{\frac{1}{N} \sum_{i=1}^{N} \Delta_i}_{\Delta} \\
&= \left(\sum_{r=1}^{4} \sum_{k \in \{a,b,c\}} rec_r(x, y; \underbrace{\Sigma_{i=1}^{N} b_{r,k}(\frac{\Theta_i}{N})}_{b_{r,k}(\Theta^{\Sigma})}) \right) - \Delta \\
&= rec(x, y, b(\Theta^{\Sigma}))
\end{aligned} \tag{3.93}$$

with $\frac{\Theta_i}{N}$ as the division of the maximal band pass amplitude of each knowledge entry in Θ_i by divisor N - just as in Equation (3.86).
It follows that the expansion to linearity leads to the same formula as the normal reconstruction, which only needs one offset and up to N overlapping intervals. In other

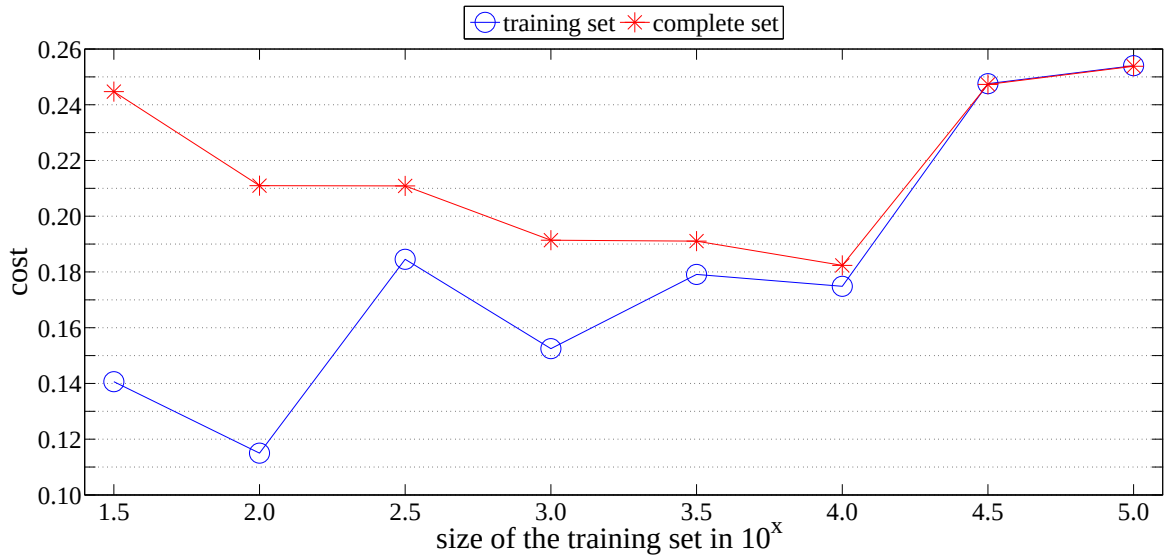


Figure 3.13.: Mapping between size of the training set and its costs.

words, we get an elegant implementation of the general filter[18].
Both approaches have different advantages. The discrete filter (3.90) is better trained since it is the sum of N potentially different filters. The linear filter (3.91) has similar characteristics up to a certain degree, whereas being intrinsically only one filter. It follows that the linear filter is faster to apply in contrast to the discrete one.

3.7. Application of the SSSF

In the last section we described the theory of the SSSF . The final parametrisation of the filter can only be found by different test runs on a real segmentation task. Therefore we will develop the filter properties on a spine segmentation task. The input images are of the same class as the image in Section 3.1, but contain the complete spine.
In this subsection we will consider the training set size and the optimal choice for the scale space. Later we will evaluate the proposed filters and the proposed cost functions to be able to define optimal presets for a given task.

3.7.1. Evaluation of Training Set Sizes

When training the filters, the size of the training set impacts the results and the evaluation time of the optimisation procedure. Due to the performance improvement, the training set is only an equidistant subset of all possible training points with the same size for object and background points. The task is to find an optimum size. With growing training set sizes, the optimisation on the training set becomes harder and its cost grows, whereas the application of the estimated knowledge becomes better. An optimal training set size is the one which barely satisfies the quality induced by the difference between the costs for the training set and the costs for the complete image.
Figure 3.13 shows a mapping between the size of the training set, tested for sizes $10^{1.5}$, 10^2, ..., 10^5 and taken from three different images of this classification. The number of test points of these 3 images is approximately $10^{5.7}$. The costs for the training set increases and the costs for the complete set decreases. The distance between the costs of the training set and the complete set reaches its minimum at 10^5. The minimum costs for the complete set is reached between $10^{3.5}$ and 10^4. Above this point, the costs for the complete set increases which can be explained as it is restricted by the cost on the training set. It follows that a larger training set size does not lead necessarily to better band pass filters. Sometimes it also generates drawbacks for the optimisation function. The optimum size depends on the motive of the image and also on the mask. In our case, a good choice is a value between $10^{3.5}$ and 10^4. In Figure 3.13 we used the cost function $\text{cost}_{\max}$. The used mask and the highest maximum scale, which will be discussed below, is certainly also responsible for the costs, but only for the absolute value and not for the relative development of the training set sizes.

3.7.2. Evaluation of the Maximum Scale

As already mentioned, another important property is the highest maximal scale s_{max}. If the maximal scale is too large - much larger as the object size - the costs increase, as the significant data is too small in relation to the complete data. In the training images and the test images, we calculated the mean size of the texture in pixels d. In this example, d is the width of the spine. The relative scale s_{rel} is a factor to calculate the highest maximum scale

$$s_{max} = d \cdot s_{rel}. \tag{3.94}$$

[18] It is not possible to apply this mapping to linearity for every example. The loss of segmentation quality from $\hat{\Xi}^{\Sigma}_{dis}$ to $\hat{\Xi}^{\Sigma}_{lin}$ is often too large.

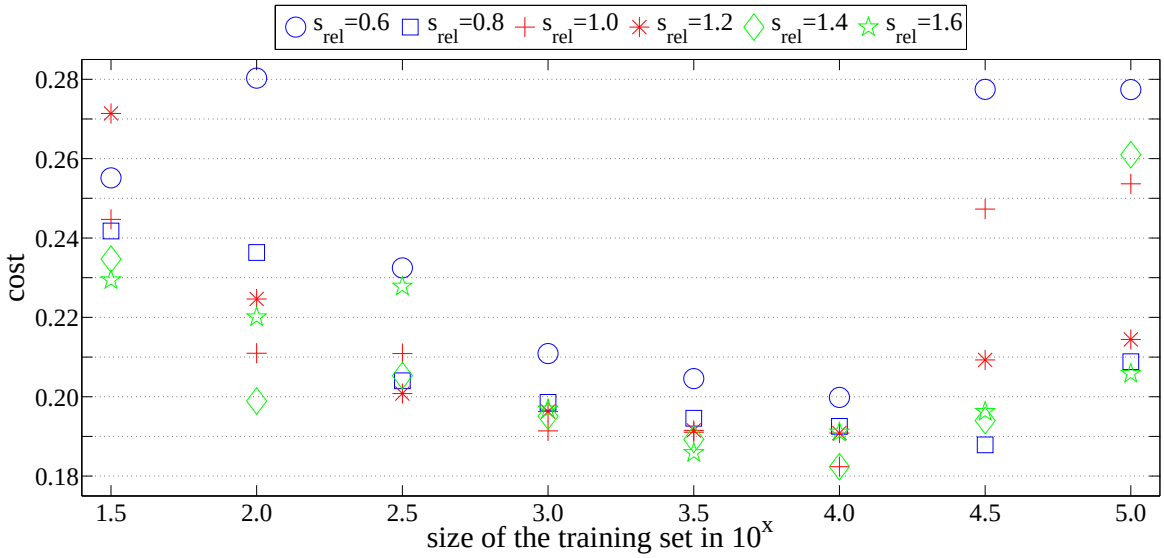


Figure 3.14.: Mapping as in Figure 3.13, but restricted to the costs for the complete set.

We tested 6 different values for s_{rel} from 0.6 to 1.6. The results are displayed in Figure 3.14. The lowest costs are reached between $10^{3.5}$ and 10^4. The curves are very comparable in quality, concerning the relative scale the lowest costs are reached at $s_{rel} = 1$ and $s_{rel} = 1.4$, whereas also the relative scales collapse more for too high training set sizes than the others. The values for $s_{rel} = 0.6$ have commonly too high costs, whereas the values for $s_{rel} = 1.4$ and $s_{rel} = 1.6$ seem to be robust, as they have only a small variance between 10^3 and $10^{4.5}$ and have also quite good costs.

3.7.3. Optimum Masks

The third property for a good segmentation is the choice of the mask. We want to separate the objects and the background, in our case, we want to separate the spine from the background. Therefore we use a mask with spine pixels represented by '1' and background pixels represented by '0' (see Figure 3.15, plot 1). It follows that the estimated knowledge accentuates the spine against the background. This could be sufficient, if the costs were close to '0'. However, this simple mask leads to costs far from '0', but still below '0.5'.
When using different knowledge types with different masks, one can also promote the quality of the segmentation. The second mask in Figure 3.15 which is evaluated is '1' at these positions, where the first mask is '1' too. It is '0' only around the spine. This leads to a better separation on the left and right side of the spine. In this way we design a set

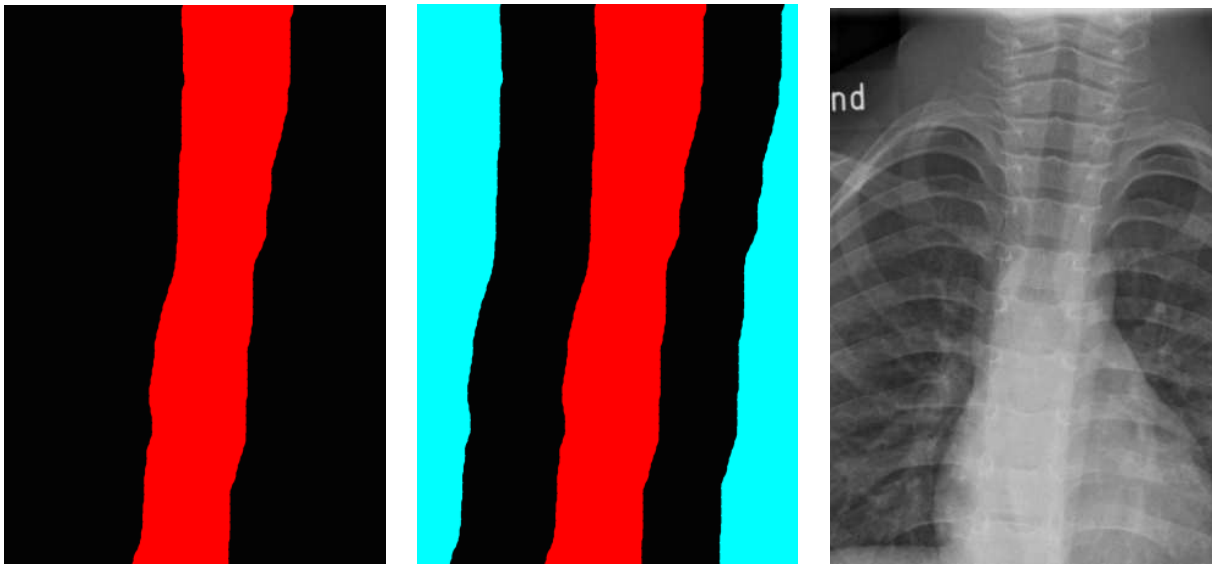


Figure 3.15.: 2 different masks with red=1, black=0, turquois=NaN: a) mask for finding the spine b) mask for getting a better segmentation at spine borders c) base image

of knowledge filters which are combined together, or which are combined with regional information.

Different masks are combined to train the segmentation especially for problematic regions. The first mask in Figure 3.15 is used to locate the spine centre as the position of the horizontal mean value on one line. The second mask in Figure 3.15 isolates the exact borders of the spine. The combination of these two masks with an one-row region-growing algorithm solves the spine segmentation problem. We start in the spine centre, induced by the first mask, and we stop at the borders, retrieved from the second mask[19]. The filters do not use any topological or regional information. Hence this filter leads to better segmentations when combining it with this regional information, as e.g. the spine is connected and it is similar to a watershed from top to bottom.

3.7.4. Evaluation of the Filter Types

For the next test runs we take the photograph and training masks from Figure 3.16. For the evaluation of the different filter types we take the 2-dimensional cost function $\text{cost}_{f,g}$ with $f = \text{cost}_{\text{mean}}$ and $g = \text{cost}^{d}_{\text{opt}}$ (see Equation 3.53).

[19] The general segmentation problem for one connected object in the middle of the image also follows this region growing approach by using a mask for the object and n others for the borders by accentuating the border itself more and more.

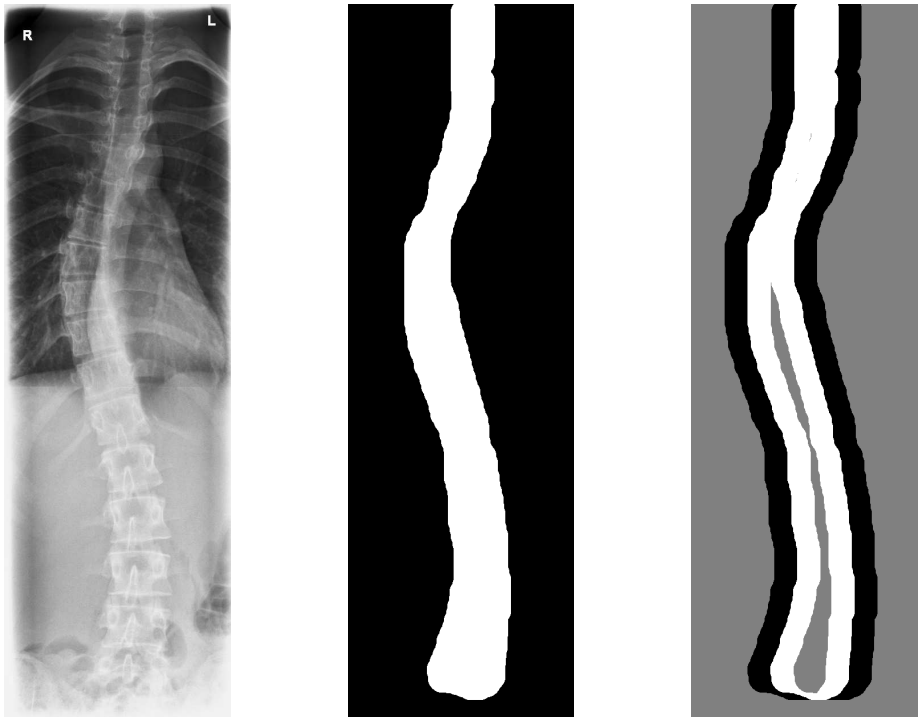


Figure 3.16.: Test image for the next examples, mask to train the spine allocation and mask to train the allocation of the spine borders.

Parametrically a filter train scheme contains three different variables:

1. The adaption of centre and width of a band pass.
2. The adaption of the left and right fuzzy edges.
3. The adaption of the maximum value for the band pass amplitude.

There are seven meaningful types of combination, depicted by the 3-dimensional binary vector: $[001], [010], \ldots, [111]$.
Each adaption starts with a maximum value $m = \{-1, 1\}$, centre $c = 0.5$ and width $w = 0.35$ which are normalised to the interval, and fuzzy edges $w_f^l = w_f^r = 0$. For types $([0 \cdot \cdot])$ ten different equivalence classes are chosen and the abort criterion for the main loop is weakened by factor 100.
The results are given in Figure 3.17. The number of applied improvements varies from 390 to above 10^4. The general filter-based approaches $([0 \cdot \cdot])$ need too many improvements, hence too much optimisation time. The fuzzy edge adaption based optimisations $([11\cdot])$ generate lowest costs, the other approaches $([10\cdot])$ are worse, but need shorter optimisation time.

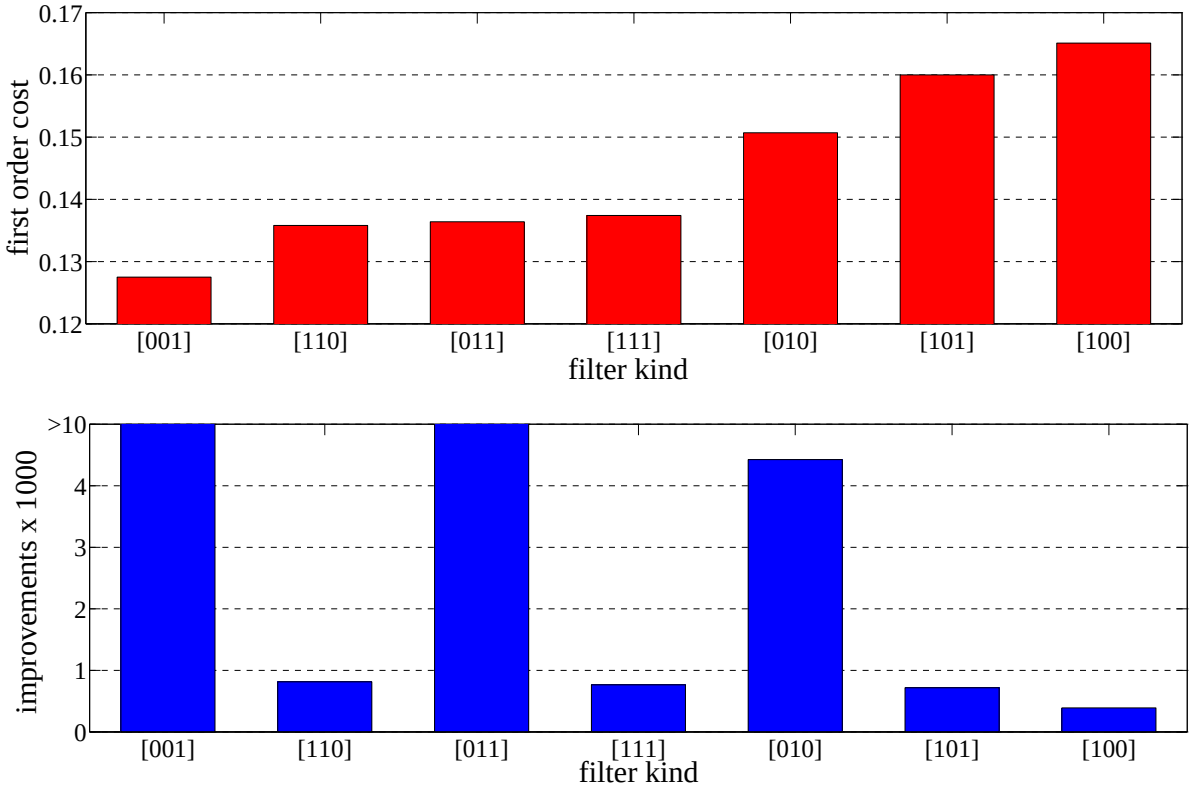


Figure 3.17.: Evaluation of the cost function. First row: first order costs. Second row: Improvements until abortion.

3.7.5. Evaluation of the Cost Functions

The evaluation of the first order cost function is done by the usage of the 2-dimensional cost function $\text{cost}_{f,g}$, with $g = \text{cost}^d_{\text{opt}}$, $f \in \{\text{cost}_{\text{mean}}, \text{cost}_{\text{max}}\}$ and filter type ([101]) (see Figure 3.19).
The cost function $\text{cost}_{\text{mean}}$ is superior in time and quality.
In 3.5.2 we proposed 5 different second order cost functions. For the evaluation purpose, these cost functions are combined with $\text{cost}_{\text{mean}}$ as the first order cost function and filter type ([111]). The results are depicted in Figure 3.18.
The first order costs for the first three functions are quite similar, whereas the third function $\text{cost}^{\Theta}_{\text{max}}$ takes the shortest optimisation time. We can drop the last three functions, as they are worse concerning costs. A significant feature for the choice of the cost function can also be the number of improvements. Concerning this characteristic the knowledge-based evaluations $(\max, t)$ and $(\min, t)$ are superior even to the empty cost function $(-)$.

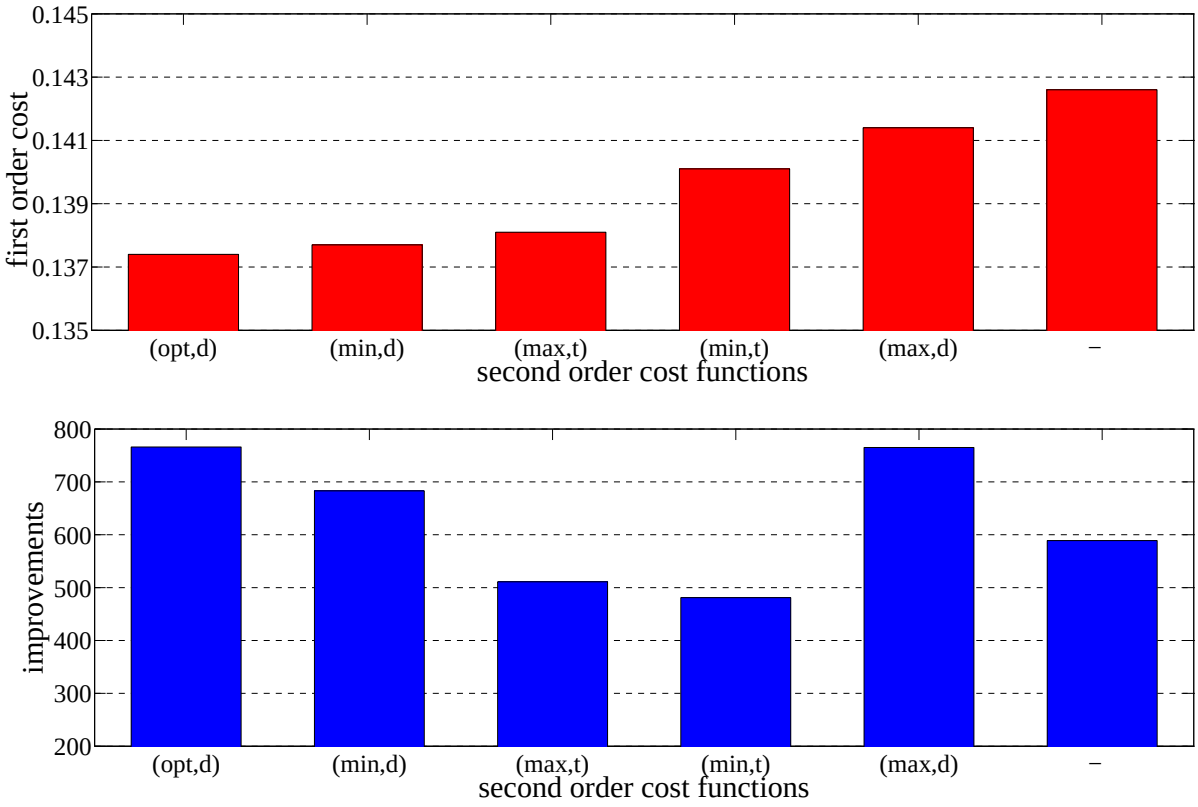


Figure 3.18.: Evaluation of the second order cost function including the empty second order cost function (-). First order costs and improvements until abortion.

rank	kind	cost	improvements
1	mean	0.1600	718
2	max	0.1792	1534

Figure 3.19.: Cost functions $\text{cost}_{\text{mean}}$ and cost_{max}

3.7.6. Final Presets

The final implemented presets are tested with 30,000 positive and 30,000 negative test points on the input image and the first mask of Figure 3.16. The execution time is growing with descending the costs. The second plot in 3.20 depicts the ratio between its execution time and the execution time of the preset 'dirty'[20].
Only preset 'best' leads to the optimum which is close to the value of 'optimal'. The others are automatically interrupted inside the search loop, because the costs-improvement got too close to zero.

[20] The execution time for the preset 'dirty' is 23 seconds on this machine. A Compaq Presario A900 Notebook PC: A 32-Bit Intel Core 2 Duo CPU T5450 with 2 × 1.66 GHz and 2GB RAM. The `Matlab` version is 7.6.0.324 (MATLAB R2008a).

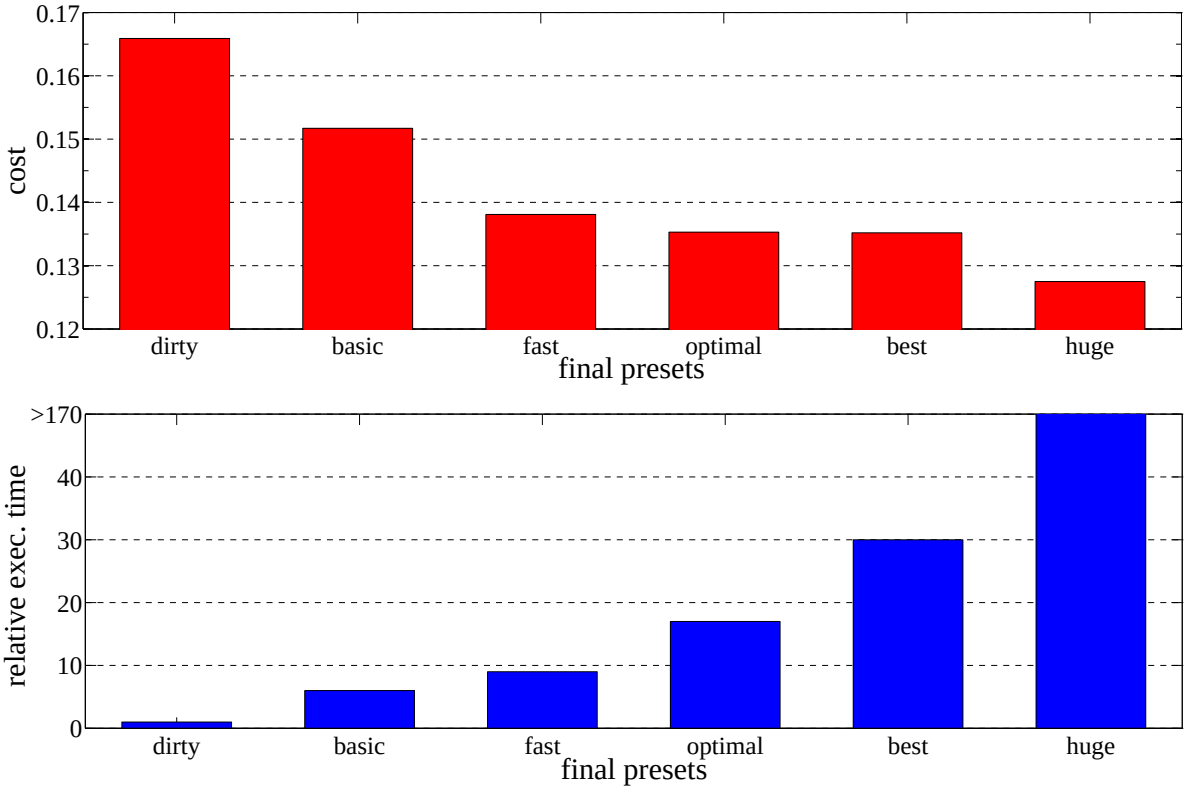


Figure 3.20.: Evaluation of the final filter presets. First order costs and execution time relative to preset 'dirty'.

3.7.7. Evaluation of the Multi-Filter

Figure 3.21 shows a segmentation on the first test image. We applied both masks, the spine detection and the spine-border detection mask. The segmentations of both filters show the following: Red or dark blue represent positive or negative responses of both filters, light blue denotes a positive response only of the spine-border filter and yellow is the colour only for a positive response of the spine filter. The first plot is the segmentation after the first iteration, the second after ten iterations and the third after 100 iterations.

The used presets are 'dirty' for the spine segmentation and 'basic' for the spine-border segmentation. The train data contains only the information of this image, therefore the absolute results become very good. If we only consider the relative improvement of the 100 iterations in 3.22 we are superior to the best standard approach with the 10th iteration (see also second image in 3.21), and reach a cost value of 0.0357 for the spine detection with the worst optimisation preset 'dirty' and 0.0229 for the spine-

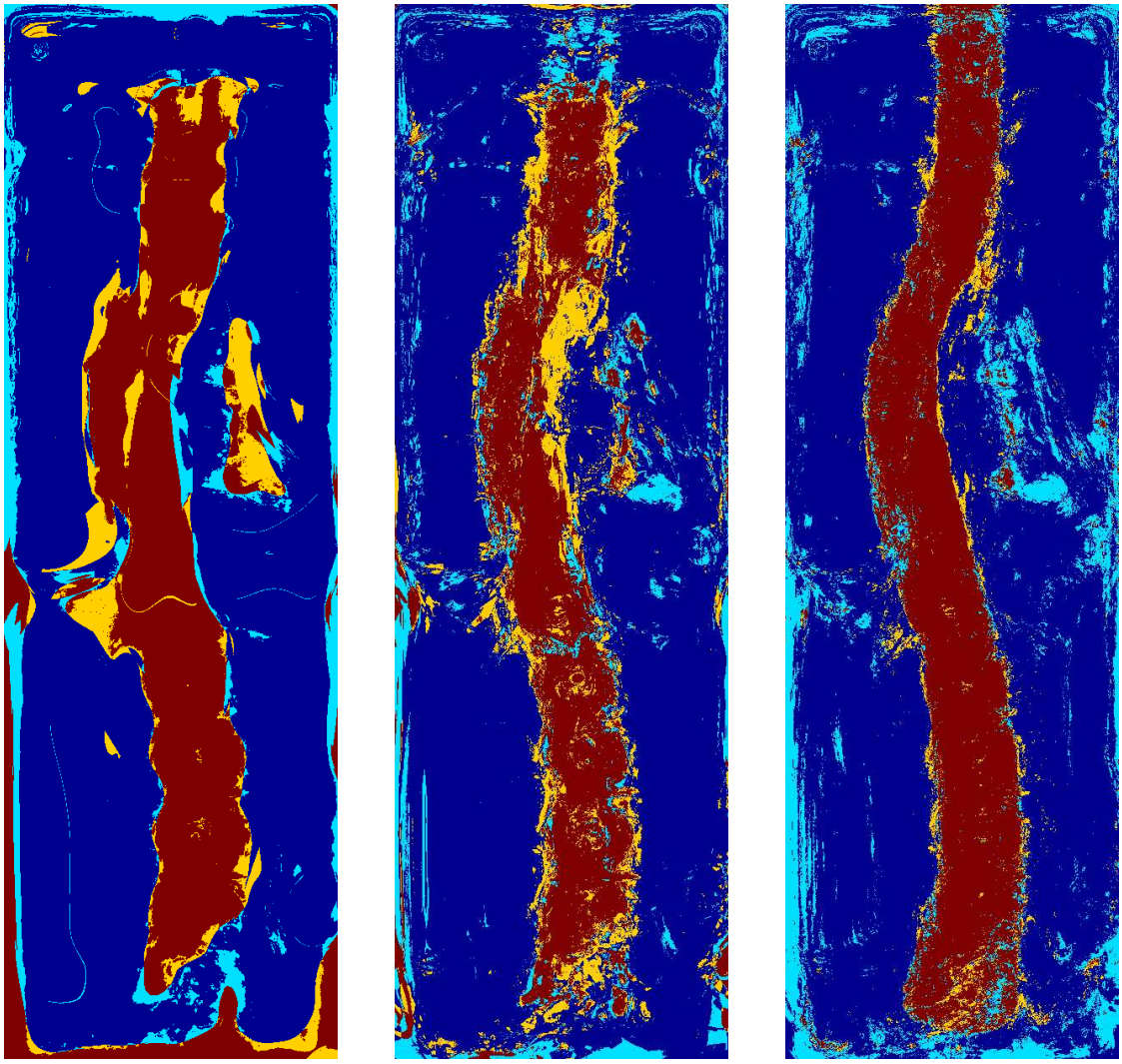


Figure 3.21.: Improvement of the multi-filter: example for 2 multi-knowledge filters induced by mask 1 and mask 2.

border detection with the second worst preset 'basic'. This segmentation is depicted in the last image of Figure 3.21. Because of the fast, but in particular highly costing optimisation types (see also first image in 3.21), the process needs 10 iterations to gain the segmentation results of the best optimisation type 'best'. But the improvement is still obvious: The execution time for the multi-filter with n iterations is restricted by n times the execution time of the used preset. If we use the fastest preset we get better results as any other single application already in the same execution time as preset 'fast'.

3.8. Results

The complete code for the implementation of the training and application for the filter is given in the Appendix A.2.
In this section we will give examples for different stand-alone SSSF -applications. These filter results are still a pre-stage to a final segmentation filter, as it is not combined with other image processing techniques which include topographical or local information[21].

3.8.1. Application on Spine Photographs

For the demonstration on spine photographs, we use different training set sizes.

1. One training image
2. Three training images
3. 14 training images

Each training set is trained for the normal spine detection and the spine-border detection. The knowledge adaption of the training images is shrinking down with the growing training set size, whereas the applicability is growing. This development is obvious as more training images contain more variances of the texture representation.
The cost graphs in Figure 3.22 have similar developments. The adaption for the spine-border detection becomes worse than the adaption for the spine detection. In the first iterations the costs come with a high oscillation. The length of oscillation is induced by the training set size. At the first plot the oscillation lasts for the first 10 iterations, in the second, it lasts for the first 20 iterations and in the third it takes 50 iterations. Also the amplitude of this oscillation is growing with growing training set size.
The final spine segmentation results meet in most cases the expectations, since the filter results are of good quality.

[21] The design of a final filter, i.e. for spine segmentation or liver volume estimation can be developed on these filter results by the use of opening and closing operators, combined with 1- or 2-dimensional region-growing or energy-minimising approaches, respectively.

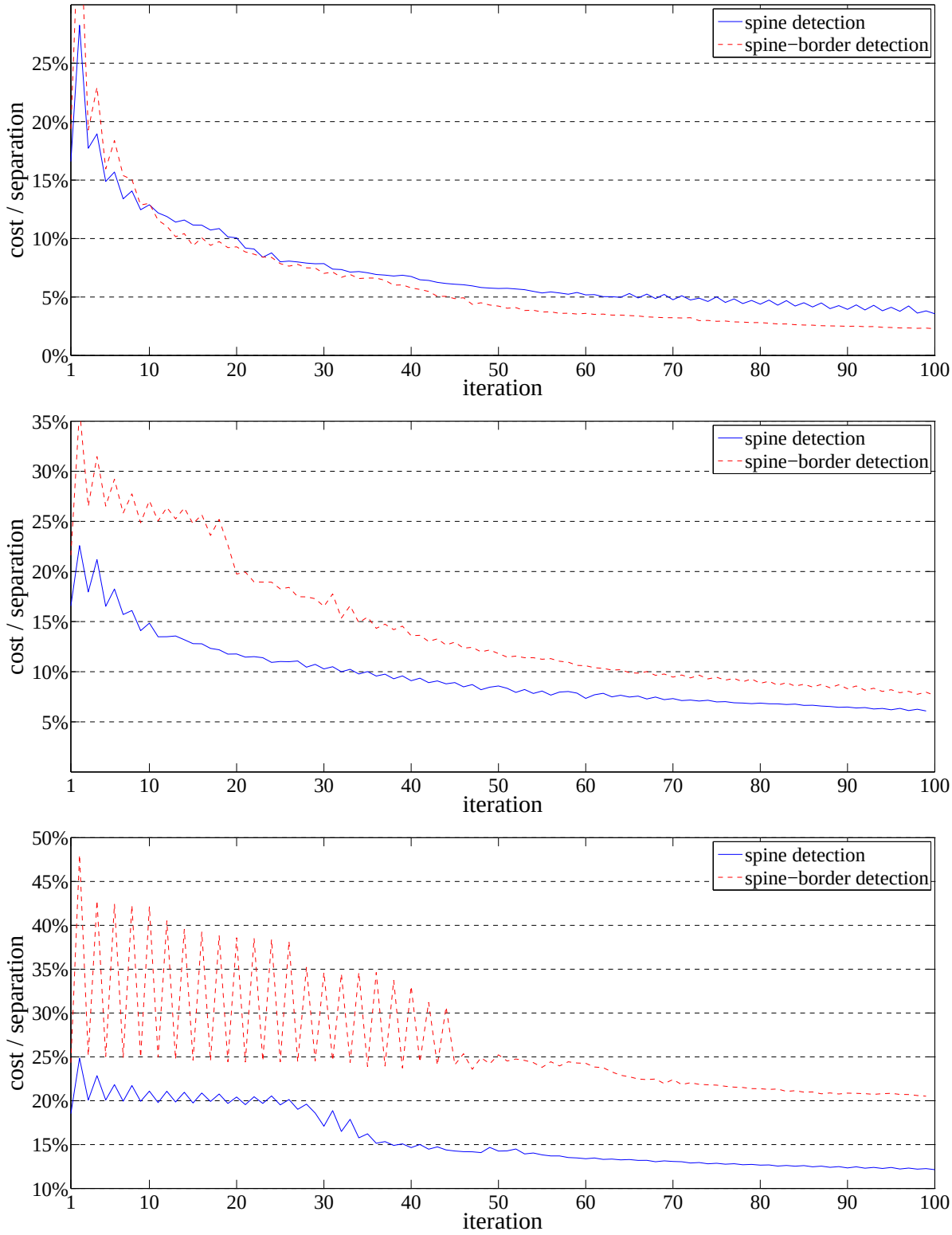


Figure 3.22.: The graphs of the costs for training improvements and different training set sizes. First row: costs for 1 training image, second row: costs for 3 training images, third row: costs for 14 training images.

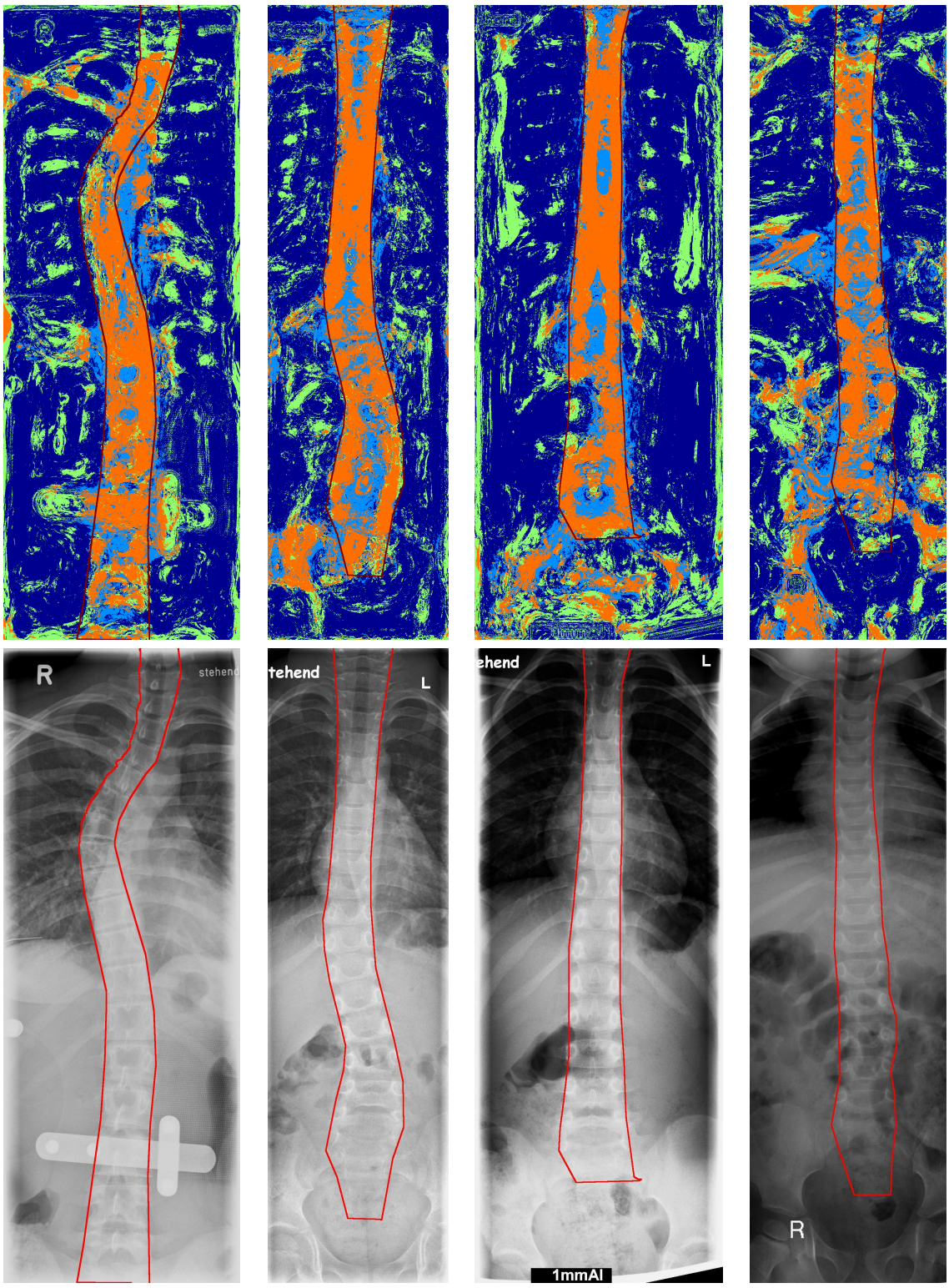


Figure 3.23.: Filter application on 4 of 14 training images. Upper row: Results of the two masks. Dark blue or orange ≙ positive or negative responses of the both filters. Green ≙ only positive response of the spine border detection, light blue ≙ only positive response of the spine detection. Red line ≙ training-mask.

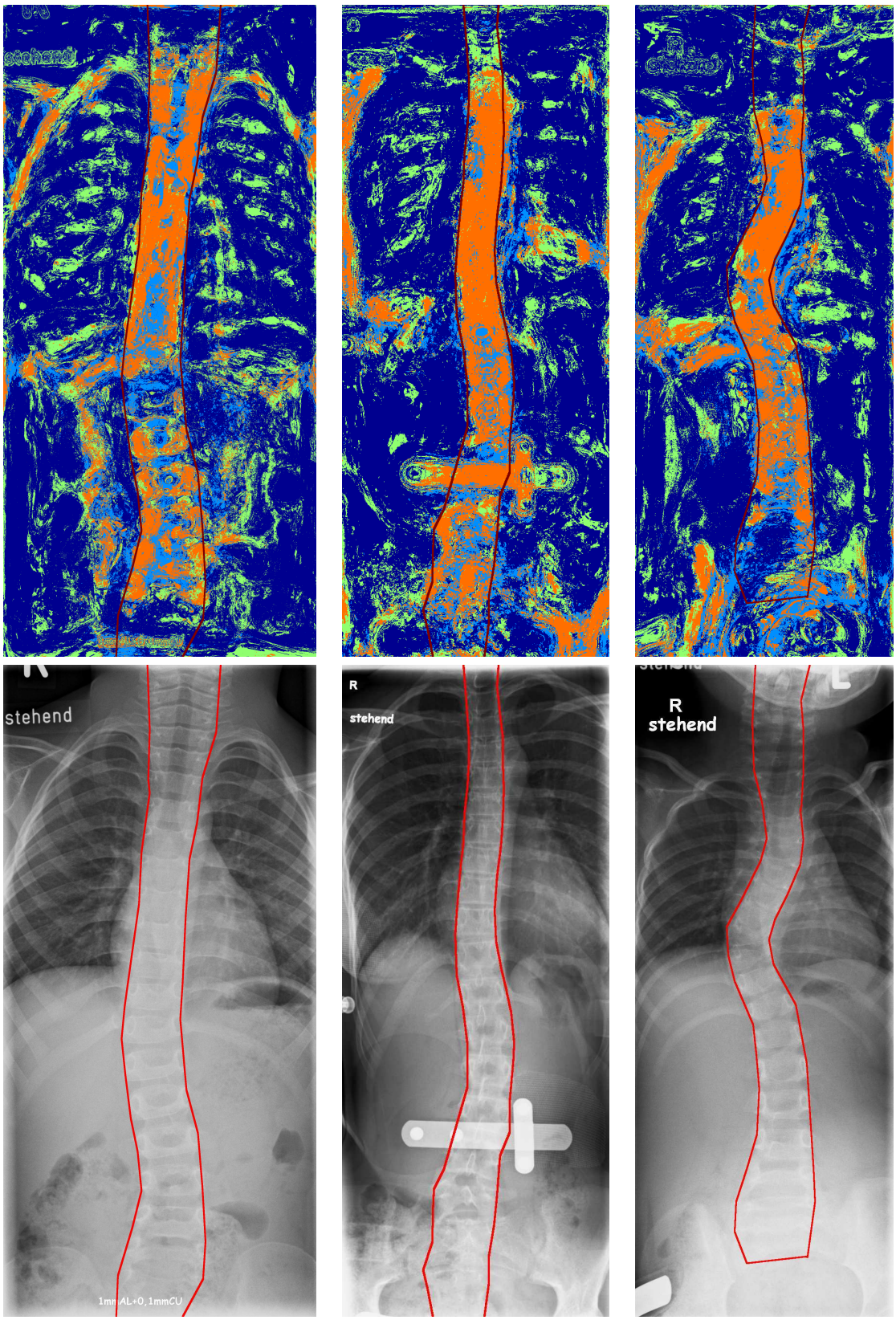


Figure 3.24.: Images 2, 4 and 6.

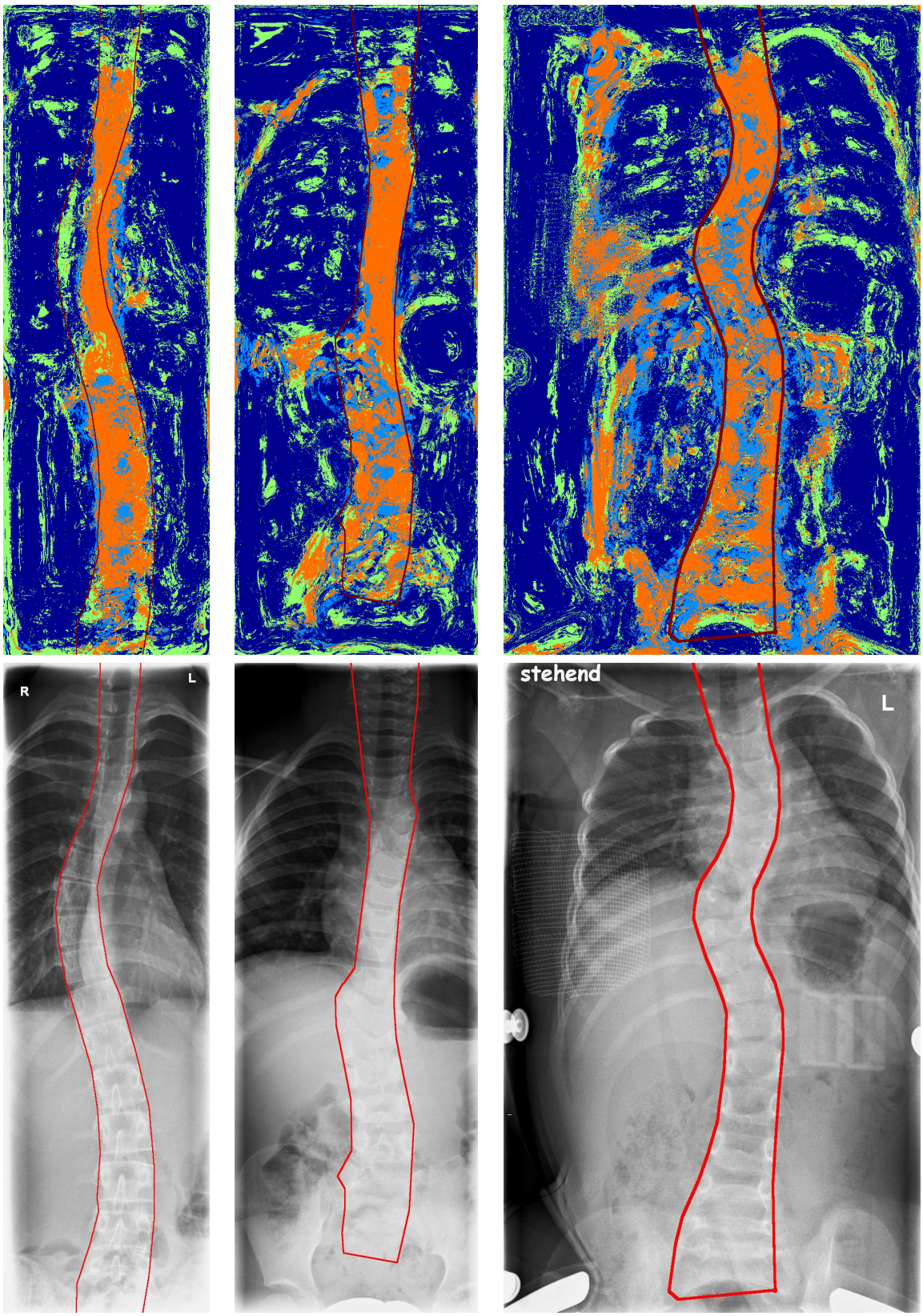


Figure 3.25.: Images 3, 8 and 10.

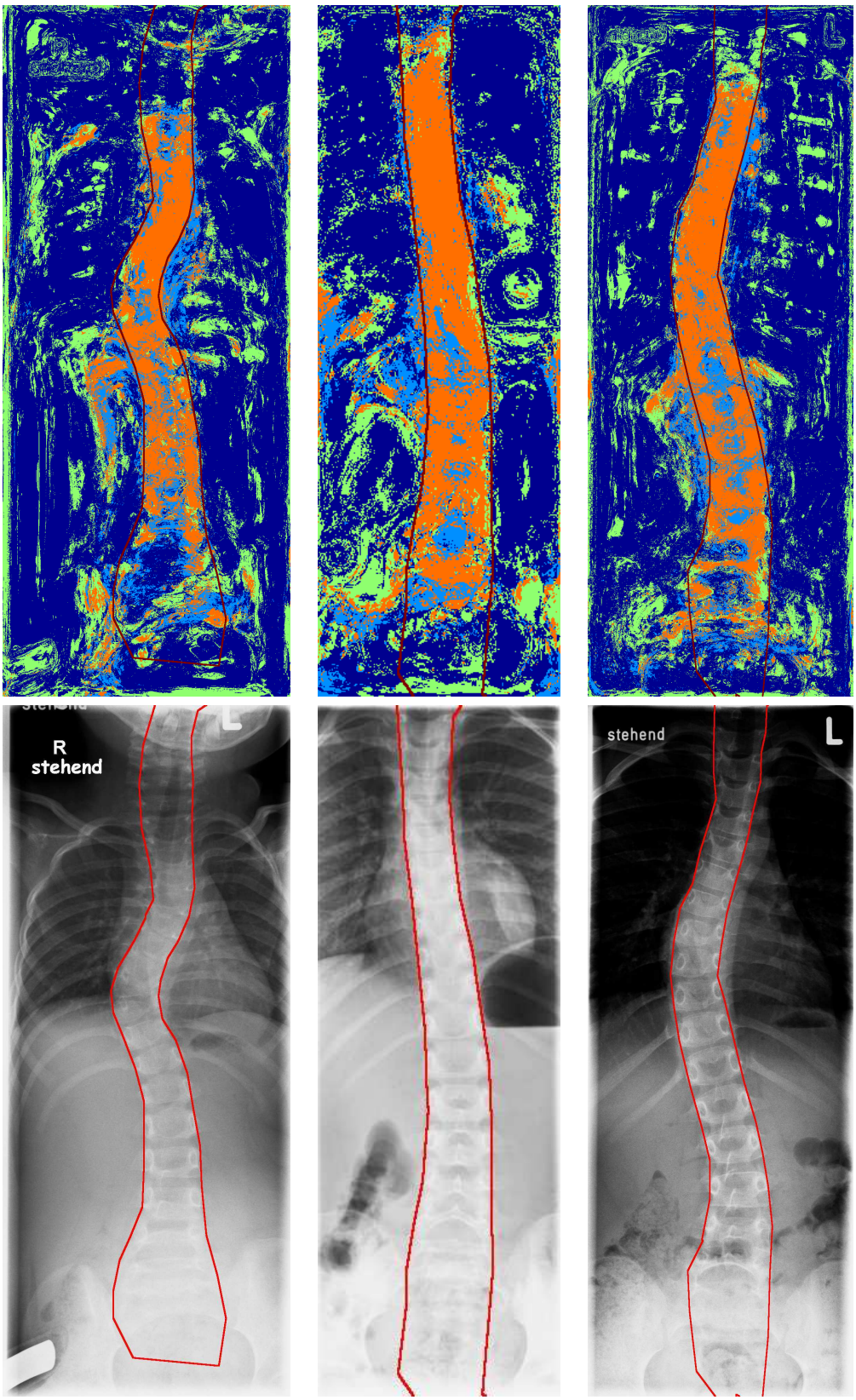


Figure 3.26.: Images 11, 12 and 13.

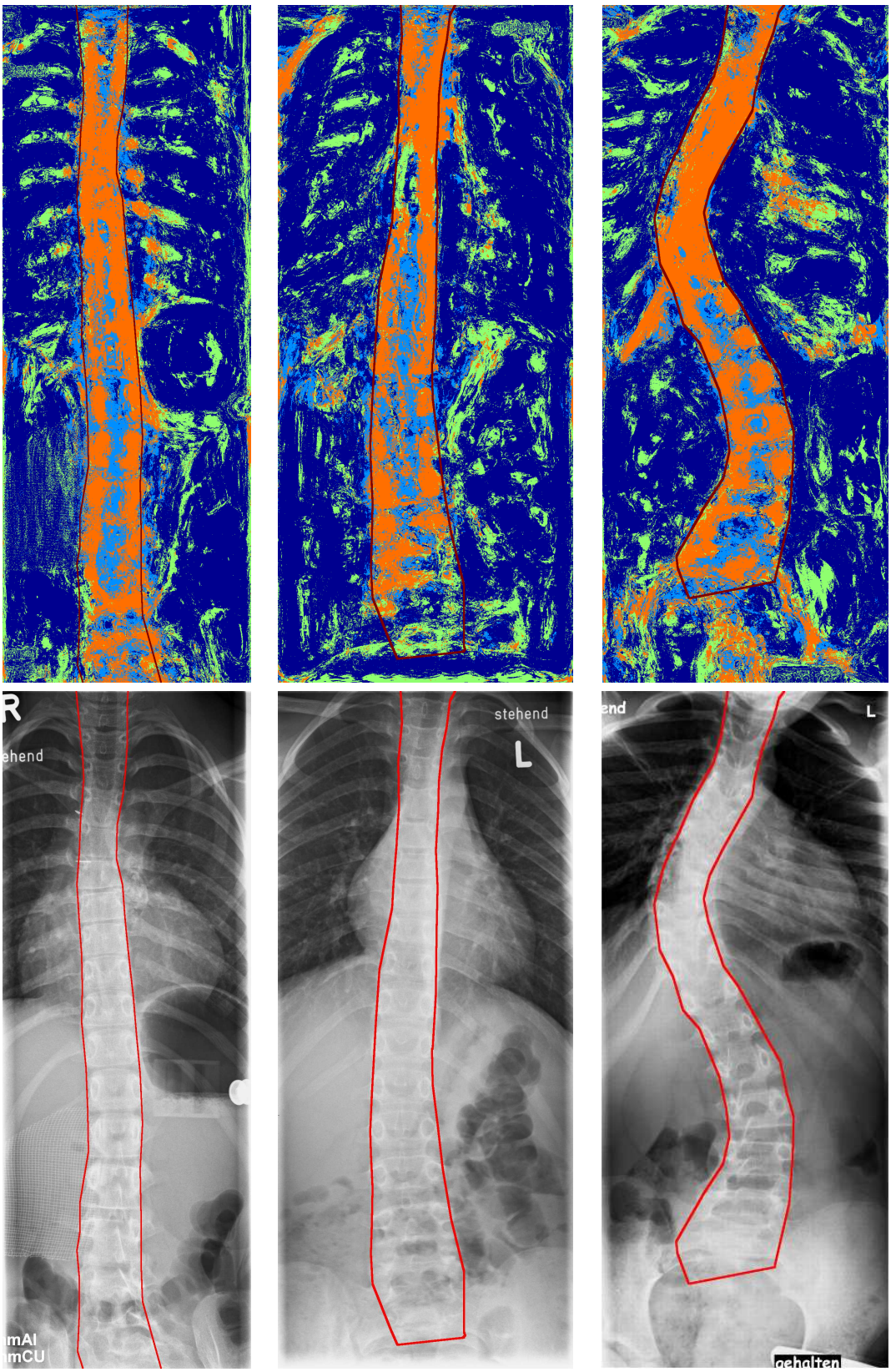


Figure 3.27.: Difficult images: 28, 31 and 36.

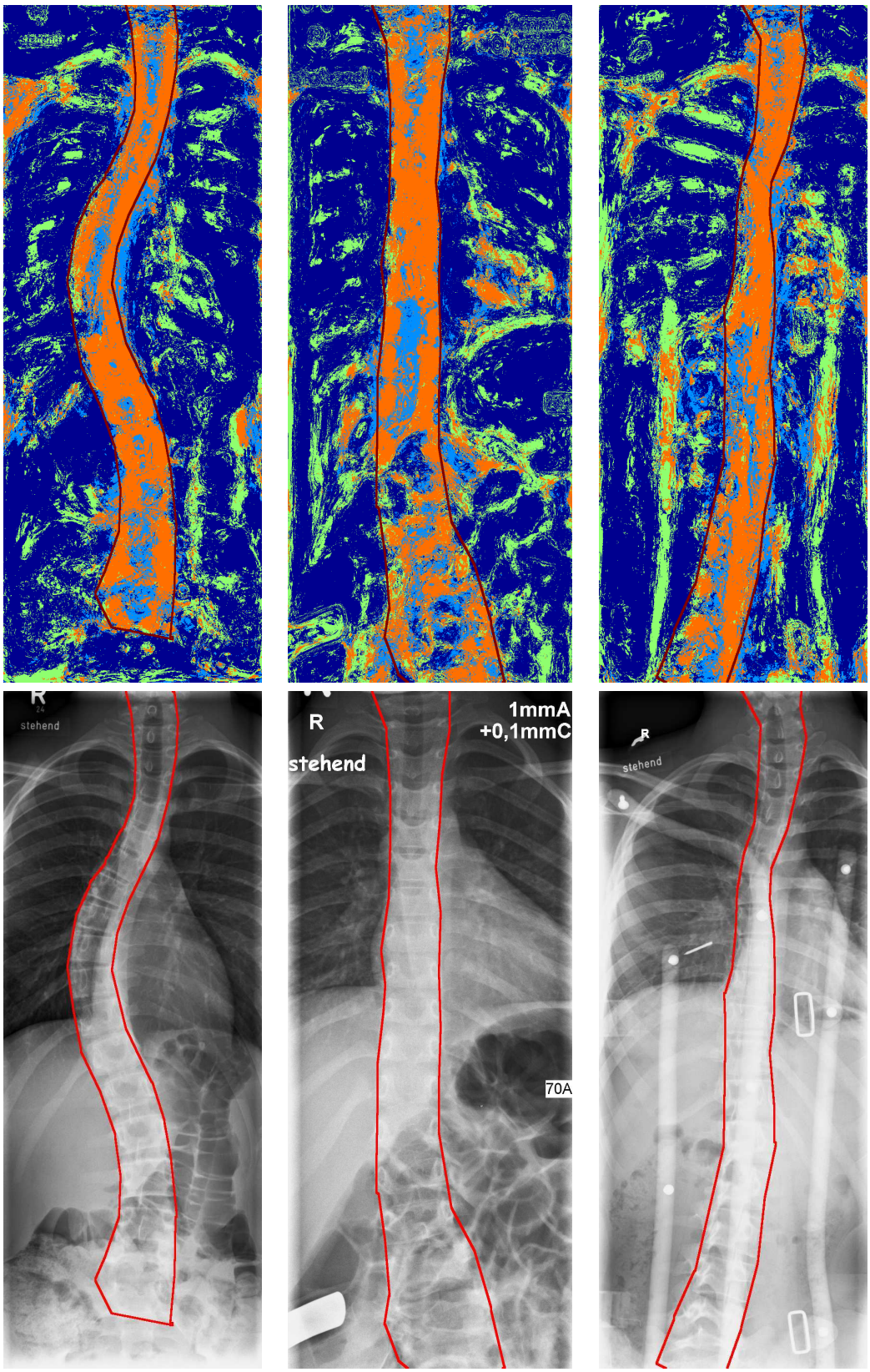


Figure 3.28.: Difficult images: 53, 69 and 90.

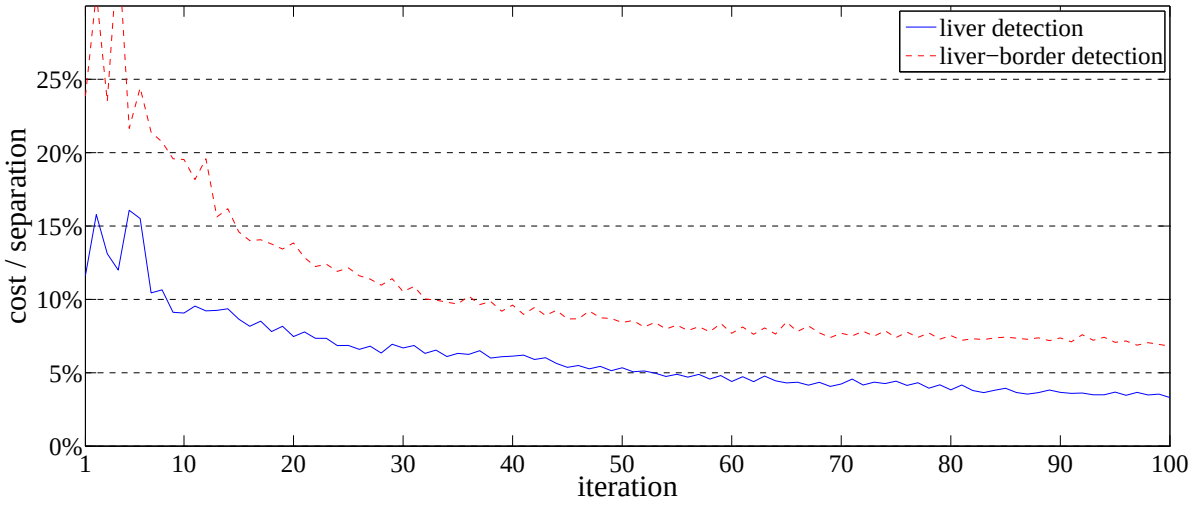


Figure 3.29.: The graphs of the costs for training improvements of the liver segmentation. We used 10 different training images.

3.8.2. Application on Liver CT-Photographs

Another difficult medical computer vision task is the segmentation of the liver in a stack of computer tomography data. The liver is a huge object on the left side in the images. As its shape varies, the current working segmentation routines always need human's pre segmentation. However, our presented results are only applications of the filter and not a liver segmentation approach itself.

For the demonstration of our method on liver photographs, we trained it on the photographs $14, 24, \ldots, 104$ and tested it on $14, 18, \ldots, 106$.

The results are depicted from Figure 3.30 to Figure 3.33.

The segmentation of the liver can be essentially upgraded by the use of interconnections between two slices. A two dimensional region growing approach could lead to a satisfying final segmentation. Another upgrade can be made by the consideration of not only the $x - y$-projection, but the projections in $x - z$ and $y - z$ direction, too. The knowledge adaption for the liver images becomes better than the knowledge adaption for the spine images (see Figure 3.29) as the images are more similar and of identical size.

The results of this image class is of the same quality as other current approaches (see also [26]).

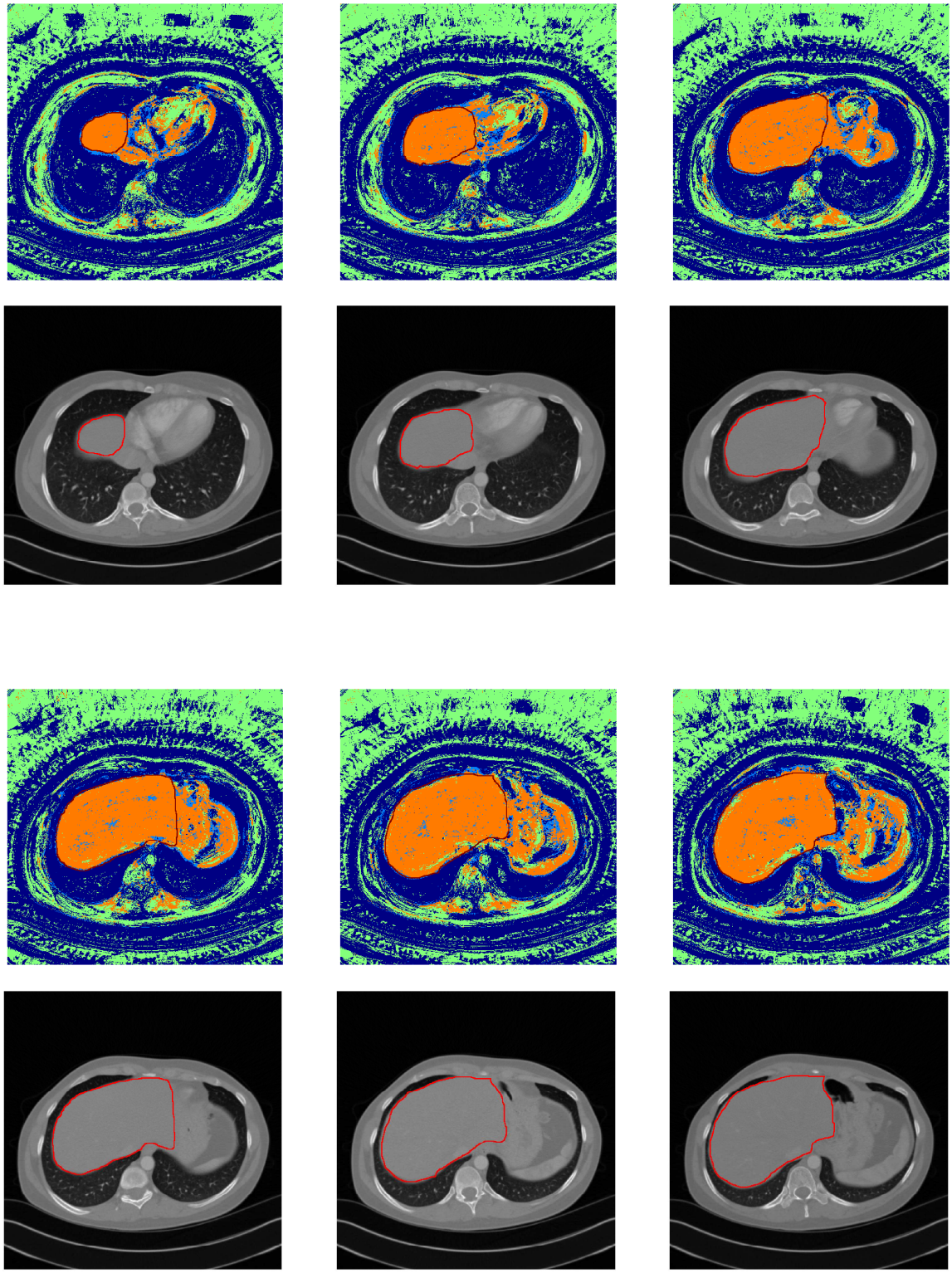


Figure 3.30.: Every fourth slice of a ct-stack. First and third row: Segmentations as for the spine detection. Second and fourth row: Input images as for the spine detection. Slices $14, 18, \ldots, 34$

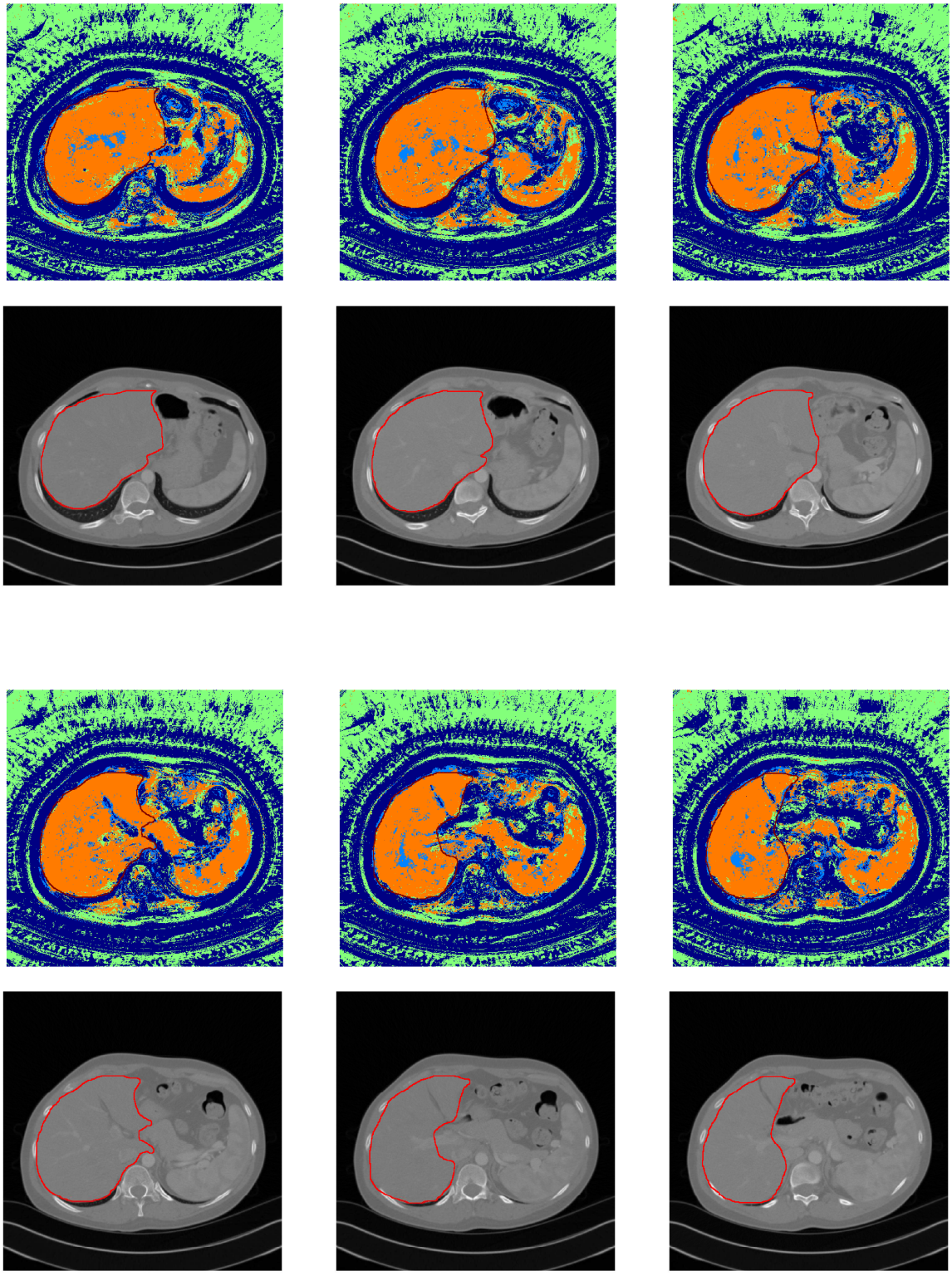


Figure 3.31.: Slices $38, 42, \ldots, 58$

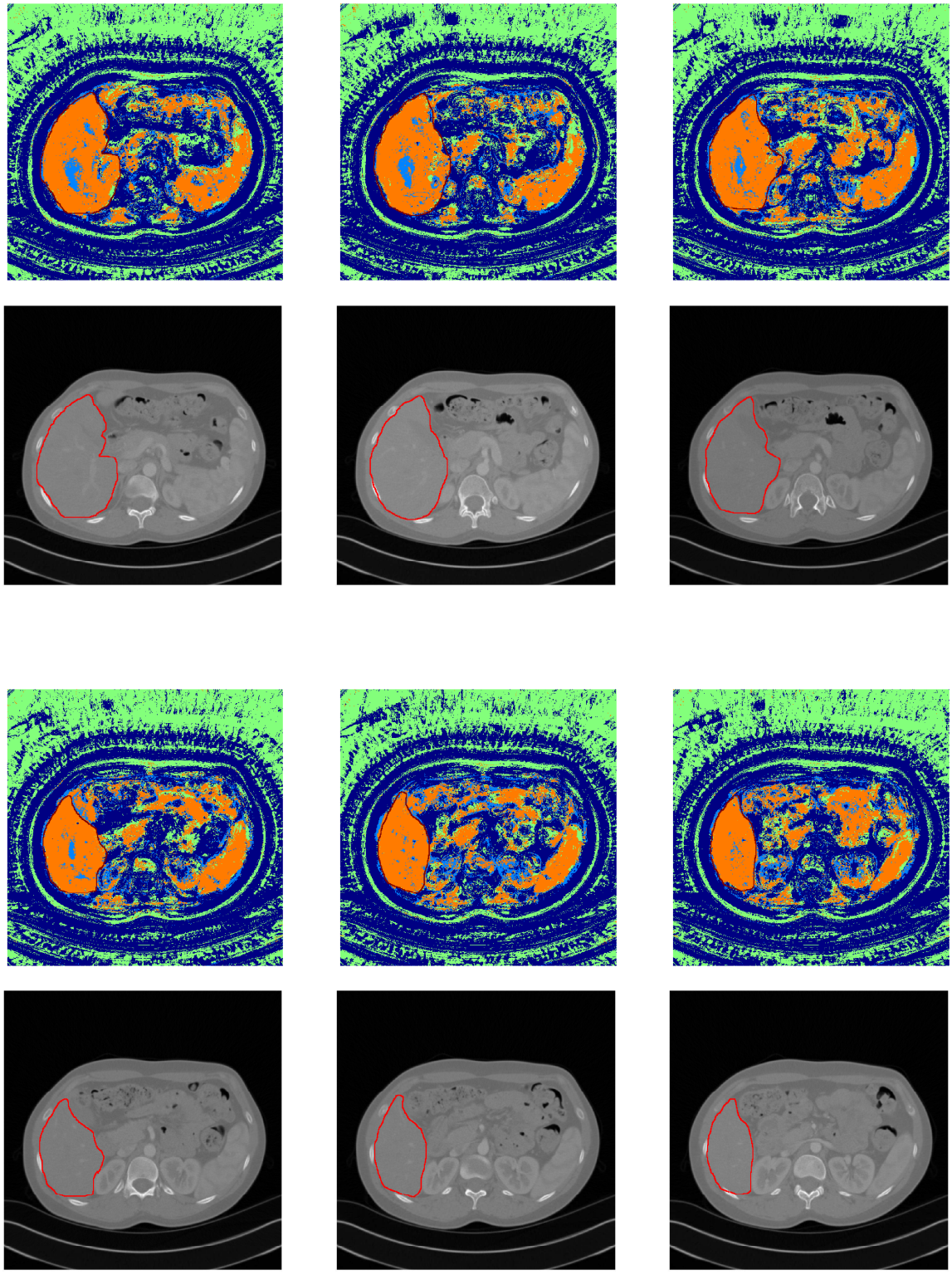


Figure 3.32.: Slices $62, 66, \ldots, 82$

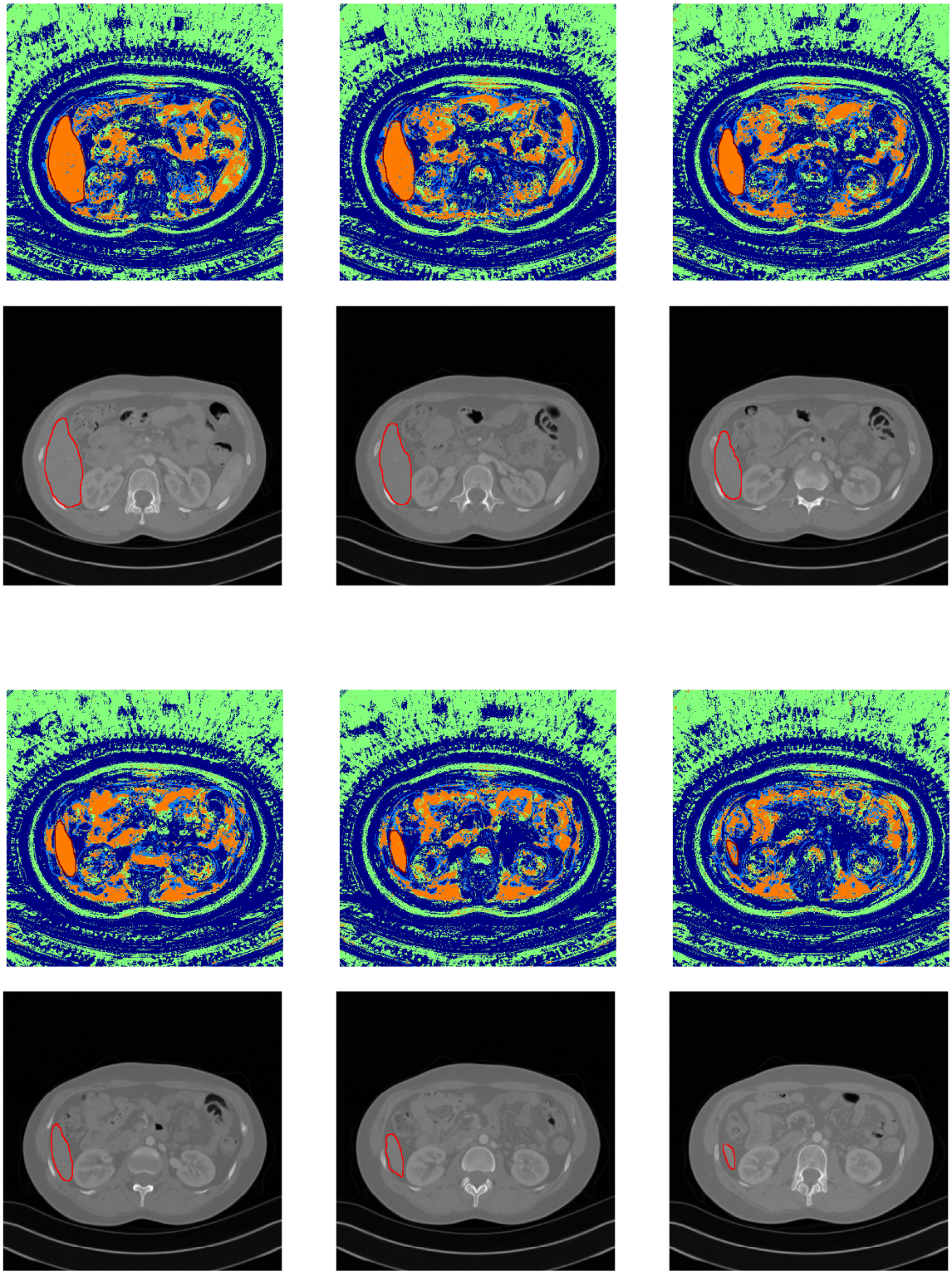


Figure 3.33.: Slices $86, 90, \ldots, 106$

3.9. Conclusion and Outlook

The proposed filter technique has some positive properties. Once having amplitude and phase signals for certain scale intervals we can train a segmentation for a special texture, such as in this example the spine or liver. The technique is relatively stable and the false-positive or false-negative error can be suppressed via a weighted evaluation function. As the segmentation result is in $\mathbb{R}$ and not in $\{0, 1\}$ as classical segmentation routines, it provides a measurement for each pixel being part of the texture or not.
Another advantage is the attribute being a single-point mapping, since each point is evaluated itself. It follows, that after the application of the analytic signal, there is no further object description necessary. As the filter is constant for each point, it can be applied very fast, for example on a GPU pixel-shader. Therefore, the filter can be executed in parallel, or in other words, in $\mathcal{O}(1)$ on an adequate hardware. We showed, that the calculation of the analytic signal can be done in less than one second for a 1 mega pixel image. The calculation of the SSSF for $n = 100$ iterations takes at most $t = 2.4$ seconds on the same system, since $t = n \times 384 \times \frac{1}{9*4*448}$seconds[22]. Hence we are able to compute one image in about three seconds on this GPU.
The next advantage concerns the false segmented points. When having some more knowledge about the considered structure, like for example some tightness or connectivity, we can expand the regions by considering the densities by convolving the result with a mask to get the mean value, which simulates the tightness.
Normal segmentations are region- or edge-based. This approach is nearly independent as it is a single-point mapping. We can find masks, which induce region based segmentations as the proposed spine segmentation and these which induce edge based ones as simple edge detectors based on the phase for small scales. In other words, this approach can be used for every kind of pixel highlighting and is therefore most abstract.
The properties of this filter are beside its potential of a high acceleration, the rotation invariance, luminance invariance and high adaptivity to any structure.
The filters only use the phases and amplitudes of the input image, therefore the filters are applied in parallel. Future work will deal with an expansion of the filter with a serial component, since the input signal for the phase and amplitude of filter i is the result of filter $i - 1$. This approach leads to much better results even if the training of this filter set lasts much longer: it requires fast hardware and good software. It looks like a neuronal network, where each neuron gets an image, computes the analytic signal and applies the filter to get the output image having same size and range as the input image. In the last chapter we show the quality of the filter by applying a final spine detection.

[22] The time for 9 pixel accesses is 448 frames / second, we only need 1 pixel access and we are able to compute 4 slots in parallel. In our implementation the size of one SSSF is maximal 384.

4. Design of a Spine Detector

In this chapter we will expand the results of the SSSF to the design of a final spine segmentation, which segments the spine from the background. This allows us to compare the proposed method with other spine segmentation programmes. We only design a very simple post processing of the spine segmentation, which underlines the power of the SSSF itself. First we consider, how medics solve this task. Afterwards, we consider the problem again from the programmer's point of view.

4.1. Image Processing from the Medical Point of View

For a human being it is typically no problem to segment the spine and spine bodies. Hence, they start immediately to measure the distortion. We only consider anterior-posterior photographs[1] which allow only a detection of the scoliosis in horizontal direction when watching from ahead. The most common measure for describing this kind of scoliosis is the Lippman-Cobb angle[2].
Figure 4.1 illustrates the calculation of the Lippman-Cobb angle. The first two figures (available at [45]) show the mapping from photographic data to scheme.
Consider the curve of the spine as a linear function $f_{\text{spine}} : \mathbb{R} \to \mathbb{R}, x = f_{\text{spine}}(y)$. The location of the Lippman-Cobb angle p^i_{cobb} is the local maximum (most right) or minimum (most left) point in f_{spine}. The angle itself is the maximum angle of the derivative f'_{spine} between the two adjacent extreme points. Hence n angles $\alpha_1, \ldots, \alpha_n$ for $n+2$ sampling points $p^0_{\text{cobb}}, \ldots, p^{n+1}_{\text{cobb}}$ are given by

$$\begin{aligned} \{p^0_{\text{cobb}}, \ldots, p^{n+1}_{\text{cobb}}\} &= \{\{0, 0, p^2_{\text{cobb}}, \ldots, p^{n-1}_{\text{cobb}}, y_{\max}, y_{\max}\}| \\ &\quad \forall i \in \{2, \ldots, n-1\} : p^{i-1}_{\text{cobb}} < p^i_{\text{cobb}} \wedge f'_{\text{spine}} = 0\} \qquad (4.1) \\ \forall i \in \{1, \ldots, n\} : \alpha_i &= \{\arctan(f'_{\text{spine}}(y^i_1)) - \arctan(f'_{\text{spine}}(y^i_2))| \\ &\quad y^i_j = \underset{y}{\operatorname{argmin}} \{f^{(2)}_{\text{spine}}(y) | y \in [p^{i-2+j}_{\text{cobb}}, p^{i-1+j}_{\text{cobb}}]\}\} \qquad (4.2) \end{aligned}$$

[1] AP radiographic projections of the spine
[2] also known as 'Cobb's angle', which was introduced in 1935.

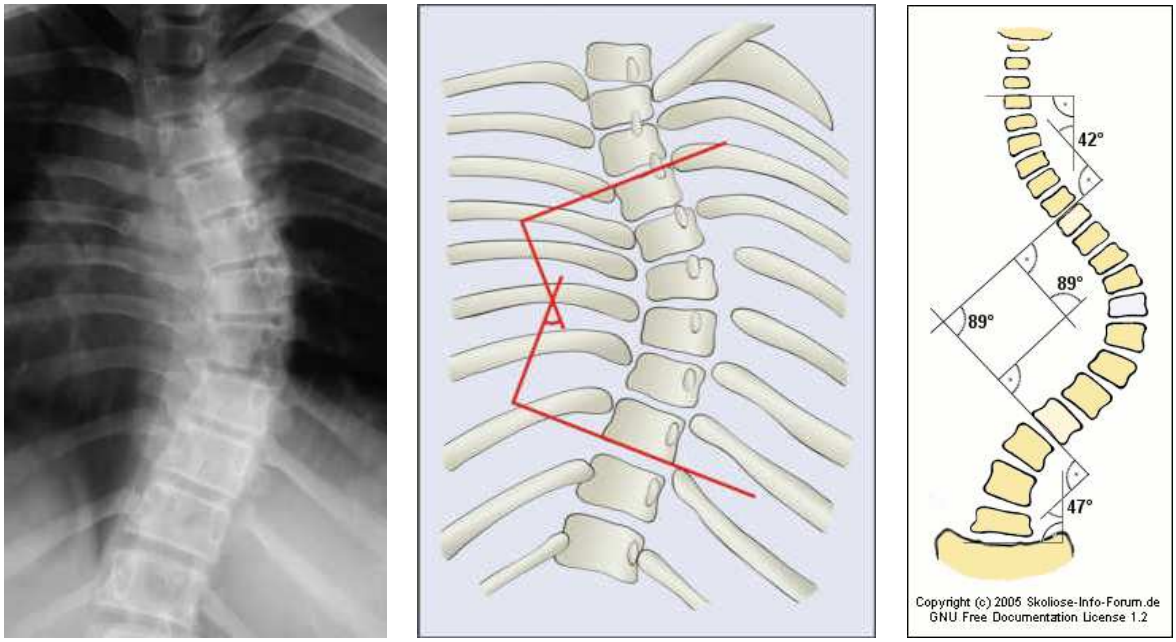


Figure 4.1.: Measuring the Lippman-Cobb angle: local spine photograph, corresponding scheme and general scheme for the Lippman-Cobb angle

with $y_{\max}$ the maximum pixel coordinate in $y-$direction. The i-th Lippman-Cobb angle is given by $\alpha^i_{\text{cobb}} = |\alpha_i|$.
Figure 4.1 illustrates the calculation. A spine curve is called decompensated if

$$\sum_{i=1}^{n} \alpha_i \neq 0. \tag{4.3}$$

The example from 4.1 (available at [46]) has values $\alpha_1 = 42°$, $\alpha_2 = -89°$ and $\alpha_3 = 47°$, this spine is called 'compensated' since the sum of the angles as $\sum_1^3 \alpha_i = 0$.
In medical diagnostics the locations must be declared in addition to the angles. Therefore spine bodies are partitioned into 5 groups from the top to the bottom cervical vertebras[3], thoracic[4], lumbar[5] vertebras, sacrum[6] and tailbone[7]. For scoliosis, generally only the spine bodies $C_4, \ldots, L_5$ are of relevance. These are 21 bodies, the radiograph is normally cut inside the cervical spine. The thoracic spine can be detected by holding up the ribs. The crossover between thoracic and lumbar spine is variable, because some individuals have 13 rib pairs.

[3] $C_1, \ldots, C_7$
[4] $T_1, \ldots, T_{12}$
[5] $L_1, \ldots, L_5$, sometimes also $(T_1, \ldots, T_{13})$ and $(L_1, \ldots, L_4)$
[6] $S_1, \ldots, S_5$
[7] at all 4 segments

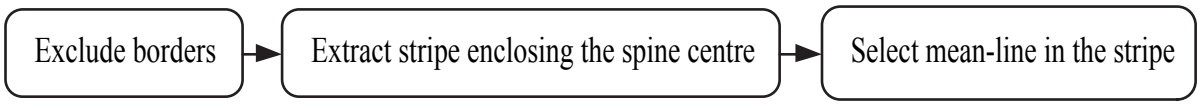


Figure 4.2.: Segmentation flow chart leading from input image to spine body segments.

There is another approach for measuring the scoliosis by using the midpoints of the vertebras called the Risser-Ferguson method. Robinson at al. have showed the equivalence of both methods in [32]. A more detailed introduction can be found in [30] and a compact overview to further methods is available in [33].
It is equivalent to select the upper and lower plates of the vertebras or the midpoints of each vertebra, as there is a mapping between the Risser-Ferguson and Lippman-Cobb method.

4.2. Programmer's Approach

The spine segmentation is the main task for an automatical system in contrast to the human approach. The principal proceeding is depicted in Figure 4.2.
The first step is the exclusion of everything, which does not contain any information, such as white border or noise. In the case of the spine segmentation this is an exclusion of some area on the left and the right sides. The second step extracts a stripe enclosing the spine centre. In the resulting neighbourhood it is easier to find the edges of the spine. The last step finally delivers the curve of the spine. This is the function f_{spine} as described in Section 4.1. After that, the calculation of the Lippman-Cobb angle is a trivial task. In the images 4.3 to 4.5 the results of the steps are marked. The result of the first step is the segmentation of the light blue area from the dark blue area. The stripe enclosing the spine centre is orange, the mean-line is yellow. Additionally the contours of the spine are marked with a red line.
It is important to note that every arrow between the boxes in 4.2 is afflicted with a special probability for finding the right mapping concerning the secondary task, as generally in computer vision the approximated function is a mapping between the input image and the conclusion, which goes along with an enormous reduction of dimensions, by simultaneously not reducing the intrinsic dimensions concerning the goal.
There are obviously many different other possibilities to join the task from Figure 4.2, but we will only consider this approach.
The Figures 3.24 to 3.28 show the results of the SSSF. The results of the first mask which only contains the object itself, are used for fulfilling the first two steps. The main

function inside these steps is the application of the analytic signal which determines the phase ϕ. The value $\cos(\phi)$ is reaching its maximum in the centre of the spine. This is cut out and expanded to get the stripe enclosing the spine centre.
For the next step we restrict the results of the spine-border detection to the stripe enclosing the spine. We calculate again $\cos(\phi)$ and apply some opening and closing procedures [37]. We get a highlighting, based on the spine-border detection and the stripe.
As the spine is tubular, we apply a track operator with a size of the mask equal to the spine width, in the next step. Its flattened result is already close to the optimum.
For the calculation of the scoliosis the segmentation of the spine bodies is not required.
The computer code of our approach is depicted in A.3.

4.3. Results

We show the results of our final segmentation function which only uses the phase, the calculation of the mean value, some noise deleting functions like opening and closing and the blob search[8].

4.3.1. Relation to other Spine Segmentation Filters

With these simple pre-processing techniques, we already reach good segmentation results. Applying active shape models, simulated annealing or the watershed transform could improve our results additionally. The percentage of overlapping of our final segmentation is only afflicted with a positive error of 5% and a negative error of 3%[9].
Beside, Brandt [4] splits down the segmentation of the spine in good and bad images and in upper parts and lower parts. He accomplishes a positive error and negative error of about 10% for images of the best quality. For images with worst quality he gains errors of 25%. However, he is not plotting any result images.
Also the average error of other spine segmentation approaches is near 10% (see also chapter 5 in [4]).
A common issue for comparing results is the fact, that many segmentation results are only given in tabular form and furthermore in millimetre, but not in the percentage of the overlap between estimated and computed results. Most of the approaches we described in the introduction of Section 3 use the energy minimisation and the region

[8]The blob search is the search for connected areas with same grey values.
[9]The positive error varies between 2.6% and 7%, the negative error varies between 1.7% and 4%.

growing approach, but need a human made calibration of the start values for each image. With this filter, we got a robust, fast, powerful and general segmentation tool which is superior to standard active shape or simulated annealing approaches at factor $3-5$.
The SSSF is neither region- nor edge-based: the colouration of our approach is more robust than these approaches, as it cannot be influenced by one false decision in the segmentation process which is not intrinsically successive.
Since the major part of our method is the computation of the SSSF, our approach still takes about three seconds per 1 mega-pixel image. The calculation of the post-processing after the SSSF does not contain any time consuming methods. In this sense, our approach is not only more accurate than any other method, but also very fast.

4.3.2. Conclusion

We demonstrated that we are able to exceed the state-of-the-art approaches with easy segmentation techniques like computing mean values, dilation, erosion and the calculation of the phase. Applying a more powerful postprocess on our filtered signals could improve the results further on.
Our approach is superior to region- and edge-based segmentations, as the pixel based approached cannot be influenced by false decisions outside the current pixel. We designed a spine segmentation, which is up to $3-5$ times more precise than the currently known approaches. We showed that our method is also applicable to difficult images which are images with implants covering the spine. The approach takes about three seconds per image, as the post-processing after the calculation of the SSSF does not contain any time consuming methods and the calculation of the input signals is possible in about three seconds.

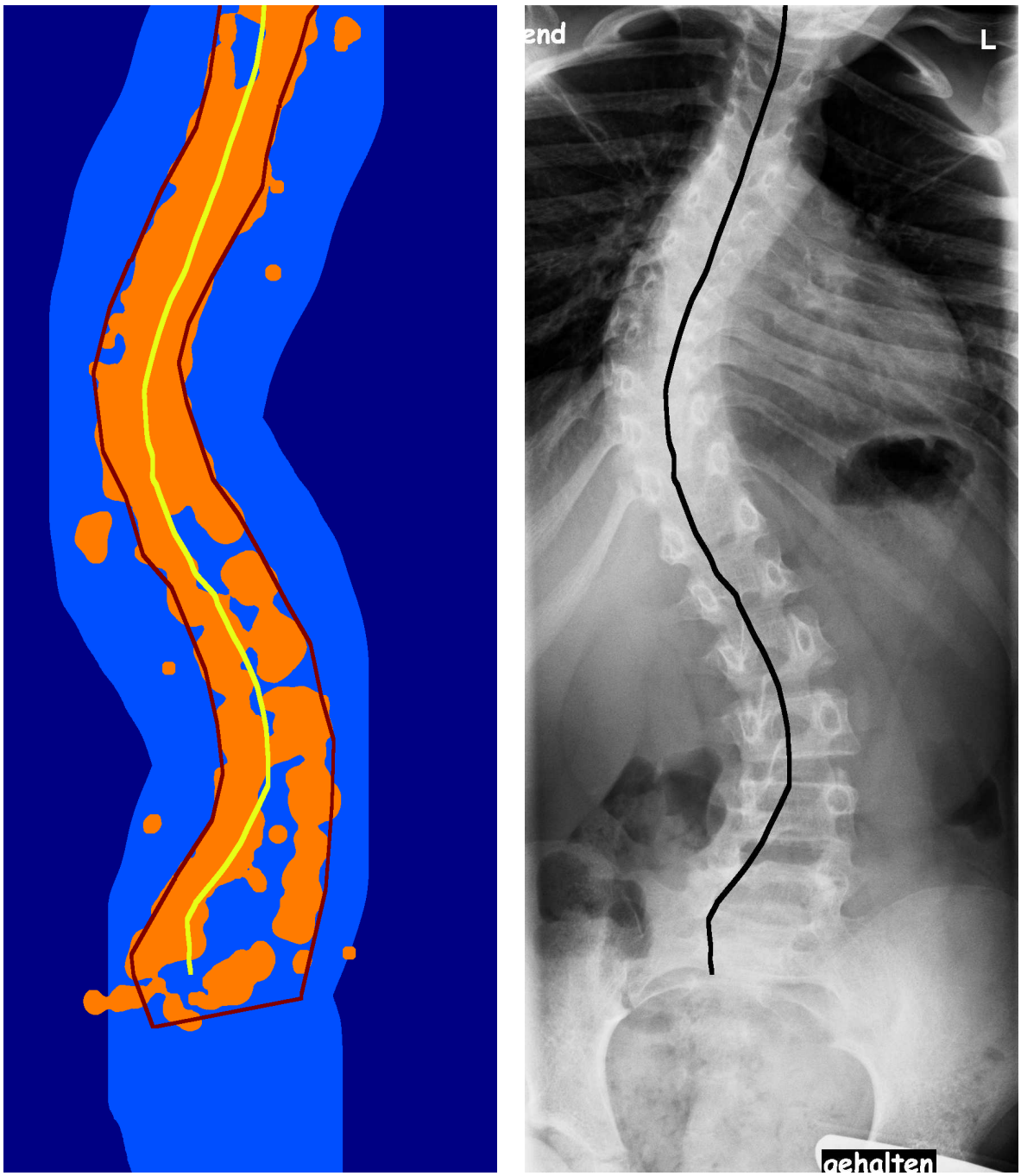


Figure 4.3.: Image 36. Left: Segmentation-steps. Right: Input and output of our function. The errors are: false negative = 7%, false positive = 1.7%.

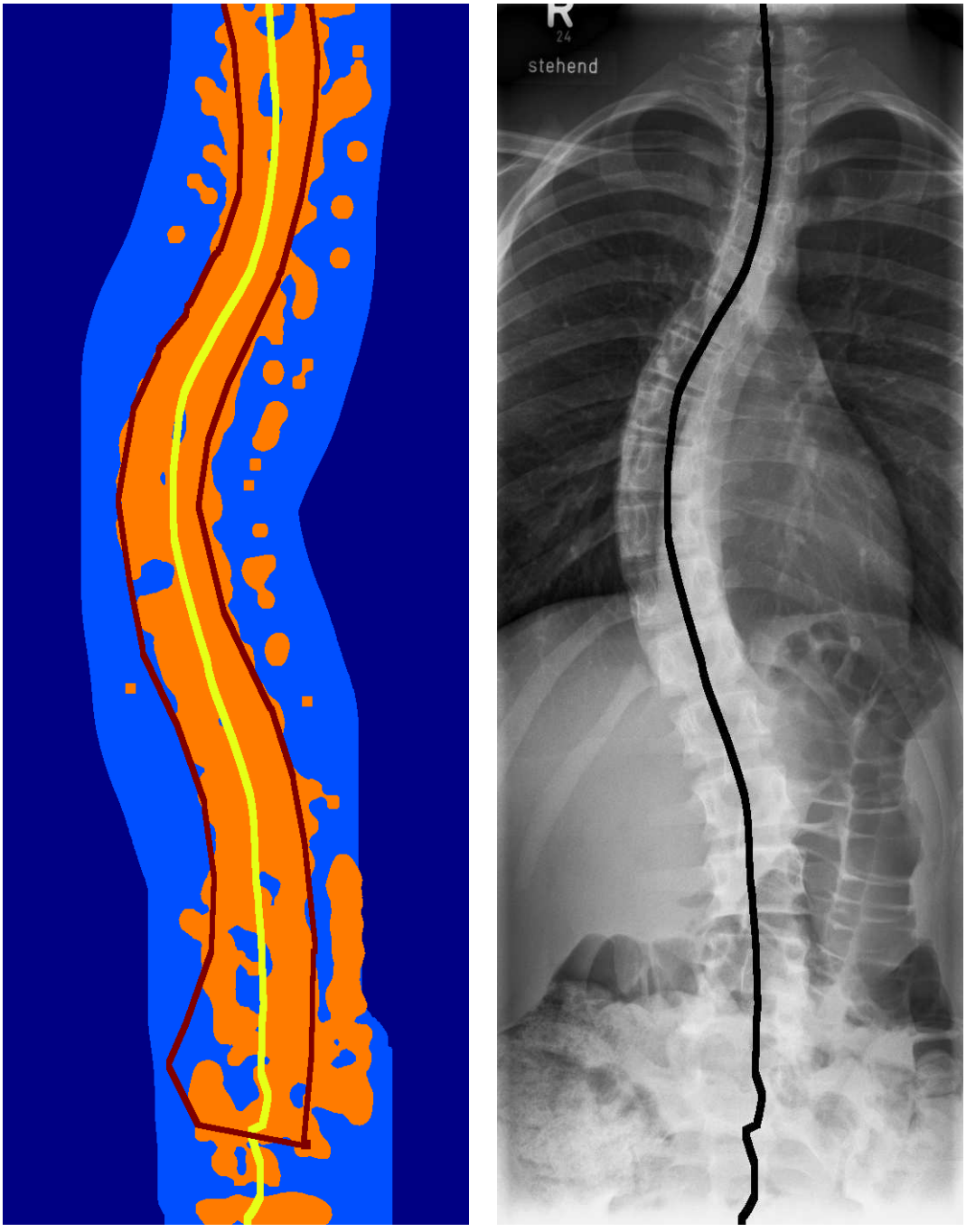


Figure 4.4.: Image 53: False negative = 3.4%, false positive = 4%.

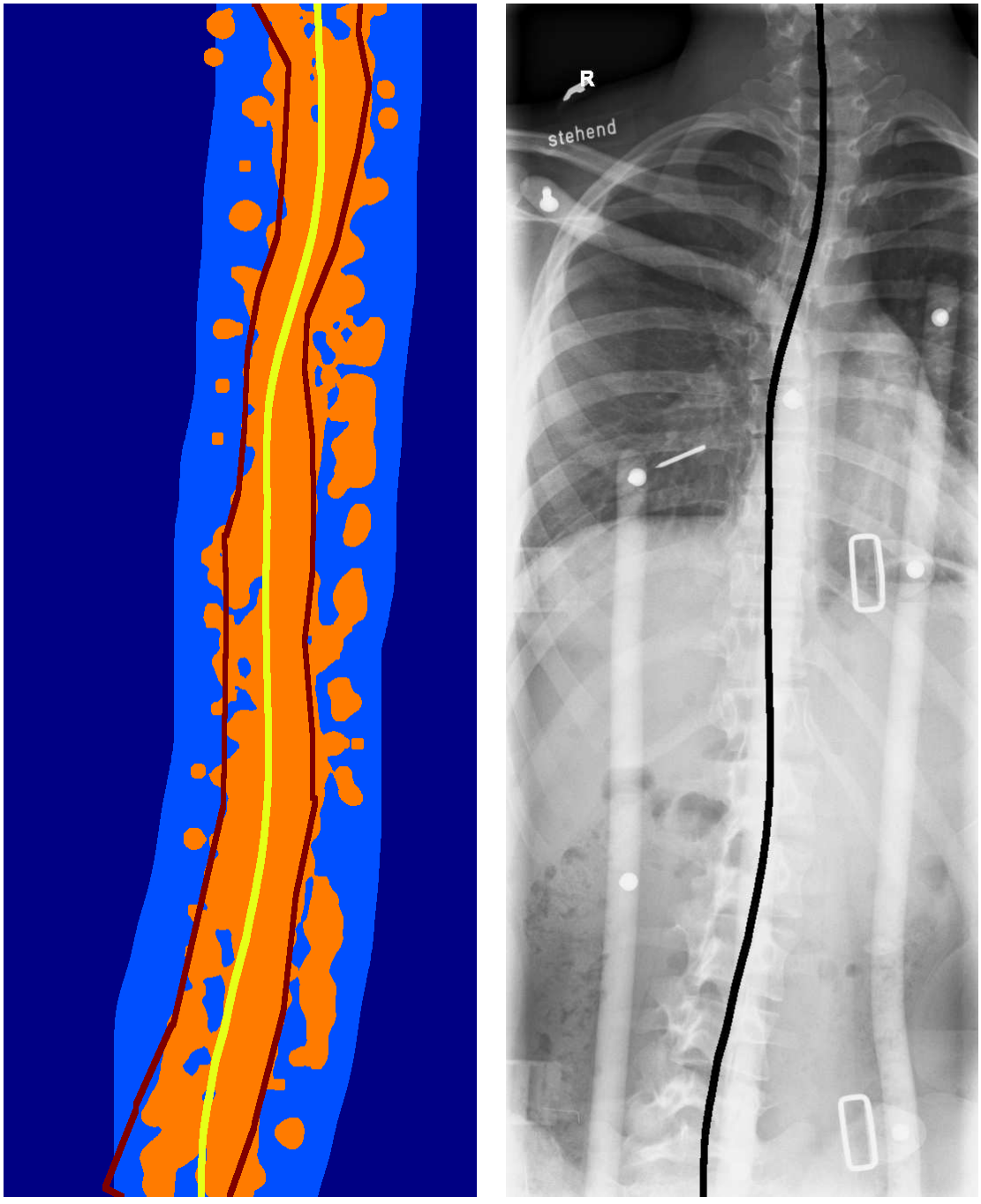


Figure 4.5.: Image 90: False negative = 2.6%, false positive = 2.8%.

Bibliography

[1] E. H. Anderson, C. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid methods in image processing. *RCA Engineer*, 29:33–41, 1984.

[2] D. H. Ballard and C. M. Brown. *Computer Vision*. Department of Computer Science, University of Rochester, 1982.

[3] R. Beare and G. Lehmann. The watershed transform in ITK - discussion and new developments. Department of Medicine, Monash University, Australia and Unité de Biologie du Développement et de la Reproduction, Institut National de la Recherche Agronomique, Jouy-en-Josas, France, 2006.

[4] A. S. Brandt. *Vollautomatische Segmentierung von lateralen Wirbelsäulenröntgenogrammen: Auswertung und Analyse*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Medizinische Fakultät, 2005.

[5] C. Chevrefils, F. Chériet, G. Grimard, and C.-E. Aubin. Watershed Segmentation of Intervertebral Disk and Spinal Canal from MRI Images. *LNCS*, 4633:1017–1027, 2007.

[6] M. Felsberg. Monogenic Signal and Scale-Space - CVL. Retrieved April 12, 2010, from `http://www.cvl.isy.liu.se/research/ima/monogenic-signal-and-scale-space`.

[7] M. Felsberg. monogenic.zip/@monogenic/private/create_DOP.m. Retrieved April 12, 2010, from `http://www.isy.liu.se/~mfe/monogenic.zip`.

[8] M. Felsberg. Low level image processing with the structure multivector. Technical report, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2002.

[9] M. Felsberg and G. Sommer. The multidimensional isotropic generalization of quadrature filters in Geometric Algebra. In *2nd International Workshop on Algebraic Frames for the Perception-Action Cycle, AFPAC 2000, Kiel*, 2000.

[10] M. Felsberg and G. Sommer. The monogenic signal. *IEEE Transactions on Signal Processing*, 49:3136–3144, 2001.

[11] M. Felsberg and G. Sommer. The monogenic scale-space: A unifying approach to phase-space. *Journal of Mathematical Imaging and Vision*, 21:5–26, 2004.

[12] M. Felsberg and G. Sommer. The Monogenic Scale Space on a Rectangular Domain and its Features. *International Journal of Computer Vision*, 64:187–201, 2005.

[13] O. Fleischmann. Local Signal Analysis by Generalized Hilbert Transforms in Conformal Space. Master's thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2008.

[14] W. T. Freeman and E. H. Adelson. The Design and Use of Steerable Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13:891–901, 1991.

[15] J. M. Gauch. Image segmentation and analysis via multiscale gradient watershed hierarchies. *IEEE Transactions on Image Processing*, 8:69–79, 1999.

[16] V. Grau, H. Becher, and J. A. Noble. Phase-Based Registration of Multi-view Real-Time Three-Dimensional Echocardiographic Sequences. *LNCS*, 4190:612 – 619, 2006.

[17] V. Grau, A. Mewes, M. Alcañiz, R.Kikinis, and S. Warfield. Improved Watershed Transform for Medical Image Segmentation Using Prior Information. *IEEE Transactions on Medical Imaging*, 23:447–458, 2004.

[18] V. Grau and J. A. Noble. Adaptive Multiscale Ultrasound Compounding Using Phase Information. *LNCS*, 3749:589 – 596, 2005.

[19] C. Grigorescu, N. Petkov, and M. A. Westenberg. Contour and boundary detection improved by surround suppression of texture edges. *Image and Vision Computing*, 22:609–622, 2004.

[20] B. Jähne. *Digitale Bildverarbeitung*. Springer Berlin, 2002.

[21] J. Kaminsky, P. Klinge, T. Rodt, M. Bokemeyer, W. Luedmann, and M. Samii. Specially adapted interactive tools for an improved 3D-segmentation of the spine. *Computerized Medical Imaging and Graphics*, 28:119–127, 2004.

[22] R. P. Kanwal. *Linear integral equations*. Birkhauser, 1997.

[23] T. Kindler, R. Wolz, C. Lorenz, A. Franz, and J. Ostermann. Spine Segmentation Using Articulated Shape Models. *LNCS*, 5241:227–234, 2008.

[24] C. Lei, L. Xiaojian, Z. Jie, and C. Wufan. Automated lung segmentation algorithm for CAD system of thoracic CT. *Journal of Medical Colleges of PLA*, 23:215 – 222, 2008.

[25] G. Mittelhäußer and F. Kruggel. Fast Segmentation of Brain Magnetic Resonance Tomograms. In *Computer Vision, Virtual Reality and Robotics in Medicine*, 1995.

[26] Y. Nakayama, Q. Li, S. Katsuragawa, R. Ikeda, Y. Hiai, K. Awai, S. Kusunoki, Y. Yamashita, H. Okajima, Y. Inomata, and K. Doi. Automated Hepatic Volumetry for Living Related Liver Transplantation At Multisction CT. *Radiology*, 240:743–748, 2006.

[27] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7:308–313, 1965.

[28] H. Nguyen and Q. Ji. Improved watershed segmentation using water diffusion and local shape priors. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2006.

[29] O. F. Olsen and M. Nielsen. Multi-Scale Gradient Magnitude Watershed Segmentation. In *LNCS*, 1997.

[30] D. Pate. Radiologic Techniques for Evaluating Scoliosis. *Dynamic Chiropractic*, 8:1–4, 1990.

[31] K. Rajpoot, A. Noble, V. Grau, and N. Rajpoot. Feature Detection from Echocardiography Images Using Local Phase Information. In *Proceedings of the 12th Annual Conference on Medical Image Understanding and Analysis*, 2008.

[32] E. F. Robinson and W. D. Wade. Statistical assessment of two methods of measuring scoliosis before treatment. *Can Med Assoc Journal*, 129:839–841, 1983.

[33] J. Sample. International Digital Technologies, Inc. Retrieved November 24, 2009, from `http://www.xraydigitizing.com/www.xraydigitizing.com/reports/bio_report.html`.

[34] A. Sedlazeck. Local Feature Detection by Higher Order Riesz Transforms on Images. Master's thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2008.

[35] K.-S. Seo, L. C. Ludeman, S.-J. Park, and J.-A. Park. Efficient Liver Segmentation Based on the Spine. *LNCS*, 3261:400–409, 2004.

[36] G. Sommer. Skriptum: Signaltheoretische Grundlagen der Bildverarbeitung. Technical report, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2006.

[37] G. Sommer. Skriptum: Stochastische, topologische und geom. Grundlagen von Computer Vision. Technical report, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2009.

[38] L. Vicent and P. Soille. Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. *IEEE Transactions on pattern analysis and machine intelligence*, 13:583–598, 1991.

[39] L. Wietzke, O. Fleischmann, and G. Sommer. 2D Image Analysis by Generalized Hilbert Transforms in Conformal Space. In *ECCV (2)*, 2008.

[40] L. Wietzke and G. Sommer. The 2D Analytic Signal. Technical report, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2008.

[41] L. Wietzke and G. Sommer. The Relation of Inverse Problems and Isotropic 2D Signal Analysis. In *Mathematics in Signal Processing 8*, 2008.

[42] L. Wietzke and G. Sommer. The Signal Multi-Vector. In *Journal of Mathematical Imaging and Vision*, 2010.

[43] L. Wietzke, G. Sommer, and O. Fleischmann. The Geometry of 2D Image Signals. In *CVPR*, 2009.

[44] D. Zang. *Signal Modeling for Two-Dimensional Image Structures and Scale-Space Based Image Analysis*. PhD thesis, Christian-Albrechts-Universität Kiel, Institut für Informatik, 2007.

[45] Cobb's angle. Retrieved November 24, 2009, from `http://www.e-radiography.net/radpath/c/cobbs-angle.htm`.

[46] Scoliosis cobb.svg. Available at `http://de.wikipedia.org/w/index.php?title=Datei:Scoliosis_cobb.svg\&oldid=52441366`.

List of Figures

Appendix A.

Code

A.1. Analytic Signal

Sample Code

Listing A.1: MainScaleExample.m

```
function scaleImage = MainScaleExample()
% scaleImage = MainScaleExample()
% Author: Felix Thomsen
% Example to understand the code structure

% take a synthetic image with orientation = pi/8, scale = 10 and
% amplitude=1:
image = SynthImage(100,[1;pi/8;0;10],1);

% take 2-dimensional analytic signal scale space with depth =16 and maximal
% coarse scale = 100
[amplitude,phase,scales] = AllAS(image,16,100);

% compute the main scale with the standard approach
scaleImage = MainScale(amplitude,phase,scales,1);

% plot the result:
surf(scaleImage);
```

Calculation of the Analytic Signal

Listing A.2: AS.m

```
function [orientation,phase,amplitude,apexAngle]=...
    AS(image,fineScale,coarseScale,nMax)
% [orientation,phase,amplitude,apexAngle,attenuation]=
% AS(image,fineScale,coarseScale,nMax)
% Author: Felix Thomsen
%
% calculates the 2-D Analytic Signal on image with fineScale and
% coarseScale and a maximal mask size nMax
% default: fineScale = 1; coarseScale = 100; nMax = 16;
if nargin<1
    error('need at least 1. parameter image');
end
if nargin<2
    fineScale = 1;
end
if nargin<3
    coarseScale = 100;
end
if nargin<4
    nMax = 16;
end
[kP,kX,kY,kXX,kYY,kXY,dx] = create_AK(fineScale,coarseScale,nMax);
% image preprocessing
if (ndims(image)==3)
    img =double(rgb2gray(image));
else
    img = double(image);
end

% convolutions
f_p = convolution(img,kP,dx);
f_x = convolution(img,kX,dx);
f_y = convolution(img,kY,dx);
f_xx = convolution(img,kXX,dx);
f_yy = convolution(img,kYY,dx);
f_xy = convolution(img,kXY,dx);

% signal calculation
f_pm = 0.5 .*(f_xx -f_yy);
f_s = 0.5 .* f_p;
f_plus = f_xy;
e = sqrt(f_pm.^2 + f_plus.^2)./abs(f_s);
q = (f_x.^2 + f_y.^2) .* 2 ./(1+e);

phase = atan2(sqrt(q),f_p);

orientation = atan2(f_y,f_x);
```

```
48 epsilon = 0.001;
49 orientation(orientation>(2*pi-epsilon)) = 0;
50
51 amplitude =  sqrt(real(f_p.^2 + q)) .*0.5;
52
53 apexAngle = real(acos(sqrt(f_y.^2+f_x.^2)./abs(f_s)));
54 end
55
56 %------------------------------------------------------------------------
57 % Create kernels and convolution
58 %------------------------------------------------------------------------
59 function [kP,kX,kY,kXX,kYY,kXY,dx]= create_AK(fineScale,coarseScale,maxN)
60 % calculates convolution mask size for static error = 5%
61 % if calculated size>maxN -> take smaller imprecise size
62
63 % calculate convolution size n for parameter E=error
64 E = 2;
65 n = coarseScale * E;
66 dx = ceil(n / maxN);
67 n  = floor(n / dx);
68 [x,y] = meshgrid(-n:n,-n:n);
69 coarseScale = coarseScale / dx;
70 fineScale = fineScale / dx;
71
72 % Kernel1FS, Kernel1CS, Kernel2FS, Kernel2CS
73 ssFS = fineScale^2;
74 ssCS = coarseScale^2;
75 z = x.^2+y.^2;
76
77 kernel1FS = 1 ./ (2 * pi .* (ssFS + z).^(3/2));
78 kernel1CS = 1 ./ (2 * pi .* (ssCS + z).^(3/2));
79 kernel2FS = (2 * pi) .* (z.^2) .*((ssFS+z).^(3/2));
80 kernel2CS = (2 * pi) .* (z.^2) .*((ssCS+z).^(3/2));
81
82 control = (kernel2FS ~= 0);
83 kernel2FS(control) = -(fineScale .* (2 * ssFS + 3.*z(control))...
84     - 2.*(ssFS + z(control)).^(3/2))./ kernel2FS(control);
85 kernel2CS(control) = -(coarseScale .* (2 * ssCS + 3.*z(control))...
86     - 2 .*(ssCS + z(control)).^(3/2))./ kernel2CS(control);
87
88 % kernels kP,kX,kY,kXX,kYY,kXY
89 kP = freeDC(fineScale .* kernel1FS - coarseScale .* kernel1CS);
90 kX = freeDC(x .*(kernel1FS - kernel1CS));
91 kXX = freeDC(x.^2 .* (kernel2FS - kernel2CS));
92 kXY = freeDC(x.*y .* (kernel2FS - kernel2CS));
93 kY = kX';
94 kYY = kXX';
95 end
96
97 function kernelB = freeDC(kernel)
98     kernelB = kernel - mean(kernel(:));
```

```
99  end
100
101 function img = convolution(img,convKernel,dx)
102 % img = convolution(img,convKernel,dx)
103 %
104 % Comnvolutes image img with mask convKernel similar to
105 % conv2(img,convKernel,'valid')
106 % dx in {1,2,...} specifies the offset between pixels.
107 % mask convKernel needs odd size and 1 =width/height
108 % if n = size(convKernel,1), the convolution result equals a normal
109 % convolution with a kernel of size
110 % m = (n-1) * dx +1
111 % Example : n = 11, dx = 2 -> m = 21
112 % for mean-value calculation one needs a pre-convolution mask of size dx.
113
114 % generates for dx=2  4 images with following structure for width=5,
115 % height=6:
116 %  image(1,1)         |   image(1,2)
117 %  1,1 - 3,1 - 5,1    |   2,1 - 4,1
118 %  1,3 - 3,3 - 5,3    |   2,3 - 4,3
119 %  1,5 - 3,5 - 5,5    |   2,5 - 4,5
120 % ----------------------------------
121 %  image(2,1)         |   image(2,2)
122 %  1,2 - 3,2 - 5,2    |   2,2 - 4,2
123 %  1,4 - 3,4 - 5,4    |   2,4 - 4,4
124 %  1,6 - 3,6 - 5,6    |   2,6 - 4,6
125 % with x,y as pixel(x,y) from image img
126 n = (size(convKernel,1)-1)/2;
127 % Mean-value calculation
128 if dx>1
129     img =padarray(padarray(img',floor(dx/2),'replicate')',...
130         floor(dx/2),'replicate');
131     flattenN = floor(dx/2)*2+1;
132     flattenMask = ones(flattenN);
133     if mod(dx,2) == 0
134         flattenMask(:,1) = flattenMask(:,1).* 0.5;
135         flattenMask(:,end) = flattenMask(:,end).* 0.5;
136         flattenMask(1,:) = flattenMask(1,:).* 0.5;
137         flattenMask(end,:) = flattenMask(end,:).* 0.5;
138     end
139     flattenMask = flattenMask ./ sum(flattenMask(:));
140     img = conv2(img,flattenMask,'valid');
141 end
142
143 % width and height calculation
144 img = padarray(padarray(img',n*dx,'replicate')',n*dx,'replicate');
145 [height,width] = size(img);
146 ii = 1:1:dx;
147 w = ceil((width-ii+1)./dx);
148 h = ceil((height-ii+1)./dx);
149
```

```
% generate and convolute dx*dx image parts
cImages = zeros(h(1)-(n*2),w(1)-(n*2),dx,dx);
for ih=1:1:dx
    for iw=1:1:dx
        curImage = zeros(h(1),w(1));
        hh = 1:1:h(ih);
        ww = 1:1:w(iw);
        curImage(hh,ww) = img((hh-1).*dx+ ih,(ww-1).*dx+ iw);
        cImages(:,:,ih,iw) = conv2(curImage(:,:),convKernel,'valid');
    end
end

% write back convolution results
img = zeros(height-(n*dx*2),width-(n*dx*2));
for hh=1:1:height-(n*dx*2);
    for ww=1:1:width -(n*dx*2);
        img(hh,ww) = cImages(ceil(hh./dx),ceil(ww./dx),...
            mod((hh-1),dx)+1,mod((ww-1),dx)+1);
    end
end
end
```

Listing A.3: AllAS.m

```
function [amplitude,phase,scales] = AllAS(image,depth,maxScale)
% [amplitude,phase,scales] = AllAS(image,elements,maxScale)
% Author: Felix Thomsen
% calculates logarithmic scale space on image concerning the 2D-Analytic
% signal with depth elements, coarseScale(depth) = maxScale
% and fineScale(1) = 1
[h,w] = size(image);
logdiff = log(maxScale)/depth;
scales = exp(log(1):logdiff:log(maxScale));
amplitude = zeros(h,w,depth);
phase = zeros(h,w,depth);
for i=1:depth
    fprintf('i= %d, fineScale=%f, coarseScale=%f\n',i,scales(i),scales(i+1))
    [or,phase(:,:,i),amplitude(:,:,i)]= AS(image,scales(i),scales(i+1));
end;
```

Attenuation Based Procedures

Listing A.4: Attenuation.m

```
function attenuation = Attenuation(amplitude,phase,scales,kind)
% attenuation = Attenuation(amplitude,phase,scales,kind)
% Author: Felix Thomsen
% calculates attenuation with kind in {1,2,3}
[h,w,d] = size(amplitude);
attenuation = zeros(h,w,d);
switch kind
    case 1; % attenuation kind= 'a'
        for dd=1:d
            attenuation(:,:,dd) = amplitude(:,:,dd)...
            ./ (log(scales(dd+1))-log(scales(dd)));
        end;
    case 2; % attenuation kind= 'b'
        amplitude = amplitude .* abs(cos(phase));
        for dd=1:d
            attenuation(:,:,dd) = amplitude(:,:,dd)...
            ./ (log(scales(dd+1))-log(scales(dd)));
        end;
    case 3; % attenuation kind= 'c'
        phase = cos(phase);
        for dd=2:d-1
            wr = log(scales(dd+2))-log(scales(dd));
            wl = log(scales(dd+1))-log(scales(dd-1));
            attenuation(:,:,dd) = 1 - (abs(phase(:,:,dd)-phase(:,:,dd-1))*wr...
            + abs(phase(:,:,dd)-phase(:,:,dd+1))*wl)./(2*(wl+wr));
        end;
    otherwise; % no action
end;
```

Listing A.5: MainScale.m

```
function [scaleImage,maxAtt] = MainScale(amplitude,phase,scales,kind)
% [scaleImage,maxAtt] = MainScale(amplitude,scales,scaleKind,phase)
% Author: Felix Thomsen
% calculates the scale with highest attenuation (main scale)
% kind in {1,2,3,4}
% with 1-3 = highest attenuations with kind 1-3
% 4 = combination of 1 and 2
[h,w,d] = size(amplitude);
scaleImage= zeros(h,w);
maxAtt = zeros(h,w);
if kind==4 % combined method
    cutOff = 0.5;
    [scaleImageA,maxAttA] = MainScale(amplitude,phase,scales,1);
    threshold = maxAttA.*cutOff;
    clear scaleImageA;
    clear maxAttA;
```

```
17      att1 = Attenuation(amplitude,phase,scales,1);
18      att2 = Attenuation(amplitude,phase,scales,2);
19      for i=1:d
20          [maxAtt,scaleImage] = Update(maxAtt,scaleImage,att2(:,:,i),...
21              att1(:,:,i)>threshold,(scales(i)+ scales(i+1))/2);
22      end;
23  else % attenuations 1,2,3
24      att = Attenuation(amplitude,phase,scales,kind);
25      iStart = 1 + (kind==3);
26      iEnd   = d - (kind==3);
27      for i=iStart:iEnd
28          [maxAtt,scaleImage] = Update(maxAtt,scaleImage,att(:,:,i),...
29              true(size(maxAtt,1),size(maxAtt,2)),(scales(i)+ scales(i+1))/2);
30      end;
31  end;
32  end
33
34  function [maxAtt,scaleImage] =Update(maxAtt,scaleImage,curAtt,indices,scale)
35  betterAtt = indices & false;
36  betterAtt(indices) = curAtt(indices)>maxAtt(indices);
37  scaleImage(betterAtt) = scale;
38  maxAtt(betterAtt) = curAtt(betterAtt);
39  end
```

Synthetic Test Image

Listing A.6: SynthImage.m

```
1   function image=SynthImage(size,parameter,n)
2   % image=SynthImage(size,parameter,n)
3   % Author: Felix Thomsen
4   % Creates a test image, which consists of n sine waves with
5   % amplitudes,orientations,phases and scales
6   if nargin~=3
7       error('need size,parameter = [amplitudes;orientations;phases;scales] and n');
8   end
9   am = parameter(1,:);
10  or = parameter(2,:);
11  ph = parameter(3,:);
12  sc = parameter(4,:);
13  delta = pi ./ (sc .* 2);
14  image = zeros(size);
15  for i=1:1:n
16      [x,y] = meshgrid(0:delta(i):(size-1)*delta(i),0:delta(i):(size-1)*delta(i));
17      image = image + am(i) .* cos(x .* cos(or(i)) + y .* sin(or(i)) + ph(i)) + am(i);
18  end
```

A.2. Scale Space Segmentation Filter

Sample Code

Listing A.7: CreateKnowledgeExample.m

```
function [knowledge0,knowledgeE] = CreateKnowledgeExample()
% [knowledge0,knowledgeE] = CreateKnowledgeExample()
% Author: Felix Thomsen
% Example to understand the code structure

% Load training data for the spine images 1,5:
[trainData0,trainDataE] = LoadTrainDataSpine([1,5]);

% Load training scheme with preset 'fast':
trainScheme = LoadTrainScheme('fast');

% Train knowledge with 20 iterations for the spine object and the spine
% edges:
knowledge0 = TrainMultiKnowledge(trainData0,trainScheme,20);
knowledgeE = TrainMultiKnowledge(trainDataE,trainScheme,20);
```

Listing A.8: ApplyKnowledgeExample.m

```
function filteredSignal = ApplyKnowledgeExample(knowledge)
% filteredSignal = ApplyKnowledgeExample()
% Author: Felix Thomsen
% Example to understand the code structure

% Load one test image number 30:
[amplitude,phase,mask,image,scales] = LoadTestImageSpine(30);

% Apply the knowledge to the signal:
filteredSignal = ApplyMultiFilter(amplitude,phase,scales,knowledge);

% plot the result:
subplot(1,2,1);
imagesc(filteredSignal);
subplot(1,2,2);
imagesc(filteredSignal>0);
```

Create Train Data

Listing A.9: CreateTrainData.m

```
function [trainData,count] = CreateTrainData(am,ph,mask,sc,count)
% [trainData,count] = CreateTrainData(am,ph,mask,sc,count)
% Author: Felix Thomsen
% Creates train data for knowledge training, needs amplitude, phase, scales
% and mask having zeros for negative points, ones for positive points and
% any other value for points to ignore. Optional parameter count as maximal
% training set size.
% Result: count = size of positive and negative training set
% trainData.limits: maximal and minimal attenuations
% trainData.signals(pixel,scale,i)
% i odd: positive set, 1 even: negative set
% i in {1,...,6}: attenuation kind(ceil(i/2))
% i in {7,...,14}: reconstruction kind(ceil((i-6)/2))
scales = size(sc,2)-1;
maxCountPos = sum(mask(:)==1);
maxCountNeg = sum(mask(:)==0);
if nargin < 5
    count = min(maxCountPos,maxCountNeg);
else
    count = min([count,maxCountPos,maxCountNeg]);
end;
trainData.signals = zeros(count,scales,14);
trainData.limits = zeros(3,2,scales);
% take indices
[h,w] = size(mask);
indicesPos = FindIndices(find(mask==1),count,h,w);
indicesNeg = FindIndices(find(mask==0),count,h,w);
% attenuation
for at=1:3
    att = Attenuation(am,ph,sc,at);
    trainData.signals(:,:,at*2-1) = GetTrainingSignal(att,indicesPos);
    trainData.signals(:,:,at*2)   = GetTrainingSignal(att,indicesNeg);
end;
% reconstruction basis
for r=1:4
    rec = GetRecBasis(am,ph,r);
    trainData.signals(:,:,(r*2)+5) = GetTrainingSignal(rec,indicesPos);
    trainData.signals(:,:,(r*2)+6) = GetTrainingSignal(rec,indicesNeg);
end;
% limits
for at=1:3
    for s=1:scales
        trainData.limits(at,1,s) = min(min(trainData.signals(:,s,(at*2-1):(at*2))));
        trainData.limits(at,2,s) = max(max(trainData.signals(:,s,(at*2-1):(at*2))));
    end;
end;
end
```

```
48
49 function indices = FindIndices(indicesIn,count,h,w)
50 % results indices with size(indices)=(count,2)
51 % indices(:,1) = x-coordinate, indices(:,2) = y-coordinate
52 % where indicesIn is 1
53 indices = zeros(count,1);
54 factor = size(indicesIn,1)/count;
55 for i=1:count
56     indices(i) = indicesIn(round(i * factor));
57 end;
58 indices2 = zeros(h,w);
59 indices2(indices) = 1;
60 [indicesY,indicesX] = find(indices2==1);
61 indices = zeros(count,2);
62 indices(:,1) = indicesY;
63 indices(:,2) = indicesX;
64 end
65
66 function tSignal = GetTrainingSignal(signal,indices)
67 % results tSignal with size(tSignal) = (count,scales)
68 % where the points are defined by the indices
69 number = size(indices,1);
70 tSignal = zeros(number,size(signal,3));
71 for i=1:number
72     tSignal(i,:) = signal(indices(i,1),indices(i,2),:);
73 end;
74 end
```

Listing A.10: AddTrainData.m

```
1  function [trainData,count] = AddTrainData(trainDataIn,signalCount,am,ph,mask,sc)
2  % [trainData,count] = AddTrainData(trainDataIn,signalCount,am,ph,mask,sc)
3  % Author: Felix Thomsen
4  % adds train data trainDataIn for signalCount different images to new
5  % train data using am,ph,mask,sc, where each training data set has same size
6  % count/(signalCount+1)
7  countIn = size(trainDataIn.signals,1);
8  scales = size(trainDataIn.signals,2);
9  count = ceil(countIn/signalCount);
10 [trainDataNew,countNew] = CreateTrainData(am,ph,mask,sc,count);
11 if count ~= countNew
12     countIn = min(countNew*signalCount,countIn);
13     trainDataIn.signals = ReduceTrainDataSize(countIn,trainDataIn.signals);
14 end;
15 count = countIn+countNew;
16 trainData.signals = zeros(count,scales,14);
17 trainData.signals(1:countIn,:,:) = trainDataIn.signals(:,:,:);
18 trainData.signals((countIn+1):(countIn+countNew),:,:) = trainDataNew.signals;
19 trainData.limits = trainDataNew.limits;
20 for j=1:3
21     for s=1:scales
```

```
22          trainData.limits(j,1,s) = min(trainDataNew.limits(j,1,s),...
23              trainDataIn.limits(j,1,s));
24          trainData.limits(j,2,s) = max(trainDataNew.limits(j,2,s),...
25              trainDataIn.limits(j,2,s));
26      end;
27  end;
28  end
29
30  function signals = ReduceTrainDataSize(countNewIndices,signalsIn)
31  countIn = size(signalsIn,1);
32  scales = size(signalsIn,2);
33  signals = zeros(countNewIndices,scales,14);
34  factor = countIn / countNewIndices;
35  for i=1:countNewIndices
36      signals(i,:,:) = signalsIn(round(i * factor),:,:);
37  end;
38  end
```

Listing A.11: TransformMask.m

```
1  function tMask = TransformMask(mask,offsetIn,offsetOut)
2  % tMask = transMask(mask,offsetIn,offsetOut)
3  % Author: Felix Thomsen
4  % transforms mask by only maintaining positive points inside the range defined by
5  % offetIn and negative points inside the range defined by offsetOut. Each
6  % offset is defined by x-axis distance to the border between positive and
7  % negative points. The mask has to consist of one positive vertical strand.
8  % A value bigger than the maximal range of a region is cut down to the maximal
9  % valid value.
10 tMask = double(mask);
11 if (offsetIn~=0)
12     tMask(imerode(mask,strel('square',(offsetIn*2)+1))==1) = -1;
13 end;
14 if (offsetOut ~= 0)
15     tMask(imdilate(mask,strel('square',(offsetOut*2)+1))==0) = -1;
16 end;
```

Listing A.12: GetBOneScale.m

```
1  function bOneScale = GetBOneScale(at,knowledgeEntry)
2  % bOneScale = GetBOneScale(at,knowledgeEntry)
3  % Author: Felix Thomsen
4  % calculates b-Matrix for one scale interval with
5  % size(bOneScale) = size(at) and
6  % knowledgeEntry = [c,w,w_f^l,w_f^r,m]
7  if (knowledgeEntry(3)<=0) && (knowledgeEntry(4)<=0)
8      bOneScale = GetBandPass(at,knowledgeEntry);
9  else
10     bOneScale = GetBandPassFuzzy(at,knowledgeEntry);
11 end;
12 if knowledgeEntry(5)~=1
```

```
13      bOneScale = bOneScale.*knowledgeEntry(5);
14  end;
15  end
16
17  function bOneScale = GetBandPass(at,knowledgeEntry)
18  bOneScale = zeros(size(at));
19  k =[knowledgeEntry(1)-knowledgeEntry(2),knowledgeEntry(1)+knowledgeEntry(2)];
20  bOneScale((k(1)<at) & (at<k(2))) = 1;
21  end
22
23  function bOneScale = GetBandPassFuzzy(at,knowledgeEntry)
24  bOneScale = zeros(size(at));
25  k = [knowledgeEntry(1)- knowledgeEntry(2) - knowledgeEntry(3),... % pos1
26      knowledgeEntry(1) - knowledgeEntry(2) + knowledgeEntry(3),... % pos2
27      knowledgeEntry(1) + knowledgeEntry(2) - knowledgeEntry(4),... % pos3
28      knowledgeEntry(1) + knowledgeEntry(2) + knowledgeEntry(4)];   % pos4
29  bOneScale(k(1)<at & at<k(2)) = (at(k(1)<at & at<k(2)) - k(1))./(k(2)-k(1));
30  bOneScale(k(2)<=at & at<=k(3)) = 1;
31  bOneScale(k(3)<at & at<k(4)) = (k(4) - at(k(3)<at & at<k(4)))./(k(4)-k(3));
32  end
```

Listing A.13: GetRecBasis.m

```
1  function recBasis = GetRecBasis(amplitude,phase,kind)
2  % recBasis = GetRecBasis(amplitude,phase,kind)
3  % Author: Felix Thomsen
4  % Calculates reconstruction basis using amplitude and phase for
5  % reconstruction kind = kind in {1,2,3,4}
6  switch kind
7      case 1; recBasis = amplitude .* cos(phase);
8      case 2; recBasis = cos(phase);
9      case 3; recBasis = amplitude .* abs(cos(phase));
10     case 4; recBasis = abs(cos(phase));
11     otherwise;
12         error('wrong kind in GetRecBasis: kind=%i',kind);
13         recBasis = NaN;
14 end;
```

Listing A.14: LoadTestImageSpine.m

```
1  function [am,ph,mask,im,sc,maxSize] = LoadTestImageSpine(number)
2  % [am,ph,mask,im,sc] = LoadTestImageSpine(number)
3  % Author: Felix Thomsen
4  % Loads signals for the spine photographs
5  % number in {1,...,95}
6  maskX = 0;
7  maskY = 0.092;
8  depth = 15;
9  if number <10
10     imSource = ['spine/b0',int2str(number),'.tif'];
11     maSource = ['spine/m0',int2str(number),'.tif'];
```

```
12 else
13     imSource = ['spine/b',int2str(number),'.tif'];
14     maSource = ['spine/m',int2str(number),'.tif'];
15 end;
16 sprintf(imSource)
17 im = imread(imSource);
18 im = double(im(:,:,1))./255;
19 mask = imread(maSource);
20 mask = (double(mask(:,:,1))./100)>1;
21 maxSize = maskY*size(im,1)+maskX*size(im,2);
22 [am,ph,sc] = AllAS(im,depth,maxSize);
23 am(isnan(am))=0;
24 ph(isnan(ph))=0;
```

Listing A.15: LoadTrainDataSpine.m

```
1  function [trainData0,trainDataE] = LoadTrainDataSpine(numbers)
2  % [trainData0,trainDataE] = LoadTrainDataSpine(numbers)
3  % Author: Felix Thomsen
4  % generates train data for specific images and masks
5  % numbers = {number(1),...,number(n)}, number(i) in {1,...,95}
6  length = size(numbers,1)*size(numbers,2);
7  count = 140000/length;
8  [am,ph,mask,im,sc,maxSize] = LoadTestImageSpine(numbers(1));
9  mask2Number = ceil(maxSize/6);
10 trainData0 = CreateTrainData(am,ph,mask,sc,count);
11 trainDataE = CreateTrainData(am,ph,...
12     TransformMask(mask,mask2Number,mask2Number),sc,count);
13 clear am ph sc im
14 for i=2:length
15     [am,ph,mask,im,sc,maxSize] = LoadTestImageSpine(numbers(i));
16     mask2Number = ceil(maxSize/6);
17     trainData0 = AddTrainData(trainData0,i-1,am,ph,mask,sc);
18     trainDataE = AddTrainData(trainDataE,i-1,am,ph,...
19         TransformMask(mask,mask2Number,mask2Number),sc);
20     clear am ph sc im
21 end;
```

Listing A.16: LoadTestImageLiver.m

```
1  function [am,ph,mask,im,sc,maxSize] = LoadTestImageLiver(number)
2  % [am,ph,mask,im,sc] = LoadTestImageLiver(number)
3  % Author: Felix Thomsen
4  % Loads signals for the liver photographs
5  % number in {11,...,107} or {202,...,298}
6  maskX = 0;
7  maskY = 0.1;
8  depth = 15;
9  if number<100
10     imSource = ['liver/stack1/image0',int2str(number),'.tif'];
11     maSource = ['liver/stack1/m0',int2str(number),'.tif'];
```

```
elseif number<200
    imSource = ['liver/stack1/image',int2str(number),'.tif'];
    maSource = ['liver/stack1/m',int2str(number),'.tif'];
else
    imSource = ['liver/stack2/image',int2str(number),'.tif'];
    maSource = ['liver/stack2/m',int2str(number),'.tif'];
end;
sprintf(imSource)
im = imread(imSource);
im = double(im(:,:,1))./255;
mask = imread(maSource);
mask = (double(mask(:,:,1))./100)>1;
maxSize = maskY*size(im,1)+maskX*size(im,2);
[am,ph,sc] = AllAS(im,depth,maxSize);
am(isnan(am))=0;
ph(isnan(ph))=0;
```

Listing A.17: LoadTrainDataLiver.m

```
function [trainData0,trainDataE] = LoadTrainDataLiver(numbers)
% [trainData0,trainDataE] = LoadTrainDataLiver(numbers)
% author: Felix Thomsen
% generates train data for specific images and masks
% numbers = {number(1),...,number(n)}, number(i) in {11,...,107} or
% {202,...,298}
length = size(numbers,1)*size(numbers,2);
count = 100000/length;
[am,ph,mask,im,sc,maxSize] = LoadTestImageLiver(numbers(1));
mask2Number = ceil(maxSize/4);
trainData0 = CreateTrainData(am,ph,mask,sc,count);
trainDataE = CreateTrainData(am,ph,...
    TransformMask(mask,mask2Number,mask2Number),sc,count);
clear am ph sc im
for i=2:length
    [am,ph,mask,im,sc,maxSize] = LoadTestImageLiver(numbers(i));
    mask2Number = ceil(maxSize/4);
    trainData0 = AddTrainData(trainData0,i-1,am,ph,mask,sc);
    trainDataE = AddTrainData(trainDataE,i-1,am,ph,...
        TransformMask(mask,mask2Number,mask2Number),sc);
    clear am ph sc im
end;
```

Create Training Scheme

Listing A.18: LoadTrainScheme.m

```
function trainScheme = LoadTrainScheme(optimizeKind)
% trainScheme = LoadTrainScheme(optimizeKind)
% author: Felix Thomsen
% Loads data for a training scheme
% optimizationKind = ['huge'|'best'|'optimal'|'fast'|'basic'|'dirty']
if nargin==0
    optimizeKind = '';
end;
found = true;
trainScheme.maxImprovements = 2000;
trainScheme.minImprovement = 10^-5;
if strcmp(optimizeKind,'huge')
    n=10;
    kF = zeros(5,n*2);
    for i=1:n
        kF(:,i) = [(i-0.5)/n,0.5/n,0,0,-1];
        kF(:,i+n) = [(i-0.5)/n,0.5/n,0,0,1];
    end;
    parameters1 = [0,0,1,0.5];
    parameters2 = [0,0,1,5,0.5];
    trainScheme.minImprovement = 10^-6;
    trainScheme.maxImprovements = 10000;
elseif strcmp(optimizeKind,'best')
    kF = [0.5,0.35,0,0,1;0.5,0.35,0,0,-1]';
    parameters1 = [1,1,0,0.5];
    parameters2 = [1,1,0,5,0.5];
elseif strcmp(optimizeKind,'optimal')
    kF = [0.5,0.35,0,0,1;0.5,0.35,0,0,-1]';
    parameters1 = [1,1,0,0.5];
    parameters2 = [1,1,0,5,0.5];
    trainScheme.maxImprovements = 1000;
elseif strcmp(optimizeKind,'fast')
    kF = [0.5,0.35,0,0,1;0.5,0.35,0,0,-1]';
    parameters1 = [1,1,1,0.5];
    parameters2 = [1,1,1,2,0.5];
    trainScheme.maxImprovements = 500;
elseif strcmp(optimizeKind,'basic')
    kF = [0.5,0.35,0,0,1;0.5,0.35,0,0,-1]';
    parameters1 = [1,0,1,0.5];
    parameters2 = [1,0,1,5,0.5];
    trainScheme.minImprovement = 10^-4;
    trainScheme.maxImprovements = 500;
elseif strcmp(optimizeKind,'dirty')
    kF = [0.5,0.35,0,0,1;0.5,0.35,0,0,-1]';
    parameters1 = [1,0,1,0.5];
    parameters2 = [1,0,1,5,0.5];
    trainScheme.minImprovement = 10^-4;
```

```
48      trainScheme.maxImprovements = 100;
49  else fprintf('LoadTrainScheme: Undefined optimize kind - use ''fast'' instead.\n')
50      trainScheme = LoadTrainScheme('fast');
51      found = false;
52  end;
53  if found
54  trainScheme.knowledgeFunction = kF;
55  trainScheme.parameters1 = parameters1;
56  trainScheme.parameters2 = parameters2;
57  end;
```

1-Dimensional Cost Functions

Listing A.19: Evaluate.m

```
1   function [cost,cV] = Evaluate(rec,kind)
2   % [cost,compareVector] = Evaluate(rec,kind,parameters)
3   % author: Felix Thomsen
4   % calculates eval for rec and kind in [-100,100] with p=mod(kind,1) and
5   % epsilon = (kind-mod(kind,1))/100 eval_2 is evaluated if kind=0
6   % eval -> 1 => good value
7   % cost >= 0 . Be rec_1 better rec_2 -> cost(rec_2)-cost(rec_1)>=1
8   % rec(:,1) = positive points, rec(:,2) = negative points
9   % cost = 1-eval
10  % compareVector = [sep,1/delta,M(rec{pos}),M(rec{neg}), eval{pos},eval{neg}]
11  if nargin == 1
12      kind = 0;
13  end;
14  cV = zeros(6,1);
15  for i=1:2
16      cV(i+2) = mean(rec(:,i));
17  end;
18  cV(1)= mean(cV(3:4));
19  cV(5) = mean(double(rec(:,1)>cV(1)));
20  cV(6) = mean(double(rec(:,2)<=cV(1)));
21  if kind==0                          %eval_2
22      cost = 1-min(cV(5:6));
23      cV(2) = size(rec,1);
24  elseif (kind~=0) && (abs(kind)<100) %eval_1
25      p = mod(kind,1);
26      epsilon =(kind-p)/100;
27      cost = 1 - (cV(5)*p+cV(6)*(1-p)-epsilon*abs(cV(5)*p-cV(6)*(1-p)));
28      % take a mathematical false value due to performance of
29      % postprocessing, only correct for kind = 0.5
30      % right value: [m,n] = divident&Divisor(p)
31      % delta = gcd(m,n) / (n * size(rec,1));
32      cV(2) = 2 * size(rec,1);
33  else  error('Evaluate: Undefined evaluation kind: %f',kind);
34  end;
```

Listing A.20: EvaluateRec.m

```
function cost = EvaluateRec(rec,mask,kind)
% cost = EvaluateRec(rec,mask,kind)
% Author: Felix Thomsen
% Calculates cost for a reconstructed image and a mask containing 1 for
% object and 0 for background. Value 'kind' is as in Evaluate.
pos = (mask==1);
neg = (mask==0);
sep = 0.5;
evalP = sum(((rec(:)>sep)&pos(:))) / sum(pos(:));
evalN = sum(((rec(:)<=sep)&neg(:))) / sum(neg(:));
if kind==0                          %cost_{max}
    cost = 1-min(evalP,evalN);
elseif (kind~=0) && (abs(kind)<100) %cost_{p,epsilon}
    p = mod(kind,1);
    epsilon =(kind-p)/100;
    cost = 1 - (evalP*p+evalN*(1-p)-epsilon*abs(evalP*p-evalN*(1-p)));
else  error('Evaluate: Undefined evaluation kind: %f',kind);
end;
```

Train Knowledge

Listing A.21: TrainKnowledge.m

```
function [knowledge,cost,newRec] =TrainKnowledge(trainData,trainScheme,knowledge)
% [knowledge,cost,newRec] =TrainKnowledge(trainData,trainScheme,knowledge)
% author: Felix Thomsen
% computes knowledge, cost>0 and final reconstruction
% needs trainData, optional trainScheme and optional start knowledge
items = size(trainData.signals,1);
scales = size(trainData.signals,2);
signalsOpt = zeros(items,3,2);
if nargin<2
    trainScheme = LoadTrainScheme();
end;
kF = trainScheme.knowledgeFunction;
maxKnowledgeEntries = size(kF,2);
if nargin<3
    knowledgeEntriesIn = zeros(5,scales,3,4,maxKnowledgeEntries);
else
    knowledgeEntriesIn = knowledge.entries;
end;
knowledgeEntries=initialiseKnowledge(trainData.limits,kF,knowledgeEntriesIn);
for at=1:3
    for scale=1:scales
        signalsOpt(:,1,1:2) = trainData.signals(:,scale,(at*2-1):(at*2));
        signalsOpt(:,2,1:2) = trainData.signals(:,scale,(at*2+5):(at*2+6));
        for rec=1:4
            for kE=1:maxKnowledgeEntries
                curKnowledge = knowledgeEntries(:,scale,at,rec,kE);
                if curKnowledge(6) >0
                    signalsOpt = updateSignals(signalsOpt,curKnowledge);
                end;
            end;
        end;
    end;
end;
cost = 0.5;
offset=0;
improvements = 0;
unusedLoops =0;
maxLoops = scales*maxKnowledgeEntries*3*4;
at = 0;
rec = 0;
scale =0;
kE = 0;
improvement = ones(1,100);
printSteps = 100;
print = true;
% Main loop
while (improvements<trainScheme.maxImprovements)&&...
```

```
48          (unusedLoops<maxLoops) && (mean(improvement)>trainScheme.minImprovement)
49      [at,rec,scale,kE] = increaseLoop(at,rec,scale,kE,scales,maxKnowledgeEntries);
50      signalsOpt(:,1,1:2) = trainData.signals(:,scale,at*2-1:at*2);
51      signalsOpt(:,2,1:2) = trainData.signals(:,scale,rec*2+5:rec*2+6);
52      curKnowledge = knowledgeEntries(:,scale,at,rec,kE);
53      curLimits = trainData.limits(at,:,scale);
54      if curKnowledge(6)==0 % inactive
55          curSignalsOpt = signalsOpt;
56          [curKnowledge,curOffset,curCost,found] =...
57              TrainKnowledgeEntry(curKnowledge(1:5),zeros(5,1),...
58              curSignalsOpt,curLimits,trainScheme.parameters1);
59      else
60          curSignalsOpt = resetSignals(signalsOpt,curKnowledge);
61          [curKnowledge,curOffset,curCost,found] =...
62              TrainKnowledgeEntry(curKnowledge(1:5),curKnowledge(1:5),...
63              curSignalsOpt,curLimits,trainScheme.parameters2);
64      end;
65      if curCost<cost
66          unusedLoops = 0;
67          improvement(2:100) = improvement(1:99);
68          improvement(1) = cost-curCost;
69      else
70          unusedLoops = unusedLoops+1;
71      end;
72      if found && (curCost<=cost)
73          improvements = improvements + 1;
74          if print && (mod(improvements-1,printSteps)==0)
75              fprintf('%i:\tcost*100=%f,\td*10^5=%f,  \timpr*10^5=%f,\tloop=%i\n',...
76                  improvements,curCost*100,(cost-curCost)*100000,...
77                  (mean(improvement)-trainScheme.minImprovement)*100000,unusedLoops)
78          end;
79          cost = curCost;
80          offset = curOffset;
81          signalsOpt = updateSignals(curSignalsOpt,curKnowledge);
82          knowledgeEntries(1:5,scale,at,rec,kE) = curKnowledge(1:5);
83          knowledgeEntries(6,scale,at,rec,kE) = knowledgeEntries(6,scale,at,rec,kE)+1;
84      end;
85  end;
86  if print
87      if (improvements>=trainScheme.maxImprovements)
88          fprintf('Exit code=1. Maximal number of iterations reached.\n')
89          fprintf('%i:cost*100=%f',improvements,cost*100)
90      elseif (mean(improvement)<=trainScheme.minImprovement)
91          fprintf('Exit code=2. Too less improvement.\n')
92          fprintf('%i:cost*100=%f\n',improvements,cost*100)
93      else
94          fprintf('Exit code=0. Regular exit.\n')
95          fprintf('%i:cost*100=%f\n',improvements,cost*100)
96      end;
97  end;
98  newRec = zeros(items,2);
```

```
99  newRec(:,1:2) = signalsOpt(:,3,1:2);
100 newRec = newRec - offset;
101 % create final knowledge
102 knowledge.entries = zeros(5,scales,3,4,maxKnowledgeEntries);
103 for at=1:3
104     for rec=1:4
105         for scale=1:scales
106             for kE = 1: maxKnowledgeEntries
107                 k = knowledgeEntries(:,scale,at,rec,kE);
108                 % add only valid knowledge entries
109                 if (k(6)~=0)&&(k(2)>0)&&(k(5)~=0)
110                     knowledge.entries(1:5,scale,at,rec,kE) = k(1:5);
111                 end;
112             end;
113         end;
114     end;
115 end;
116 knowledge.offset = offset;
117 [knowledge,newRec] = normaliseKnowledge(knowledge,newRec);
118 end
119 %--------------------------------------------------------------------------
120 % outsourced methods for TrainKnowledge
121 %--------------------------------------------------------------------------
122 function [at,rec,scale,kE]=increaseLoop(at,rec,scale,kE,scales,maxKnowledgeEntries)
123 % Increases loop parameters. You are able to abort the loop with regular
124 % instructions and get plainer code.
125 kE = mod(kE,maxKnowledgeEntries)+1;
126 if kE==1
127     scale = mod(scale,scales)+1;
128     if scale==1
129         rec = mod(rec,4)+1;
130         if rec==1
131             at = mod(at,3)+1;
132         end;
133     end;
134 end;
135 end
136
137 function knowledgeEntries = initialiseKnowledge(limits,knowledgeFunction,...
138     knowledgeEntriesIn)
139 % maps old knowledge entries to new ones.
140 % if old knowledge ~= empty : knowledge entry = old knowledge entry
141 % otherwise: knowledgeFunction,limits -> knowledge entry
142 [foo,scales,foo2,foo3,maxKE] = size(knowledgeEntriesIn);
143 knowledgeEntries = zeros(6,scales,3,4,maxKE);
144 maxKE2 = size(knowledgeFunction,2);
145 for scale=1:scales
146     for at=1:3
147         for rec = 1:4
148             for kE=1:maxKE
149                 k = [knowledgeEntriesIn(:,scale,at,rec,kE);1];
```

```
150                 if (k(5)==0) || (k(2)==0)
151                     curLimits = limits(at,:,scale);
152                     kE2 = min(maxKE2,kE);
153                     k(1) = curLimits(1) + knowledgeFunction(1,kE2)...
154                         * (curLimits(2)-curLimits(1));
155                     k(2) = knowledgeFunction(2,kE2)...
156                         * (curLimits(2)-curLimits(1));
157                     k(3:4) = knowledgeFunction(3:4,kE2) *k(2);
158                     k(5) = knowledgeFunction(5,kE2);
159                     k(6) = 0;
160                 end;
161                 knowledgeEntries(:,scale,at,rec,kE) = k;
162             end;
163         end;
164     end;
165 end;
166 end
167
168 function signalsOpt = updateSignals(signalsOpt,knowledgeEntry)
169 % applies knowledgeEntry
170 for i=1:2
171     signalsOpt(:,3,i) = signalsOpt(:,3,i) +...
172         signalsOpt(:,2,i) .* GetBOneScale(signalsOpt(:,1,i),knowledgeEntry(1:5));
173 end;
174 end
175
176 function signalsOpt = resetSignals(signalsOpt,knowledgeEntry)
177 % subtracts knowledgeEntry
178 signalsOpt = updateSignals(signalsOpt,[knowledgeEntry(1:4);-knowledgeEntry(5)]);
179 end
180
181 function [knowledge,rec] = normaliseKnowledge(knowledge,rec)
182 % "normalises" knowledge with
183 % mean(recPos) = 1, mean(recNeg) = -1
184 scales = size(knowledge.entries,2);
185 mKE = size(knowledge.entries,5);
186 factor = 2/(mean(rec(:,1))-mean(rec(:,2)));
187 rec = factor.*rec;
188 for scale=1:scales
189     for kE=1:mKE
190         for at=1:3
191             for recKind=1:4
192                 entry = knowledge.entries(:,scale,at,recKind,kE);
193                 knowledge.entries(5,scale,at,recKind,kE)= entry(5)*factor;
194             end;
195         end;
196     end;
197 end;
198 offset = knowledge.offset;
199 knowledge.offset = offset*factor;
200 end
```

```
201 %-------------------------------------------------------------------------
202 % Train knowledge entry
203 %-------------------------------------------------------------------------
204 function [knowledgeEntry,offset,cost,found] =...
205     TrainKnowledgeEntry(knowledgeEntry,knowledgeEntryOld,signals,limits,parameter)
206 % [knowledgeEntry,offset,cost,found] =...
207 %     TrainKnowledgeEntry(knowledgeEntry,knowledgeEntryOld,signals,limits,parameter)
208 % signals(:,1,1) = atPos, signals(:,1,2) = atNeg,
209 % signals(:,2,1)= recBasisPos, signals(:,2,2)= recBasisNeg,
210 % signals(:,3,1) = recOldPos, signals(:,3,2) = recOldNeg
211 % limits(1) = minAt, limits(2) = maxAt
212 % knowledgeEntry = startKnowledge
213 % parmeter = [b,f,m,evType1,evType2]
214 % cost : cost1 + cost2/delta2
215 % knowledgeEntry(1:5) = [c',w',f_w^l',f_w^r',m']
216 % -> knowledgeNew = [c',w',f_w^l',f_w^r',m'] .* ~[b,b,f,f,m] +
217 % [c,w,f_w^l,f_w^r,m] .* [b,b,f,f,m]
218 type = parameter(1) + 2*parameter(2) + 4*parameter(3);
219 parameterSize = size(parameter,2);
220 evType = parameter(4:parameterSize);
221
222 startCost = getStartValues(signals,evType,limits,knowledgeEntryOld);
223 maxFunEvals = 20;
224 %-------------------------------------------------------------------------
225 % Optimization
226 %-------------------------------------------------------------------------
227 switch type
228     case 1; % 100 bandpass
229         value = fminsearch(@(minimiser)...
230             costFunction(signals,evType,startCost,limits,...
231             [minimiser(1:2);knowledgeEntry(3:5)]),knowledgeEntry(1:2),...
232             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
233         knowledgeEntry(1:2) = value(1:2);
234     case 2; % 010 fuzzy
235         value = fminsearch(@(minimiser)costFunction(signals,evType,...
236             startCost,limits,[knowledgeEntry(1:2);...
237             minimiser;knowledgeEntry(5)]),knowledgeEntry(3:4),...
238             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
239         knowledgeEntry(3:4) = value(1:2);
240     case 3; % 110 bandpass, fuzzy
241         value = fminsearch(@(minimiser)...
242             costFunction(signals,evType,startCost,limits,...
243             [minimiser(1:4);knowledgeEntry(5)]),knowledgeEntry(1:4),...
244             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
245         knowledgeEntry(1:4) = value(1:4);
246     case 4; % 001 maxValue
247         value = fminsearch(@(minimiser)...
248             costFunction(signals,evType,startCost,limits,...
249             [knowledgeEntry(1:4);minimiser]),knowledgeEntry(5),...
250             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
251         knowledgeEntry(5) = value;
```

```
252     case 5; % 101 bandpass, maxValue
253         value = fminsearch(@(minimiser)...
254             costFunction(signals,evType,startCost,limits,...
255             [minimiser(1:2);knowledgeEntry(3:4);...
256             minimiser(3)]),[knowledgeEntry(1:2);knowledgeEntry(5)],...
257             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
258         knowledgeEntry(1:2) = value(1:2);
259         knowledgeEntry(5) = value(3);
260     case 6; % 011 fuzzy, maxValue
261         value = fminsearch(@(minimiser)...
262             costFunction(signals,evType,startCost,limits,...
263             [knowledgeEntry(1:2);minimiser]),knowledgeEntry(3:5),...
264             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
265         knowledgeEntry(3:5) = value(1:3);
266     case 7; % 111 bandpass, fuzzy, maxValue
267         value = fminsearch(@(minimiser)...
268             costFunction(signals,evType,startCost,limits,...
269             minimiser),knowledgeEntry,...
270             optimset('MaxFunEvals',maxFunEvals,'Display','off'));
271         knowledgeEntry = value;
272     otherwise; % no change
273 end;
274 knowledgeEntry = map2validKnowledge(knowledgeEntry,limits);
275 ev = costFunction(signals,evType,startCost,limits,knowledgeEntry);
276 [cost,compareVector] = CallEvaluate(signals,knowledgeEntry,evType(parameterSize-3));
277 offset = compareVector(1);
278 found = (ev<=0);
279 end
280 %--------------------------------------------------------------------------
281 % knowledgeEntry design constraints
282 %--------------------------------------------------------------------------
283 function knowledgeEntry = map2validKnowledge(knowledgeEntry,limits)
284 % Computes valid knowledgeEntry using knowledgeEntry and limits
285 % c,m in R, w,w_f^l,w_f^r in R>=0
286 % w_f^l<w, w_f^r<w
287 height = abs(knowledgeEntry(5));
288 maxHeight = 10;
289 minHeight = 0.01;
290 if height ~=0
291     height = max(min(height,maxHeight),minHeight);
292 end;
293 knowledgeEntry(5) = sign(knowledgeEntry(5)) * height;
294 knowledgeEntry(2:4) = abs(knowledgeEntry(2:4));
295 knowledgeEntry(3) = min(knowledgeEntry(2),knowledgeEntry(3));
296 knowledgeEntry(4) = min(knowledgeEntry(2),knowledgeEntry(4));
297 lower =  knowledgeEntry(1)-knowledgeEntry(2)-knowledgeEntry(3);
298 higher = knowledgeEntry(1)+knowledgeEntry(2)+knowledgeEntry(4);
299 if (lower>limits(2))||(higher<limits(1))
300     knowledgeEntry(1) = (limits(1)+limits(2))/2;
301 else
302     lower = knowledgeEntry(1)-knowledgeEntry(2)+knowledgeEntry(3);
```

```
303      higher = knowledgeEntry(1)+knowledgeEntry(2)-knowledgeEntry(4);
304      if lower < limits(1)
305          knowledgeEntry(3) = 0;
306      end;
307      if higher > limits(2)
308          knowledgeEntry(4) = 0;
309      end;
310      lower = knowledgeEntry(1)-knowledgeEntry(2)+knowledgeEntry(3);
311      if lower < limits(1)
312          center=(knowledgeEntry(1)+knowledgeEntry(2)-knowledgeEntry(3)+limits(1))/2;
313          width=(knowledgeEntry(1)+knowledgeEntry(2)+knowledgeEntry(3)-limits(1))/2;
314          knowledgeEntry(1) = center;
315          knowledgeEntry(2) = width;
316      end;
317      higher = knowledgeEntry(1)+knowledgeEntry(2)-knowledgeEntry(4);
318      if higher > limits(2)
319          center=(knowledgeEntry(1)-knowledgeEntry(2)+knowledgeEntry(4)+limits(2))/2;
320          width=(-knowledgeEntry(1)+knowledgeEntry(2)+knowledgeEntry(4)+limits(2))/2;
321          knowledgeEntry(1) = center;
322          knowledgeEntry(2) = width;
323      end;
324  end;
325  knowledgeEntry(2:4) = abs(knowledgeEntry(2:4));
326  end
327  %-------------------------------------------------------------------------
328  % cost functions for trainKnowledgeEntry
329  %-------------------------------------------------------------------------
330  function [cost,deltaInv]=EvaluateCVOrKE(compareVector,knowledgeEntry,limits,evType)
331  % computes evaluation based only on knowledgeEntry design and limits for
332  % evType in {1=minimise,2=maximise}
333  % computes evaluation based only on compareVector for
334  % evType in {3=minimise,4=maximise,5=near 2} distance between positive
335  % and negative points
336  % supplies empty evaluation for evType =0
337  maxHeight= 10;
338  maxWidth = limits(2)-limits(1);
339  distance = abs(compareVector(4)-compareVector(3));
340  deltaInv = 10000;
341  switch evType
342      case 0; % no action
343          cost = 0;
344          deltaInv = 1;
345      case 1; % minimise
346          cost = abs(knowledgeEntry(5)*knowledgeEntry(2))/ (maxHeight*maxWidth);
347      case 2; % maximise
348          cost = abs(knowledgeEntry(5)*knowledgeEntry(2))/-(maxHeight*maxWidth)+1;
349      case 3; %minimise
350          cost = abs(1/exp(distance)-1);
351      case 4; %maximise
352          cost = 1/exp(distance);
353      case 5; % distance near 2
```

```
354          cost = abs(1/exp(abs(distance-2))-1);
355      otherwise;
356          error('EvaluateCVOrKE: Undefined evType: %f',evType);
357  end;
358  % discretise cost:
359  cost = (floor(cost*deltaInv))/deltaInv;
360  end
361
362  function [cost,compareVector] = CallEvaluate(signals,knowledgeEntry,evType)
363  % calls Evaluate by firstly computing all relevant parameters
364  rec = zeros(size(signals,1),2);
365  for i=1:2
366      rec(:,i) = signals(:,3,i) +...
367          signals(:,2,i).*GetBOneScale(signals(:,1,i),knowledgeEntry(1:5));
368  end;
369  [cost,compareVector] = Evaluate(rec,evType);
370  end
371
372  function startCost = getStartValues(signals,evType,limits,knowledgeEntry)
373  % computes start costs to optimize from
374  knowledgeEntry = map2validKnowledge(knowledgeEntry,limits);
375  n = size(evType,2);
376  startCost = zeros(n,1);
377  [startCost(n),compareVector] = CallEvaluate(signals,knowledgeEntry,evType(n));
378  for i=1:n-1
379      startCost(i) = EvaluateCVOrKE(compareVector,knowledgeEntry,limits,evType(i));
380  end;
381  end
382
383  function [cost,costValues]=costFunction(signals,evType,startCost,limits,...
384      knowledgeEntry)
385  % n-dimensional cost function:
386  % cost = cost1-startCost1 + deltaInv2(cost2-startCost2 +
387  % deltaInv3(...(costn-startCostn)...))
388  % cost <0 -> better value, cost>0 worse value
389  % most right evType stands for evaluation in CallEvaluate
390  % all others for EvaluateCVOrKE
391  knowledgeEntry = map2validKnowledge(knowledgeEntry,limits);
392  n = size(evType,2);
393  costValues = zeros(n,1);
394  [costValues(n),compareVector] = CallEvaluate(signals,knowledgeEntry,evType(n));
395  cost = costValues(n)-startCost(n);
396  deltaInv = compareVector(2);
397  for i=n-1:-1:1
398      cost = deltaInv * cost;
399      [costValues(i),deltaInv] =...
400          EvaluateCVOrKE(compareVector,knowledgeEntry,limits,evType(i));
401      cost = cost + costValues(i) - startCost(i);
402  end;
403  end
```

Train Multi-Knowledge

Listing A.22: TrainMultiKnowledge.m

```
function [multiKnowledge,costs,binaryRec]=TrainMultiKnowledge(trainData,...
    trainScheme,maxIterations)
% [multiKnowledge,costs,binaryRec] =
% TrainMultiKnowledge(trainData,trainScheme,maxIterations)
% author: Felix Thomsen
% computes set of knowledge sets: multiKnowledge, which has better issues
% concerning separation as only one knowledge set.
% uses trainData, optional trainScheme and maxIterations.
if nargin<2
    trainScheme = LoadTrainScheme();
end;
if nargin<3
    maxIterations = 100;
end;
items = size(trainData.signals,1);
scales = size(trainData.signals,2);
newItems = ceil(items/3);
curCosts = ones(maxIterations,1);
% first knowledge
[curMultiKnowledge(maxIterations),curCosts(1),curRec] =...
    TrainKnowledge(trainData,trainScheme);
curMultiKnowledge(1) = curMultiKnowledge(maxIterations);
binaryRecPos = [(curRec(:,1)>0)-0.5,(1:items)'];
binaryRecNeg = [(curRec(:,2)<=0)-0.5,(1:items)'];
% optimisation - loop
for iteration=2:maxIterations
    sortedPos = sortrows(binaryRecPos,1);
    sortedNeg = sortrows(binaryRecNeg,1);

    % take only these indices which have bad segmentation issues.
    curNewItems = min(newItems,max(ceil(sum(binaryRecPos(:,1)<=1)*1.5),...
        ceil(sum(binaryRecNeg(:,1)<=1)*1.5)));
    %take indices from sorted Vectors
    curSignals = zeros(curNewItems,scales,14);
    for i=1:curNewItems
        for j=1:7
            curSignals(i,:,j*2-1)=trainData.signals(sortedPos(i,2),:,j*2-1);
            curSignals(i,:,j*2)=trainData.signals(sortedNeg(i,2),:,j*2);
        end;
    end;
    limits = trainData.limits;
    curTrainData.signals = curSignals;
    curTrainData.limits = limits;
    curMultiKnowledge(iteration) = TrainKnowledge(curTrainData,trainScheme);
    curRec = ApplyFilter2(trainData,curMultiKnowledge(iteration));
    binaryRecPos(:,1) = binaryRecPos(:,1) + (curRec(:,1)>0)-0.5;
    binaryRecNeg(:,1) = binaryRecNeg(:,1) + (curRec(:,2)<=0)-0.5;
```

```
48      binaryRec = [binaryRecPos(:,1),-binaryRecNeg(:,1)];
49      curCosts(iteration) = Evaluate(binaryRec,0.5);
50      fprintf('TrainMultiKnowledge:%i:training set size=%i',iteration,curNewItems)
51      fprintf(',\t current cost*100=%f\n',curCosts(iteration)*100)
52  end;
53  iteration = find(curCosts==min(curCosts));
54  costs = ones(iteration,1);
55  costs(1:iteration) = curCosts(1:iteration);
56  multiKnowledge(iteration) = curMultiKnowledge(iteration);
57  for i=1:iteration-1
58      multiKnowledge(i) = curMultiKnowledge(i);
59  end;
60  end
61
62  function rec = ApplyFilter2(trainData,knowledge)
63  % rec = ApplyFilter2(trainData,knowledge)
64  % author: Felix Thomsen
65  % applies the knowledge set on trainData
66  % very similar to ApplyFilter(am,ph,sc,knowledge)
67  % rec(:,1) = positive points, rec(:,2) = negative points
68  items = size(trainData.signals,1);
69  scales = size(trainData.signals,2);
70  maxKnowledgeEntries = size(knowledge.entries,5);
71  rec = zeros(items,2);
72  recBasis = zeros(items,2,scales);
73  att = zeros(items,2,scales);
74  for recKind=1:4
75      recBasis(:,1,:) = trainData.signals(:,:,(recKind*2)+5);
76      recBasis(:,2,:) = trainData.signals(:,:,(recKind*2)+6);
77      for at=1:3
78          att(:,1,:) = trainData.signals(:,:,at*2-1);
79          att(:,2,:) = trainData.signals(:,:,at*2);
80          for kE=1:maxKnowledgeEntries
81              for scale=1:scales
82                  knowledgeEntry = knowledge.entries(:,scale,at,recKind,kE);
83                  if (knowledgeEntry(5)~=0) && (knowledgeEntry(2)>0)
84                      rec = rec+GetBOneScale(att(:,:,scale),knowledgeEntry)...
85                          .*recBasis(:,:,scale);
86                  end;
87              end;
88          end;
89      end;
90  end;
91  rec = rec-knowledge.offset;
92  end
```

Apply Scale Space Segmentation Filter

Listing A.23: ApplyFilter.m

```
function rec = ApplyFilter(am,ph,sc,knowledge)
% rec = ApplyFilter(am,ph,sc,knowledge)
% author: Felix Thomsen
% applies the knowledge set on amplitudes am, phases ph, scales sc
[h,w,scales] = size(am);
rec = zeros(h,w);
maxKnowledgeEntries =size(knowledge.entries,5);
attenuations = zeros(h,w,scales,3);
for kind=1:3
    attenuations(:,:,:,kind) = Attenuation(am,ph,sc,kind);
end;
for recKind=1:4
    recBasis = GetRecBasis(am,ph,recKind);
    for at=1:3
        att = attenuations(:,:,:,at);
        for kE=1:maxKnowledgeEntries
            for scale=1:scales
                % extract one knowledge entry
                knowledgeEntry = knowledge.entries(:,scale,at,recKind,kE);
                % if knowledge entry contains any information
                if (knowledgeEntry(5)~=0) && (knowledgeEntry(2)>0)
                    b = GetBOneScale(att(:,:,scale),knowledgeEntry);
                    rec = rec + b.*recBasis(:,:,scale);
                end;
            end;
        end;
    end;
end;
rec = rec-knowledge.offset;
```

Listing A.24: ApplyMultiFilter.m

```
function [binaryRec,rec] = ApplyMultiFilter(am,ph,sc,multiKnowledge)
% [rec,binaryRec] = ApplyMultiFilter(am,ph,sc,multiKnowledge)
% author: Felix Thomsen
% applies several filters on signals am,ph and sc
% rec = sum(rec_1,...,rec_n), binaryRec=sum((rec_1>0)-0.5,...,(rec_n>0)-0.5)
knowledgeDepth = size(multiKnowledge,1)*size(multiKnowledge,2);
[h,w,scales] = size(am);
rec = zeros(h,w);
binaryRec = zeros(h,w);
attenuations = zeros(h,w,scales,3);
recBasises = zeros(h,w,scales,4);
for at = 1:3
    attenuations(:,:,:,at) = Attenuation(am,ph,sc,at);
end;
for rec = 1:4
    recBasises(:,:,:,rec) = GetRecBasis(am,ph,rec);
end;
for depth = 1:knowledgeDepth
    fprintf('F:%i\t',depth)
    if mod(depth,10)==0
        fprintf('\n')
    end;
    maxKnowledgeEntries = size(multiKnowledge(depth).entries,5);
    curRec = zeros(h,w);
    for rec=1:4
        recBasis = recBasises(:,:,:,rec);
        for at=1:3
            att = attenuations(:,:,:,at);
            for kE=1:maxKnowledgeEntries
                for scale=1:scales
                    knowledgeEntry = multiKnowledge(depth).entries...
                        (:,scale,at,rec,kE);
                    if (knowledgeEntry(5)~=0) && (knowledgeEntry(2)>0)
                        b = GetBOneScale(att(:,:,scale),knowledgeEntry);
                        curRec = curRec + b.*recBasis(:,:,scale);
                    end;
                end;
            end;
        end;
    end;
    curRec = curRec - multiKnowledge(depth).offset;
    rec = rec + curRec;
    binaryRec = binaryRec + (curRec>0)-0.5;
end;
fprintf('\n')
```

A.3. Spine - Detection

Sample Code

Listing A.25: SpineDetectionExample.m

```
function [spineCentre,spine] = SpineDetectionExample(knowledge0,knowledgeE)
% spineCentre = SpineDetectionExample(knowledge0,knowledgeE)
% Author: Felix Thomsen
% Example to understand the code structure

% Load one test image number 30:
[amplitude,phase,mask,image,scales] = LoadTestImageSpine(30);

% Apply the knowledge sets to the signal:
rec0 = ApplyMultiFilter(amplitude,phase,scales,knowledge0);
recE = ApplyMultiFilter(amplitude,phase,scales,knowledgeE);

% Take first reduction:
region = GetRegion(rec0);

% Take further reduction:
closerRegion = GetCloserRegion(recE,region);

% Take final segmentation:
[spine,spineCentre] = FinalSegmentation(closerRegion,recE);

% plot the result:
[he,we] = size(mask);
im2 = zeros(he,we,3);
im2(:,:,1) = im .*(TransformMask(mask,3,3)==-1)+(TransformMask(mask,3,3)~=-1);
im2(:,:,2) = im .*(TransformMask(mask,3,3)==-1);
im2(:,:,3) = im .*(TransformMask(mask,3,3)==-1);

im3 = double(spine) ./2 + double(mask)./4;
im3(imdilate(spineCentre,strel('square',5))==1) = 1;
subplot(1,2,1);
imshow(im2);
subplot(1,2,2);
imshow(im3);
colormap(jet);
```

Helper functions

Listing A.26: FindBlobs.m

```
function [binImage,maxTallness] = FindBlobs(binImage)
% [binImage,maxTallness] = FindBlobs(binImage)
% Author: Felix Thomsen
% Searches connected objects with the mehtod 'Tallness'
[h,w] = size(binImage);
maxTallness = 0;
for yy=1:h
    for xx=1:w
        if binImage(yy,xx) == 1
            [tallness,binImage] = Tallness(binImage,yy,xx,'links');
            maxTallness = max(tallness,maxTallness);
        end;
    end;
end;
binImage = -binImage;
end

function [tallness,signal] = Tallness(signal,posy,posx,dir)
% [tallness,signal] = Tallness(signal,posy,posx,dir)
% Author: Felix Thomsen
% searches connected objects with Tschebycheff neighbourhood at position
% posy,posx in the binary image, directed in 'links'|'rechts'.
[h,w] = size(signal);
signal = double(signal);
% Freeman coded direction : left up = 0, up = 1 etc.
if strcmp(dir,'links')
    direction = 6; % left down
else % 'rechts' = right
    direction = 2; % right up;
end;
dirLeft = ((direction==6)*2)-1; % at 'links'->1, at 'rechts'->-1
lookUpDir = [-1 0 1 1 1 0 -1 -1;
             -1 -1 -1 0 1 1 1 0];
startpos = [posy,posx];
pos = startpos;
savePos = pos;
% Wize of the window:
deepIndices = [pos(1),pos(1),pos(2),pos(2)];
% mark start position
signal(pos(1),pos(2)) = signal(pos(1),pos(2)) ...
    + (pos(2) == 1 || signal(pos(1),pos(2)-1)==0)...
    + (pos(2) == w || signal(pos(1),pos(2)+1)==0)*2;
pos(:) = savePos(:) + lookUpDir(:,direction+1);
looked = 0;
toLookRightStartPos = true;
% Run around the object at the borders
while (pos(1)~=startpos(1) || pos(2)~=startpos(2)|| toLookRightStartPos )&& looked<8
```

```
48      if pos(1) == startpos(1) && (pos(2)==startpos(2)+dirLeft)
49          toLookRightStartPos = false;
50      end;
51      if pos(1)>=1 && pos(2)>=1 && pos(1)<=h && pos(2)<=w && signal(pos(1),pos(2))>0
52          deepIndices = [min(deepIndices(1),pos(1)),max(deepIndices(2),pos(1)),...
53              min(deepIndices(3),pos(2)),max(deepIndices(4),pos(2))];
54          if signal(pos(1),pos(2))==1
55              signal(pos(1),pos(2)) = signal(pos(1),pos(2)) ...
56              + (pos(2) == 1 || signal(pos(1),pos(2)-1)==0)...
57              + (pos(2) == w || signal(pos(1),pos(2)+1)==0)*2;
58          end;
59          savePos = pos;
60          direction = mod(direction+2,8);
61          looked = 0;
62      else
63          direction = mod(direction-1,8);
64          looked = looked + 1;
65      end;
66      pos(:) = savePos(:) + lookUpDir(:,direction+1);
67  end;
68  % add elements
69  tallness = 0;
70  open = false;
71  s2 = signal(deepIndices(1):deepIndices(2),deepIndices(3):deepIndices(4));
72  for yy = 1:deepIndices(2)-deepIndices(1)+1
73      for xx = 1:deepIndices(4)-deepIndices(3)+1
74         if s2(yy,xx)>1
75             open = (s2(yy,xx)==2);
76         end;
77         if (s2(yy,xx)==1 && open)||s2(yy,xx)>1
78             tallness = tallness+1;
79             s2(yy,xx) = NaN;
80         end;
81      end;
82  end;
83  s2(isnan(s2)) = -tallness;
84  signal(deepIndices(1):deepIndices(2),deepIndices(3):deepIndices(4)) = s2(:,:);
85  end
```

Listing A.27: BorderTransform.m

```
1   function positions = BorderTransform(image,dir)
2   % positions = BorderTransform(image,dir)
3   % Author: Felix Thomsen
4   % extracts the borders, hence vertical edges in binary signal image.
5   [h,w] = size(image);
6   image = double(image);
7   positions = nan(h,w);
8   maxPos = 0;
9   if nargin==1
10      dir = 1;
```

```
end;
if dir==1
    sW= 2;
    eW= w;
    dW =1;
else
    sW = w-1;
    eW = 1;
    dW = -1;
end;
for hh = 1:h
    c = 1;
    pos = 0;
    for ww=sW:dW:eW
        if (c == 0)
            if image(hh,ww)==1
                pos = pos+1;
                maxPos = max(maxPos,pos);
                positions(hh,pos) = ww;
                c = 1;
            end;
        else % c=1
            c = image(hh,ww);
        end;
    end;
end;
positions = positions(:,1:maxPos);
```

Listing A.28: HeightTransform.m

```
function image = HeightTransform(image)
% image = HeightTransform(image)
% Author: Felix Thomsen
% maps the grey-values of the pixels in one row to the order in [0,1].
% Hence the highest pixel in one row becomes 1 the lowest becomes 0, etc.
image=transform(transform(image));
end

function trans = transform(image)
[h,w] = size(image);
%find n heighest pixels per row:
trans = zeros(h,w);
line = zeros(w,2);
for hh=1:h
    line(:,1) = image(hh,:);
    line(:,2) = (0:w-1)./(w-1);
    ss = sortrows(line,1);
    trans(hh,:) = ss(:,2);
end;
end
```

Spine-Segmentation Functions

Listing A.29: GetRegion.m

```
function region = GetRegion(rec0)
% region = GetRegion(rec0)
% Author: Felix Thomsen
% tries to drop about the half of the background pixels,
% which do not contain the object
h=size(rec0,1);
atomic = h/240;
s1 = imopen(HeightTransform(rec0)>0.8,strel('square',2));
[or,ph] = AS(s1,atomic*8,atomic*16,25);
phase = HeightTransform(cos(ph));
line = (phase>0.85);
[binImage,t] = FindBlobs(line);
line = (binImage==t);

posR = BorderTransform(line,2);
posL = BorderTransform(line,1);
centres = (posR(:,1)+posL(:,1))./2;
line(:,:) = 0;
for hh=1:h
    if ~isnan(centres(hh))
    line(hh,ceil(centres(hh))) = 1;
    end;
end;
region = imdilate(line,strel('square',ceil(atomic*40)));
```

Listing A.30: GetCloserRegion.m

```
function closerRegion = GetCloserRegion(recE,region)
% closerRegion = GetCloserRegion(recE,region)
% Author: Felix Thomsen
% excludes some more background
h  =size(recE,1);
length = h/36;
edge = recE.*region;
s1 = imopen(HeightTransform(edge)>0.85,strel('square',2)).*region;
sprintf('AS1')
[or,ph] = AS(s1,length,length*2,25);
phase = cos(ph);
phase(region==0) = -2;
phase = HeightTransform(phase);
line = (phase>0.9);

[binImage,t] = FindBlobs(line);
line = (binImage==t);

region2 = imdilate(line,strel('square',ceil(h/12)));

```

```
l2 = ceil(h/500);
edge2 = recE.*region2;
s1 = imopen(HeightTransform(edge2)>0.85,strel('square',2)).*region2;

s2 = imdilate(imerode(s1,strel('square',l2)),strel('square',l2*3));
sprintf('conv')
s3 = conv2(s2,ones(ceil(length/2))./(length/2)^2,'same');
closerRegion  = (s3>0.2);
```

Listing A.31: FinalSegmentation.m

```
function [spine,centre] = FinalSegmentation(closerRegion,recE)
% [spine,centre] = FinalSegmentaton(closerRegion,recE)
% Author: Felix Thomsen
% computes the final spine-centre
% and a region 'spine', which is used for the computation of the
% false positive and false negative error
[h,w] = size(recE);
atomic = h/240;

signal = closerRegion .* recE;
s1 = conv2(signal,ones(ceil(18*atomic),ceil(18*atomic))./(ceil(648*atomic)),'same');
s2 = HeightTransform(s1).*closerRegion;
line = nan(h,2);
delta = 1.2/w;
for hh=1:h
    if (hh>1)
        lastValue = line(hh-1);
        weight = zeros(1,w);
        for i=1:w
            weight(i) = 1-abs(lastValue-i)*delta;
        end;
        values = s2(hh,:).*weight;
    else
        values = s2(hh,:);
    end;
    f = find(values==max(values),1,'first');
    line(hh)=f(1);
end;

s1 = zeros(h,w);
for hh=1:h
    s1(hh,ceil(line(hh))) = 1;
end;

closestRegion = imdilate(s1,strel('square',ceil(atomic*18)));
signal2 = closestRegion .* signal;
s3 = imclose(signal2,strel('square',ceil(atomic*6)));
spine = (s3>0).*closestRegion;

meanSize = atomic * 12;
```

```
41  s5 = nan(h,1);
42  centre = zeros(h,w);
43  l2 = line;
44  l2(l2<=1) = nan;
45  for hh=1:h
46      s5(hh) = mean(l2(max(1,ceil(hh-meanSize)):min(h,ceil(hh+meanSize))));
47      if (s5(hh)>0)
48          centre(hh,ceil(s5(hh))) = 1;
49      end;
50  end;
```

A.4. Samples

In the first three figures we give the output images of the implemented examples. The console inputs are the following listings:

Listing A.32: Main-scale detection

```
1  >> scaleImage = MainScaleExample();
```

Listing A.33: Knowledge

```
1  >> [knowledge0,knowledgeE] = CreateKnowledgeExample();
2  >> filteredSignal = ApplyKnowledgeExample(knowledge0);
```

Listing A.34: Spine detection

```
1  >> [knowledge0,knowledgeE] = CreateKnowledgeExample();
2  >> [spineCentre,spine] = SpineDetectionExample(knowledge0,knowledgeE);
```

In Figure A.4 we give a sample band pass.

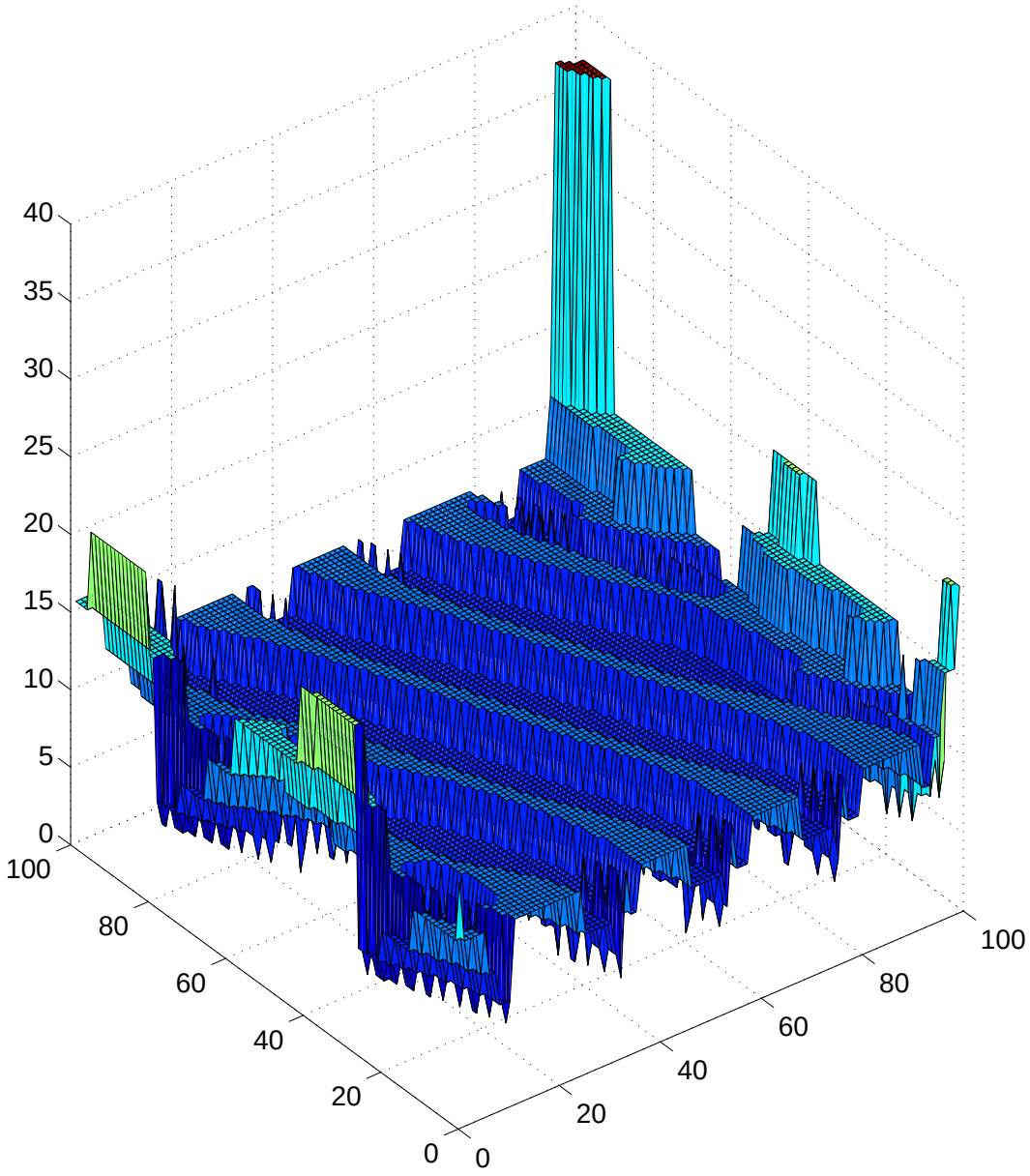


Figure A.1.: Output of listing A.32

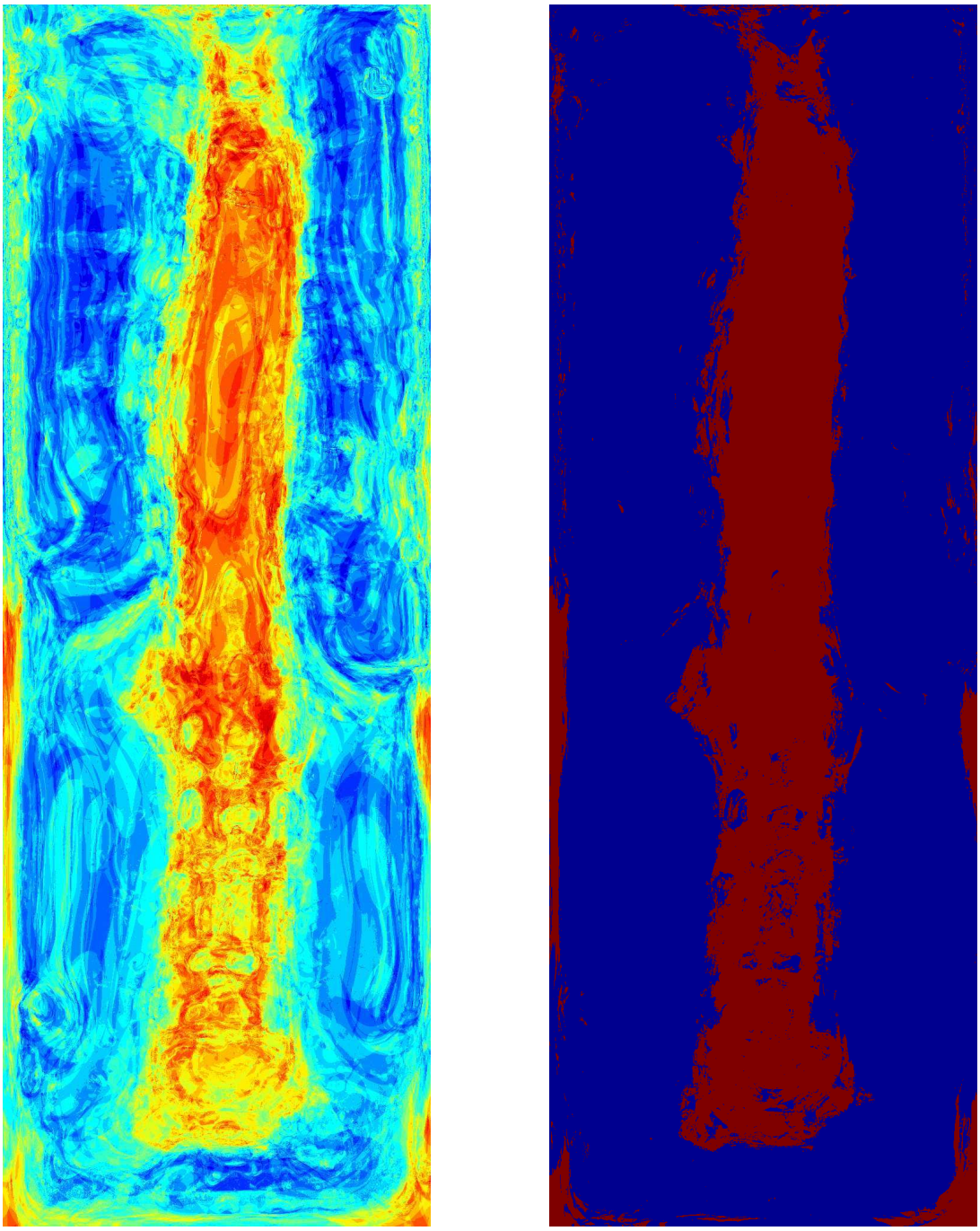


Figure A.2.: Output of listing A.33

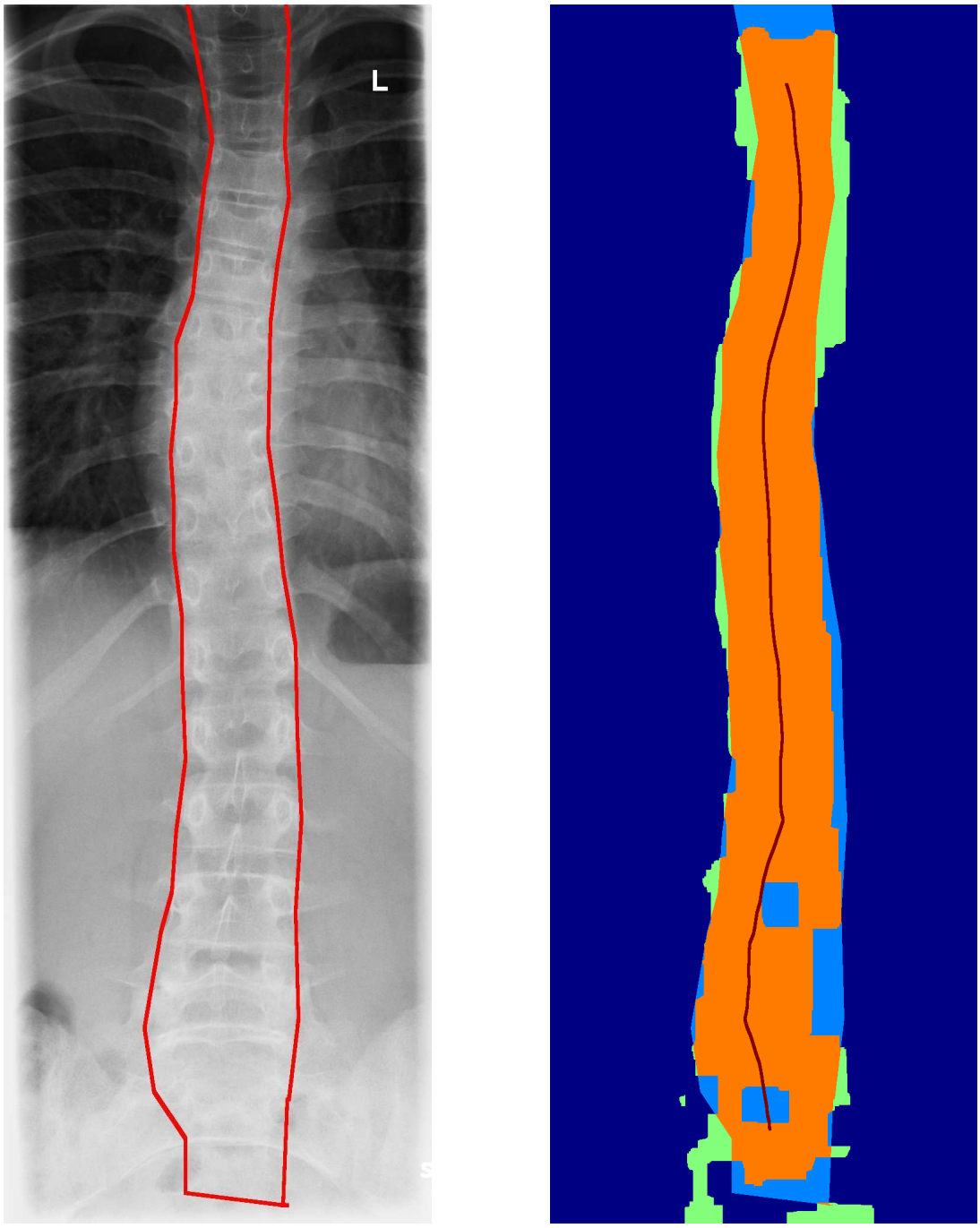


Figure A.3.: Output of listing A.34

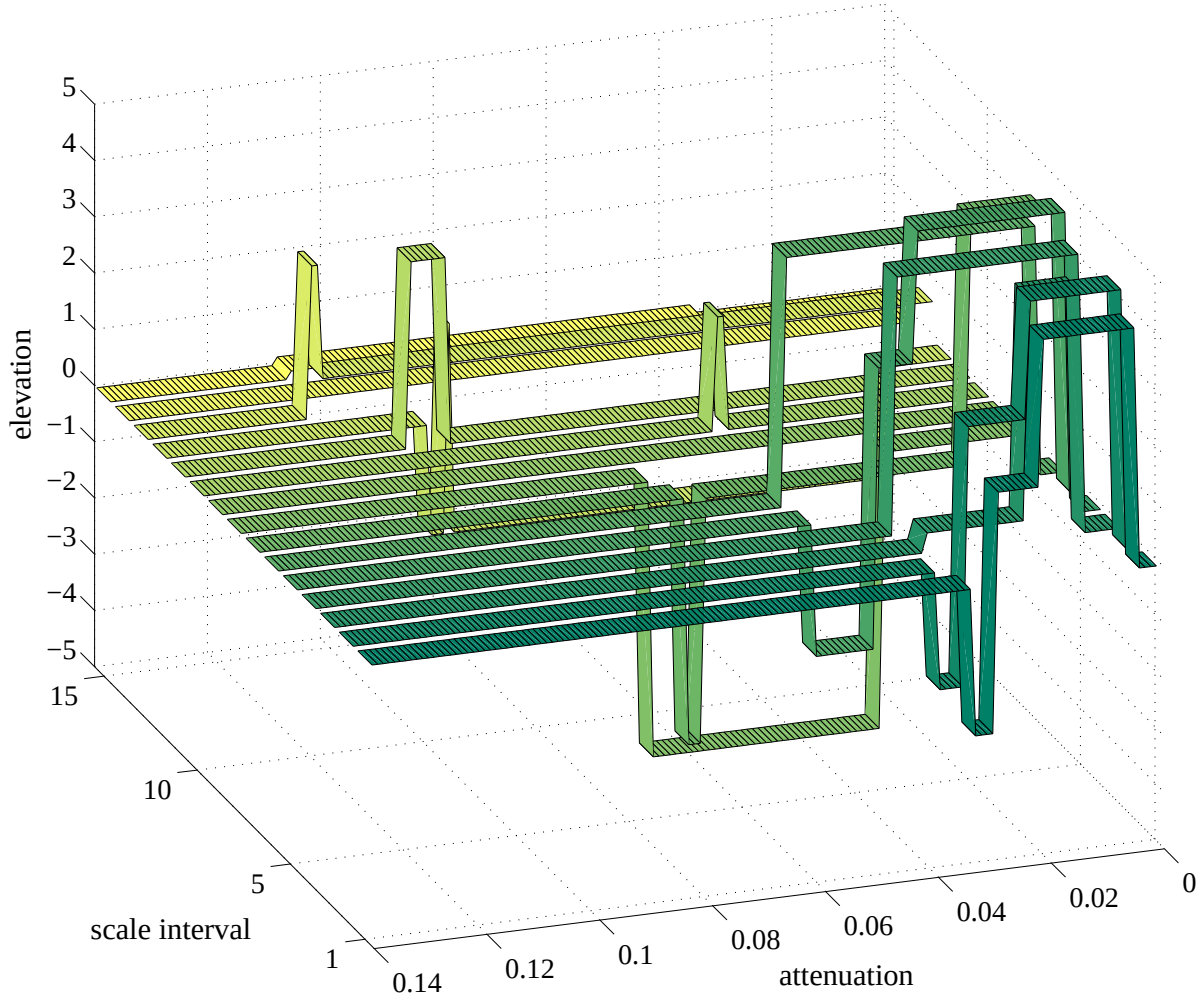


Figure A.4.: Band passes for the first attenuation type and the third reconstruction type of the spine border detection.

Appendix B.

GPU

B.1. Specifications of our GPU

The specifications of our GPU are as follows:

```
Vendor: NVIDIA Corporation
Version: 3.0.0
Renderer: GeForce 9600 GS/PCI/SSE2

GL_MAX_TEXTURE_UNITS: 4
GL_MAX_VERTEX_ATTRIBS_ARB: 16
GL_MAX_VERTEX_UNIFORM_COMPONENTS_ARB: 4096
GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS_ARB: 32
GL_MAX_VARYING_FLOATS_ARB: 60
GL_MAX_TEXTURE_IMAGE_UNITS_ARB: 32
GL_MAX_TEXTURE_COORDS_ARB: 8
GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB: 32
GL_MAX_FRAGMENT_UNIFORM_COMPONENTS_ARB: 2048
```

with

```
Stream-Processors: 48
Kernel-Clock: 500 MHz
Shader-Clock: 1200 MHz
Storage-Clock: 500 MHz
```

B.2. GPU-Code

We list the source code of the vertex shader we used in Figure 2.13.

Listing B.1: $place_{min}$

```
1  // tensorBig.frag - as place_min
2
3  // CONSTANTS:
4   const int MAX_HALF_N = 16;
5   const float PI = 3.14159265358979323846;
6   const float TWOPI = PI * 2.0;
7
8  // VARIABLES:
9      uniform sampler2D texture;
10
11  uniform vec4 kernelPoisson[MAX_HALF_N *(MAX_HALF_N-1)/2];
12  uniform vec4 kernel1[MAX_HALF_N * (MAX_HALF_N-1) / 2];
13  uniform vec4 kernel2[MAX_HALF_N * (MAX_HALF_N-1) / 2];
14
15  uniform vec4 diagPoisson[MAX_HALF_N];
16  uniform vec4 diag1[MAX_HALF_N];
17  uniform vec4 diag2[MAX_HALF_N];
18
19  uniform vec2 randPoisson[MAX_HALF_N];
20  uniform vec2 rand1[MAX_HALF_N];
21  uniform vec2 rand2[MAX_HALF_N];
22
23  uniform float middlePoisson;
24  uniform int halfN;
25  uniform float n;
26  uniform vec2 offsetFactor;
27  uniform float amplitudeScale;
28  float f_x,f_y,f_p,f_xx,f_yy,f_xy;
29
30 // FUNCTION-DECLARATIONS:
31  vec4 AnalytikSignal();
32  void Convolution(vec2 coord);
33
34 // PROGRAMME:
35 //-------------------------------------------------------------
36 //----------Main - Function-----------------------------------
37 //-------------------------------------------------------------
38 // The main function calls AnalyticSignal()
39 void main()
40 {
41  gl_FragColor = AnalytikSignal();
42 }
43 //-------------------------------------------------------------
44 //----------Signal - Calculation-----------------------------
45 //-------------------------------------------------------------
```

```
46 // Analytic Signal computes the vector:
47 // vec4(phase,orientation,amplitude,apexAngle).
48  vec4 AnalytikSignal()
49 {
50  // Texture coordinates:
51  vec2 coord= gl_TexCoord[0].xy;
52  vec4 ret;
53  // Signal computing excepts the boundary
54  if((coord.x<((n+1.0)*offsetFactor.x))||(coord.x>(1.0-n* offsetFactor.x))
55   ||(coord.y<(n*offsetFactor.y))||(coord.y>(1.0- (n+1.0)* offsetFactor.y)))
56  {
57   ret = vec4(0.0,0.0,0.0,0.0);
58  }
59  else
60  {
61   // Convolution
62   Convolution(coord);
63   // Signal computation
64   float f_pm = 0.5 *(f_xx-f_yy);
65   float f_s = 0.5 * f_p;
66   float e = sqrt(pow(f_pm,2.0)+pow(f_xy,2.0))/abs(f_s);
67   float q = (pow(f_x,2.0)+pow(f_y,2.0))* 2.0 /(1.0+e);
68
69   float phase = atan(sqrt(q),f_p) / TWOPI + 0.5;
70   float orientation;
71   if (phase ==0.5)
72    orientation = atan(f_xy,f_pm) /TWOPI +0.5;
73   else
74    orientation = atan(f_y,f_x) /TWOPI +0.5;
75   float amplitude = 0.5 * sqrt(pow(f_p,2.0)+q);
76   amplitude = pow(1.0 - 1.0 /(1.0 + amplitude),(1.0/amplitudeScale));
77   float apexAngle = (atan(sqrt(pow(f_s,2.0)-pow(f_xy,2.0)-pow(f_pm,2.0)),sqrt(pow(
         f_xy,2.0)+pow(f_pm,2.0)))) /PI;
78   ret = vec4(phase,orientation,amplitude,apexAngle);
79  }
80  return ret;
81 }
82 void Convolution(vec2 coord)
83 {
84  //Coordinates of one element of kernel1SfSc,kernel1 und kernel2
85  // -  -  -  -  |  -  -  -  -
86  // -  -  - lu1 | ru1 -  -  -
87  // -  -  -  -  |  -  -  -  -
88  // - lu2 -  -  |  -  - ru2 -
89  // ------------+------------
90  // - ld2 -  -  |  -  - rd2 -
91  // -  -  -  -  |  -  -  -  -
92  // -  -  - ld1 | rd1 -  -  -
93  // -  -  -  -  |  -  -  -  -
94  // Coordinates of one element of rand1SfSc,rand1 und rand2(r,l,d,u) or diag1Sfsc,
       diag1 und diag2 (rd,ru,lu,ld), respectively
```

```
95  // -  -  -  -  |  -  -  -  -
96  // - lu  -  -  u  -  - ru  -
97  // -  -  -  -  |  -  -  -  -
98  // -  -  -  -  |  -  -  -  -
99  // ---l--------+--------r---
100 // -  -  -  -  |  -  -  -  -
101 // -  -  -  -  |  -  -  -  -
102 // - ld  -  -  d  -  - rd  -
103 // -  -  -  -  |  -  -  -  -
104 // Stored data from array kernels..(k[Index][x..w]), Rand..(r[Index][x..y]) and
        Diag..(d[Index][x..w])
105 // for n=6 in the lower right quadrant
106 //+---+-----------------------+
107 //|r3x|r0x r0y r1x r1y r2x r2y|
108 //|---+-------+---------------|
109 //|r0x|d0x d0y|k0x k0z k1x k1z|
110 //|r0y|d0z d0w|k0y k0w k1y k1w|
111 //|   |-------+-------+       |
112 //|r1x|k0x k0y|d1x d1y|k2x k2z|
113 //|r1y|k0z k0w|d1z d1w|k2y k2w|
114 //|   |       +-------+-------|
115 //|r2x|k1x k1y k2x k2y|d2x d2y|
116 //|r2y|k1z k1w k2z k2w|d2z d2w|
117 //+---+---------------+-------+
118 //+---+-----------------------+
119 //| M |    b o r d e r'       |
120 //|---+-------+---------------|
121 //|   | d     |               |
122 //| b |    i  +  k e r n e l' |
123 //| o |------+a+------+       |
124 //| r |       +  g    |       |
125 //| d |       |     o +       |
126 //| e |       +------+n+------|
127 //| r | k e r n e l   +  a    |
128 //|   |               |     l |
129 //+---+---------------+-------+
130 // Convolution storage
131 vec4 f_pM = vec4(0.0,0.0,0.0,0.0),
132   f_xM = vec4(0.0,0.0,0.0,0.0),
133   f_yM = vec4(0.0,0.0,0.0,0.0),
134   f_xxM = vec4(0.0,0.0,0.0,0.0),
135   f_yyM = vec4(0.0,0.0,0.0,0.0),
136   f_xyM = vec4(0.0,0.0,0.0,0.0);
137 // start points of the texture coordinates for the diagonale /kernels
138 vec4 startRight,startLeft;    //x,y = up; z,w = down;
139 startRight.xy = coord + offsetFactor * vec2(1.0,2.0);
140 startRight.zw = coord + offsetFactor * vec2(1.0,-1.0);
141 startLeft.xy = coord + offsetFactor  * vec2(-2.0,2.0);
142 startLeft.zw = coord + offsetFactor  * vec2(-2.0,-1.0);
143 //start points of texture coordinates for boundary
144 vec4 startHorizontal,    //x,y = right; z,w = left;
```

```
145     startVertikal;      //x,y = up; z,w = down;
146   startHorizontal.xy = coord + offsetFactor * vec2(1.0,0.0);
147   startHorizontal.zw = coord + offsetFactor * vec2(-2.0,0.0);
148   startVertikal.xy = coord + offsetFactor  * vec2(0.0,2.0);
149   startVertikal.zw = coord + offsetFactor  * vec2(0.0,-1.0);
150   // Change of the texture coordinates per loop
151   // Kernel
152   vec4 deltaX = vec4(offsetFactor.x,0.0,offsetFactor.x,0.0) * 2.0;
153   vec4 deltaY = vec4(0.0,offsetFactor.y,0.0,-offsetFactor.y) * 2.0;
154   //vec4 deltaKY = deltaY
155   // Boundaries
156   vec4 deltaHorizontal = vec4(offsetFactor.x,0.0,-offsetFactor.x,0.0) * 2.0;
157   // deltaVertikal = deltaY
158   // Diagonals
159   vec4 deltaDiagRight = vec4(offsetFactor.x,offsetFactor.y,offsetFactor.x,-
          offsetFactor.y) * 2.0;
160   vec4 deltaDiagLeft = vec4(-offsetFactor.x,offsetFactor.y,-offsetFactor.x,-
          offsetFactor.y) * 2.0;
161
162   const vec4 startXValue = vec4(1.0,2.0,1.0,2.0);
163   const vec4 deltaValue = vec4(2.0);
164
165   vec4 yValue = vec4(1.0,1.0,2.0,2.0);
166   vec4 xValue;
167   int i= 0;
168   for(int y=1;y<halfN;y++)
169   {
170    float yF = float(y);
171    vec4 coordR1 = startRight + yF * deltaY;
172    vec4 coordL1 = startLeft + yF * deltaY;
173    vec4 coordR2 = startRight + yF * deltaX;
174    vec4 coordL2 = startLeft - yF * deltaX;
175    yValue += deltaValue;
176    xValue = startXValue;
177    for(int x=0;x<y;x++)
178    {
179     // Read texture
180     vec4 textureRU1 = texture2D(texture,coordR1.xy).zwxy;
181     vec4 textureRD1 = texture2D(texture,coordR1.zw).xyzw;
182     vec4 textureRU2  = texture2D(texture,coordR2.xy).zxwy;
183     vec4 textureRD2  = texture2D(texture,coordR2.zw).xzyw;
184
185     vec4 textureLU1 = texture2D(texture,coordL1.xy).wzyx;
186     vec4 textureLD1 = texture2D(texture,coordL1.zw).yxwz;
187     vec4 textureLU2 = texture2D(texture,coordL2.xy).wyzx;
188     vec4 textureLD2 = texture2D(texture,coordL2.zw).ywxz;
189
190     vec4 xSquareValue = xValue * xValue;
191     vec4 ySquareValue = yValue * yValue;
192
193     // Convolution
```

```
194     f_pM += (textureRD1 + textureLD1 + textureRU1 + textureLU1
195       +  textureRD2 + textureLD2 + textureRU2 + textureLU2) * kernelPoisson[i];
196     f_xM += (xValue * (textureRD1 - textureLD1 + textureRU1 - textureLU1)
197       +  yValue * (textureRD2 - textureLD2 + textureRU2 - textureLU2)) * kernel1[i];
198     f_yM += (yValue * (textureRD1 + textureLD1 - textureRU1 - textureLU1)
199       +  xValue * (textureRD2 + textureLD2 - textureRU2 - textureLU2)) * kernel1[i];
200     f_xxM += (xSquareValue * (textureRD1 + textureLD1 + textureRU1 + textureLU1)
201       +  ySquareValue * (textureRD2 + textureLD2 + textureRU2 + textureLU2)) *
              kernel2[i];
202     f_yyM += (ySquareValue * (textureRD1 + textureLD1 + textureRU1 + textureLU1)
203       +  xSquareValue * (textureRD2 + textureLD2 + textureRU2 + textureLU2)) *
              kernel2[i];
204     f_xyM +=  (textureRD1 + textureLD1 + textureRU1 + textureLU1
205       +  textureRD2 + textureLD2 + textureRU2 + textureLU2)* xValue * yValue *
              kernel2[i++];
206     // actualise indices
207     coordR1 += deltaX;
208     coordL1 -= deltaX;
209     coordR2 += deltaY;
210     coordL2 += deltaY;
211     xValue += deltaValue;
212    }
213   }
214   // Convolution of boundary and diagonals
215   vec2 xyValue = vec2(1.0,2.0);
216   xValue= vec4(1.0,2.0,1.0,2.0);
217   yValue= vec4(1.0,1.0,2.0,2.0);
218   for(i=0;i<halfN;i++)
219   {
220    // Read texture
221    vec2 textureR = texture2D(texture,startHorizontal.xy).xy;
222    vec2 textureL = texture2D(texture,startHorizontal.zw).yx;
223    vec2 textureU = texture2D(texture,startVertikal.xy).zx;
224    vec2 textureD = texture2D(texture,startVertikal.zw).xz;
225    vec4 textureRU = texture2D(texture,startRight.xy).zwxy;
226    vec4 textureRD = texture2D(texture,startRight.zw).xyzw;
227    vec4 textureLU = texture2D(texture,startLeft.xy).wzyx;
228    vec4 textureLD = texture2D(texture,startLeft.zw).yxwz;
229    // Convolution
230    f_pM.xy += (textureR+textureL+textureU+textureD) * randPoisson[i];
231    f_pM += (textureRU+textureRD+textureLU+textureLD) * diagPoisson[i];
232    f_xM.xy += xyValue* (textureR-textureL) * rand1[i];
233    f_xM += xValue * (textureRU+textureRD - textureLU - textureLD) * diag1[i];
234    f_yM.xy += xyValue * (textureD-textureU) * rand1[i];
235    f_yM += yValue * (textureLD + textureRD - textureLU - textureRU) * diag1[i];
236    f_xxM.xy+= xyValue * xyValue * (textureR+textureL) * rand2[i];
237    f_xxM += xValue * xValue * (textureRD + textureRU + textureLD + textureLU) * diag2
           [i];
238    f_yyM.xy+= xyValue * xyValue * (textureU+textureD) * rand2[i];
239    f_yyM += yValue * yValue * (textureRD + textureRU + textureLD + textureLU) * diag2
           [i];
```

```
240     f_xyM +=  (textureRD + textureRU + textureLD + textureLU)* xValue * yValue  *
            diag2[i];
241     xValue +=deltaValue;
242     yValue +=deltaValue;
243     xyValue += deltaValue.xy;
244     startHorizontal += deltaHorizontal;
245     startVertikal += deltaY;
246     startRight += deltaDiagRight;
247     startLeft += deltaDiagLeft;
248   }
249   //Convolution with centre point
250   f_p = texture2D(texture,coord).x * middlePoisson;
251   // Convolution with convolution storage
252   f_p += f_pM.x+f_pM.y+f_pM.z+f_pM.w;
253   f_x = f_xM.x+f_xM.y+f_xM.z+f_xM.w;
254   f_y = f_yM.x+f_yM.y+f_yM.z+f_yM.w;
255   f_xx = f_xxM.x+f_xxM.y+f_xxM.z+f_xxM.w;
256   f_yy = f_yyM.x+f_yyM.y+f_yyM.z+f_yyM.w;
257   f_xy = f_xyM.x+f_xyM.y+f_xyM.z+f_xyM.w;
258 }
```

Listing B.2: $place_{opt}$

```
// tensorFast.frag - as place_opt
#version 120
// CONSTANTS:
 const float PI = 3.14159265358979323846;
 const float TWOPI = PI * 2.0;
 const int HN_MAX = 8; // N_MAX = HN_MAX*2
// VARIABLES:
 uniform sampler2D texture;
 // Precalculated Convolution kernels:
 //   f_p:  |   f_x   |   f_y   |   f_xx  |   f_yy  |   f_xy  |
 //     |   |     |   |     |   |     |   |     |   |     |   |
 //  a x a  | -b 0 b  | -c-y-c  |  d u d  |  e z e  |  f 0-f  |
 //  x w x  | -y 0 y  |  0 0 0  |  z u z  |  u u u  |  0 0 0  |
 //  a x a  | -b 0 b  |  c y c  |  d u d  |  e z e  | -f 0 f  |
 //     |   |     |   |     |   |     |   |     |   |     |   |
 uniform vec4 kernelFP[HN_MAX * HN_MAX];  //a
 uniform vec4 kernelFX[HN_MAX * HN_MAX];  //b
 uniform vec4 kernelFY[HN_MAX * HN_MAX];  //c

 uniform vec4 kernelFXX[HN_MAX * HN_MAX]; //d
 uniform vec4 kernelFYY[HN_MAX * HN_MAX]; //e
 uniform vec4 kernelFXY[HN_MAX * HN_MAX]; //f

 uniform vec2 kernelFPRand[HN_MAX];   //x
 uniform vec2 kernelFXRand[HN_MAX];   //y
 uniform vec2 kernelFXXRand[HN_MAX];   //z

 uniform float middlePoisson;    //w
 uniform float offsetFXX;     //-u

 uniform bool odd;
 uniform int halfN;
 uniform float n;
 uniform vec2 offsetFactor;

 // Scale of the amplitude:
 uniform float amplitudeScale;
 float f_x,f_y,f_p,f_xx,f_yy,f_xy;

// FUNCTION-DECLARATIONS:
 vec4 AnalytikSignal();
 void Convolution(vec2 coord);

// PROGRAMME:
//-------------------------------------------------------------
//----------Main - Function------------------------------------
//-------------------------------------------------------------
void main()
{
```

```
50    gl_FragColor = AnalytikSignal();
51  }
52  //-----------------------------------------------------------
53  //----------Signal - Calculation----------------------------
54  //-----------------------------------------------------------
55   vec4 AnalytikSignal()
56  {
57   // Texture coordinates:
58   vec2 coord= gl_TexCoord[0].xy;
59   vec4 ret;
60   // Signal computing excepts the boundary
61   if((coord.x<((n+1.0)*offsetFactor.x))||(coord.x>(1.0-n* offsetFactor.x))
62    ||(coord.y<(n*offsetFactor.y))||(coord.y>(1.0- (n+1.0)* offsetFactor.y)))
63   {
64    float value = texture2D(texture,coord).x;
65    ret = vec4(value,value/2.0,value/2.0,value/2.0);
66   }
67   else
68   {
69    // Convolution
70    Convolution(coord);
71    // signal computing
72    float f_pm = 0.5 *(f_xx-f_yy);
73    float f_s = 0.5 * f_p;
74    float e = sqrt(pow(f_pm,2.0)+pow(f_xy,2.0))/abs(f_s);
75    float q = (pow(f_x,2.0)+pow(f_y,2.0))* 2.0 /(1.0+e);
76
77    float phase = atan(sqrt(q),f_p) / TWOPI + 0.5;
78    float orientation;
79    if (phase ==0.5)
80     orientation = atan(f_xy,f_pm) /TWOPI +0.5;
81    else
82     orientation = atan(f_y,f_x) /TWOPI + 0.5;
83    float amplitude = 0.5 * sqrt(pow(f_p,2.0)+q);
84    amplitude = pow(1.0 - 1.0 /(1.0 + amplitude),(1.0/amplitudeScale));
85    float apexAngle = (atan(sqrt(pow(f_s,2.0)-pow(f_xy,2.0)-pow(f_pm,2.0)),sqrt(pow(
          f_xy,2.0)+pow(f_pm,2.0)))) /PI;
86    phase-=0.5;
87    phase = max(phase,0.0);
88    ret = vec4(phase,orientation,amplitude,apexAngle);
89    float threshold = 1.0;
90    if(ret.x>threshold)
91    ret.x = 1.0;
92   }
93   return ret;
94  }
95
96  void Convolution(vec2 coord)
97  {
98   // Texture storage
99   vec4 tRightDown,tLeftDown,tRightUp,tLeftUp;
```

```
100  vec2 tRight,tLeft,tUp,tDown;
101  // Convolution storage
102  vec4 f_pM = vec4(0.0,0.0,0.0,0.0),
103    f_xM = vec4(0.0,0.0,0.0,0.0),
104    f_yM = vec4(0.0,0.0,0.0,0.0),
105    f_xxM = vec4(0.0,0.0,0.0,0.0),
106    f_yyM = vec4(0.0,0.0,0.0,0.0),
107    f_xyM = vec4(0.0,0.0,0.0,0.0);
108  // start points of texture coordinates for the inner of the kernels
109  vec4 startRight,startLeft; //x,y = up; z,w = down;
110  startRight.xy = coord + offsetFactor * vec2(1.0,2.0);
111  startRight.zw = coord + offsetFactor * vec2(1.0,-1.0);
112  startLeft.xy = coord + offsetFactor * vec2(-2.0,2.0);
113  startLeft.zw = coord + offsetFactor * vec2(-2.0,-1.0);
114  // Change of texture coordinates per loop
115  vec4 deltaOffsetY = vec4(0.0,offsetFactor.y,0.0,-offsetFactor.y) * 2.0;
116  vec4 deltaOffsetX = vec4(offsetFactor.x,0.0,offsetFactor.x,0.0) * 2.0;
117  vec4 coordRight,coordLeft;
118  int i= 0;
119  for(int y=0;y<halfN;y++)
120  {
121   float yF = float(y);
122   coordRight = startRight + yF * deltaOffsetY;
123   coordLeft = startLeft + yF * deltaOffsetY;
124   for(int x=0;x<halfN;x++)
125   {
126    // Read texture
127    tRightUp = texture2D(texture,coordRight.xy).zwxy;
128    tRightDown = texture2D(texture,coordRight.zw).xyzw;
129    tLeftUp  = texture2D(texture,coordLeft.xy).wzyx;
130    tLeftDown = texture2D(texture,coordLeft.zw).yxwz;
131    // Convolution
132    f_pM += (tRightDown + tLeftDown + tRightUp + tLeftUp) * kernelFP[i];
133    f_xM += (tRightDown - tLeftDown + tRightUp - tLeftUp) * kernelFX[i];
134    f_yM += (tRightDown + tLeftDown - tRightUp - tLeftUp) * kernelFY[i];
135    f_xxM += (tRightDown + tLeftDown + tRightUp + tLeftUp) * kernelFXX[i];
136    f_yyM += (tRightDown + tLeftDown + tRightUp + tLeftUp) * kernelFYY[i];
137    f_xyM += (tRightDown - tLeftDown + tRightUp - tLeftUp) * kernelFXY[i];
138    // actualise indices
139    i++;
140    coordRight += deltaOffsetX;
141    coordLeft -= deltaOffsetX;
142   }
143  }
144  // Convolution near boundary
145  vec4 coordRightLeft, // x,y = rechts; z,w= links;
146    coordUpDown; // x,y = oben; z,w = unten;
147   coordRightLeft.xy =  coord + offsetFactor * vec2(1.0,0.0);
148   coordRightLeft.zw =  coord + offsetFactor * vec2(-2.0,0.0);
149   coordUpDown.xy = coord + offsetFactor * vec2(0.0,2.0);
150   coordUpDown.zw = coord + offsetFactor * vec2(0.0,-1.0);
```

```
151   deltaOffsetX *= vec4(1.0,0.0,-1.0,0.0);
152
153  vec2 vecOffsetFXX = vec2(offsetFXX);
154  for(i=0;i<(halfN-1);i++)
155  {
156   // Read texture
157   tRight = texture2D(texture,coordRightLeft.xy).xy;
158   tLeft = texture2D(texture,coordRightLeft.zw).yx;
159   tUp  = texture2D(texture,coordUpDown.xy).zx;
160   tDown = texture2D(texture,coordUpDown.zw).xz;
161   // Convolution
162   f_pM.xy += (tRight+tLeft+tUp+tDown) * kernelFPRand[i];
163   f_xM.xy += (tRight - tLeft) * kernelFXRand[i];
164   f_yM.xy += (tDown - tUp) * kernelFXRand[i];
165   f_xxM.xy += (tRight + tLeft) * kernelFXXRand[i] + (tUp+tDown) * vecOffsetFXX;
166   f_yyM.xy += (tUp + tDown) * kernelFXXRand[i] + (tLeft+tRight) * vecOffsetFXX;
167   // actualise indices
168   coordRightLeft += deltaOffsetX;
169   coordUpDown += deltaOffsetY;
170  }
171  if(odd)
172  {
173   // read texture
174   tRight = texture2D(texture,coordRightLeft.xy).xy;
175   tLeft = texture2D(texture,coordRightLeft.zw).yx;
176   tUp  = texture2D(texture,coordUpDown.xy).zx;
177   tDown = texture2D(texture,coordUpDown.zw).xz;
178   // Convolution
179   f_pM.xy += (tRight+tLeft+tUp+tDown) * kernelFPRand[i];
180   f_xM.xy += (tRight - tLeft) * kernelFXRand[i];
181   f_yM.xy += (tDown - tUp) * kernelFXRand[i];
182   f_xxM.xy += (tRight + tLeft) * kernelFXXRand[i];
183   f_yyM.xy += (tUp + tDown) * kernelFXXRand[i];
184   f_yyM.x += (tLeft+tRight).x * offsetFXX;
185   f_xxM.x += (tUp+tDown).x * offsetFXX;
186  }
187  else
188  {
189   // read texture
190   tRight = texture2D(texture,coordRightLeft.xy).xy;
191   tLeft = texture2D(texture,coordRightLeft.zw).yx;
192   tUp  = texture2D(texture,coordUpDown.xy).zx;
193   tDown = texture2D(texture,coordUpDown.zw).xz;
194   // Convolution
195   f_pM.xy += (tRight+tLeft+tUp+tDown) * kernelFPRand[i];
196   f_xM.xy += (tRight - tLeft) * kernelFXRand[i];
197   f_yM.xy += (tDown - tUp) * kernelFXRand[i];
198   f_xxM.xy += (tRight + tLeft) * kernelFXXRand[i] + (tUp+tDown) * vecOffsetFXX;
199   f_yyM.xy += (tUp + tDown) * kernelFXXRand[i] + (tLeft+tRight) * vecOffsetFXX;
200  }
201  // Convolution with centre point
```

```
202   float middle = texture2D(texture,coord).x;
203   f_p = middle * middlePoisson;
204   f_xx = middle * offsetFXX;
205   f_yy = middle * offsetFXX;
206   // Convolution of convolution storages
207   f_p += f_pM.x+f_pM.y+f_pM.z+f_pM.w;
208   f_x = f_xM.x+f_xM.y+f_xM.z+f_xM.w;
209   f_y = f_yM.x+f_yM.y+f_yM.z+f_yM.w;
210   f_xx += f_xxM.x+f_xxM.y+f_xxM.z+f_xxM.w;
211   f_yy += f_yyM.x+f_yyM.y+f_yyM.z+f_yyM.w;
212   f_xy = f_xyM.x+f_xyM.y+f_xyM.z+f_xyM.w;
213  }
```