computer graphics & vision

Inffeldgasse 16/2, A-8010 Graz

www.icg.tu-graz.ac.at

TUG

Computing and Visualizing Uncertainties in Radiofrequency Ablation Treatment Simulations

Christoph Ebner

Inst. of Computer Graphics and Vision
Graz University of Technology, Austria

Master's Thesis
Advisor: Dipl.-Ing. Philip Voglreiter
Graz, November 27, 2017

contact: Christoph Ebner c.ebner@student.tugraz.at

Abstract

In the past years, developments in the domain of cancer treatment have led to the rise of minimally invasive techniques such as radiofrequency ablation (RFA). To master the complexity of the RFA procedure and its dependency on several treatment-specific parameters, RFA treatment simulations may help interventional radiologists to predict the outcome and further improve the treatment success rate. However, the parameterizations of such simulations are often only estimates that may lead to considerable discrepancies with respect to real treatment. This thesis introduces a plugin for an existing RFA software called "RFA Guardian" that considers uncertainties in the parameter space and visualizes the possible range of results. The "Parameter Uncertainty" plugin consists of two distinct modules for parameter sampling and simulation ensemble visualization. The visualization technique provides a concise representation of simulation ensembles while laying focus on factors important for RFA treatment planning such as blood vessels that penetrate the ablation zone. Ensembles are visualized using the concept of contour boxplots, which is a generalization of boxplots for simulation ensembles. In this concept, a measure in which each simulation outcome is ranked by its depth within the whole ensemble is employed. This measure is known as "band depth". In order to provide real time rendering of a simulation ensemble in the Parameter Uncertainty plugin, new approaches for fast band depth computation are introduced.

Keywords: *Radiofrequency ablation, boxplots, band depth, uncertainty visualization, ensemble visualization*

Acknowledgements

First, I want to sincerely thank my advisor, Dipl.-Ing. Philip Voglreiter, for his generous support and scientific advice throughout the past years. Further, I owe a great thank to Prof. Dr. Dieter Schmalstieg for enabling my work at the ICG which triggered my interest in the exciting field of computer graphics.

Special thanks also go to my parents for their continuous support, without which my studies at the Graz University of Technology would not have been possible.

Lastly, but perhaps most importantly, I would particularly like to thank my girlfriend for her day-to-day encouragement during writing this thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

....

Ort und Datum Unterschrift

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....

Place and date Signature

Contents

List of Figures

List of Abbreviations

BD	Band depth
CCW	Counter-clockwise
CW	Clockwise
GPGPU	General-purpose computing on graphics processing units
GPU	Graphics processing unit
IQR	Interquartile range
KDE	Kernel density estimation
lBD	Lesion band depth
LUT	Lookup table
mBD	Modified band depth
RFA	Radiofrequency ablation
RNG	Random number generator
sBD	Set band depth
SIMD	Single instruction, multiple data
SSBO	Shader Storage Buffer Object
UI	User interface

Part I
Introduction

I Introduction

1 Motivation and related work

1.1 Motivation

From a global perspective, malignant neoplasms (commonly referred to as "tumors" [1, p16]) are a major cause of death, leading to more fatalities than tuberculosis, HIV/AIDS, diabetes mellitus and Alzheimer's disease combined. In 2015, an estimated 16% of all deaths were attributed to neoplasms, making it the second-most cause of death of non-communicable diseases in the world, only surpassed by cardiovascular diseases [2]. Due to the prevalence of tumor related deaths, several treatment methods, such as surgery, chemotherapy, radiation therapy and others[1], were established during the 20th century in order to conquer neoplasms. While those three methods still remain the "three pillars" of tumor treatment, a wide range of minimally invasive, image-guided techniques has become available in the last decades [4].

Among those techniques, a procedure called radiofrequency ablation (RFA) has been widely used to thermally ablate tumors in various organs of the body. In this procedure, an AC current is applied locally at the tumor through a needle electrode, causing resistive heating of the tumor and surrounding tissue [5]. However, performing an RFA requires a significant degree of clinical experience. Less experienced interventional radiologists may benefit from an appropriate planning tool for the treatment [6]. One of these tools is the "RFA Guardian", which provides pre-interventional, interventional and post-interventional assistance in RFA liver tumor treatment. The interventional phase includes a sophisticated simulation algorithm of the treatment outcome for heating protocols used during an RFA intervention [7]. However, critical simulation parameters, such as the perfusion of the healthy tissue surrounding the tumor and tumor perfusion itself, have a strong influence on the simulation outcome, but - at the current stage - need to be estimated by the user.

The goal of this thesis is to extend the RFA Guardian with a plugin that allows for visualizing the uncertainty of the simulation results arising from sampling the treatment parameter space in order for users to get a non-cluttered view of different simulation outcomes depending on their re-

[1] Other treatment methods include adjuvant therapy, hormonal therapy or immunotherapy. For further reading see Sudhakar [3].

spective parameterizations and to provide support in subsequent treatment decisions. This thesis will first give a rough introduction to the medical background of RFA in general and elaborate on certain difficulties that need to be addressed by the visualization technique. Subsequently, related work in the field of boxplot visualization techniques is introduced. Part II introduces new fast approaches to band depth computation, as these methods will be needed for real-time visualization of the simulation ensembles. Part III will focus on the main methods and algorithms used for generating the sampling space and visualization of the simulation ensemble where the new approaches for fast band depth computation are used to generate a contour boxplot of the ensemble. The fourth part includes a performance analysis, as well as qualitative results and the last part contains concluding words and an outlook on future work.

1.2 Radiofrequency ablation

Radiofrequency ablation is a minimally invasive medical procedure used for the treatment of several types of diseases, including tumors in the breast [8], liver [9, 10], lung [11, 12], kidney [13, 14] and bone [15, 16]. RFA is widely used due to its safety, ablation efficiency, effectiveness and acceptable complication profile [5, 17, 18]. A comparison of percutaneous ablation technologies for malignant liver tumors concluded that RFA should be considered "the first-line treatment modality in the treatment of primary and secondary liver malignancies", though also noticing its limitations for large hepatocellular carcinomas [19]. Due to its advantages and widespread application area, Gillams [17] and Friedman et al. [20] referred to RFA as the "frontrunner" of available image-guided minimally invasive ablation methods. However, other methods, such as cryotherapy ablation and microwave ablation are on the rise, too, as the whole field of ablation techniques has seen large improvements over the last years [17]. Especially microwave ablation offers advantages over RFA when dealing with tumor masses larger than 3cm in diameter [21]. However, in terms of clinical end-points of hepatocellular carcinoma treatment, further studies have to be conducted in order to favor one over the other [22].

1.2.1 Radiofrequency ablation procedure

In performing an RFA treatment, an electrode called "probe" is inserted into the patients body to a target location (e.g., a tumor in the liver). For guiding the probe through the human body, medical imaging techniques such as CT or ultrasound can be used [23]. Several types of probes are available: orig-

inally, standard stock needles (Figure 1.1a) were used. However, the extent of the thermal injury was relatively small compared to the lesion size achievable by modern equipment. Nowadays, coaxial needles that house seven or nine retractable electrodes are commonly used. During probe placement, the electrode tips stay retracted and are not expanded until the target location has been reached. Figures 1.1b and c show two modern probe designs [24].

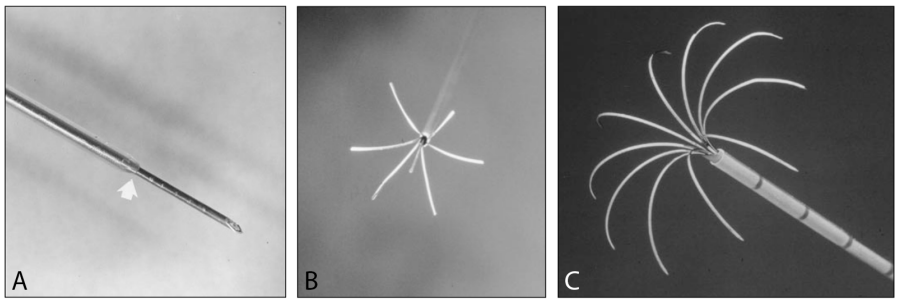


Figure 1.1: Examples of probes used in RFA treatment. (A) Standard stock needle insulated to distal tip. The arrow marks the edge of the insulation. (B) Probe with "Christmas tree" configuration, manufactured by RITA Medical Systems, Mountain View, CA. (C) Probe with "umbrella" configuration, manufactured by Radiotherapeutics, Mountain View, CA. Figure taken from McGahan and Dodd [24].

After successful probe placement, an AC current in the radio frequency range (typically 450-500kHz) is applied between the electrodes and grounding pads. This leads to the generation of resistive heat around the electrode tips (as well as the grounding pads, although temperatures are lower on the pads due to their larger surface area). The heat stems from an effect called "ionic agitation": Through application of an AC current, the ions and water molecules in the body tissues begin to move. As a result, heat - directly proportional to the applied current density - originates through friction. The amount of time it takes for cells to undergo coagulation necrosis (damage triggered cellular death) depends on the resulting temperature. Optimal temperatures for ablation are in the range of 50°C to 100°C [23, 25].

In order to be sure to remove all tumor-related tissue, usually a surgical (tumor-free) margin of about 1cm has to be ablated around the visible tumor (however, there exists an exception when treating target volumes in kidneys where a margin of 1cm may be too much). If this margin cannot be achieved, several consecutive ablations have to be performed in order to ablate all neoplasmic tissue [20]. Figure 1.2 provides a short summary of the whole ablation process in four schematic depictions.

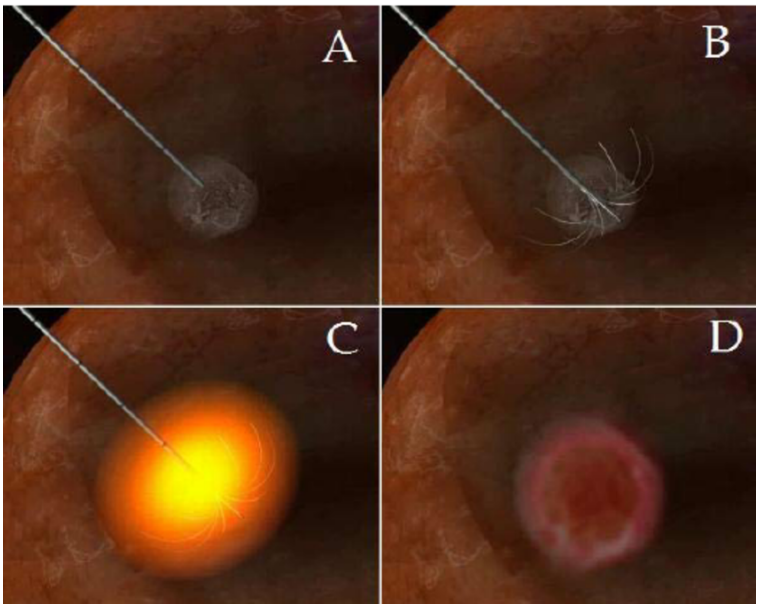


Figure 1.2: Schematic summary of the RFA procedure: First, the probe is inserted into the target volume (A). Second, the retracted probe electrodes are extended (B) and ablation of the tumor with a safety margin is carried out (C). Finally, the tumor tissue is ablated and the probe is removed from the body (D). Taken from Julianov [26].

1.2.2 Difficulties and shortcomings

A major difficulty arises when larger blood vessels are located near the tumor, as increased blood flow shows a strong negative correlation with lesion size through the resulting heat sink effect. Blood flow in large vessels acts as a heat sink, exerting a cooling effect on the tissue. This leads to a smaller lesion extent and may subsequently prevent the treatment to fully ablate the tumor [27]. According to Lu et al. [28], this effect was consistently observable beyond a vessel diameter of 2-4mm, and often tumor recurrence after treatment is attributed to the heat sink effect [29]. Due to the significance of the effect, it is reasonable to assume that the outcome of the ablation is especially important in regions near vessels. Taking this fact into consideration, the uncertainty visualization technique pays close attention to the simulation outcome in these particular regions in the following way: For one, the estimated probability of a blood vessel slice being ablated on a certain

2D view can be shown[2]. Furthermore, the user is able to only display simulation results that fail to ablate a specific vessel, as well as results that fully ablate the vessel in question. Finally, the user can define a threshold with respect to the median outcome that leads to visualizing only the results that lie within or outside of these results.

Apart from the heat sink problem, RFA suffers from additional disadvantages which will only be mentioned briefly, due to the fact that they are not directly relevant for this thesis. One disadvantage that was already mentioned is the restriction to smaller target volumes. For example, Solbiati et al. [30] reported a significant difference in recurrence rate between liver metastases that were less than 3cm in diameter and metastases greater or equal than 3cm, with a recurrence rate of 16.5% and 56.1% respectively. Yu and Burke [19] also mentioned that for liver tumors that are larger than 5cm, RFA is limited due to incomplete tumor ablation. Further disadvantages over other ablation methods are listed in Maini [21] and include - among others - preferential heating of fat tissue and dependency of the current flow to local electrical tissue characteristics.

1.2.3 Radiofrequency ablation simulation

RFA treatment is a fairly complex procedure, considering the aforementioned difficulty caused by the heat sink effect and the dependency of the resulting lesion shape from several patient specific and non-patient specific parameters. In order for the treatment to be successful, the interventional radiologist has to ensure full ablation of the tumor including a safety margin, whilst preserving as much healthy tissue as possible.

A study concerning the outcome of RFA treatments dependent on the experience of the operators showed that RFA treatments bear a considerable learning curve and, through the utilization of a specialized RFA team, simulation outcomes could be improved significantly. The 2-year survival rate was 89%, when treatment was performed by an experienced clinician, and, 40%, with a less experienced clinician [31]. This circumstances make adequate treatment planning and simulation an important factor, and, according to Mariappan et al. [6], especially operators with little experience could benefit from software that visualizes simulation results based on planned treatment.

The interventional phase of the RFA Guardian contains a graphics processing unit (GPU) simulation for predicting the resulting lesion shape and

[2]The RFA Guardian displays the 3D simulation results as projections on orthogonal 2D slices. For details, refer to Section 5.2.

size. For calculating the temperature distribution in the tissue, the algorithm uses a finite element method to solve for a Pennes bioheat model [32]. The used model incorporates tissue parameters as well as blood-related parameters, such as thermal conductivity of the tissue, tissue convection, specific heat capacity and density of both blood and tissue, as well as perfusion.

According to a study in Mariappan et al. [6], comparison between real and simulated lesions led to a maximum deviation of 3.5mm for nine cases. Furthermore, it is proposed that with the availability of more patient-specific data, the simulation could achieve even greater accuracy. Due to exploitation of the parallel processing capabilities on the GPU, the algorithm performs significantly faster than the real treatment protocol. For example, the outcome in a patient undergoing 62 minutes of RFA treatment (with three ablations), could be simulated in five to six minutes[3]. This allows for running the simulation before treatment and giving interventional radiologists better insight on whether to continue the planned treatment or switch to a different treatment protocol [6].

1.2.4 Factors influencing the treatment outcome

The actual extent and shape of the resulting lesion depends on various parameters of the target tissue and the tissue surrounding the target volume as variations of those parameters in RFA simulation algorithms have shown. For example, Hall et al. [33] stated that main contributors of patient-specific parameters to the outcome of hepatic RFA simulations (and consequently treatments, provided that the simulations reflect in-vivo conditions) are blood perfusion-parameters, electrical parameters of the tissue and cell death severity. Other influencing factors include probe placement - where placement inaccuracies have been reported to may have severe effects on the outcome of RFA treatments [34], as well as probe properties (cannula diameter and tip lengths), temperature and treatment duration [35].

In order to carry out RFA simulations, the mentioned parameters have to be provided to the simulation algorithm. However, the used parameters are only estimates that may not reflect actual treatment conditions accurately. Thus, the plugin introduced in this thesis aims to provide a mechanism for sampling the parameter space and carrying out simulations for each sample in order to get better insight on which lesion extents and shapes likely to expect during treatment, as each simulation invocation potentially leads to different results. Thus, the user is presented with a whole ensemble of simulation results instead of only one. The uncertainty sampling technique introduced

[3]Using a Intel Core i7 CPU with 3.5GHz and 6 Cores and a NVIDIA GeForce Titan Black GPU with 3.5 GHz. For details, refer to Mariappan et al. [6]

in this thesis includes the possibility to sample perfusion parameters of the tumor and healthy tissue as well as the electrode tip placement. However, if needed, additional parameters can easily be added to the plugin.

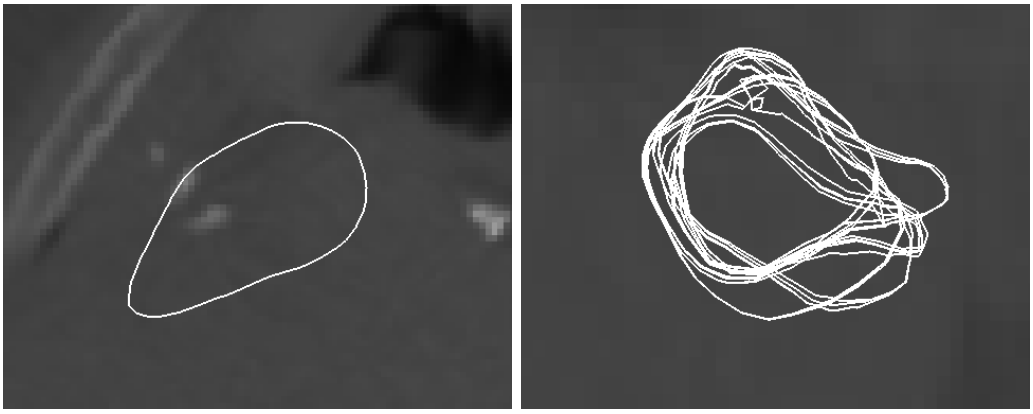
1.2.5 Visualizing simulated lesions

The RFA Guardian visualizes the resulting lesion as 2D projections of the lesion outline on orthogonal anatomic planes, whereas a lesion outline represents an iso surface of the simulation mesh where the cell death probability crosses 80%. A typical example of this representation in the axial plane is given in Figure 1.3a. The depicted outline or "contour", as it is referred to subsequently in the thesis, is the result of a single simulation invocation using a specific treatment parameterization.

An improved visualization technique of a single simulated lesion for the RFA Guardian was established by Voglreiter et al. [36], who used a level of detail approach to display the exact distance of the resulting lesion contour to the tumor. Furthermore, their method enables the users to visualize iso-values of several tissue parameters within the contour using a bivariate approach. Examples of their rendering technique can be seen in Figure 1.4.

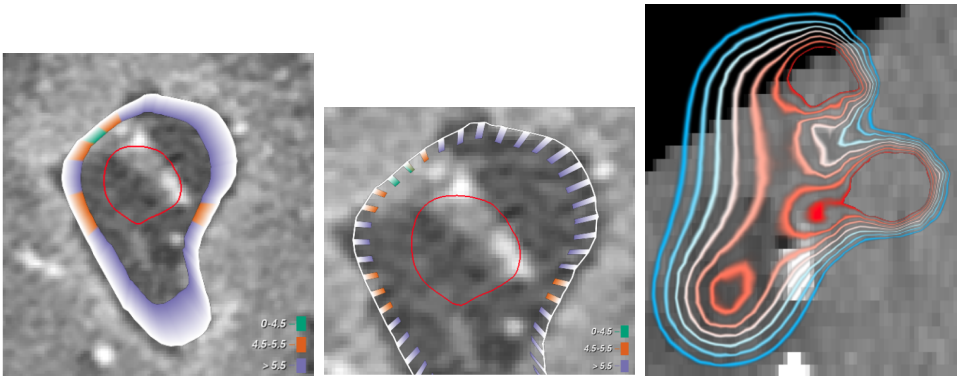
However, as noted above in Section 1.2.4, due to the uncertainty in the simulation parameterization, the user is presented with a whole ensemble of contours instead of a single one. Thus, the plugin needs to provide an appropriate visualization technique for contour ensembles, as the current method is not capable of delivering a suitable way for the user to investigate such an ensemble. Figure 1.3b shows a visualization of a simulation ensemble consisting of 25 contours resulting from different parameterizations. As is visible in the figure, the visualization is cluttered and provides no additional statistical relevant information. Furthermore, no connection to the respective parameterization of each simulated contour is established. Both shortcomings are addressed by the plugin which visualizes the ensemble using an adapted version of a concept called "Contour Boxplots" [37] for visualizing the median result, the 50% central region[4] and simulation outliers. To get an overview of this concept, the notion of contour boxplots as a generalization of functional boxplots is further elaborated in Section 1.3.

[4]The 50% central region contains half of the simulation results which are considered the most "central" within the whole ensemble.



(a) Single simulation outcome (b) Simulation ensemble

Figure 1.3: The RFA Guardian represents the resulting lesion as 2D projections of the lesion outline on orthogonal anatomic planes (a). These outlines are referred to simply as "contours" in the thesis. Visualizing a whole ensemble of simulation results leads to a cluttered representation (b).



(a) Zoomed out (b) Zoomed in (c) Iso-contours

Figure 1.4: Examples of the visualization technique of Voglreiter et al. [36]. The visualization employs a level detail approach. For one, the distance of the tumor is color-coded for safety margin evaluations (refer to Section 1.2), whereas different zoom levels yield different representations (the red contour is the tumor outline) (a & b). Additionally, iso-contours of various tissue parameters can be visualized using color coding (c).

1.3 Concepts of Boxplots and the notion of data depth

A large part of the visualization technique developed in this thesis is visualizing a certain simulation ensemble using a concept called contour boxplots proposed by Whitaker et al. [37], which is a generalization of functional boxplots. In order to generate a contour boxplot of a simulation ensemble, each contour has to be ranked to create an ordered statistic. Using the ordered statistic, one can identify statistically significant contours representing certain quartiles (or outliers) of the data set, analogous to a conventional boxplot. In order to establish data ranking, both functional boxplots and contour boxplots use a variant of a concept called band depth (BD), introduced by López-Pintado and Romo [38].

This section will first introduce functional boxplots and BD in order to give the reader a suitable background of these concepts before addressing their generalization with the notion of contour boxplots. Sometimes, comparisons to the original boxplot (introduced as a statistical concept by Tukey [39]) are made. Whenever this is the case, they are referred to as "conventional boxplots". A conventional boxplot (also called "box-and-whisker plot") typically consists of the following features: the median, the "box", which is constrained by the upper and lower quartiles (containing 50% of the data), the "whiskers" which comprise all data points not in the box, however, within 1.5 times the interquartile range (IQR) and, finally, outliers - all remaining data points - often visualized as dots.

1.3.1 Functional boxplots

Background Functional boxplots, proposed by Sun and Genton [40], port the concept of conventional boxplots to functional data. The functional boxplot adopts the statistical descriptors used in conventional boxplots. However, the generation of the ordered statistic is quite different. While it is easy to rank one-dimensional data (as is the case for conventional boxplots), ranking of multivariate (functional) data is less trivial. To accomplish this task, Sun and Genton [40] used the concept of BD. However; to make sense of the notion of BD, the concept of functional bands is defined first. Both, the definition of the functional band and the derivation of BD is taken from López-Pintado and Romo [41].

Functional bands A functional band or simply "band" in $\mathbb{R}^2$, restricted to an interval I and consisting of k functions $f_{i_r} : t \mapsto \mathbb{R}, t \in I$ with $k \geq 2$

and $r \in 1, ..., k$ is defined as:

$$B^k(f_{i_1}, ..., f_{i_k}) \coloneqq \{(t, y) : t \in I, \min_{r \in 1,...,k} f_{i_r}(t) \leq y \leq \max_{r \in 1,...,k} f_{i_r}(t)\} \quad (1)$$

Intuitively, a band is defined as the set enclosed by the envelope of the functions that it is built upon. An example of a band consisting of three functions is shown in Figure 1.5.

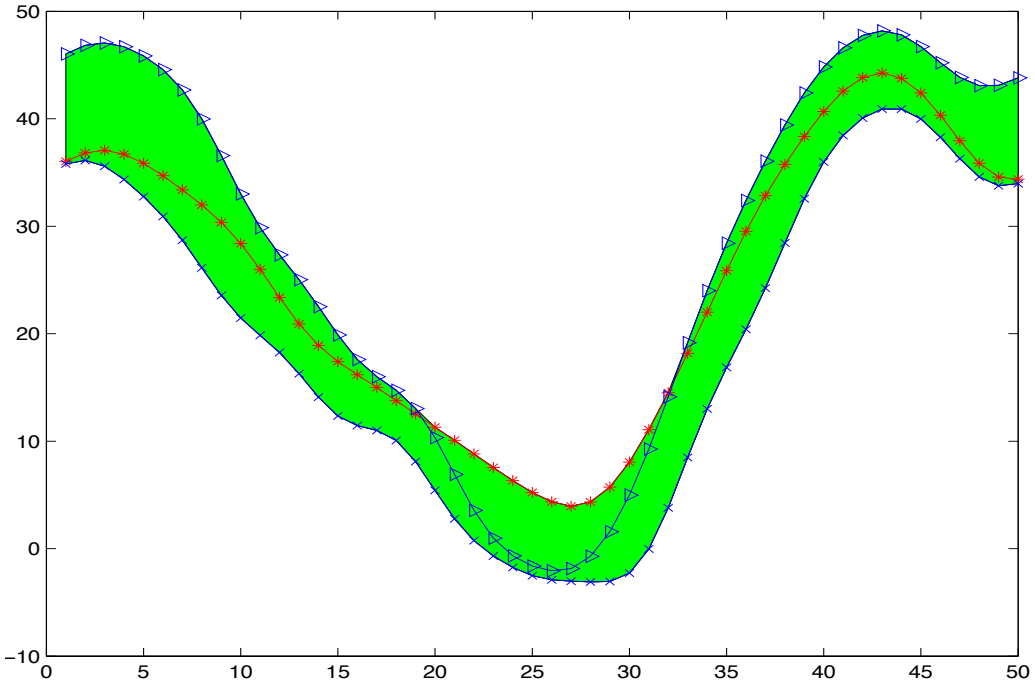


Figure 1.5: An example of a band consisting of three functions in the interval $t \in [1; 50]$. The band is depicted as the green region enclosed by all three functions. Picture taken from López-Pintado and Romo [38].

Band depth Considering a certain function f_i taken out of an ensemble of n total functions, the graph of this function is defined as the set of tuples in the form of:

$$G(f_i) \coloneqq \{t, f_i(t) : t \in I\} \quad (2)$$

Hence, the graph of f_i is said to be included in a certain band if:

$$G(f_i) \subset B^k \tag{3}$$

Defining the number of all bands consisting of k number of functions that satisfy Condition 3 for a particular function f_i as B_i^k (e.g., if $G(f_i)$ is the subset of five bands, $B_i^k = 5$), the fraction of bands B^k that include the graph of a certain function f_i can be defined as:

$$S_n^k(f_i) = \binom{n}{k}^{-1} B_i^k \tag{4}$$

with $S_n^k(f_i) \in [0; 1]$. In order to calculate the BD for the function f_i, one has to consider $S_n^j(f_i)$ for all $j \in [2; k]$. The BD for f_i is calculated as:

$$BD_{n,k} = \sum_{j=2}^{k} S_n^j(f_i) \tag{5}$$

As can be seen in Equations 4 and 5, the BD is dependent on the free variable k which can be chosen to suit the needs of a particular application. Moreover, it should be noted that choosing high k values increases the computational costs disproportionately. For example, if we assume an ensemble of 30 functions and, for each of them, the BD is calculated, choosing $k = 2$ results in a total of 435 bands to be computed, and, moreover, $BD_{30,2} = S_{30}^2$. However, choosing $k = 3$ results in 4060 bands to be computed, an increase of 833%. However, $k = 3$ is recommended over $k = 2$ by López-Pintado and Romo [41] due to the fact that when bands consisting of two functions cross, the band is reduced to a single point. Consequently, no other function will entirely be included in this band. A method for fast BD computation regardless of which k parameter is used for constructing the bands, will be introduced in Section 7.4.4. Subsequently in this thesis, the number of functions a band consists of will be referred to as the "order" of the band.

Modified band depth The notion of BD discussed above is fairly restrictive when it comes to functions leaving a certain band only for a short interval. The previously defined concept of BD does not distinguish whether a certain function is mostly outside a certain band, or if it only leaves the band for a negligible interval. If Equation 3 is not satisfied, B_i^k treats both situations in the same manner. Furthermore, the previous BD concept may lead to lots of ties in the computation of the functional depth.

Being aware of that, López-Pintado and Romo [41] introduced a slightly modified version of their BD concept, termed modified band depth (mBD).

Both methods only differ in the calculation of B_i^k. While, in the original evaluation of B_i^k, an all-or-nothing approach was taken (if a certain band contained the function f_i, then 1 was added, otherwise not). The mBD approach, however, adds the normalized fraction for which the function f_i stays inside the test band to B_i^k, dismissing condition 3 and making it possible (and likely) for B_i^k, and subsequently for S_n^k and $BD_{n,k}$, to to have a fractional value. In contrast to BD, which is more dependent on the shape of the functions, the modified version depends more on the function's magnitude. Furthermore, mBD is said to be very stable in the band order [41]. However, the computational costs increase rapidly for higher orders in the same manner as for BD. An example of the calculation of mBD is presented in Figure 1.6.

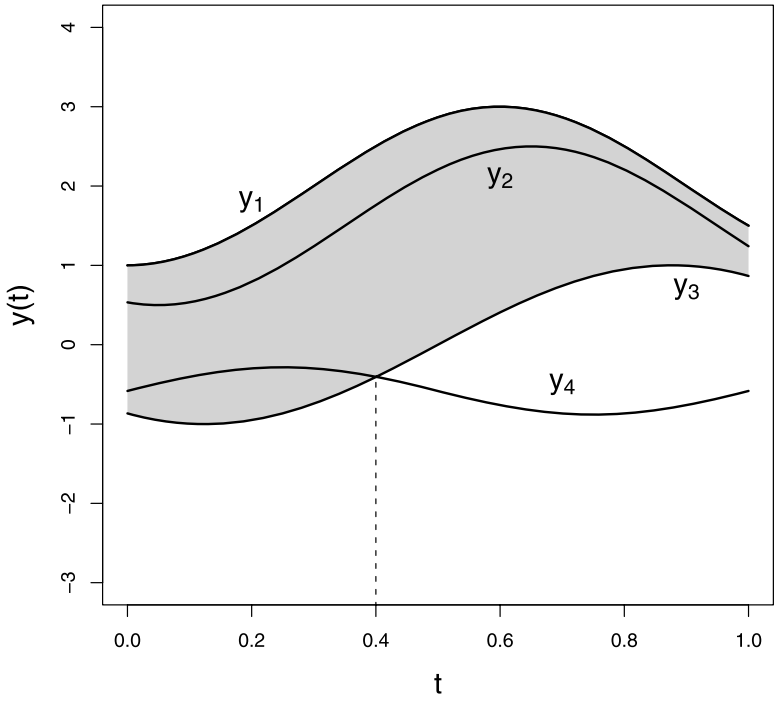


Figure 1.6: An example of the calculation of mBD of four contours. k was chosen to be 2, and the interval I is normalized. In the current step, y_4 is tested against the band formed by y_1 and y_3. Since y_4 is only an amount of 0.4 in the test band, B_4^2 is increased by 0.4 in this calculation step. Overall, B_4^2 takes the value 3.8. The final mBD values are: $mBD_{1,2} = 0.5$, $mBD_{2,2} = 0.83$, $mBD_{3,2} = 0.7$, $mBD_{4,2} = 0.63$. Picture taken from Sun and Genton [40].

Construction of functional boxplots With the utilization of mBD, an ordered statistic can be created to construct functional boxplots. The construction of functional boxplots includes the following statistical parameters [40]:

- The median, which is the function with the highest mBD value (which makes intuitive sense, since a high mBD values describes functions that are deep within an ensemble).
- 50% of the functions with highest mBD rank - forming a region that is analogous to the IQR of convention boxplots.
- The "fence", consisting of functions not included in the 50% region but ranked higher than $1.5 * \text{IQR}$.
- The outliers: all remaining functions.

1.3.2 Contour boxplots

Contour boxplots can be viewed as generalized functional boxplots. Developed for uncertainty characterization of numerical simulation results, the concept is well applicable to the uncertainty visualization technique of RFA simulation ensembles presented in this thesis. In order to generalize the idea of the functional boxplot, the definition of a band in Equation 1 has to be revised. Every non-parametric contour that takes part in constructing a band is viewed as a set C_i, containing all points that it encloses (as well as all points on the contour itself). The band formed by k numbers of sets is the set-theoretic difference of the union and intersection of the k sets:

$$B_c^k(C_1, ..., C_k) \coloneqq \Big(\bigcup_{i=1}^{k} C_i\Big) \setminus \Big(\bigcap_{i=1}^{k} C_i\Big) \tag{6}$$

In general terms, the band can be viewed as consisting of all points, bounded by the union and intersection of the k contours.

Similar to the graph of a function (refer to Equation 2), a contour is tested against a band by checking if all points on the contour form a subset of the band; refer to Figure 1.7a for an example. Thus, the calculation of B_i^k and, subsequently, $BD_{n,k}$ remains unchanged. However, in order to distinguish contour band depth from functional BD, this method is referred to as "set band depth" (sBD). If all topologies can be mapped to the graph of a function, sBD converges to BD, see Figure 1.7b. There exists also an analogous concept to mBD, called "contour band depth". However, since the visualization technique presented in this thesis does not make use of contour band depth, further explanation of the concept is omitted [37].

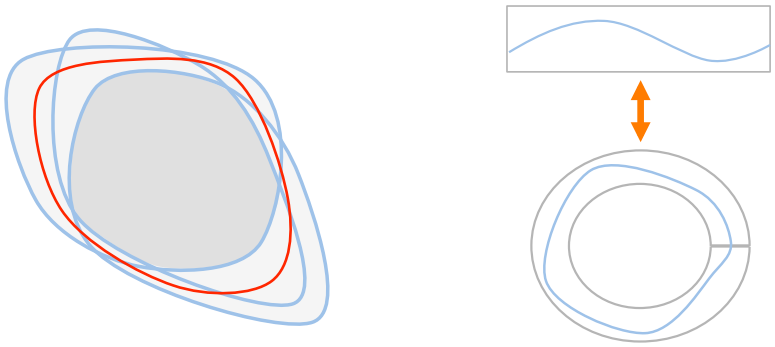


Figure 1.7: a) A band constructed by three blue contours. The band is the region delimited by the intersection and union of the contours. The red contour is entirely contained in the band, i.e. all points on the contour form a subset of the band. b) An example of sBD converging to BD. Picture taken from Whitaker et al. [37].

2 Overview

As we have seen in the introduction to RFA treatment in Section 1.2, visualizing only a single RFA simulation result may give an inaccurate representation of the actual case due to uncertainties in the simulation parametrization. Thus, the first extension to the RFA Guardian the Parameter Uncertainty plugin provides, is a module to generate an ensemble of simulations by performing exhaustive parameter sampling. The implementation of the Parameter Sampling module is given in Section 6.

Considering the cluttered visualization of a simulation ensemble presented in Figure 1.3b, the Parameter Uncertainty plugin aims to present the ensemble using the contour boxplot technique, introduced in Section 1.3.2. However, the boxplot is not generated with set/contour band depth measures developed by Whitaker et al. [37], but by employing the mBD measure. This is accomplished by sampling the whole ensemble with outward pointing rays. To accelerate the band depth computation, new algorithms for fast BD and mBD computation are introduced in Section 4. Using the introduced algorithms, real time generation and rendering of the boxplot can be achieved. The whole process to establish the contour boxplot is given in detail in Section 7.4.

Due to the heat sink effect, the behavior of lesions around blood vessels is of great importance. Thus, the Parameter Uncertainty plugin provides several means to investigate the behavior around vessels: on top of the contour boxplot, the estimated probability of a blood vessel slice being ablated on a certain 2D view can be shown and the user is able to only display simulation results that fail to ablate a specific vessel, as well as results that fully ablate the vessel in question. Vessel related functionalities of the Parameter Uncertainty plugin are described in Section 7.5.

The contour boxplot is a global visualization technique using the whole simulation ensemble. While this gives a concise view of the whole ensemble, local characteristics of single contours are lost. Hence, the visualization technique provides automatic grouping of locally similarly behaving simulation results, where the probability of a result being included in a certain group can be displayed. This gives the user the possibility to investigate the behavior of contours in a local area. Implementation details are given in Section 7.6.

In summary, the main contributions of this thesis are:

- Developing an approach for fast $BD_{n,2}$ and mBD computation for functional bands which are used in the Parameter Uncertainty plugin for contour boxplot generation. The algorithms are explained in Part II of the thesis.

- Introducing the Parameter Uncertainty plugin for the RFA Guardian. The plugin is capable of generating RFA simulation results using exhaustive parameter sampling of two parameters. Furthermore, the Ensemble Visualization module of the plugin is able to visualize the generated simulation ensemble using the concept of contour boxplots. Contour ordering is established through the aforementioned fast mBD algorithm. In addition, the visualization technique focuses on the behavior of contours around blood vessels and is capable of performing local grouping of simulation results based on contour density. The implementation of the Parameter Uncertainty plugin is given in Part III.

Part II
Fast functional band depth computation

II Fast functional band depth computation

Due to the increasing computational demand for higher band orders, a few methods for fast BD and mBD computation have emerged in the past years. Kwon and Ouyang [42] list the steps required to obtain BD, mBD and their computational complexities using a trivial, intuitive approach: calculating all bands and testing each function on its containment. Sampling the n functions with m steps, this takes $\mathcal{O}(mn^3)$ and $\mathcal{O}(mn^4)$ time for $BD_{n,2}/mBD_{n,2}$ and $BD_{n,3}/mBD_{n,3}$ respectively. As these time complexities are not suitable for real time visualization of an contour ensemble of arbitrary size, the visualization module of the plugin presented in this thesis uses an acceleration technique for the BD and mBD computation. This section first gives insight into two already established methods for fast BD and mBD computation before new approaches are introduced in Section 4.

3 Established methods for fast band depth computation

3.1 Acceleration through data resampling

López-Pintado and Jornsten [43] proposed a simple data resampling method for fast computation of BD and mBD (as well as other depth measures). In this method, the data (e.g. an ensemble of functions) X is divided randomly into K roughly equal sized subsets X_k. For each subset, the desired band depths of the included objects are computed independently. The resampling-based depth of curve x is then defined as the average of the depths of x in all subsets, where $D(x|K_k)$ refers to the depth of curve x in subset X_k [43]:

$$D_r(x|X) = \frac{1}{K} \sum_{k=1}^{K} D(x|K_k) \tag{7}$$

While providing equivalent results for mBD computation, López-Pintado and Jornsten [43] pointed out that deviations exist for BD computation, where results for the deepest data sets were not accurate due to the occurrence of many ties in subsets with smaller size.

The computational savings of this method depend on the subset size; thus, on the selection of K. Smaller samples lead to a decrease in the number of bands to compute. This method may be used as an extension for

the fast mBD method introduced in this thesis, to accomplish even larger computational gains. However, in most applications, is not necessary to use the resampling algorithm as the approach introduced in this thesis is already fast enough.

3.2 Acceleration with an exact approach

A fast technique for exact $BD_{n,2}$ and $mBD_{n,2}$ computation was proposed by Sun et al. [44]. First, with the help of an $n \cdot m$ rank matrix, where m is the number of sampling points and n is the number of functions, all functions are ordered from smallest to largest value for each sampling point. The BD and mBD then calculate as follows:

For the BD computation, the maximum rank R_{max} and minimum rank R_{min} for a given function f are extracted from the rank matrix. Furthermore, the number of functions completely below and above f are denoted as $n_b = R_{min} - 1$ and $n_a = n - R_{max}$ respectively. $BD_{n,2}$ then calculates as stated in Equation 8. For mBD computation, n_b and n_a are evaluated for every sample point and the proportion a function resides inside a band is taken into account. Refer to Sun et al. [44] for pseudo code implementations of both algorithms.

$$BD_{n,2} = \binom{n}{2}^{-1} (n_b \cdot n_a + n - 1) \tag{8}$$

This method reduces the time complexities of both $BD_{n,2}$ and $mBD_{n,2}$ to $\mathcal{O}(mn \log n)$ [42]. However, the algorithms also have a downside: Hong et al. [45] pointed out that the algorithms do not consider ranking ties during sorting, making it unsuitable for binary indicator functions. Consequently, Hong et al. [45] developed a fast algorithm for band depth calculation of binary indicator functions that runs in $\mathcal{O}(mn)$ time.

Additionally, Kwon and Ouyang [42] stated that the method for computing BD is wrong. Unfortunately, this observation is confirmed by tests performed during developing the Parameter Uncertainty plugin. Consider the function ensemble in Figure 3.1. The Ensemble consists of five continuous functions sampled at three locations denoted by circles. All BD values are 0.4, except for the orange function, which has a BD value of 0.5. The computation with the algorithm according to Sun et al. [44] is depicted in Figure 3.2. As visible in the figure, the algorithm produces wrong results for the blue function and the orange function. The problem with the algorithm in this particular example is that the number of functions completely above the orange and blue curves are overestimated. Consequently, the number

of bands that include these two functions is too high. In light of this fact, Section 4.2 introduces a different approach for $BD_{n,2}$ computation that still exhibits reasonable performance.

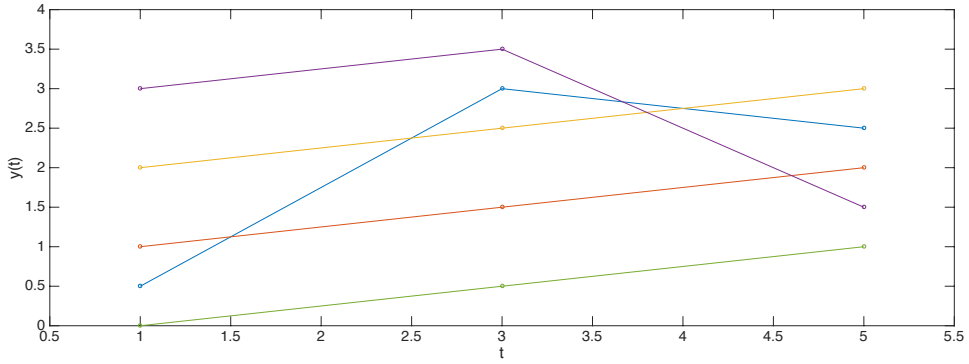


Figure 3.1: Function ensemble consisting of five different continuous functions. Sampling locations are marked by a circle. The band depths of the functions are: $BD_{purple} = 0.4$, $BD_{yellow} = 0.4$, $BD_{orange} = 0.5$, $BD_{blue} = 0.4$, $BD_{green} = 0.4$.

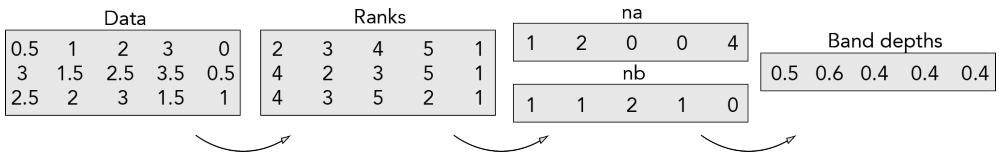


Figure 3.2: Steps of $BD_{n,2}$ computation using the algorithm presented by Sun et al. [44]. The $m \cdot n$ data matrix contains the sampled function values of the ensemble in Figure 3.1. First, the function ranks are computed for each time point. Second, the n_a and n_b vectors are calculated. Each element in the vectors should contain the number of functions completely above and below the corresponding function respectively. However, this is not the case for the blue function and the orange function in n_a as both values are too high. This error propagates and leads to wrong BD values for the mentioned functions.

4 New approaches

This section introduces two new approaches for fast mBD and $BD_{n,2}$ computation which will be used in the Parameter Uncertainty plugin in order to achieve real-time visualization of a simulation ensemble. The evaluation of the performance of the new algorithms is given in Section 8.1.

4.1 Modified band depth

The mBD of a function is computed by investigating the normalized fraction in which the function resides in every band. The intuitive approach entails calculating every possible band and testing each function for inclusion. The problem with this approach is the computation of all possible bands which has the complexity $\mathcal{O}(n^k)$ for $k < n - k$. Because every function has to be tested for inclusion and this test has to be carried out for every sampling step, the total cost adds up to $\mathcal{O}(mn^{k+1})$.

The key to improving the performance of this process is to address the elephant in the room: the computation of each individual band. A functions mBD value does not imply in *which* bands the function resides, but is rather a collective measure of the summed up fraction that a function is included in all possible bands of the ensemble. In other words, the mBD measure does not tell you about the fraction of inclusion of a function with respect to a certain band, it only gives you the average fraction across all bands. Thus, the information gain we get from computing all different bands has no relevance for the result.

Consider a certain time point t and the sampled values of all functions in the ensemble which consists of n total functions. Similar to the method presented in Sun et al. [44], a rank matrix is populated with the ranks of every function for each sample point. The rank of the i-th function in the sample point t is denoted by $r_{i,t}$. It is important to note that depending on how many ties there are between functions, the maximum rank for a sample point $r_{max,t}$ is a number between $[1; n]$. This means that whenever two or more functions values are equal, their ranks are equal as well.

Now, to calculate the mBD value for each function, the average fraction of inclusion across all bands has to be computed. For a given band order k and the number of functions n, the maximum number of bands a function is included in is given by the binomial coefficient $\binom{n}{k}$. Since both n and k are

positive numbers, the binomial coefficient can be expressed as:

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!\cdot(n-k)!} & n \geq k \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

To obtain mBD for a given function, we need to calculate in how many bands the function is included in each sample point and averaging the result by the number of total possible bands and sample steps. The other way to address the problem is to compute the number of bands that do not include the function for each sample point and subtracting it from the total number of bands possible before averaging the result. The bands a particular function is not included in, are the ones formed by functions that either are ranked higher or lower than the particular function. For a function with rank $r_{i,t}$, the number of functions with higher rank is $r_{a,i,t} = r_{max,t} - r_{i,t}$ and the number of functions with lower rank is $r_{b,i,t} = r_{i,t} - 1$. Consequently, the total number of bands that do not include the function equals the number of bands formed by the $r_{a,i,t}$ functions and the $r_{b,i,t}$ functions. Figure 4.1 illustrates this circumstance. Subtracting this number from the total number of bands and averaging the result yields the mBD for this function:

$$mBD_{n,k,i} = \frac{1}{m \cdot \binom{n}{k}} \sum_{t=1}^{m} \left(\binom{n}{k} - \binom{r_{a,i,t}}{k} - \binom{r_{b,i,t}}{k} \right) \tag{10}$$

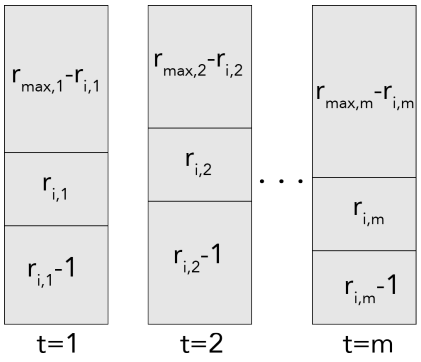


Figure 4.1: Illustration of three columns of the rank matrix corresponding to three sampling points. When considering a particular function i, the number of functions ranked above and below this function within the scope of a sampling point have direct influence on the number of bands in which this function is included.

Using this method to compute mBD reduces the computational complexity from $\mathcal{O}(mn^{k+1})$ to $\mathcal{O}(mn \log n)$. Note that the complexity does not depend on the band order k any more. However, when considering all subbands, i.e. for a given band order lower band orders are considered as well, the complexity increases to $\mathcal{O}(kmn \log n)$:

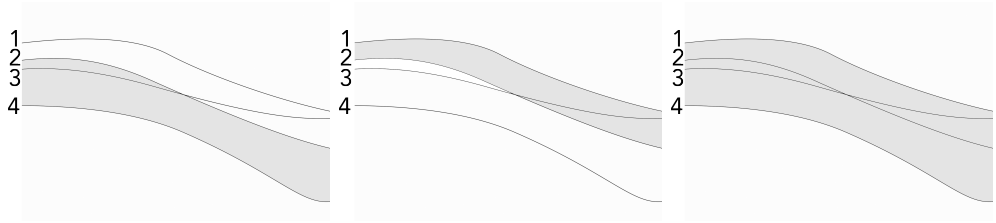
$$mBD_{n,k,i} = \frac{1}{m} \sum_{j=2}^{k} \binom{n}{j}^{-1} \sum_{t=1}^{m} \left(\binom{n}{j} - \binom{r_{a,i,t}}{j} - \binom{r_{b,i,t}}{j} \right) \tag{11}$$

Equation 11 can be viewed as a generalization of the $mBD_{n,2}$ algorithm proposed by Sun et al. [44], as it also employs a rank matrix and counts functions ranked higher and lower as the function for which mBD is computed. An advantage of Equation 11 lies in the fact that the mBD using bands of any order can be computed (without sacrificing computation time). Furthermore, ties in function values are treated properly. However, with no ties present and the band order $k = 2$, the approach by Sun et al. [44] and Equation 11 yield the same results. The Application of this mBD algorithm in the Parameter Uncertainty plugin is detailed in Section 7.4.4

4.2 Band depth

An algorithm for fast $BD_{n,2}$ computation can be found using similar considerations as in the previous section. As before, the number of bands that contain the function needs to be calculated. Unlike computing the mBD however, if a band contains the current function, but the function escapes the band in any other sample point, the band does not contribute to the functions BD. This makes knowledge about the behavior of each function in the ensemble across all sampling points with regards to the current function inevitable.

As in the case of mBD computation, bands formed by functions located either above or below the current function do not contribute to the functions BD. Additionally, functions crossing the current function do not contribute either. That means that only bands made by a function fully below and a function fully above the current function contribute to BD as illustrated in Figure 4.2.



(a) Band of f_2 and f_4 (b) Band of f_1 and f_2 (c) Band of f_1 and f_4

Figure 4.2: Example of three bands in an ensemble of four functions. Assume that $BD_{n,2}$ of function f_3 needs to be computed. Bands built with the function f_2 (which crosses f_3) never fully contain f_3 (a) & (b). Only the band of f_1 and f_4 - built by functions fully above and fully below f_3 contains f_3 (c).

Defining the number of functions that cross f_i as c_i and the number of functions fully above and below f_i as a_i and b_i respectively, the $BD_{n,2}$ of f_i can be computed in the following way:

$$BD_{n,2,i} = \binom{n}{2}^{-1}\left(\binom{n}{2} - \binom{a_i + c_i}{2} - \binom{b_i + c_i}{2} + \binom{c_i}{2}\right) \tag{12}$$

Note that in Equation 12 the term $\binom{c_i}{2}$ is added. Otherwise combinations of crossing functions would have been subtracted twice in the equation. In the implementation of this method, a rank matrix is utilized again. However, in order to get a_i, b_i and c_i, two additional matrices of size $n \cdot n$ are utilized. These matrices, termed F_a and F_b record functions ranked higher and lower than the current function. First, the matrices are populated with zeros. Whenever a function f_j has a higher rank than the current function f_i, $F_a(i, j)$ is set to one. The same holds for functions with lower rank than f_i and F_b. Summation over all column entries for a particular row i gives the number of functions that are below and/or above function f_i. The number of functions that cross f_i can be computed by a logical `AND` operation of F_a and F_b and a subsequent summation over the columns of the resulting matrix for row i. Finally, a_i and b_i are computed by subtracting c_i from the number of functions that are either above or below f_i. Summarized:

$$c_i = \sum_{l=1}^{j} (F_a(i,l) \wedge F_b(i,l)) \tag{13}$$

$$a_i = \sum_{l=1}^{j} F_a(i,l) - c_i \tag{14}$$

$$b_i = \sum_{l=1}^{j} F_b(i,l) - c_i \tag{15}$$

With Equation 12, the computation of $BD_{n,2}$ takes $\mathcal{O}(mn^2)$. Although this is larger than the complexity of the algorithm proposed by Sun et al. [44], the algorithm produces exact results for ties too, while still being faster than the trivial approach, which has $\mathcal{O}(mn^3)$ complexity. Furthermore, the algorithm works with ensembles for which the algorithm of Sun et al. [44] produces wrong results, refer to Section 3.2. Figure 4.3 demonstrates the algorithm using the ensemble presented in Figure 3.2. For a performance comparison of the BD and the mBD algorithm with the trivial approach, refer to Section 8.1.

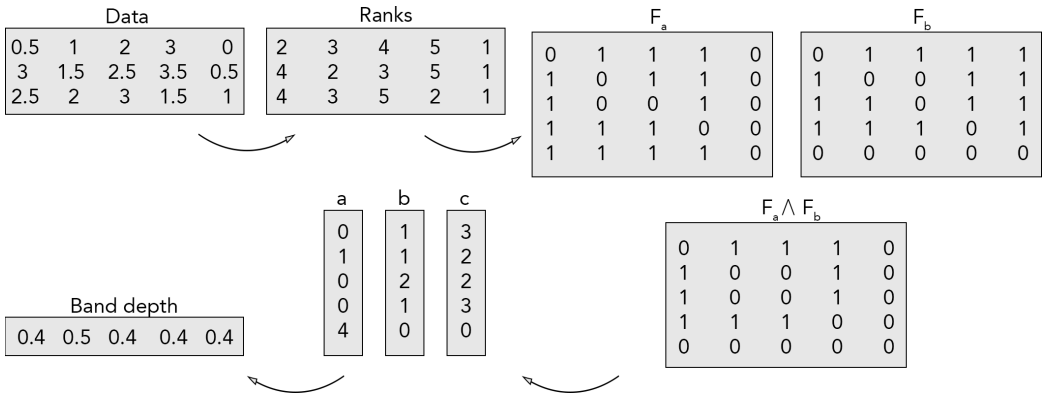


Figure 4.3: Illustration of the $BD_{n,2}$ computation using a new approach. The data matrix corresponds to the example in Figure 3.1. The first step is to generate a matrix of function ranks. Second, the matrices F_a, F_b and $F_a(i,l) \wedge F_b(i,l)$ are generated by recording functions with higher rank and lower rank respectively. Subsequently, the vectors a, b and c can be computed using equations 13-15. Finally, $BD_{n,2}$ is obtained by using Equation 12.

Part III
The Parameter Uncertainty plugin

III The Parameter Uncertainty plugin

The goal of the Parameter Uncertainty plugin is to extend the RFA Guardian with the capability to visualize the uncertainty of the RFA simulation outcomes resulting from different parameterizations. As described in Section 1.2.4, the outcome of the treatment depends on several treatment-specific parameters. Those parameters need to be provided to the simulation routine of the RFA Guardian in order to perform the simulation. Changing parameters leads to different outcomes, and the different shapes and extents of those generated lesions need to be visualized in a non-cluttered manner in order to get an insight in the statistic relevance of the entered parameters.

The Parameter Uncertainty plugin is separated into two independent modules. One module for sampling the input parameter space of two simulation parameters and another module designed for the actual visualization of simulation ensembles. Most of the time, the user interacts with both parts in a linear fashion: generating the sampling space first and then visualizing it. However, this is not always the case. Sometimes previously sampled parameters are loaded and visualized without interacting with the parameter sampling interface at all. At other times, after having seen the uncertainty visualization, another sampling step is carried out to acquire additional data for the visualization. Designing both modules of the plugin in an independent manner enables this back-and-forth interaction.

Before the plugin itself is described in detail, the RFA Guardian is introduced. Afterwards, following the structure of the plugin, this chapter will describe several important aspects concerning the parameter sampling part of the plugin. Secondly, the larger part of the chapter deals with producing the visualization for displaying the uncertainty in the simulation results where the mBD algorithm introduced in Section 4 will be used to generate a contour boxplot.

5 RFA Guardian

5.1 Workflow

The RFA Guardian is an RFA simulation tool for assisting interventional radiologists in treatment of liver tumors. The tool provides assistance in planning, guidance and validation of treatments. Treatment assistance is divided into three distinct temporal phases: the goal of the *pre-interventional* phase is to create a patient model: first, already available patient images are loaded into the software and automatic segmentation of the liver, vessel tree

and semi-automatic tumor segmentation is carried out. Second, volumetric meshing of the segmented data takes place. In the *peri-interventional* phase, the tips of the probe electrode are defined and registered into the patient model. When finished, simulation parameters are defined and the simulation of the treatment can be performed. Finally, in the *post-interventional* phase the treatment result is compared with the simulation result [7]. A flow chart of the three phases is depicted in Figure 5.1.

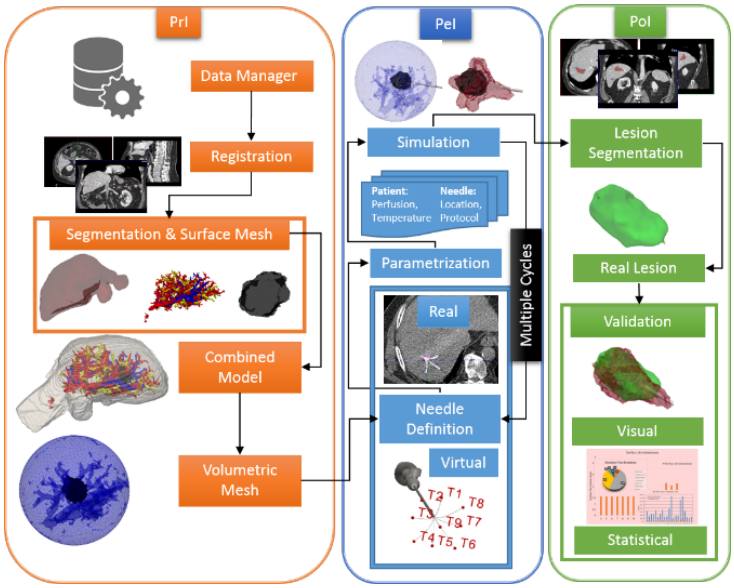


Figure 5.1: Technical workflow of an RFA treatment in the RFA Guardian. Pre-interventional (PrI), peri-interventional (PeI) and post-interventional (PoI) phases are outlined in distinct colors [46].

5.2 Frameworks and user interface

The RFA Guardian is built upon the "Medical Imaging Interaction Toolkit" (MITK) [47], an open source software framework that combines the "Insight Segmentation and Registration Toolkit" (ITK) [48] with the "Visualization Toolkit" (VTK) [49] and provides a platform for clinical applications and research. The incorporation of ITK and VTK allows developers to access a large number of algorithms and data structures potentially useful for medical applications. As will be further elaborated in Section 7, several VTK filters are utilized in the Uncertainty Visualization module as well. Note that all referenced VTK filters and data structures in this thesis are documented in the VTK API documentation at [50].

MITK based applications are often built on top of the MITK "workbench", and the RFA Guardian is no exception. The workbench offers a convenient, extendable user interface (UI) that, in its standard configuration, contains three orthogonal[5] windows for a 2D view in the axial, sagittal and coronal anatomic plane, as well as a 3D window for volumetric representations. The cursors in the 2D views can be used to focus on a specific part of the body. Each crosshair in a window represents the intersection of both of the perpendicular planes. In order to select a different slice of the current view, the user may scroll through the plane normal, and different slices of the body will be displayed (Figure 5.2). The UI of the RFA Guardian consists of three main parts: a database of all loaded and generated data, the window view and controls for interaction with plugins. A screenshot of the UI is shown in Figure 5.3.

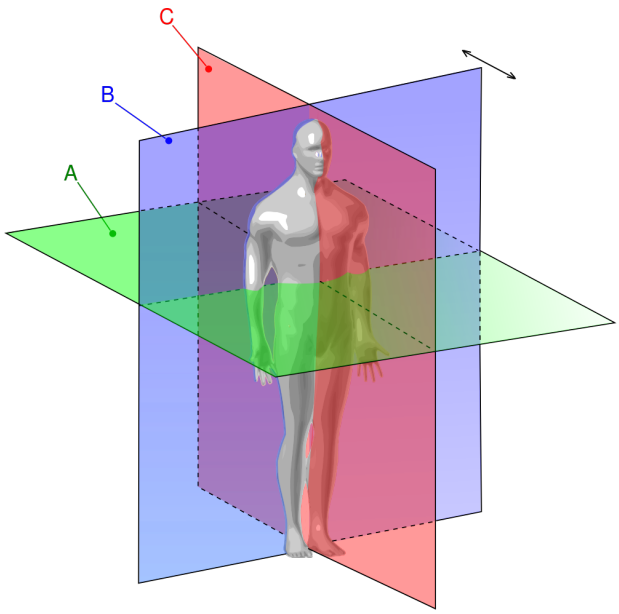


Figure 5.2: Schematic drawing of the orthogonal anatomic planes. Visible are the axial plane (A), the coronal plane (B) and the sagittal plane (C). In the MITK workbench, the intersection of two planes represents the cursor location in the third plane. The user may scroll through any of the views in order to select different slices of the current plane, as depicted by the black double arrow in the coronal plane. Adapted from Mrabet [51].

[5]The 2D windows are actually only orthogonal in the default view and can be adjusted to represent arbitrary planes.

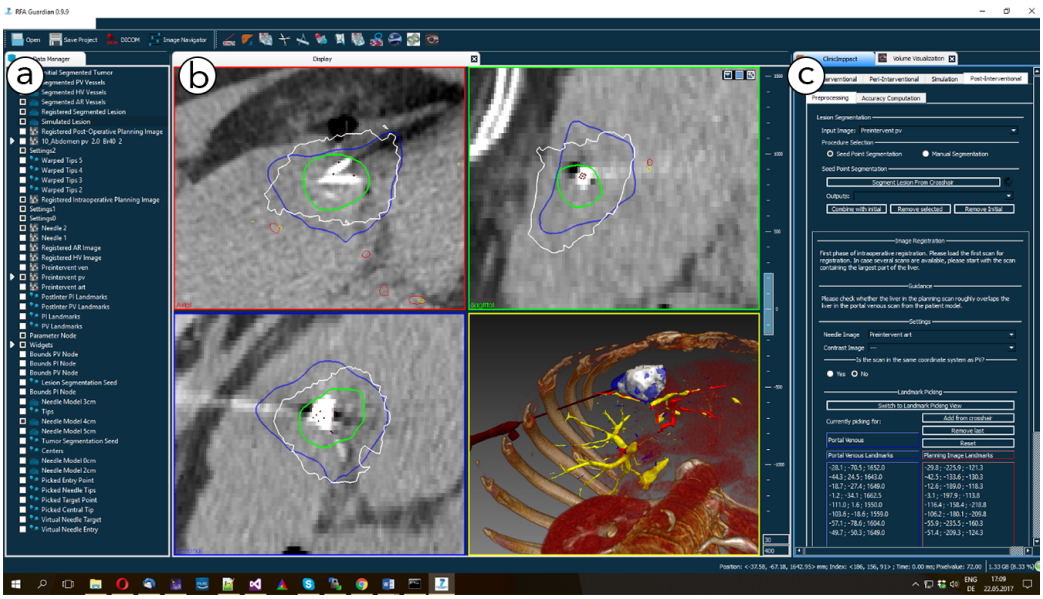


Figure 5.3: User interface of the RFA Guardian. The view is split up into three main parts. First, the MITK data storage of all data sources (e.g., patient images or meshes among others) (a). Second, the visualization of selected data in four views (b): axial view in upper-left corner; sagittal view in upper-right corner; coronal view in lower-left corner and a 3D view in lower-right corner, in which an RFA probe can be seen punctuating a liver tumor. Third, a pane for plugin interaction (c) [52].

6 The Parameter Sampling module

6.1 User interface

The UI of the Parameter Sampling module allows the user to select two parameters. For each parameter, a sampling method, the number of samples and parameter specific options can be set. Additionally, the UI contains a log window and a progress bar for providing feedback of the simulation progress to the user. The UI is located at the right pane of the RFA Guardian (refer to Figure 5.3). A depiction of the interface can be seen in Figure 6.1. The individual parts of the UI are elaborated in more detail in the following sections.

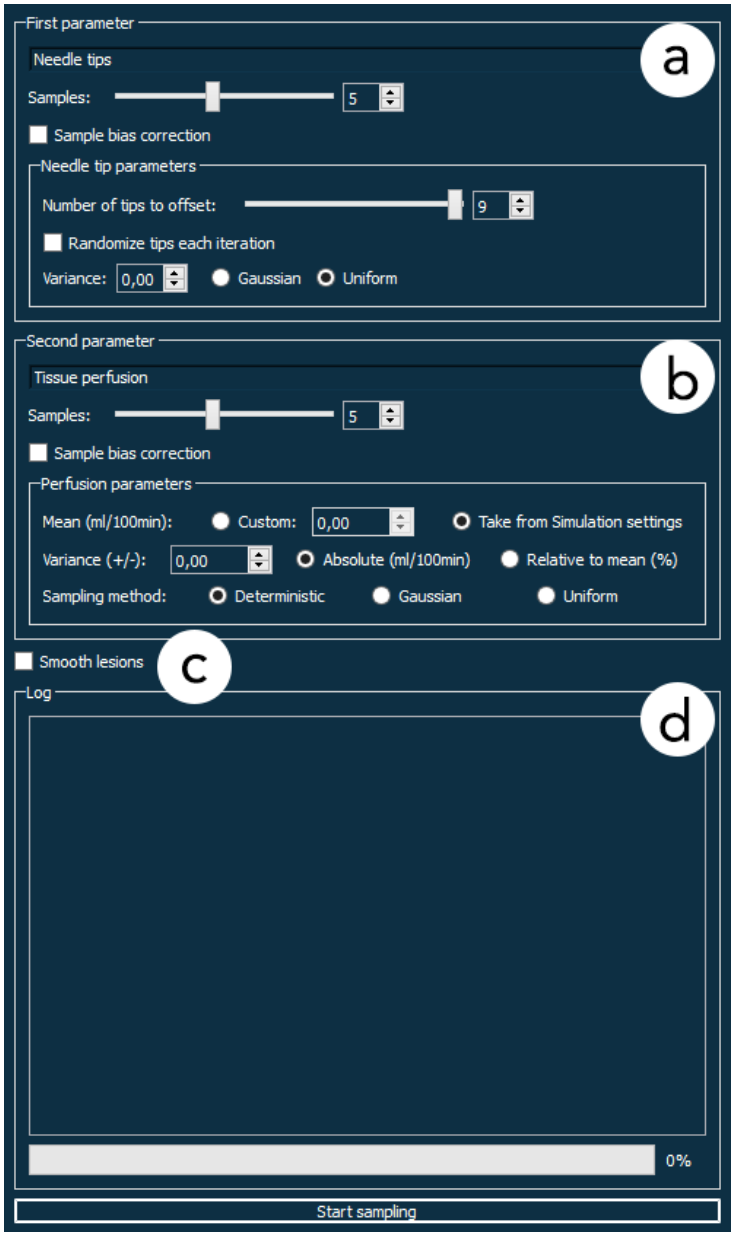


Figure 6.1: User interface of the Parameter Sampling module. Two parameters can be selected by the user. Sampling properties such as the number of samples, the parameter distribution and sample bias correction, as well as parameter specific options can be set independently for the first parameter (a) and the second parameter (b). The UI additionally includes the post-processing option to smooth the simulation results (c). Finally, a log window and progress bar for the current sampling procedure is shown in (d).

6.2 Workflow

In order to perform the parameter sampling, the user may select two parameters, a sampling method and the number of samples for each parameter. Specific settings may be available depending on the chosen parameter type. In the default setting, three parameters are available: tissue perfusion, tumor perfusion and the location of the electrode probe tips; however, it is easy to extend the module with additional parameters to sample from. The module performs exhaustive sampling and produces a simulation ensemble of $n \cdot m$ simulation results in total, where n is the number of samples in the first parameter and m is the number of samples in the second parameter. For each of the $n \cdot m$ parameter combinations, the simulation routine of the RFA Guardian is invoked and performs a single simulation. The simulation result is read back to the parameter sampling module, and either the next simulation iteration takes place, or the module halts. In either case, the simulation results are saved to the MITK data storage together with the parameter types and values of the sampled tuple that was injected into the simulator to generate the particular prediction. This allows for saving the data on the filesystem and loading it later without losing information about the used parameters. Due to the fact that each additional parameter leads to an exponential increase in simulation invocations, the module restricts sampling to two parameters. The workflow of generating a simulation ensemble is shown in Figure 6.2.

6.3 Sampling methods and properties

There are three available sampling methods the user might choose from. The statistical methods include generating *Gaussian distribution*s or *uniform distribution*s and the deterministic approach samples a parameter in *fixed intervals*. Since both simulation parameters are sampled independently, the user might choose to generate a Gaussian distribution of one parameter and a uniform distribution of the other or even mix the deterministic approach with a statistical one. Which sampling method one should use depends on the knowledge about the specific parameter. If the user is fairly certain about the value of a parameter, a Gaussian distribution might be the right choice, since about 68.3% of the sampling values will lie within the region of $\pm$ one standard deviation around the mean [53, p.384]. Hence, the resulting simulation ensemble will give a well resolved estimation about this region. If, on the other side, the parameter value is relatively uncertain, a uniform distribution or the deterministic approach might be more appropriate. Both methods differ in the sense that the uniform distribution is generated by

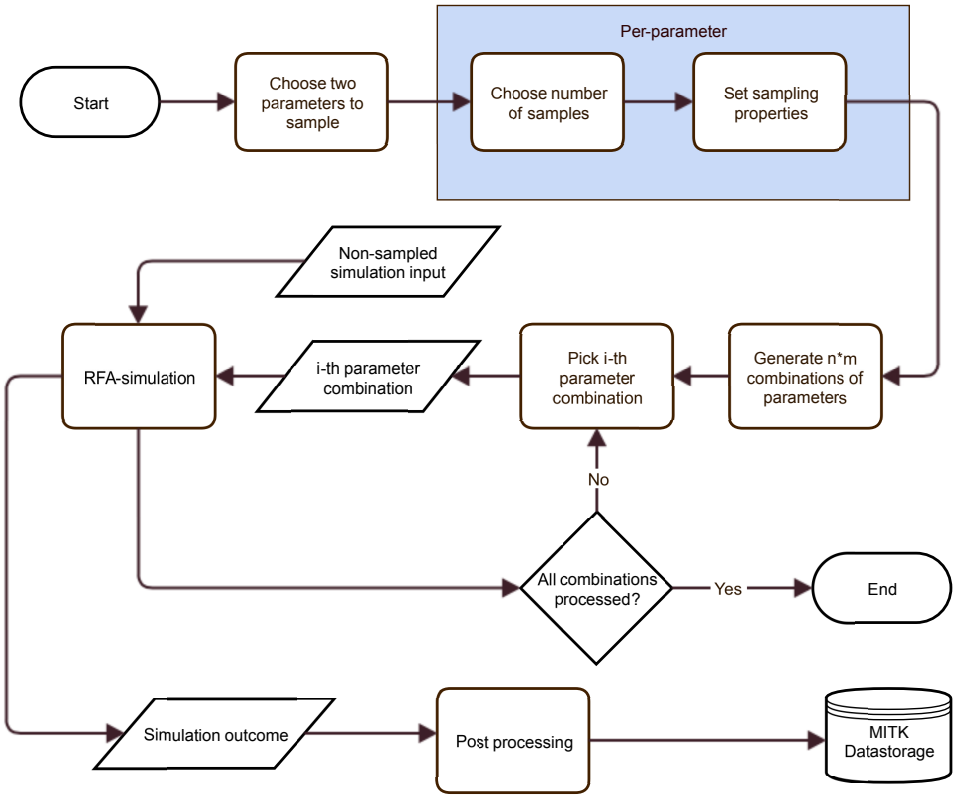


Figure 6.2: Flowchart depicting the procedure of generating a simulation ensemble.

using mean and variance values, whereas the deterministic method requires the specification of a minimum and maximum value. Hence, if the user has more knowledge about the range of the parameter than its variability, the deterministic approach may be better suited to produce a sensible output. The mathematical definitions of the probability density function of the statistical sampling methods are listed in Equations 16 and 17 where μ represents the mean and σ the standard deviation [53, p.331&371].

Gaussian distribution:

$$f(x) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{16}$$

Uniform distribution:

$$f(x) = \begin{cases} \frac{1}{2\sqrt{3}\sigma} & \mu - \sqrt{3}\sigma \leq x \leq \mu + \sqrt{3}\sigma \\ 0 & \text{else} \end{cases} \tag{17}$$

In order to produce distributions that correspond to those definitions for a given μ and σ, the C++11 standard library header *random* was used. This header contains several random number generators (RNGs), as well as mappings for common statistical distributions. Generally, most RNGs produce uniformly distributed pseudo-random numbers in the unit interval $(0; 1)$ (however, RNGs usually produce integer values first and then scale them back to the unit interval). Those numbers then need to be mapped onto the specific distribution that needs to be realized. For the statistical sampling method utilized in the Parameter Sampling module, the "Mersenne Twister" (MT19937 to be precise) RNG is used. Although its initial release back in 1998 had some flaws, the MT19937 is a widely-used generator nowadays with solid distributional properties [54, p.5&42]. For the sake of brevity, the actual algorithm of the MT19937 is not listed here, but the interested reader may refer to the original publication of the algorithm [55] and a throughout review of the MT19937 and variants for single instruction, multiple data (SIMD) architecture [56].

For mapping the uniform distribution in the unit interval to the distributions defined in Equations 16 and 17, different approaches have to be taken. The mapping for the user defined uniform distribution is simply achieved with a scaling function: Let $f_{a,b}(x)$ be a specific realization of a random variable, uniformly distributed in $[a; b]$. Furthermore, consider a transformation $T_{c,d}$ which scales and translates the realization to the interval $[c; d]$, and let $f_{c,d}(x)$ be the scaled version of $f_{a,b}(x)$. Thus, the relationship between $f_{a,b}$ and $f_{c,d}$ can be expressed as:

$$f_{c,d}(x) = T_{c,d}\{f_{a,b}(x)\} = \frac{d-c}{b-a} \cdot (f_{a,b}(x) - a) + c \tag{18}$$

As a consequence, $f_{c,d}(x)$ can be viewed as a specific realization of a different random variable, uniformly distributed in the interval $[c; d]$ with mean: $\mu = \frac{c+d}{2}$ and variance: $\sigma^2 = \frac{1}{12}(d-c)^2$. Hence, in the module for transforming the uniform distribution of the RNG $R(0, 1)$ into the desired distribution $V(c, d)$ defined by the user, this simply means transforming R by $T_{c,d}$ with $c = \mu_V - \sqrt{3}\sigma_V$ and $d = \mu_V + \sqrt{3}\sigma_V$.

For the Gaussian distribution, the mapping is not as trivial as it is in the uniform distributed case, and several approaches exist. For example, the Box-Muller transform (one of the "polar methods") applies a polar transformation of two uniform, independently distributed random numbers $R_1(0, 1)$

and $R_2(0,1)$ in order to get two independently distributed random variables in the standard normal distribution $N_1(0,1)$ and $N_2(0,1)$:

$$N_1(0,1) = \sqrt{-2\log(R_1)} \cdot \cos(2\pi R_2) \tag{19}$$

$$N_2(0,1) = \sqrt{-2\log(R_1)} \cdot \sin(2\pi R_2) \tag{20}$$

Since the Box-Muller method is fairly slow, an acceleration is provided by other polar methods, for example the acceptance/rejection algorithm [54, p.171-173]. However, the mappings of the Gaussian and uniform distributions do not have to be implemented by hand, since the C++11 *random* header already includes convenient mapping functions for generating the desired distributions.

Regardless of which statistical parameter distribution is generated, a sample bias correction can be performed in order to align the sample mean with the mean provided by the statistical distribution. For many samples of a parameter, the deviations of the sample mean and the mean of the generated distribution will vanish, as the sample mean is a consistent estimator. This means that, with an increasing number of samples, the sample mean converges to the provided mean, which makes it asymptotically unbiased. Mathematically expressed, in a sample $S_n := \{X_1, X_2, ..., X_n\}$ consisting of n independently distributed random variables X_i, produced by an RNG and mapped to a distribution with predefined mean μ, the sample expectation $\mathbb{E}$ has the following asymptotic property:

$$\mathbb{E}\{\lim_{n\to\infty} S_n\} = \mu \tag{21}$$

However, this property is only approximated well, if n is a fairly large number, and, especially for a small sample size, the actual mean of the produced sample may diverge from the predefined mean in an undesirable manner. The sample bias correction method simply re-aligns the sample mean with the desired mean, in order to get a parameter distribution around the specified mean. A pseudocode implementation is given in Listing 1.

Listing 1: Pseudocode of sample bias correction in the parameter sampling module

```
function sampleBiasCorrection
  set sample_mean = 0

  for each of the values in sample S:
    add value to sample_mean
  end

  divide sample_mean by sample size
  set correction_factor = provided_mean  - sample_mean

  for each of the values in sample S:
    add correction_factor to value
  end

end
```

6.4 Parameter dependent sampling properties

Depending on which parameters are to be sampled, different sampling options are available. For sampling the tips of the probe electrode, the mean of each tip is automatically obtained from the RFA Guardian and corresponds to the current location of the registered tips. The user has to specify the variance and how many needle tips are to be sampled - this can be a number from zero to nine, which is the total number of tips available. The uncertainty sampling module will then pick the specified number of tips randomly from all nine available tips. Optionally, re-randomization of the tip selection can be performed on every invocation. The location of each tip that is processed in the current pass is varied along the x, y, and z coordinate, where each component is treated as an independent random variable. Thus, samples of tip locations are taken from a probability distribution in $\mathbb{R}^3$.

For tissue perfusion and tumor perfusion, the user may specify a mean or take the mean from the simulation settings as well. The variance can be selected relative to the mean or as an absolute value. Since perfusion is a one-dimensional parameter, the probability distribution of the perfusion parameters is in $\mathbb{R}^1$ as well.

After all options have been set, the parameters are gathered and injected for simulation, and the simulation routine processes each of the $n \cdot m$ parameter combinations iteratively. Feedback is provided to the user through the log window and the progress bar (Figure 6.1d). Once the simulation routine has finished, post processing takes place.

6.5 Post processing

Before a simulation result is saved in the MITK data storage, several post-processing steps are performed. This ensures being able to save the data onto the filesystem and loading it in another session without loss of information about the parameterization of this specific lesion. Additionally, the sampling module provides the possibility to smooth the resulting lesion, in order to enhance the visualization of the simulation ensemble in the visualization module. The data type of a simulation outcome is *vtkPolyData*, a universal geometry data type of VTK, able to hold 3D vertices, lines, polygons, and/or triangle strips. This data is wrapped in an MITK container called *DataNode*, which has the advantage of assigning names to the underlying data and enabling it to add customizable properties to the stored data. Properties can be basic data types like strings, integers, floating point numbers, and others. The MITK *DataNode* can be directly saved on the filesystem through the MITK workbench. Utilizing these functionalities, the sequence of post-processing steps is as follows:

- Giving the data a unique name, in order to recognize it later as a data structure that was produced by the parameter sampling module,
- saving the values and types of the parameters that were used to generate the simulation result,
- (optional) smoothing the VTK data with the *vtkSmoothPolyDataFilter*,
- saving the whole data node to the MITK data storage.

Smoothing of the VTK data is done using a filter called the *vtkSmoothPolyDataFilter*. This algorithm uses Laplacian filtering in order to generate a mesh with evenly distributed vertices. Basically, the filter gathers directly connected vertices of each vertex in the first step. Then, for each vertex, the average position of its neighborhood is calculated, and the vertex is moved to the resulting location. This procedure is repeated several times until the mesh is sufficiently smooth.

7 The Ensemble Visualization module

Before the actual module is described, Section 7.1 will give relevant introductory information about the OpenGL rendering pipeline. Subsequently, computational geometry algorithms used throughout the module are described in Section 7.2.

7.1 The Graphics Processing Unit and OpenGL

Serving as massively parallel processors in many consumer devices, such as desktop computers or mobile devices, GPUs have seen vast improvements in terms of performance and capabilities over the last decades. While initially being restricted to a fixed function, single purpose pipeline, modern-day GPUs offer a flexible, programmable rendering pipeline and general-purpose computation abilities [57]. Aside from their graphics rendering capabilities, the advantages of general purpose computation on GPUs (GPGPU) are utilized in many different fields that can benefit from their massive parallel functionalities, e.g., medical physics [58] or machine learning [59, 60].

In the uncertainty visualization technique presented in this thesis, the Open Graphics Library (OpenGL) is used to make use of hardware-accelerated rendering and GPGPU. OpenGL is a platform independent, open source API for interacting and controlling the graphics subsystem on various devices. Using a standardized interface for interacting with the GPU has the advantage of adding an abstraction layer to the specific requirements of the underlying platform [61, p.45].

7.1.1 The OpenGL rendering pipeline

In order to render a scene, OpenGL provides a rendering pipeline that consists of both fixed-function stages, as well as programmable stages. For programmable stages, the developer might provide code written in the OpenGL Shading Language, a high-level, C-style programming language for executing shader programs on GPUs. When issuing a drawing command, vertices pass the following stages of the pipeline on their way to a framebuffer: first, the vertices are fetched automatically to the input of the *vertex shader*, the first programmable part of the pipeline. This shader is necessary to perform per-vertex processing tasks, such as applying transformations (rotations, scaling, etc...) to each input vertex. After the vertex processing stage is finished, tessellation takes place - the task of splitting primitives into smaller pieces: first, the programmable *tessellation control shader* is invoked for defining the

tessellation-level of a particular patch (a collection of a variable number of vertices, by default three). The output of this shader stage goes directly into the fixed-function tessellator, which processes the receiving patches according to properties defined in the tessellation control shader and passes the resulting primitives into the *tessellation evaluation shader*, which is invoked once for each produced vertex and used to manipulate the tessellation result. After the tessellation phase, the *geometry shader* is invoked for each produced primitive. The main purpose of this programmable shader lies in generating, removing and manipulating passed primitives.

Before the last programmable part, the *fragment shader* is invoked, a number of fixed-function routines process the primitives. First, primitive assembly takes place, in which vertices are grouped to lines and triangles. Second, the result is clipped against the viewport (the area in which the scene is drawn). Any triangle (partly) visible in the viewport is transformed to normalized device coordinates, a coordinate system ranging from -1 to 1 in the x and y axis and from 0 to 1 in the z-axis. An optional last stage, before rasterization takes place is culling - the rejection of triangles depending on vertex ordering for dismissing triangles that face away from the viewer. Rasterization is the step in which it is tested which fragments belong to a triangle. All triangles that pass this test are sent to the fragment shader. Finally, in the fragment shader, individual colors of fragments are manipulated and the fragment is sent to the framebuffer [61, ch.3]. A simplified concept of the pipeline can be viewed in Figure 7.1.

The stencil buffer There are several per-fragment tests that can be used to prevent writing certain fragments to the framebuffer. Those tests are the scissor test, the stencil test and the depth test. Because stencil testing is used extensively throughout the visualization of simulation ensembles, this section focuses on briefly explaining the basic concepts of the stencil buffer. A great analogy of what stencil testing does is given in Sellers et al. [61, p.416]: basically, it can be thought of as cutting out a shape in a cardboard and spray-painting the shape on a wall. Thus, stencil testing is used to "cut out" a certain portion of the framebuffer and prevent fragments to be drawn outside of that portion. A stencil buffer that specifies the region to be written to has to be attached to a framebuffer in order to work. Once attached, the stencil filters write operations to the framebuffer as specified by the developer. In order to configure the behavior of a stencil buffer, three functions are frequently used in the visualization technique presented in this thesis. First, *glStencilMask*, which specifies which bit planes of the stencil buffer are modifiable. The stencil buffers that are created in the Uncertainty

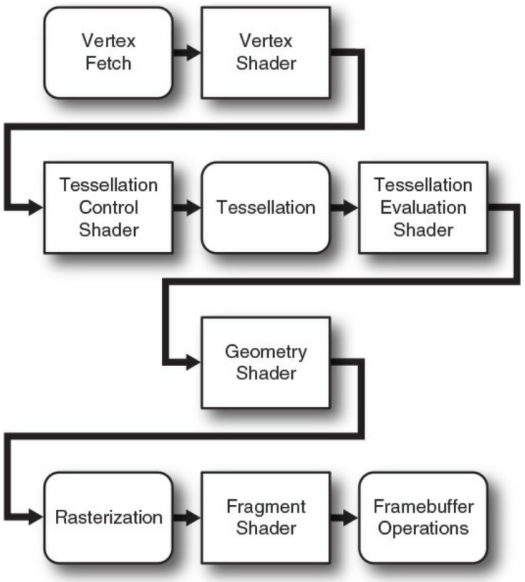


Figure 7.1: Simplified version of the OpenGL implementation of the rendering pipeline concept. Non-programmable shaders have rounded corners and programmable shaders have squared corners [61, p.47].

Plugin all have a bit depth of eight bits per fragment, thus enabling the developer to use up to one byte of independent tests per fragment. The function *glStencilFunc* is used to set on which conditions the stencil test passes. The function takes a comparison operator c (e.g. $\leq$, $\neq$, etc.), a reference value r and a bit mask m as parameters. For a certain value v in the stencil buffer, the test passes if: $(r \ \wedge \ m) \ c \ (v \ \wedge \ m)$. If this test passes, the fragment is written to the stencil buffer - otherwise not. Finally, the third function, *glStencilOp*, specifies what happens to the values in the stencil buffer when the stencil test passes or fails.

7.1.2 Compute shaders

Introduced in OpenGL 4.3, compute shaders provide access to the GPGPU capabilities of modern GPUs. Being standalone shaders, they do not fit anywhere in the rendering pipeline described in Section 7.1.1. In some aspects, compute shaders act like other shaders (access to uniform variables and buffer objects for example), however, because they are isolated from other shaders, they do not have any output types. Providing data to the compute shader and reading data back is implemented using a special buffer called the Shader

Storage Buffer Object (SSBO). SSBOs may hold arbitrary data and have decent mapping capabilities to access the buffer the same way in CPU-related code and shader code [62].

In order to dispatch a task to a compute shader and to take advantage of the parallel capabilities of GPUs, it should be possible to split the task into so called workgroups - parallel entities that concurrently work on the task. A workgroup consists of a grid of work items in which each item performs a single invocation of the shader code. The whole dispatch consists of a grid of work groups in which each work group gets assigned a unique id [61, ch.10]. Additionally, each work item can be identified with a global invocation id and an id that is relative to its work group, Figure 7.2. Those ids are accessible in the shader and can be used to determine the current invocation.

Because several invocations are executed at the same time, and an SSBO is shared among the invocations, necessary precautions have to be taken when writing to an SSBO in order to avoid race conditions. This can be achieved by using atomic counters: an atomic counter buffer is provided to the dispatch and invocations may increment the counter via a built-in shader function. The function returns the value of the counter before the increment took place, giving the invocation a unique value that can be used as an index for the SSBO.

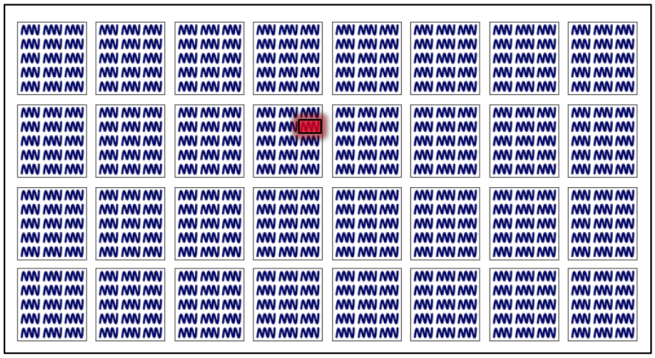


Figure 7.2: Illustration of work groups in a compute shader. Note: this picture only shows work groups aligned in a two-dimensional grid while in reality, a third dimension exists. Each blue spring represents a work item. Work items are grouped into work groups. The whole dispatch consists of several work groups aligned in a grid. Each work group and invocation is identified with a unique three-dimensional id. The id count starts at the lower-left corner; thus, the highlighted invocation has the global id: $(11, 13, z)$, and the local id: $(2, 3, z)$. The workgroup id of the workgroup containing the highlighted invocation is: $(3, 2, z)$. Picture taken from Hafner [63].

7.2 Relevant computational geometry algorithms

Working with 2D contours, which are projections of the resulting 3D lesion iso-surfaces into planes, is an integral part of the visualization technique presented in this thesis. Those contours are defined by their ordered vertices and may be treated as polygons. This brief section will introduce some of the algorithms used for computing several polygon-related tasks that keep recurring throughout the thesis. For convenience, all calculations in this section are restricted to $\mathbb{R}^2$.

7.2.1 Segments, rays, lines and their intersections

It is important to properly define and distinguish the notion of segments, rays and lines and their associated properties when dealing with intersections among them for subsequent sections of this thesis.

Terminology The general equation that can be used to describe all of the three concepts is as follows:

$$\mathbf{x}(s) = \mathbf{x_0} + s\mathbf{d} \tag{22}$$

This linear equation can be used as a parameterized representation of a segment, ray or line, where $\mathbf{x(s)}$ represents a point in $\mathbb{R}^2$, defined by an origin $\mathbf{x_0}$, a direction vector $\mathbf{d}$ and a scalar s.

Segments are constrained on both ends and often described by two vertices. Thus, s is bound by two finite scalars: $0 \leq s \leq l$, where $\mathbf{x_0}$ represents one of the constraining vertices and, if the direction vector $\mathbf{d}$ is normalized, l is the length of the segment.

Rays are often defined by an origin and a direction. The notion of length does not make sense for rays, since they extend to infinity. Consequently, s is only bound by a single scalar, namely zero: $s \in [0; \infty)$; hence, it does not matter whether d is normalized or not.

Lines have neither an origin nor an end. They extend in both directions to infinity, and any point on the line may be used to define $\mathbf{x_0}$. The scalar s is not constrained at all: $s \in (-\infty; \infty)$.

Intersections Despite the different constraints imposed on s, intersection calculation of segments, rays and lines share a similar mathematical foundation. Somehow counter-intuitively, line-line intersection calculation has the fewest computations of all the concepts, although lines (as well as rays) have infinite length. This is based on the fact that parameterized intersection

calculation implicitly treats the linear transformation in Equation 22 as being a line. Further constraints on the parameter s need to be implemented separately, making segment-segment intersection the most demanding computation. Consider two intersecting lines in parameter form:

$$\mathbf{u}(s) = \mathbf{u_0} + s\mathbf{u_d} \tag{23}$$

$$\mathbf{v}(t) = \mathbf{v_0} + t\mathbf{v_d} \tag{24}$$

The intersection point of the lines is obtained by equating $\mathbf{u}(\mathbf{s})$ and $\mathbf{v}(\mathbf{t})$, solving for either s or t and putting the calculated scalar back into the line equation [64, p.115-116]. An example implementation of an algorithm that computes segment intersection is given in Listing 2.

Listing 2: Pseudocode example for computing the intersection of two segments. Returns true if segments intersect.

```
function segmentIntersection
  //The two segments are defined by four vertices in total:
  //begin_first, end_first, begin_second, end_second

  //direction of first segment:
  set dir_first = end_first - begin_first

  //direction of second segment:
  set dir_second = end_second - begin_second

  //tmp vector for calculating numerator:
  set tmp = begin_second - begin_first

  //calculate denominator:
  set denom = dir_second.x * dir_first.y - dir_first.x * dir_second.y;
  if denom < EPS
    return false //segments are colinear
  end

  set denom_is_positive = denom > 0;

  //both numerators:
  set first_numer  = dir_second.x * tmp.y + dir_second.y * tmp.x;
  set second_numer = dir_first.x  * tmp.y + dir_firs.y   * tmp.x;

  //performing tests for parameters:
  if (first_numer < 0) equals denom_is_positive or
     (first_numer > denom) equals denom_is_positive or
     (second_numer < 0) equals denom_is_positive or
     (second_numer > denom) equals denom_is_positive
    return false; //segments do not intersect
  end

  //segments intersect. Calculating parameter of first segment:
  set param = first_numer / denom;

  //Actual intersection:
  set intersection = begin_first + param * dir_first;

  return true
```

7.2.2 Polygons

As noted before, a large part of the visualization algorithm deals with polygon operations. All processed polygons do not self-intersect; i.e., the plugin only processes "simple" polygons. Narrowing the type of polygons down to this benign subclass is important, since many algorithms are only applicable to simple polygons.

Convex hull of simple polygons The convex hull of a polygon is the smallest convex subset of vertices that still contains the original polygon [65, ch.3]. The reason to approximate a polygon with its convex hull is mainly the simplification (and therefore acceleration) of several algorithms, like the "point-in-polygon test". In a convex polygon, each point within the polygon lies on the same side of all segments and one can draw a straight line between two arbitrary polygon vertices, and the line will never be outside the polygon. Furthermore, the interior angle of two consecutive segments of a convex polygon is always $\leq 180°$. A comparison of a non-convex polygon and its convex hull is given in Figure 7.3.

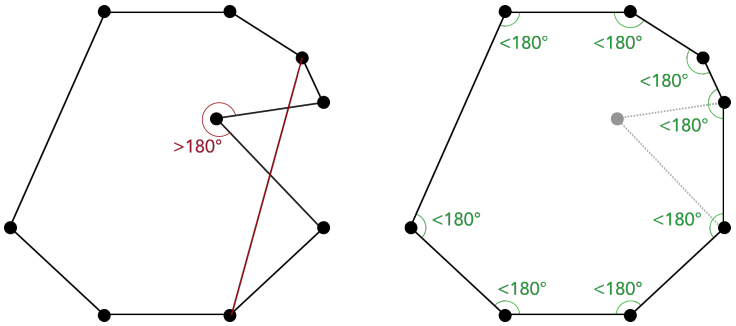


Figure 7.3: Comparison between a non-convex polygon (left) and its convex hull (right). It can be seen that the non-convex polygon violates some conditions of convex polygons, as not all interior angles between segments are $\leq 180°$, and connecting certain vertices results in intersections between polygon segments and the line.

Several algorithms have been proposed in an attempt to compute the convex hull of a simple polygon in $\mathcal{O}(n)$ time. However, many of them have been proven to be wrong later on and some of them are difficult to implement (for an overview, refer to e.g. [66]). In this thesis, Melkman's algorithm [67] has been used to compute the convex hull of simple polygons, because it both has $\mathcal{O}(n)$ complexity and is easy to implement. Furthermore, it is an online algorithm which distinguishes it from other convex hull algorithms. The

algorithm uses a double-ended queue (deque) to calculate the hull. The deque contains the current convex hull at any point in the algorithm. New vertices are added and already included vertices are removed depending on where new vertices lie with respect to the current convex hull. For a thorough description of the algorithm and a pseudocode implementation, refer to Melkman [67].

Point-in-polygon test On several occasions in the uncertainty visualization it is necessary to examine whether a point lies within a polygon. For convex polygons, this task is fairly simple, because one only needs to check if the point lies on the right-hand side of each segment of a polygon with clockwise (CW) orientation in order to be inside the polygon (in case of counter-clockwise (CCW) orientation, the point has to lie on the left side of each segment). Therefore, the complexity of the algorithm is linear with respect to the number of vertices.

For non-convex polygons, Jordan's point-in-polygon test is used to determine if a certain point is inside a polygon. In this algorithm, a ray is shot from the test point in an arbitrary direction, and the number of intersections with the polygon is counted. If the resulting number is even, the point is outside the polygon; if it is odd, the point is inside. This algorithm also runs in $\mathcal{O}(n)$, however, it may be a bit slower in practice due to the potential need for calculating several intersections. A visual representation of the algorithm is shown in Figure 7.4.

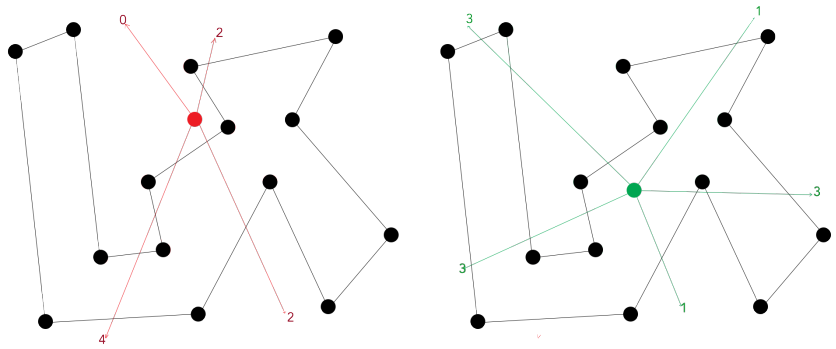


Figure 7.4: Visualization of the point-in-polygon test. Left: a point that lies outside the polygon. Regardless in which direction the ray is shot, the intersection count with the polygon is always an even number. Right: a point that lies inside the polygon. In this case, an arbitrary ray shot from this point will always result in an odd intersection count with the polygon.

7.3 Overview and user interface

The core of the Parameter Uncertainty plugin is the ensemble visualization module, which gives the user a non-cluttered method to concurrently visualize and investigate the entire ensemble of simulation results. In order to visualize a particular ensemble of simulated lesions, the user has to first load the data and select the lesions that are to be visualized. For convenience, the UI of the ensemble visualization module contains two lists, each representing the range of values for one of the specific parameters. Each item in these lists is directly connected to the corresponding simulation parameterizations and allows toggling the specific value on or off for the upcoming analysis tasks. The user may select a single sample value of one parameter, while including the whole sample range of the other parameter. If the selection changes later on, the visualization adapts to the changes and visualizes the current selection accordingly.

In addition to specifying the lesions to be visualized, the UI contains options to adjust several aspects of the ensemble visualization as well as control panes for the "vessel filter" and the "threshold filter", which are described in Section 7.7. The basic workflow is shown in Figure 7.5 and the UI is shown in Figure 7.6.

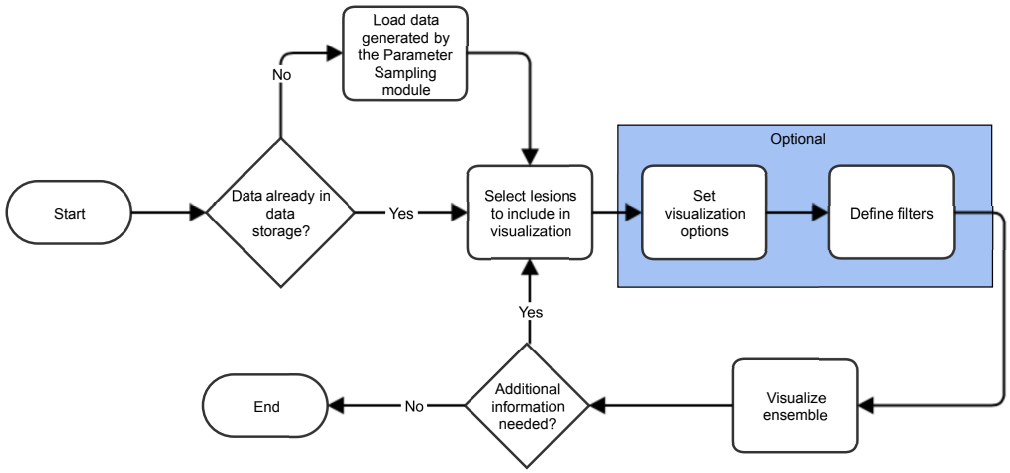


Figure 7.5: Workflow of the ensemble visualization module.

The following sections will sequentially describe the steps taken during the visualization process, i.e., the techniques within the "Visualize ensemble" box in Figure 7.5. Each of these steps are performed whenever the main view

of the RFA Guardian needs an update, i.e., whenever the user interacts in any way with the axial, sagittal, and coronal windows. Thus, the ensemble is rendered in real time. Interactions with a specific window may trigger the need for other windows to be updated, too, because moving the cursor to a specific position in one view leads to the need for an update of the rendered data layers in the other views as well in order to stay consistent (recall Section 5.2). Hence, most of the time all three views are simultaneously updated, which requires high performance in order to stay interactive. For accomplishing this task, several performance related optimizations, such as multithreading and GPU rendering and computing, are employed throughout the rendering process.

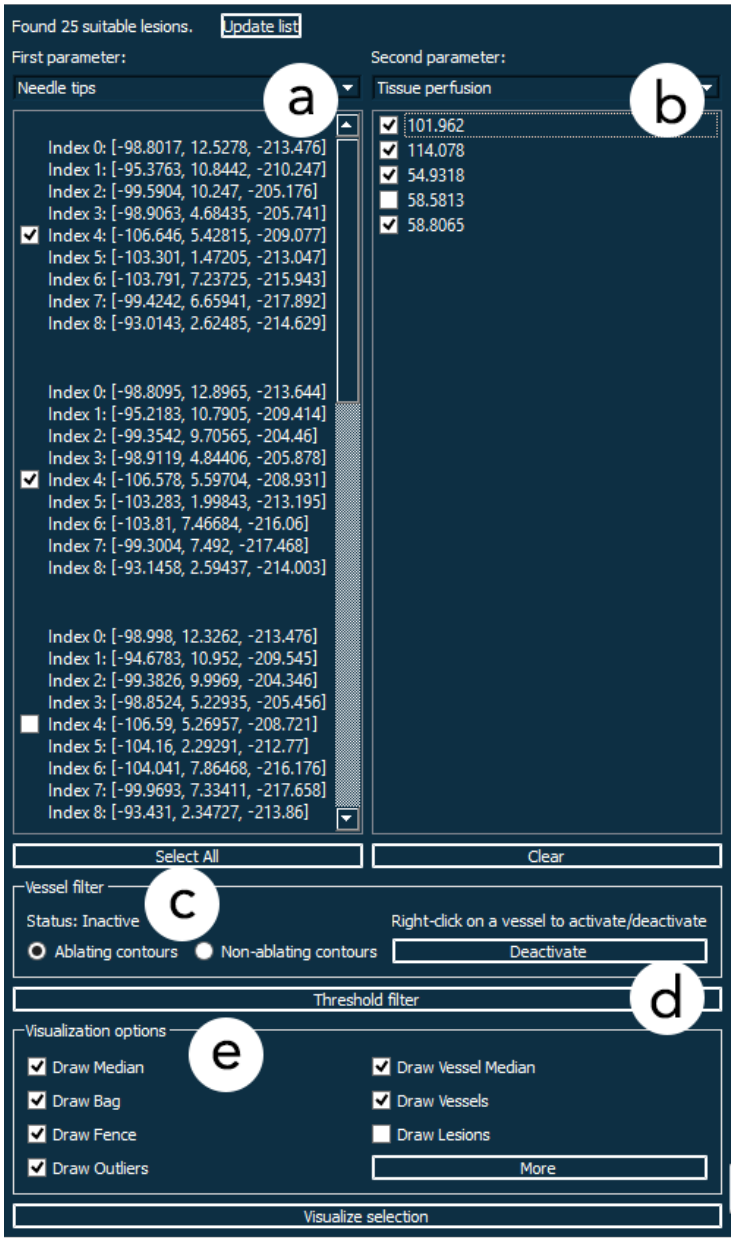


Figure 7.6: The UI contains a list of sampled values of the first parameter (a) and the second parameter (b). In this particular example, the tips of the electrode probe and the tissue perfusion have been sampled. In the case of electrode tips, locations of all tips are displayed. Sample values can be selected by ticking the boxes on the left of the values. The UI also includes the option to set the vessel filter (c) and the threshold filter (d) (see Section 7.7). Finally, the visualization can be adjusted through several options (e).

7.4 Generating the contour boxplot

The section gives detailed information about how the contour boxplot of an simulation ensemble is generated. An example of an ensemble visualized using the contour boxplot can be seen in Figure 7.18 at the end of this section.

7.4.1 Generating 2D data

The first step of rendering in the 2D windows (axial, sagittal, and coronal views) of the RFA Guardian is to actually slice out the correctly aligned 2D view of the lesion data for each view. As stated in Section 6.5, the simulation output is a 3D mesh of the resulting lesion. In detail, the simulation mesh consists of an iso-surface where the cell death probability of the tissue crosses 80%. This iso-surface is stored in a *vtkPolyData* data set. Therefore, it seems natural to use VTK filters for slicing the data. The filter that comes into question is the *vtkCutter*, which can be used to slice through a VTK data set and consequently reduce the n-dimensional data set to $(n-1)$ dimensions. For the specific application of slicing the 3D lesion meshes, an implicitly defined plane is needed in order to get the correct window-aligned views. The result will consequently be a projection of the 3D mesh onto the provided plane, Figure 7.7.

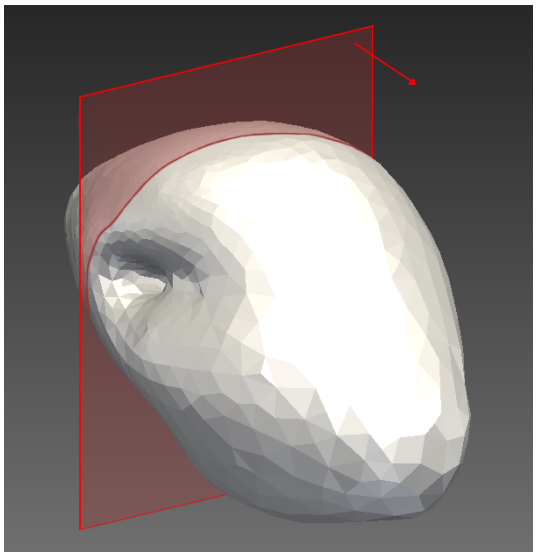


Figure 7.7: Schematic depiction of slicing a 3D lesion (rendered in gray as solid mesh) onto a plane (red). The slice contour of the lesion and the normal of the plane are visible.

The plane itself is defined by a (unit length) normal $\mathbf{n}$, orthogonal to the plane and an origin $\mathbf{p}$ that lies on the plane. This is sufficient to define an implicit equation of the plane, since the orientation of the plane is defined by the normal and the position of the plane is given by the origin. Consequently, every point $\mathbf{x}_i$ that satisfies the following equation lies on the plane, because the dot product of orthogonal vectors is zero:

$$(\mathbf{x}_i - \mathbf{p}) \cdot \mathbf{n} = 0 \tag{25}$$

It is important to consider that the plane normal and origin must be given in world coordinates, since the lesion is defined in world coordinates, too. This, however, is trivial, since MITK already provides the relevant algorithms to transform from display to world coordinates. The actual intersection algorithm is not listed in the *vtkCutter* reference, but essentially all segments of the mesh are tested on whether they intersect with the given plane or not. This can simply be calculated using the parametric linear equation for lines (Equation 22), putting $\mathbf{x}(s)$ into the plane equation and solving for s:

$$\begin{aligned} (\mathbf{x}(s) - \mathbf{p}) \cdot \mathbf{n} &= 0 \\ (\mathbf{x_0} + s\mathbf{d} - \mathbf{p}) \cdot \mathbf{n} &= 0 \\ \Rightarrow s &= \frac{(\mathbf{p} - \mathbf{x}_0) \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}} \end{aligned} \tag{26}$$

If s meets the requirements for segments, i.e., if $s \in [0; l]$, where l is the segment length (refer to Section 7.2.1), the segment and the plane intersect and the vertices of the intersecting segment are projected onto the plane. However, the special case where the segment lies in the plane, i.e., both vertices of the segment satisfy Equation 25, needs to be handled distinctively. In this case, $\mathbf{d} \cdot \mathbf{n}$ will be zero and s is undefined. Thus, both vertices are included in the 2D slice of the mesh without the need to project them onto the plane. If the plane normal $\mathbf{n}$ corresponds to a coordinate vector of the given coordinate system, the whole process is accelerated by testing if the two vertices of each segment lie on the opposite side of the plane. This can be accomplished easily by examining the vector component that corresponds to the plane normal of both vertices and the plane origin $\mathbf{p}$. If one vertex component is smaller and the other vertex component is greater than the respective component of the plane origin, an intersection occurs.

In order to project a vertex $\mathbf{x}_v$ onto the plane, the distance v_d of the vertex to the plane has to be computed first. This is accomplished by computing the dot product of the direction vector $\mathbf{v}$ from the plane origin to the vertex with the plane normal $\mathbf{n}$ (the dot product will first project $\mathbf{v}$ onto $\mathbf{n}$ and

additionally calculate the resulting length of the projected vector):

$$\begin{aligned} \mathbf{v} &= \mathbf{x}_v - \mathbf{p} \\ v_d &= \mathbf{v} \cdot \mathbf{n} \end{aligned} \tag{27}$$

Furthermore, the projected point $\mathbf{x}_p$ is simply obtained by plugging in v_d as parameter in Equation 22 and using $\mathbf{n}$ as negative direction and $\mathbf{x}_v$ as origin:

$$\mathbf{x}_p = \mathbf{x}_v - v_d \cdot \mathbf{n} \tag{28}$$

This procedure requires testing all segments in the mesh and is therefore a fairly expensive operation[6]. The user may select several lesions to be processed, and the complexity of this algorithm is $\mathcal{O}(n)$, where n is the total number of segments of all lesions. The whole procedure has to be executed three times (once for each plane); therefore, this can be a major bottleneck for the visualization of the ensembles right at the beginning. In order to get a reasonable performance, the whole process is multithreaded. Portions of lesions are assigned to specific threads (lesion slicing can be performed independently), which reduces the cost to $\mathcal{O}(\frac{n}{m})$, where m is the number of threads that can run concurrently on the CPU.

After the lesion is sliced, the resulting data structure is again a *vtkPolyData* that contains the projected vertex tuples (each tuple corresponds to a segment in the lesion). However, since each segment is defined separately, each vertex is contained twice in the data struct, and the vertices are unordered on top. Therefore, a filter called *vtkStripper* is used to transform the unstructured *vtkPolyData* into a polyline. The stripper simply checks for duplicate vertices, removes them and arranges the corresponding segments in a sequential order. The resulting polyline, having CW orientation, is then transferred into the C++ container *std::vector* for later use without depending on the VTK framework.

In some cases, vessels may appear as separate polygons within outer contours of a projected lesion, since they penetrate the surface of the lesion and can appear as 'tunnels' or similar structures (bottom-left in Figure 7.8). Such vessels have - in contrast to the lesion outline - CCW orientation (as a consequence, their right-hand side normals face in the opposite direction of the right-hand side normals of lesions). Those contours are immediately classified as vessels, before the lesion tree generation (see Section 7.4.2) takes place. The classification is performed as follows: If a plane intersects vessels, the output of the *vtkStripper* consists of separate contours for the lesion outline and the vessels. In order to determine which contour represents a vessel,

[6]Equations 25 to 28 are acquired from [64, p.122-127]

Jordan's point-in-polygon test (refer to Section 7.2.2) is used to assign the vessel contours to the correct lesion outline contour, where a vertex in the contour serves as the test point.

If vessels penetrate the lesion, it may as well be the case that vessels lie in the projection plane and split the projected contour into separate closed polygons. Furthermore, vessels may appear within separated closed polygons as well, making proper assignment to the appropriate outline contour even more important. Figure 7.8 shows the result of a lesion being sliced in all three 2D views.

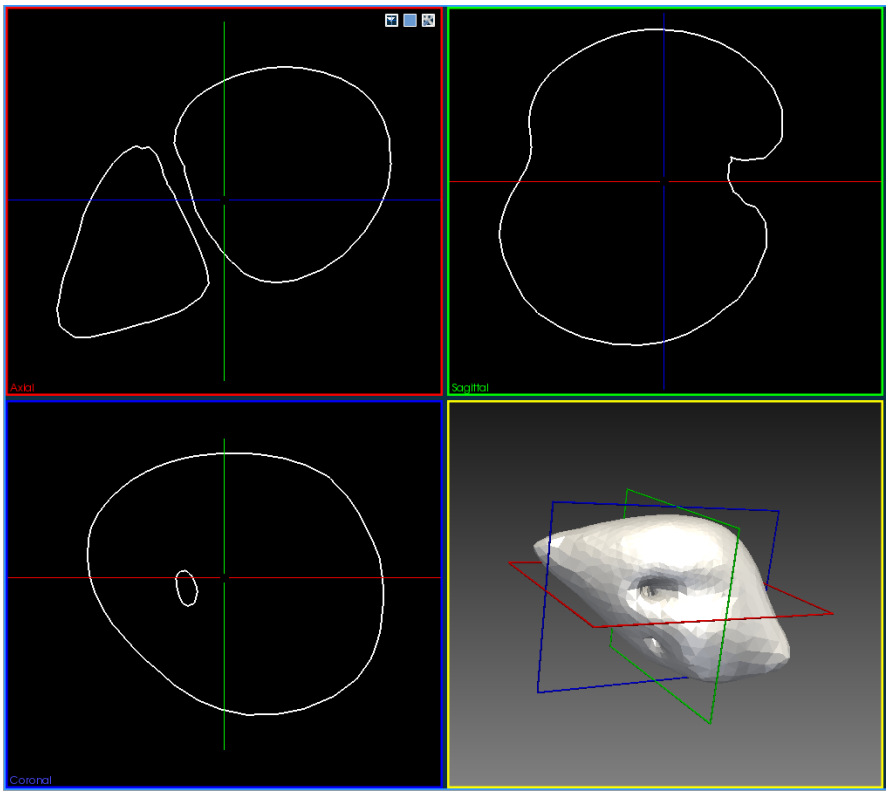


Figure 7.8: Visualization of a single 3D lesion (lower-right window) and its projection onto the 2D view planes. 3D representation of the 2D planes are rendered as well in their respective colors. The hole in the coronal 2D view (lower-left window) represents a vessel. As visible in the axial 2D view (upper-left window), the lesion projection may be split into separate contours, if large vessels penetrate the lesion.

7.4.2 The lesion tree data structure

As already mentioned in Section 7.4.1, a projected lesion may be split into several contours, as vessels penetrate the projected plane of the lesion. In order to visualize the whole ensemble of projected lesions, it is important to build a (semi) hierarchical structure of the contours and group similarly behaving contours of different lesion projections together, as it may be the case that not all contours are split in the same manner. Furthermore, it is important to classify the contours appropriately and to identify which contours ablate particular vessels and which contours fail to ablate them.

Components of the lesion tree There are three distinguished classes of contours: *vessels*, *single-component* contours and *multi-component* contours. Vessels are already described in Section 7.4.1. They occur within another contour and are classified as early as possible. For the other two classes of contours, however, another classifier relates contours to specific vessels: *vessel ablating* contours and *vessel non-ablating* contours. The contour in which a vessel is identified is immediately classified as *vessel ablating* with respect to the found vessel, because the lesion extends beyond the vessel geometry. An example of such a case is shown in the coronal (lower-left) view in Figure 7.8.

The lesion tree data structure is best introduced using an artificial example, as shown in Figure 7.9. In this case, an ensemble consisting of several projected lesions is processed. The associated lesion tree can be viewed in Figure 7.10.

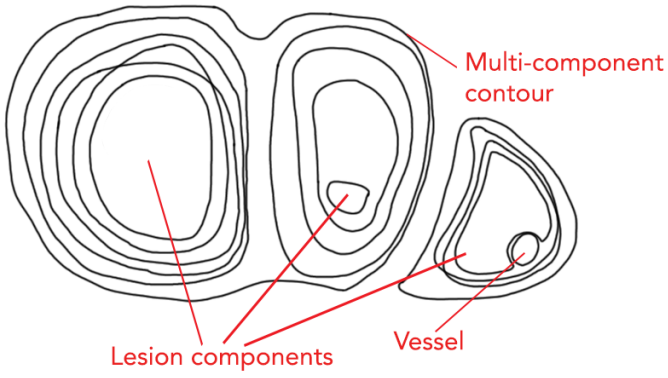


Figure 7.9: Artificial example of a lesion ensemble consisting of three distinct components. Only one contour belongs to two lesion components. The rightmost lesion component contains a vessel. Without proper vessel classification, it is hard to distinguish small lesion contours from vessels as is the case in the middle component.

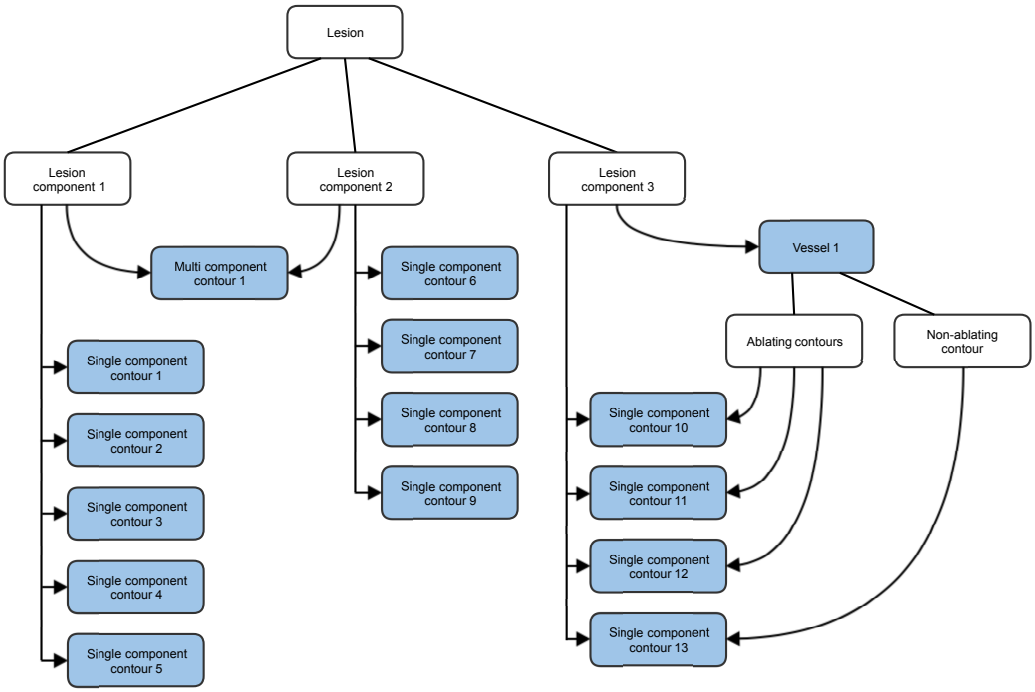


Figure 7.10: Lesion tree data structure that belongs to the lesion ensemble in Figure 7.9. There are three distinct classes in the tree: *Lesion*, the top node, *Lesion components* which are collections of several contours based on spatial criteria, and *Contours*, which are classified accordingly. The blue fields denote actual contours, and the white fields denote containers. Arrows represent pointers.

The top element of every lesion tree is the lesion ensemble itself. This node corresponds to the all projected lesions in a single 2D view and contains all processed contours. Each lesion ensemble is structured by one or more lesion components. In Figure 7.9, the ensemble is split up into three components due to penetrating vessels. However, it is important to note that not every projected lesion splits in the same manner. Some parameterizations may ablate a certain vessel, and, while other lesions split into components, those lesions may not. This leads to the notion of multi-component contours. A multi-component contour belongs to more than one lesion component. In the example in Figure 7.9, this is the case for exactly one contour. In the lesion tree, this contour belongs to the lesion components 1 and 2, which both include a pointer to the multi-component contour 1. All other contours, except for the vessel in lesion component 3, are classified as single-component

contours, because they belong to only a single lesion component. As shown in the data tree in Figure 7.10, each lesion component includes pointers to all contours that belong to that component, even if those contours belong to other components as well. As stated above, for each vessel, every non-vessel contour in the same component is tested on whether the contour ablates the vessel or not. Consequently, a vessel has knowledge about how other contours of the same lesion component behave with respect to the vessel.

Building the lesion tree The lesion tree is built from top to bottom in a two-step manner. First, the lesion components are generated by identifying multi-component and single-component contours with subsequent grouping. Secondly, vessels are assigned to the appropriate lesion component and ablating and non-ablating contours of this component are assigned to vessels. In order to generate the lesion tree, a set of definitions and criteria is formulated for all included tree nodes. Contours are then classified programmatically by referring to those definitions and criteria. For the sake of brevity, the term "contour" is used for non-vessel contours throughout the remaining text.

Lesion components consist of contours and vessels. In order to add a contour to an existing lesion component, the following criterion has to be fulfilled: *A contour C_i is part of a lesion component if the center of the set-theoretic union of all contours already included in the lesion component lies within the convex hull of C_i.* The center of the set-theoretic union of all contours of a lesion component is subsequently referred to as simply the center of the lesion component. In practice, the lesion components are built as follows: all contours are sequentially processed and tested according to the given criterion above. This simply entails checking if the center point of the lesion component lies within the convex hull as described in Section 7.2.2. If the test succeeds, the contour is added to the lesion component. A new lesion component is created whenever a contour does not belong to any of the already existing components with respect to the criterion. Since this test creates lesion components on the fly, the whole procedure is executed twice in order to take components into account that were just created. This usually leads to redundant lesion components that need to be merged later. Consequently, contours might be added to several components, even if they classify as single-component contours. A lesion component is merged, if *the center of a lesion component lies within the convex hull of any of the contours in the other component that is associated with only this lesion component.* Merging entails removing one of the two merged lesion components and transferring all associated contours to the other lesion component. Furthermore, the number of associated lesion components is decreased for each contour.

Through merging, the surplus of created lesion components is reduced to the actual number of existing components, and contours can be finally classified. If a contour still belongs to two or more lesion components, it is classified as multi-component contour. Otherwise, the contour is a single-component contour. The whole procedure is shown in pseudocode in Listing 3.

Listing 3: Pseudocode of the procedure to build lesion components.

```
function buildLesionComponents
  //Create lesion components and assign contours:
  for i in 1,2 //execute two times
    for each contour:
      set convex_hull = convex hull of contour

      for each lesion component:
        set center = center of lesion component
        if center lies in convex_hull
          add contour to lesion component
          increase lesion component counter of contour
      end

      if lesion component counter of contour equals 0
        create new lesion component and include contour
        increase lesion component counter of contour
      end
    end
  end

  //Check whether some lesion components can be merged:
  for each lesion component:
    set center = center of lesion component
    for each other lesion component:
      for each contour in the other lesion component:
        if lesion component counter of contour equals 1
          set convex_hull = convex hull of contour

          if center lies in convex_hull
            merge components
            goto contour classification
          end

        end
      end
    end
  end

  //contour classification:
  for each contour:
    if lesion component counter of contour > 1
      classify  contour as multi-component contour
    else
      classify contour as single-component contour
    end
  end
end
```

The second and final step is taking care of the vessels. One part of this procedure is to categorize the contours of the lesion component to which a vessel belongs into two groups: ablating contours and non-ablating contours. As described earlier in this section, whenever a vessel is identified, a contour of the same lesion can be immediately categorized as ablating said vessel. Thus, at this point, each vessel has knowledge of exactly one single contour that ablates the vessel. Depending on whether this contour is a multi-component contour or a single-component contour, the assignment of a vessel to a lesion component is more or less computationally demanding. In the case of a multi-component contour, the following criterion is used to determine if a vessel belongs to a lesion component or not: *If the center of the vessel lies within the convex hull of any contour belonging to lesion component S_i, the vessel itself belongs to S_i, too.* In case of a single-component contour, the vessel can be immediately assigned to the same lesion component as the contour itself, because the criterion above is always trivially true. After assigning a vessel to a component, the last step consists of identifying which contours ablate the vessel and which do not. The definition of a non-ablating contour is: *If at least a single vertex of the vessel V_i is outside a particular contour, the contour is classified as "non-ablating" with respect to V_i.* Note that the convex hull is not used in this definition, since it is often the case that the vessel is completely inside the convex hull of a contour, but outside of the actual contour. This would lead to many false negatives when testing whether a contour is non-ablating. Thus, Jordan's point-in-polygon test is used to determine if a single vertex of the vessel is outside a contour. All contours that do not fall into the category of being non-ablating are automatically classified as ablating with respect to the vessel in question. The pseudocode of this procedure is shown in Listing 4.

Once the lesion tree is built, the contours within the tree are ready for further processing. In general, each lesion component is visualized separately. However, multi-component contours and single-component contours are not visualized together due to their differing behavior. Either the single-component contours are visualized, or the multi-component contours. In general, three contours are required in order to create the contour boxplot (technically only two contours are required, but creating a contour boxplot of only two contours would not make much sense). Hence the module automatically switches to the visualization that includes more than three contours. E.g., in the tree in Figure 7.9, there is only a single multi-component contour for the lesion components 1 and 2, and, therefore, their single component contours are visualized separately. In such a case, the visualization module has the option to show the actual lesions too, which is crucial for a thorough investigation of the data.

If, however, the number of both multi-component and single-component contours exceed three, the user can decide what should be visualized.

Listing 4: Pseudocode of processing vessels.

```
function processVessels
  //assign vessels to lesion components
  for each vessel
    if ablating contour of vessel is single-component
      add vessel to same component as contour
    else
      set center = center of vessel
      for each lesion component:

        for each contour of the lesion component:
          set convex_hull = convex hull of contour
          if center lies in convex_hull
            add vessel to lesion component
            process next  vessel
          end
        end
      end
    end
  end

  //classify contours:
  for each lesion component:
    for each vessel of the lesion component:
      for each contour of the lesion component:
        for each vertex of the vessel:
          if vertex is outside contour
            assign contour to non-ablating contours of vessel
          end
        end

        if contour is not non-ablating contour of vessel
          assign contour to ablating contours of vessel
        end

      end
    end
  end
end
```

7.4.3 Contour sampling

By building the lesion tree, contours with similar spatial behavior are grouped together and ready for further processing. In order to generate the contour boxplot, the BD of each contour has to be calculated (refer to Section 1.3.2). However, constructing bands and calculating the exact BD for each contour is computationally demanding. Exact computation of the BD is certainly important in cases where contours behave radically different. In the case of this application, however, the lesion contours of a lesion component behave in a very similar manner. Thus, the visualization module makes use of this fact and samples the contours with rays shot from the inside of a contour

ensemble to the outside. The direction of these rays resembles the approximate direction of normals to the contour segments. Thus, they are referred to as "contour normals", Figure 7.11. This approach has two major advantages: first, the set-theoretic intersection of all contours has to be calculated only once - in contrast to calculating each band by its own (for details, see Section 7.4.4). Second, the direction of the contour normals provides a convenient way for grouping the contours locally (refer to Section 7.6).

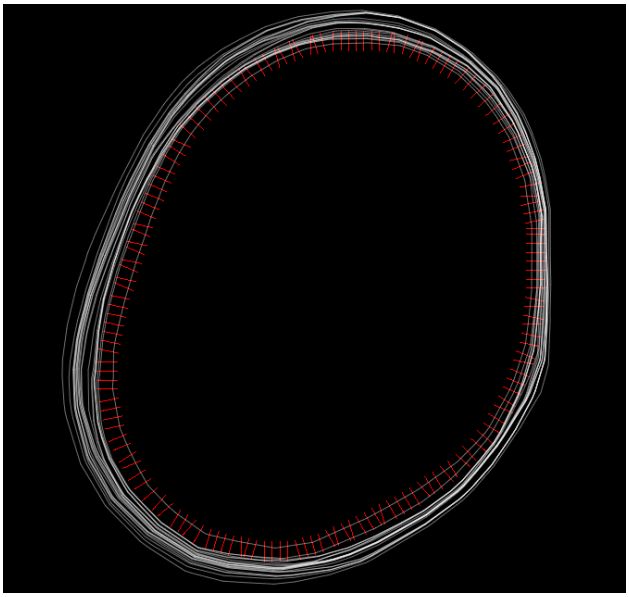


Figure 7.11: Illustration of contour normals (red) that are used to sample the contours (white). In the actual application, a higher number of normals is used to sample the ensemble. The number of normals has been reduced for illustrative purposes.

In order to compute the origin and direction of the contour normals, the innermost contour has to be identified. It is important to note that this contour is not necessarily an actual contour in the lesion tree. Instead, it may consists of vertices of several existing contours in the tree due to the fact that contours may intersect. In other terms, the band of all contours has to be calculated. The normal origins correspond to vertices on the inner edge of the band, and the normal directions point outside towards the outer edge of the band. The band, as defined in Equation 6, is computed by calculating the set-theoretic difference of the union and intersection of the contours. However, for computing the normal, only the inner edge of the band is needed, and, since the inner edge of the band corresponds to the

outer edge of the intersection, it is sufficient to compute the intersection only. This is done facilitating the GPU using multi-pass rendering and a compute shader. First, the intersection is rendered to a texture using stencil buffer techniques, and, subsequently, the edge of the intersection is extracted with a compute shader. It is important to note that, by using this technique, only an approximation of the actual intersection edge is computed, because rendering to a texture inherently discretizes the data. However, this does not pose a problem for computing the contour normals as is shown later in this section.

Computing the contour intersection Contour intersections are computed by rendering the contours sequentially to a texture, while making use of a stencil buffer in a multi-pass approach. First, the stencil buffer has to be configured in a way that allows for rendering non-convex polygons in general. There is no mode in OpenGL for drawing filled, non-convex polygons. Thus, if a filled polygon needs to be rendered in OpenGL, the polygon is triangulated first and subsequently drawn. We use a different approach for drawing filled non-convex polygons using a stencil buffer [68, ch.14], since the stencil buffer is needed for rendering the intersection of the polygon, (saving the computational cost for triangulating the polygon). Consider the non-convex polygon in Figure 7.12. Let us assume several triangles consisting of the vertices of this polygon are rendered in the following way: 123, 134, 145, 156, 167; where each digit corresponds to the vertex number in Figure 7.12.

Now, recalling Jordan's point-in-polygon test in Section 7.2.2, every ray for which the origin lies inside the polygon crosses the polygon an odd number of times. The same is true for the areas covered by the aforementioned triangles. Areas that are rendered an odd amount of time are inside the polygon, while areas that are rendered an even amount of time are outside the polygon. In order to draw the polygon, only the areas that are rendered an odd amount of time need to be written to the framebuffer. This can be realized utilizing a stencil buffer. As mentioned in Section 7.1.1, the stencil buffer used in this application is allocated with eight bit planes. For rendering a non-convex polygon, a single bit in the stencil buffer is used for each fragment. This bit is inverted each time the fragment is rendered. Thus, if it is initially set to `0`, only those fragments rendered an odd number of times will end up with this bit being `1` in the end. In practice, the procedure works as follows: firstly, set all stencil bits to zero and disable writing to the color buffer. Secondly, specify the stencil parameters: use the stencil mask: `0x01` for manipulating only the first bit. The stencil operation should invert the bit only if the stencil test is passed and the stencil test function should be

configured to allow rendering of all triangles. Thirdly, render triangles of the polygon in the way presented above (i.e., the triangles, 123, 134, ...). This can easily be achieved in OpenGL using the "triangle fan" rendering mode. Now, the final polygon can be rendered to the texture by enabling the color buffer again, specifying the test function $(r \wedge m)\ c\ (v \wedge m)$ to only allow rendering when the first bit is `1` (i.e., $r =$`0xFF`, $m =$`0x01` and $c =$ "equal-operator") and drawing a full screen quad over the stencil. However, since the intersection of all polygons should be rendered, and not each polygon on its own, the last step has to be adapted.

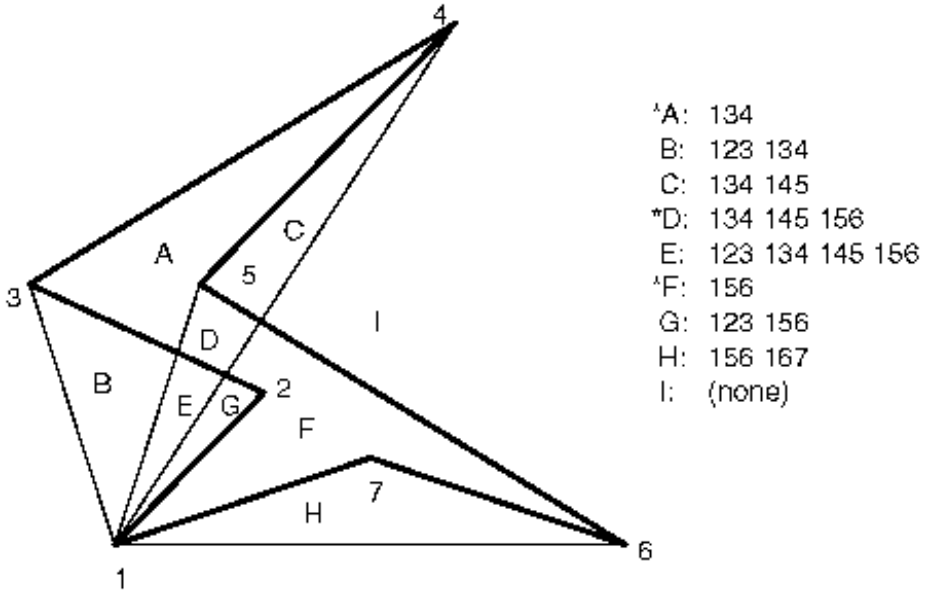


Figure 7.12: An example of a simple polygon with vertices 1-7. The triangles 123, 134, 145, 156 and 167 cover the areas A-I. When drawing the aforementioned triangles, those areas are covered an even or odd number of times, where areas that are covered an odd amount of times are inside the polygon. The table on the right side lists all areas and the triangle coverage. Areas inside the polygon are marked with a star [68, ch.14].

The contour intersection can be drawn using another bit (the "intersection bit") which is initially set to `1` and, instead of enabling the color buffer in the last step, another stencil test is executed. The test function is set to the same values as if the polygon would have been written to the color buffer, however this time, the intersection bit of the stencil buffer itself is modified: whenever the stencil test fails, the bit is set to zero. Hence, if all polygons are rendered sequentially, only the area of the polygon intersection will have

the intersection bit set to `1`. The whole procedure can be summarized as follows:

- Initialize the bits of the stencil buffer and disable the color buffer. The first bit is used for rendering the polygons, and the second one is used as intersection bit. Thus, set `00000010` for the whole stencil buffer. Set the stencil function to always pass.
- Render each polygon in sequential order: set the stencil mask to only manipulate the first bit (`0x01`), and render the triangle fan. In case the stencil test passes, the first bit is inverted for each fragment, so in areas covered by the polygon, the first bit is `1`, otherwise it is `0`. Now draw a full screen quad and set the stencil test to only pass if the first bit is `1`, and ignore the value of the intersection bit. This time the stencil mask is set to `0x02` in order to only manipulate the second bit in the stencil buffer. The intersection bit should be set to zero if the stencil test fails. Reset the first bit and draw the next polygon.
- After all polygons have been rendered, draw a full screen quad and set the stencil test to only pass if the intersection bit is `1`. Ignore the other bits and do not manipulate the stencil buffer itself. Enable the color buffer again and draw the polygon intersection to a texture.

This procedure leads to a binary texture where the inside of the polygon intersection is `1` and the outside area is `0`. The texture has a grid of $256 \cdot 256$ pixels. The bounding box of the whole processed ensemble is calculated, and, during rendering, the whole bounding box is mapped onto the grid for maximum resolution.

Before the edge of the intersection is extracted with a compute shader, a morphological opening operation is applied using OpenGL fragment shaders. This operation consists of an erosion operation followed by a dilation operation. First, the erosion filter is applied to slightly shrink the intersection and to structurally smooth the edge. An erosion filter is a morphological filter that uses a structuring element to "thin out" objects in an image. The structuring element defines the manner in which the object is thinned out. Erosion as set operation can be defined as follows:

$$A \ominus B = \{z | B_z \subseteq A\} \tag{29}$$

Here, z denotes coordinates, and B_z are translations of the structuring element B by z. This equation can be interpreted as "erosion of A by the

structuring element B". In a nutshell, the resulting set consists of all coordinates z by which the translation of B by z (B_z) is a subset of A. To interpret the resulting set in terms of a binary image, all coordinates z which satisfy Equation 29 represent `1`s in the final image. The structuring element used for eroding the contour intersection is a 3x3 neighborhood. An example of an erosion operation with this structuring element is given in Figure 7.13.

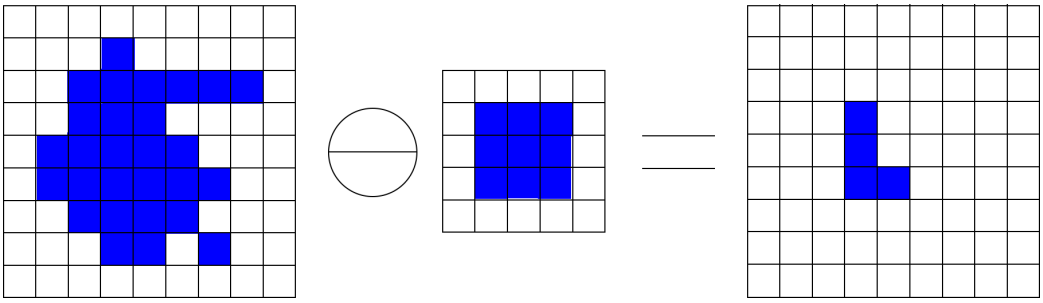


Figure 7.13: Example of an erosion operation of a binary image expressed as set A using a 3x3 neighborhood as structuring element B. The resulting set consists of the coordinates z where $B_z \subseteq A$.

The erosion operation is easily implemented in OpenGL by fetching a 3x3 neighborhood of the input texture for the current texel in the fragment shader and checking whether a single texel value in the neighborhood is `0`. If so, draw `0`, otherwise draw `1`.

After the erosion operation has finished, the dilation filter is applied, where the same symmetric structuring element as in the erosion filter is used. Thus, the operation itself is defined as:

$$A \oplus B = \{z | \hat{B}_z \cap A \neq \varnothing\} \tag{30}$$

where $\hat{B}_z$ refers to all translated-by-z versions of B. The dilation filter can be viewed as the counterpart to the erosion filter. The result of the dilation operation consists of all coordinates where the intersection of $\hat{B}_z$ and A is a non-empty set. The dilated image of the result in Figure 7.13 is shown in Figure 7.14. [69, ch.9]

Once the image is dilated, the edge of the contour intersection can finally be extracted with a compute shader. For this operation, a 3x3 neighborhood is utilized as well. If the center pixel of the neighborhood is 1 (i.e. the pixel is inside the dilated area) and any other pixel in the neighborhood is 0 (i.e. outside the area), the compute shader tries to extract two neighboring pixels

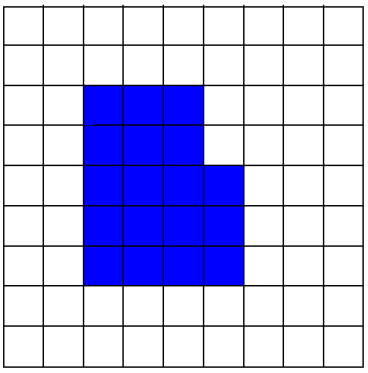


Figure 7.14: Dilation of the resulting image in Figure 7.13. The same structuring element as in the erosion operation was used.

of the center pixel that lie on the edge contour. Since this is done for the whole image, the final contour can be extracted by following the extracted neighbors until the start has been reached again. Due to the opening operation, each center has only two neighboring pixels, and each neighbor that lies on the edge contour can be extracted in a deterministic way. Furthermore, the neighbors can be determined in a way to ensure CW contour orientation, see Figure 7.15. There exist several possible neighborhoods that may arise during processing. In practice, a lookup texture is used for determining where the contour neighbors of a pixel are located. If an eligible neighborhood is found by the compute shader, the morphology of the neighborhood is first encoded in a single byte, where each bit represents a pixel in the neighborhood (with exception of the center). The upper-left pixel of a neighborhood corresponds to the first bit (`0x01`), and the bottom-right pixel to the last bit (`0x80`). E.g., the first neighborhood in Figure 7.15 corresponds to the code `11110100`. This makes it possible to encode up to 256 different neighborhoods, although only 36 are needed. Each code corresponds to a texture cell in the lookup texture, which has the dimension: 1x1x256. In each of the 36 texture locations, the offsets of the CW neighbor (viewed from the center) are stored. Getting back to the previous example: The code `11110100` corresponds to the lookup texture location 1x1x244. In this location, the texture returns the tuple (1,1), which represent the offsets in x- and y-direction from the center to the neighbor. Now the compute shader has knowledge about the relative coordinates of the CW neighbor. In order to obtain the absolute coordinates of the neighbor (i.e., the coordinates of the edge contour in texels), the tuple has to be added as offset to the global invocation id (refer to Section 7.1.2).

The next step is to transfer the coordinates of the center and the neighbor

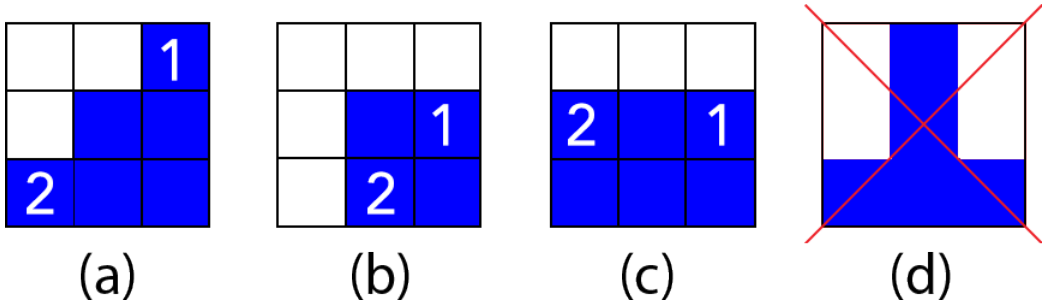


Figure 7.15: Examples of 3x3 neighborhoods in which the center pixel is inside the dilated area and at least one pixel is not. Due to the morphological opening operation, every center has only two neighbors that also lie on the contour edge (a)-(c). Situations that would lead to more than two neighbors of a center are not possible (d). When viewed from the center pixel, the neighbor denoted by a `1` lies on the CW direction of the edge and the neighbor denoted by 2 lies on the CCW direction.

back to the CPU. This is done using an SSBO and an atomic counter, as described in Section 7.1.2. As stated earlier, the texture dimensions of the dilated image are 256x256. This means that two bytes are needed to store every possible absolute coordinate, and 4 bytes are needed in total, in order to store the center and one neighbor. Thus, one unsigned integer (32 bit) sufficiently meets the storage requirements of both coordinates. They are stored in the following way in an unsigned integer in the compute shader and written to the SSBO:

$$u = (((x_c << 8) \mid y_c) << 16) \Big| ((x_n << 8) \mid y_n) \tag{31}$$

where (x_c, y_c) are the center coordinates and (x_n, y_n) are the coordinates of the neighbor in CW direction. On the C++ side, first an array containing 16bit integers is allocated that is capable of representing every possible center coordinate. Thus, the array needs to be able to store 256x256 values. Afterwards, the SSBO is read out, and the unsigned integer values are decoded into two 16bit values in the following manner:

$$c = u \ \& \ \texttt{0xFFFF0000} >> 16 \tag{32}$$

$$n = u \ \& \ \texttt{0xFFFF} \tag{33}$$

where c is the center coordinate and n is the neighbor coordinate. The center is used as the position in the array in which the neighbor is to be saved.

Once the whole SSBO is read out and the neighbor for each center coordinate has been saved in the array, the array is processed. Each neighbor coordinate is used to locate the next center, until the start coordinate has been reached again and the contour is closed. During this procedure, every fourth center coordinate is transformed back into world space using the following transform:

$$x_{\text{world}} = \frac{x_{ub} - x_{lb}}{256} \cdot x + x_{lb} \tag{34}$$

$$y_{\text{world}} = \frac{y_{ub} - y_{lb}}{256} \cdot y + y_{lb} \tag{35}$$

where $x_{\text{world}}, y_{\text{world}}$ represent the world coordinates, and x_{ub}, y_{ub}, the upper bounding box coordinates, and x_{lb}, y_{lb}, the lower bounding box coordinates of the contour ensemble, respectively. These transformed coordinates are the final origins of the contour normals. The directions of the contour normals are simply obtained by calculating the outward pointing normal of the direction of the two adjacent origins. A higher number of normals can easily be achieved with linear interpolation between existing normals.

Intersecting the normals and contour segments Once the contour normals are computed, the intersection of each normal with the contour segments need to be obtained in order to sample the contours. For each contour, this process looks as follows: First, the algorithm looks for the segment that intersects the first normal. Since both the contour and the set-theoretic contour intersection (and consequently, the sequence of normals) have CW direction, intersections of following normals are either located on the current segment or the next segment of the contour. Despite the fact that $n \cdot m$ intersections need to be calculated (n = number of contours, m = number of normals), the algorithm exhibits a reasonable performance by exploiting the equal directions of contours and sequence of normals. The most demanding step is finding the segment on each contour that intersects the first normal. The whole algorithm, however, can easily be multithreaded by splitting the processed contours into different threads. Furthermore, the actual intersections of the normals do not have to be calculated - only the parameter s of the ray (refer to Equation 22) needs to be computed, due to the fact that, for calculating the BD, only the position of the contours relative to the normal origin is important.

7.4.4 Band depth calculation

The BD measure used in this thesis is based on the concepts of mBD, as introduced in Section 1.3.1. Hence, the technique merges the concept of con-

tour boxplots with a band depth measure used in functional boxplots. This is possible, because the hull of contour normal origins resembles the shape of the processed contour ensemble. While the mBD in functional boxplots is obtained by measuring the amount by which a certain function stays within all bands with respect to the abscissa using the Lebesgue measure, the method employed in the visualization module actually measures the amount that a contour stays within a contour band with respect to the overall shape of the contour ensemble itself. In order to distinguish this measure from mBD, it is subsequently referred to as "lesion band depth" (lBD).

In order to calculate the lBD of each contour, bands of second and third order may be used. As stated in Section 1.3.1, using more than two contours to construct a band increases the computational costs of computing the bands in a superlinear fashion. However, due to contour sampling and a lookup table (LUT), the Parameter Uncertainty plugin is able to keep the computational costs for computing bands of second and third order the same. What makes this possible is the fact that actual bands do not need to be computed. The lBD value for a particular contour is calculated by taking the location of the contour intersection relative to the intersecting contour normal origin into account and averaging this measure across all contour normals as described by the new approach to compute the mBD in Section 4.1.

First, consider an example of a single contour normal and its intersection with five contours. Furthermore, assume that a band consists of three contours. In total, there are ten distinct combinations of bands that need to be taken into account. Within that scope of this contour normal, the sampled contours exhibit different probabilities of being included in a band, depending on the distance to the normal origin, i.e., the location of the contour on the normal, analogous to the rank matrix in Section 4.1. This probability is mirrored at the "innermost" contour, Figure 7.16. By collecting these probabilities for different numbers of sampled contours, a probability table can be generated. This table contains the likelihood that a certain contour is enclosed in a band dependent on the location of the contour on the normal. Due to the aforementioned symmetry of the probabilities, about half of the values in the table can be omitted, see Table 1. In order to implement this table in the plugin for lookups, Equation 11 is utilized to calculate the probability for a certain location of a contour on a contour normal and a given number of sampled contours. In the Parameter Uncertainty plugin, only bands of second and third order are used to compute the lBD. Therefore, a closed formula (the "lookup equation") is derived from Equation 11 for computing the lBD using bands of third and second order only.

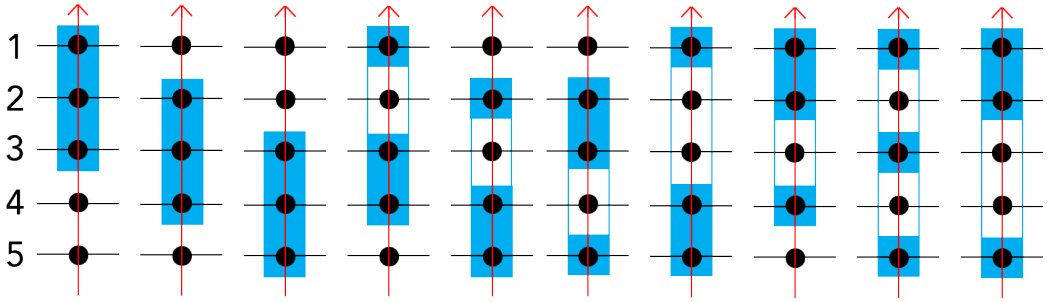


Figure 7.16: An example of a single contour normal (red), that samples five contours (horizontal black lines). Intersections of contours with the contour normal are marked by a circle. Each instance shows a different band being formed by three contours, which are shaded in blue. The probability of a contour being included in a band depends on the location of the contour with respect to the origin of the contour normal. There are six instances in which contours on location #1 and #5 are inside a band, nine instances where contours on the location #2 and #4 are inside a band and contours on location #3 are always encased by a band - regardless of which band is currently processed.

Derivation of the lookup equation In order to derive the formula, certain properties of the probability table are exploited. For the sake of brevity, the number of sampled contours is subsequently referred to as i, and the location on the normals is referred to as j. First, consider the number sequence of the elements where $j = i$. The first five elements are $1, 3, 6, 10$ and 15. This sequence can be viewed as a function in one variable $y = f(x)$, where $x = j = i$. By looking at the rate of increase of the number sequence and the rate change itself, one can infer the derivative of the function, which is $x - 1.5$. Thus, integrating the rate change and specifying the correct offset yields the first term of the lookup equation:

$$\begin{aligned} \frac{df(x)}{dx} = x - \frac{3}{2} \Rightarrow \int f'(x)dx &= \frac{1}{2}x^2 - \frac{3}{2}x + c\Big|_{c=1} \\ &= \frac{1}{2}x^2 - \frac{3}{2}x + 1 \end{aligned} \tag{36}$$

With this term, the first probability value in each row can be calculated. For example, for $x = j = i = 5$, the formula yields the value 6, which corresponds to the first column entry in the fifth row in Table 1. Note that the term probability is used vaguely here, as Table 1 does not contain

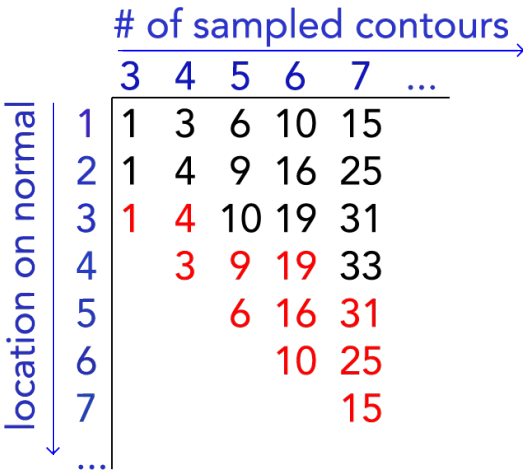


Table 1: Number of instances that contours are included in bands of third order depending on the location of the contour on a contour normal. The rows represent the location of the contour intersection on the contour normals - analogously to Figure 7.16 -, and the columns show the number of sampled contours. The values in red are actually redundant and are omitted in a software implementation in order to save memory.

relative probabilities in the interval [0;1), but rather the absolute number of occurrences of the contours in the possible bands. The next step is to obtain the difference between the first column entry in a particular row and the second one. With five contours sampled and the location value being four, the difference between the first and second probability would be six (refer to Table 1). Those differences are again viewed as a number sequence, which is a shifted version of Equation 36, where $x = j$. Hence, the second term is:

$$\frac{1}{2}(j+1)^2 - \frac{3}{2}(j+1) + 1 \tag{37}$$

For a particular tuple (i, j), this term is added $(i - j)$ times to the first term to get to the second entry in the corresponding row. Thus, with this knowledge, every tuple (i, j), where $j = i + t, t \in \{0, 1\}$, can be computed. However, in order to calculate any value in the table, a third term is needed that takes the increase of differences in the columns into account.

For instance, for the third row (normal location = 3), the difference between the cells increases by three when read from left to right. Furthermore, the increase in difference in the fourth row is four and so on. As a generalization, this circumstance can be expressed by the following term:

$$\frac{j}{2}(i-j)^2 - (i-j)) \tag{38}$$

With all terms summarized and simplified, the whole formula looks like:

$$\frac{1}{2}(3j^2 - ij^2) + \frac{j}{2}(i^2 - 2i - 3) + 1 \tag{39}$$

This, however, is not the final equation, since bands formed by two contours have to be taken into account too (because all possible sub-bands have to be considered too, see Section 1.3.1). In cases where bands are formed by two contours, Formula 39 can be simplified a bit. The derivation is not explicitly stated here, because it can be performed in the same manner as above. The formula for bands of two contours is:

$$(j - 1) + j(i - j) \tag{40}$$

The last step is to find the equation that is valid for both the cases where bands consist of two and three contours. Consequently, this equation has to depend on the band order. The chosen approach incorporates the band order k into the exponents and the number "3" of Equation 39:

$$j^2 \to j^{k-1}$$
$$3 \to k$$

Thus, the equation that combines both the Formulas 39 and 40 is dependent on the number of contours that form a band (k) and the tuple (i, j). For $k \in \{2, 3\}$, this equation is as follows:

$$P_{abs}(i, j, k) = \sum_{m=2}^{k} \Big((\frac{1}{2}(mj^2 - ij^{m-1}) + \frac{j}{2}(i^{m-1} - 2i - m) + 1)(-1)^{m-1} \Big) \tag{41}$$

The sum is needed due to the fact that, for a given band order k, the lower orders are now considered as well. Finally, the lookup equation is obtained by normalizing the absolute number of inclusions of a particular contour by the number of possible bands (refer to Equation 4):

$$P_{rel}(i, j, k) = \sum_{m=2}^{k} \binom{i}{m}^{-1} \Big((\frac{1}{2}(mj^2 - ij^{m-1}) + \frac{j}{2}(i^{m-1} - 2i - m) + 1)(-1)^{m-1} \Big) \tag{42}$$

Using this equation, the LUT can be generated programmatically. As stated before, only half of the actual table needs to be generated due to the underlying symmetry of the probabilities. Thus, the range of the location values can be restricted to: $j \in [1; \lceil \frac{i}{2} \rceil]$. Consequently, when using the

LUT, the location values that are greater than $\lceil \frac{i}{2} \rceil$ need to be remapped back: $j_{remapped} = i - j + 1$. In this way, the location values get effectively mirrored around the center. In most applications, the number of sampled contours is consistent throughout the visualization; thus, the LUT is reduced to a one-dimensional array. Listing 5 shows the computation of the LUT for a given number of sampled contours in pseudocode.

Listing 5: Pseudocode of computing the LUT for lBD calculation. The variable i corresponds to the number of sampled normals and k corresponds to the band order.

```
function computeLUT(i,k)

  if k < 2 or k > 3
    error //k has to be in [2;3]
  end

  set table_size = ceil(i/2) //ceil(x) returns x rounded up to next integer
  allocate lut of size table_size and fill with zeros

  for m in [2;k]
    for j in (0;table_size)

      //Note: binom(i,m) returns the binomial coefficient
      lut[j] = lut[j] + binom(i,m)^(-1) * ((0.5*(m*j^2 - i*j^(m-1)) +
                        j/2*(i^(m-1)-2*i-m)+1)*(-1)^(m-1))
    end
  end
  return lut
end
```

Lesion band depth calculation Now the LUT can be generated before the visualization actually starts. The lBD values for each of the contours is then obtained as follows: The probabilities in the LUT are recorded for each contour normal, added together, and divided by the number of contour normals that were used. To be precise, if all contours are sampled with the same number of normals, the division is not really necessary, since the number of normals would be an invariant across all contours. However, when the division is not omitted, the lBD measure actually corresponds to the mBD measure used for functions. In this case, Equation 42 and Equation 11 yield the same results. An example of a contour ensemble sampled by seven contour normals and the contours respective lBDs is shown in Figure 7.17. Due to the use of the LUT, the computational cost for calculating the lBD values is comparatively low. In total, there are n additions and lookups and a single division for a particular contour when sampled with n normals. However due to the independency of the lBD measure for individual contours, the whole process can easily be multithreaded, where the calculation for individual contours is split up among different threads.

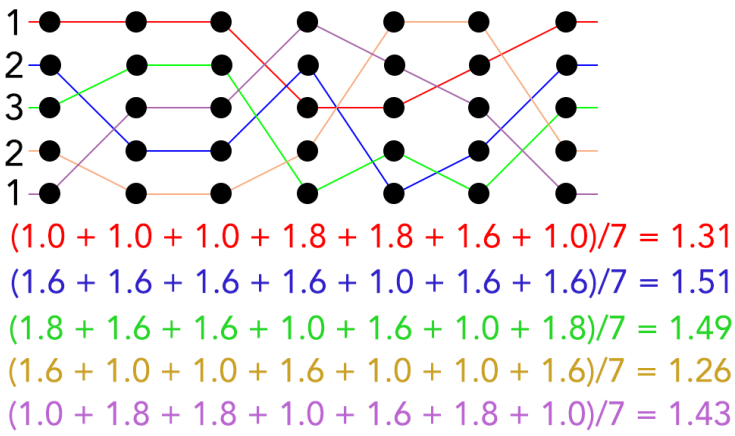


Figure 7.17: Top: example of an ensemble of five contours (drawn in different colors) sampled by seven contour normals. The contours are displayed denatured. In reality, the end of each contour connects to the start. The normals themselves are not shown; however, the intersections of the normals with the contours are depicted by a circle. Furthermore, the mirrored location values are shown on the left of the ensemble. Bottom: the corresponding probability values of each contour taken out of the LUT, added together for each normal and divided by the number of normals, give the lBD value for each contour. A band is formed by three contours. The contours can also be viewed as discrete functions, in which case their lBD equals the mBD.

7.4.5 Visualization of the contour boxplot

Before the boxplot can be visualized, the contours have to be ranked by their lBD values, as described in Section 1.3.1. There are four statistic parameters to be visualized: the median, the 50% region, the fence and the outliers. Visualization of the outliers and the median is trivial, since those parameters are contours. The 50% region and the fence, however, are bands themselves; thus, a stencil buffer is used to achieve the desired visualization. First, recall the definition of a contour band in Equation 6. A band is the set-theoretic difference of the union and the intersection of the contours that form the band. A stencil buffer was already used for rendering the set-theoretic intersection of all contours in Section 7.4.3. When rendering a band, an additional stencil bit for calculating the contour union needs to be used. Briefly recalling the method for rendering the contour intersection in Section 7.4.3, two stencil buffer bits were used to draw the intersection. The first one was needed for rendering the actual non-convex polygons and the second one (the "intersection bit") was initially set to `1`, and set to `0` outside

of each polygon. Consequently, the intersection bit remained `1` only in regions where all polygons shared a common area (i.e., in the area of contour intersection). The third bit that needs to be occupied in the stencil buffer when rendering a band is the "union bit". This bit is initially set to `0` and each time a polygon is drawn, it is set to `1` in areas where the polygon is drawn. Consequently, there are three distinct bit combinations with respect to the intersection bit and the union bit (the second bit in the buffer is used as intersection bit, and the third one, as union bit) that may occur:

- The area outside of all polygons: `00000000`
- The area within the polygon intersection: `00000110`
- The area inside the polygon union, but outside of the polygon intersection: `00000100`

The third area is the one that needs to be rendered, i.e., when enabling the color buffer again, the test function $(r \wedge m)\ c\ (v \wedge m)$ has to be set with $r =$`00000100`, $m =$`0xFF` and $c =$ "equal-operator".

For the whole boxplot to be complete, both the 50% region and the fence need to be rendered. Because the fence covers a larger area than the 50% region, it would be a waste of resources to render both bands on top of each other. Instead, the 50% region is rendered first, and the stencil buffer is used again for rendering the fence. For this to work, a fourth bit is used in the stencil buffer that marks the region where the 50% band has been rendered. Before the fence is rendered, the stencil buffer has to be initialized with `00001000`, where, previously, it was `00000100`. This basically means transferring the third bit to the fourth by setting the stencil buffer function $(r \wedge m)\ c\ (v \wedge m)$ to $r =$`00001100`, $m =$`00000110` and $c =$ "equal-operator" and using the "replace" operation to replace the third bit with the union bit. The stencil mask has to bet set to `00001000` to only change the fourth bit. In order to apply the transformation, a full screen quad can be drawn with deactivated color buffer. Now there are four distinct areas with different combinations of stencil buffer bits:

- The area outside of all polygons: `00000000`
- The area within the fence intersection : `00000110`
- The area where the 50% region was rendered: `00001000`
- The area inside the fence union but outside of the fence intersection and outside the 50% region: `00000100`

Since the fourth area is the one that needs to be drawn, the stencil test function can again be specified as r =`00000100`, m =`0xFF` and c = "equal-operator" Figure 7.18 gives an example of a boxplot where each component and the values of the stencil buffer in different areas are delineated.

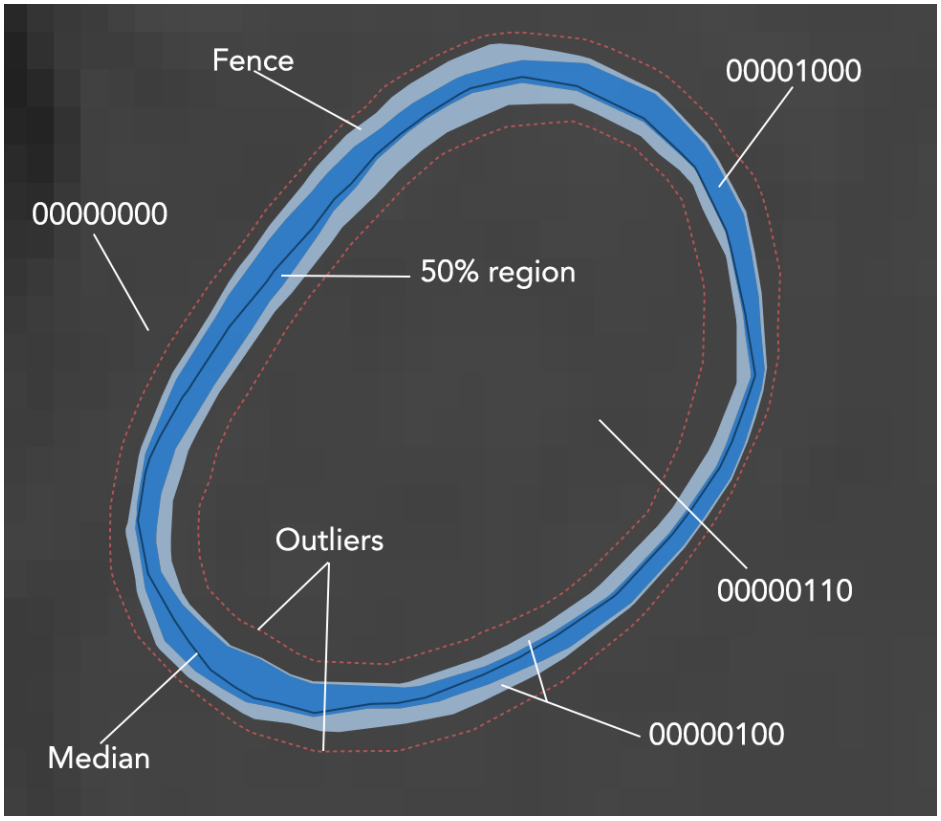


Figure 7.18: Example of a contour ensemble visualized as a contour boxplot. A CT scan of liver tissue is used as background. The components of the boxplot are labeled accordingly. Furthermore, the values of the stencil buffer in different regions are shown, after the 50% region and the fence have been rendered.

7.5 Processing vessels

Special attention needs to be drawn to situations where vessels are present. As explained in Section 7.4.2, some contours may ablate a vessel and others may not (in the same slice). This leads to diverging spatial behavior of the contours, and using the same contour normals for all contours to compute

the lBD would potentially lead to non-ideal results. Consequently, in areas where the vessel is located and contours exhibit a different spatial behavior, contours that do, and do not, ablate vessels, need to be sampled with separate contour normals. Contours that fail to ablate a vessel completely surround the vessel in a pronounced non-convex way. The first step is to identify which contour normal origins follow the spatial characteristics of those non-ablating contours. These normals can be identified by a convexity analysis of the contour normal origins. First, all non-convexities of the contour normal origins are identified. Subsequently, the bounding boxes of the non-convex areas are computed. The box that includes the center of a vessel corresponds to the non-convex area that embraces the vessel. Due to the fact that the ablating contours exhibit fairly convex behavior, it is most of the time sufficient to just connect the first and the last vertex of the non-convex area and interpolate contour normals across that line, see Figure 7.19. When this approach leads to degenerate contour normals, i.e., normals that do not intersect with all of the vessel-ablating contours, the whole contour intersection has to be recomputed using only the ablating contours for a certain vessel. The relevant portion of normals can be extracted by searching for the closest normal origins to the first and last vertex of the non-convex region. Though this approach may be more accurate in some situations, it is also slower than just connecting the first and last vertex by a line.

Since the contour boxplot is a global representation of the underlying contours and some of the contours are now sampled with different normals, measures have to be taken in order for the probabilities to be correctly fetched from the LUT, when processing non-vessel-ablating contours or vessel-ablating contours only. For looking up the probabilities, the number of sampled contours has to resemble the number of contours in the whole ensemble, regardless of how many contours were sampled with a contour normal. In order for the probabilities to be correct, however, the location values of the contours need to be adjusted accordingly. Ablating-contours are always on the outside of non-ablating contours; thus, the number of non-ablating contours has to be added to the location values of vessel-ablating-contours. For non-ablating contours, the location values do not have to be adjusted, because they lie on the inside of the ablating-contours. Using these correction methods, the global lBD for each contour can be computed.

When a vessel is present in the contour ensemble, it may be of great interest for the user to know how likely it is for the vessel to be ablated, given the underlying dataset. The likelihood of ablation for the given dataset is calculated by the number of contours that ablate the vessel, divided by

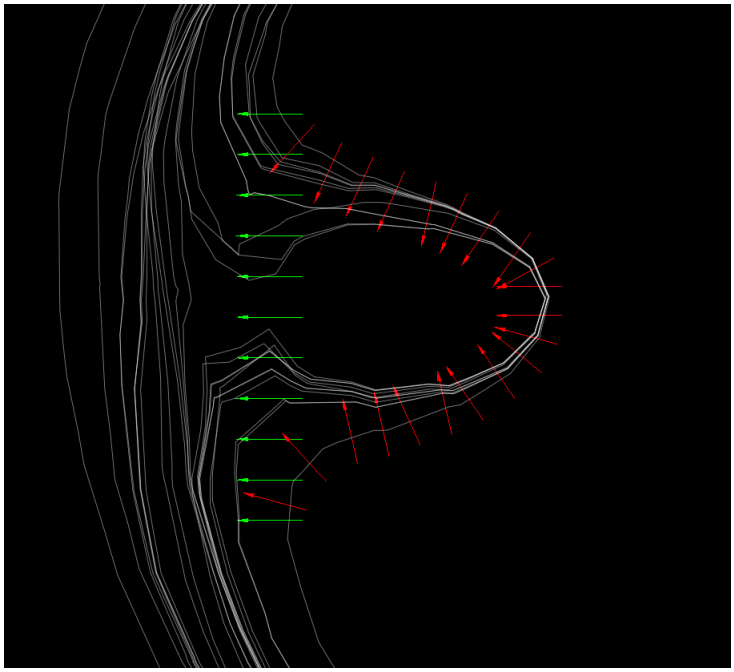


Figure 7.19: When a vessel is present in the slice, contours that ablate the vessel behave in a different way than contours that do not fully ablate the vessel. Consequently, both types of contours need separate contour normals in this region. The normals in red originate from the actual contour intersection of the whole ensemble and are used for sampling the non-ablating contours. The relevant portion around the vessel of all origins is extracted by a convexity-analysis. The green normals are used for sampling the vessel-ablating contours. They are created by connecting the first and last origin of the red normals and interpolating normals across the resulting line.

the number of all contours in the ensemble. The resulting percentage is a per-slice measure; i.e., slices other than the current one are not taken into account in this measure, because the visualization module does not include a segmentation algorithm. Therefore, vessels on different slices cannot be associated to one another. The likelihood is displayed when the user hovers the mouse over a certain vessel. After 500ms of resting on the vessel, the label appears.

Furthermore, it may also be of interest for the user to get a sense of what the medians around a vessel of only ablating contours and only non-ablating contours are. The median contour of ablating contours and non-ablating contours is easily obtained by using their respective normals. This time,

location correction and adjustment of the number of contours do not have to be performed. The median is visualized as an overlay to the global contour boxplot when the user zooms into a vessel. Both concepts, the likelihood of ablation and the median of ablating and non-ablating contours, are shown in Figure 7.20.

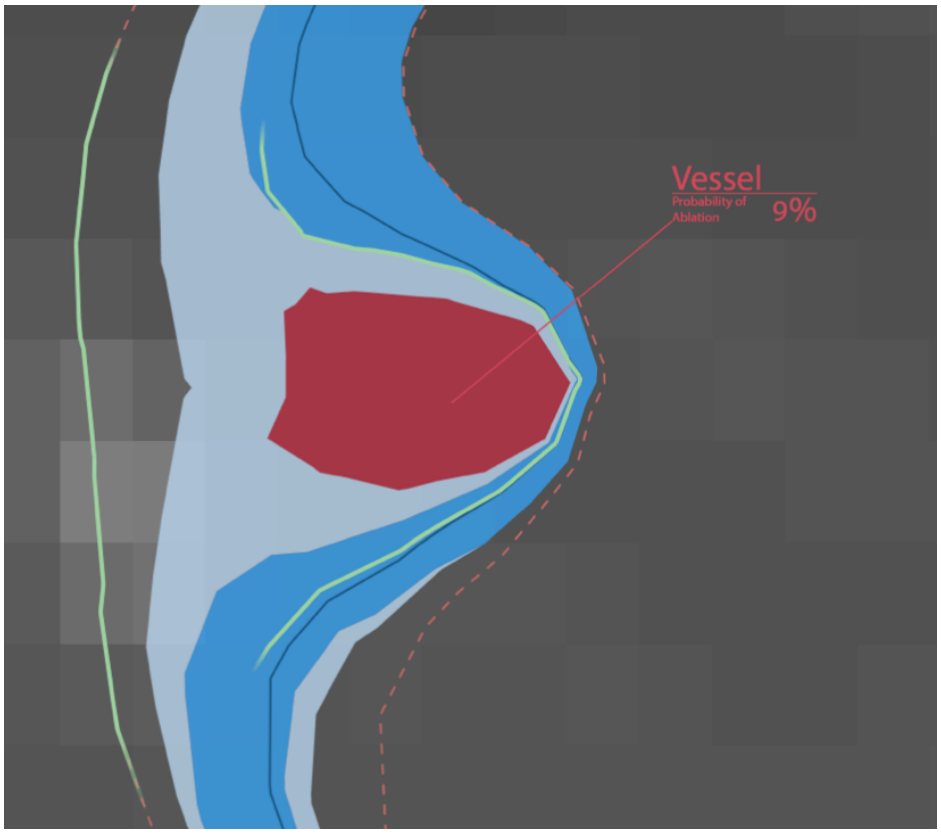


Figure 7.20: Medians of ablating- and non-ablating-contours of a specific vessel are visualized as an overlay on top of the contour box plot (green). The vessel itself and the likelihood of ablation for the given dataset is visualized in red.

7.6 Local grouping of contours

In addition to the global contour boxplot, the Uncertainty Visualization module also displays a local overlay of groups of contours when the user zooms into a window. Contours that lie close together belong to the same group, where grouping is performed along the contour normals with which the ensemble is sampled using kernel density estimation (KDE).

Kernel density estimation As the name suggests, KDE performs an estimation of the underlying density of data points. In statistics, this is often used to estimate the probability density function of a random variable. In the visualization plugin, contours that lie closely together should be assigned to the same group. The notion of lying close together along a contour normal can also be viewed in terms of how dense the intersections of the contours with a contour normal are. The more intersections are included in a certain area along a contour normal, the closer together the contours lie and the denser the intersection distribution. Thus, KDE is used as a univariate group classification algorithm of the contours based on their densities. All KDE related equations and definitions are taken from Silverman [70, ch.13]. First of all, the univariate KDE is defined as:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{x - X_i}{h}\Big) \tag{43}$$

where $\hat{f}(x)$ is the estimated probability density function, $K(x)$ is a kernel, h is the kernel bandwidth and $X_i, i \in [1; n]$ are the data points. As can be seen in Equation 43, both the kernel and its bandwidth have to be chosen manually in order to calculate the estimated probability density. In general, every possible kernel $K(x)$ that satisfies the following condition can be used:

$$\int_{-\infty}^{\infty} K(x)dx = 1 \tag{44}$$

In practice, however, few kernels have emerged that are regularly used, such as the Cauchy kernel, the Gaussian kernel and others. The resulting shape of the estimated probability density depends on which kernel was used for the KDE. For the particular application of grouping the contours, the Gaussian function appears most suitable. However, other kernel types could be used as well, because the actual shape of the probability density is irrelevant in this context. When choosing a symmetric kernel, such as the Gaussian one, Equation 43 can also be viewed as the convolution of the data points with the kernel itself, where the data points are viewed as shifted unit pulses. Choosing an appropriate kernel bandwidth is a crucial task when performing a KDE. Depending on the bandwidth, the resulting distribution may be under-smoothed or over-smoothed, see Figure 7.21. To address this problem, several methods have been developed to select the bandwidth automatically, and there is still an ongoing discussion in literature on which method to use under which circumstances [71]. For grouping contours in the visualization module, however, what needs to be estimated is not the underlying probability density function of the contour intersections along a contour normal.

Rather, groups based on the density of those intersections are of interest. Thus, the user should actually be able to specify which density measure accounts for building separate groups. Depending on whether the user wants a more fine-grained view, i.e., many different groups, or if they want a rough overview, e.g. a classification into only two groups, the kernel bandwidth has to be adjusted accordingly. Hence, the kernel bandwidth remains an adjustable parameter in the visualization module.

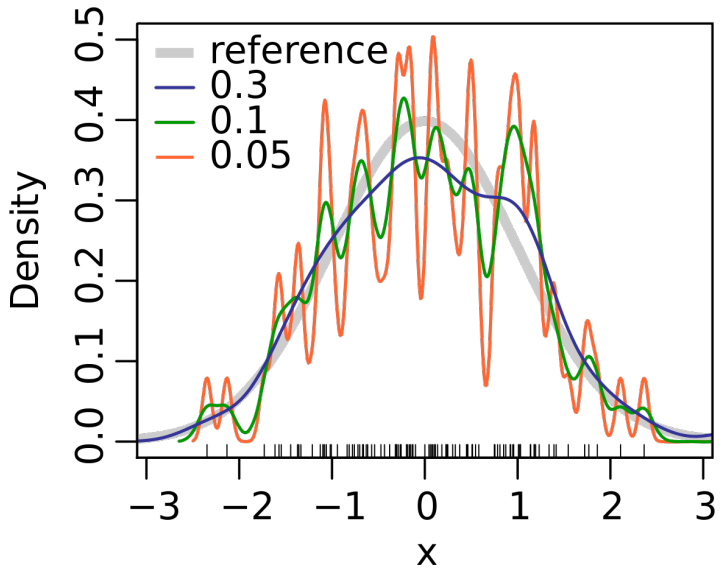


Figure 7.21: A comparison of different kernel bandwidths of a KDE using a Gaussian kernel. The gray curve is the original reference. Other curves are generated using the respective kernel bandwidth in the plot legend. The black strokes on the bottom represent the data points. As can be seen in the figure, the red and green curves exhibit a pronounced under-smoothing of the original probability density. The blue function is closer to the original, but still introduces erroneously an additional local minimum. Figure taken from Toews [72].

In order to perform the classification, contours are split into different groups where $\hat{f}(x)$ contains local minima. Each local minimum corresponds to a group border which separates two groups from one another. Due to the fact that, with this approach, only the local minima need to be computed, Equation 43 can be simplified. The final equation for extracting the local

minima out of the density distribution of $\hat{f}(x)$ is as follows:

$$\min_x \Big\{ \frac{1}{nh} \sum_{i=1}^{n} K\Big(\frac{x - X_i}{h}\Big) \Big\} = \min_x \Big\{ \sum_{i=1}^{n} K\Big(\frac{x - X_i}{h}\Big) \Big\} \tag{45}$$

Adjusting the contour normals The contour normals, generated as explained in Section 7.4.3, do not necessarily have the optimal direction for applying the KDE. The direction of the normals depends on the overall set-theoretic contour intersection of the ensemble. Consequently, the approximation of the actual segment normals with the computed contour normals is usually better in segments near the ensemble center. For contours that are further away, however, the contour normal does not approximate the respective segment normal well. For the computation of the contour boxplot, this circumstance does not influence the outcome in a significant way, since only the locations of the contour intersections relative to each other are important. For estimating the density of the contours, the absolute distances of the intersections are important in order to get reasonable results. Thus, a correction of each contour normal is performed by minimizing the deviation from $\frac{\pi}{2}$ of the angles between the contour normal and the segments it intersects. This is analogous to minimizing the deviation of the dot product of the contour normal and the segments it intersects from zero:

$$\min_{\phi \in [-\frac{\pi}{2}; \frac{\pi}{2}]} \Big\{ \sum_{i=0}^{k} (\mathbf{n_d}(\phi) \cdot \mathbf{s_d}_i)^2 \Big\} \tag{46}$$

where $\mathbf{s_d}_i$ are the directions of the segments that the contour normal intersects with, and $\mathbf{n_d}(\phi)$ is the direction of the normal to correct. The normal itself is defined as:

$$\mathbf{n}(\phi) = \mathbf{p} + s \cdot \mathbf{n_d}(\phi) = \mathbf{p} + s \begin{bmatrix} \cos(\phi) \\ \sin(\phi) \end{bmatrix} \tag{47}$$

where $\mathbf{p}$ is the origin of the normal, $s \in [0; \infty)$ is the parameter of the ray (refer to Section 7.2.1) and ϕ is the angle of the normal with respect to the abscissa. Once the ideal direction has been found, the intersections with the segments are re-calculated. Due to the fact that the KDE along a contour normal is a local measure, intersections of the contour normals can be tested if they are inside the viewing bounding box. If at least one intersection is inside the bounding box, the correction has to be performed - otherwise not. Because the local group overlay is only active in a deep zoom level, the amount of normals to correct is reduced to a small portion of the total number of contour normals.

Visualizing the contour groups The groups are visualized by connecting the group borders of adjacent contour normals, if the border is located between intersections of the same two contours, and those contours did not cross between the normals. If a border connection contains only a small amount of contour normals, its opacity is reduced to signify that the group extent is comparatively small.

When the user hovers over a group, the probability that a contour falls into that group in the given dataset is displayed in a similar manner to the label that displays the probability of ablating a vessel (Section 7.5). The probability is calculated by dividing the number of the contours in the group by the number of contours in all groups. In addition to the label, an interpolated contour normal is shown beneath the mouse cursor to indicate where the probabilities are obtained. An example of the visualization can be seen in Figure 7.22.

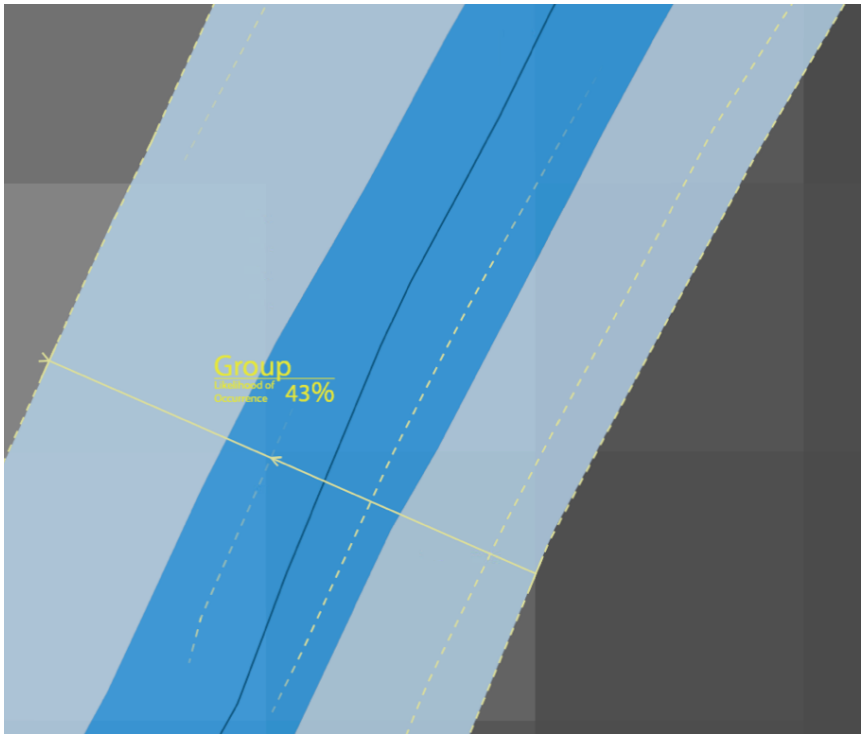


Figure 7.22: Example of the visualization of locally grouped contours. Connected group borders are rendered as dashed yellow lines. An interpolated contour normal is displayed when the mouse hovers above a certain group. The arrows on the normal denote the group border. The label displays the probability that a contour falls into that group.

7.7 Vessel filter and threshold filter

The last functionality of the visualization module is the ability to specify filters for certain criteria. There are two filters available: the "vessel filter" and the "threshold filter".

The vessel filter allows the user to filter contours based on whether they ablate a certain vessel or not. There are two options available: either all vessel-ablating contours are displayed, or all contours that fail to ablate said vessel. Once the filter is applied, the contours that are filtered out are deselected in the parameter list of the UI (refer to Figure 7.6). This provides a convenient way for the user to check which parameters lead to an ablation of a vessel and which do not. Furthermore, the filter gives indication of how the ensemble looks if a certain vessel is ablated or non-ablated. The filter can be activated by right-clicking on a vessel. The visualization module analyzes which contours ablate the vessel and which contours do not ablate the vessel and apply the parameter selection accordingly. The filter can be disabled again through the vessel filter preference pane of the UI, see Figure 7.6.

The threshold filter allows the user to specify a certain margin around the median contour. Contours outside this margin are disregarded during visualization. Hence, the margin is used as a threshold to select relevant contours. More precisely, in order for a contour to be displayed, the distance of every contour vertex to the median has to be smaller than the specified threshold. The same way as in the vessel filter, the parameter lists are adjusted where only those parameters are selected that lead to contours that are located within the specified threshold. Thus, the filter provides a convenient way for the user to sort out parameterizations that do not fall within a certain desired range.

In order to compute the distances of all contours to the median, a distance field is generated using a VTK filter called *vtkDistancePolyDataFilter*. This filter is used to generate an image consisting of the absolute distances from each pixel to the median. After the image is generated, each vertex of every contour is transformed into the image space, and the resulting coordinate is used to look up the distance from the vertex to the median in the distance field. If just one distance is larger than the specified threshold, the contour is filtered out, and the corresponding parameters are deselected in the parameter list in the visualization UI.

Part IV
Results

IV Results

8 Performance

This section deals with analyzing the performance of the functional band algorithms, as well as the Parameter Uncertainty visualization technique as a whole and particular steps of the visualization process that were expounded in the previous sections.

8.1 Comparison of band depth algorithms

Methods In this section, the performance of the mBD and $BD_{n,2}$ algorithms introduced in Section 4 are compared with trivial approaches for mBD and $BD_{n,2}$ computation. The reference system is a personal computer running macOS 10.12 with 32GB of RAM and an Intel-Core i7-7700 CPU with 4.2Ghz. All algorithms are implemented in MATLAB R2015b. The test ensemble consists of sine waves with added amplitude and frequency noise. The whole ensemble is sampled with 200 steps between $t \in [1; 10]$. For the individual tests, several different ensemble sizes are compared. An example of a test ensemble with $n = 200$ functions can be viewed in Figure 8.1.

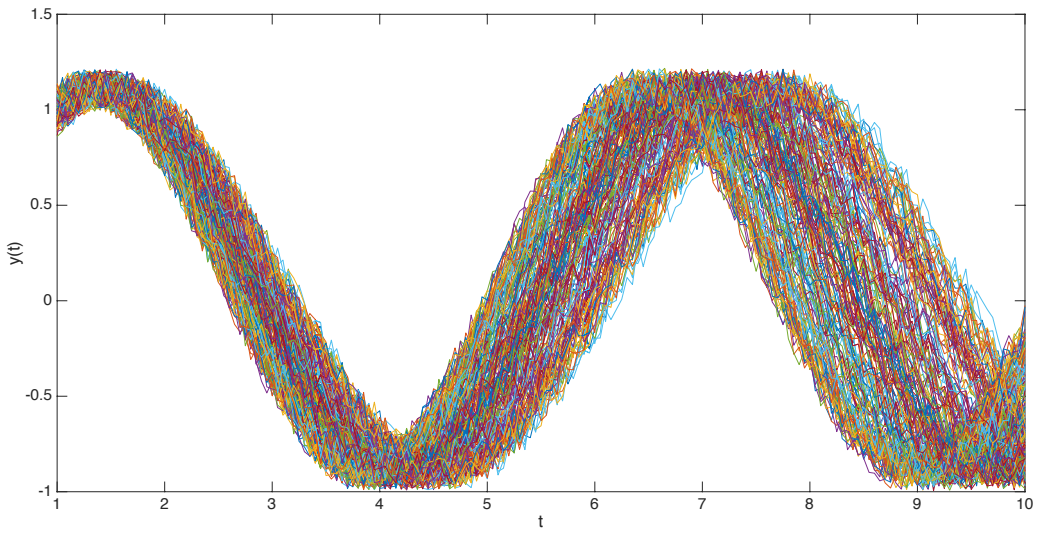


Figure 8.1: Example of a test ensemble consisting of $n = 200$ sine waves with added frequency noise and amplitude noise.

8.1.1 Modified band depth

In order to compare the new mBD algorithm with the trivial approach, $mBD_{n,2}$ and $mBD_{n,3}$ values are computed. Table 2 gives the mean±std.dev. times needed for computing the mBD values for every function in the ensemble. For each ensemble, the sample size was 100. Due to the fact that the computational complexity of the new approach does not depend on k, the times for $k = 2$ and $k = 3$ are comparable. However, it is important to note that subbands of second order are not considered in the evaluation in the case of $k = 3$. Otherwise, the times for $k = 2$ have to be added in order to get a sense of how long the computation takes in both cases for individual ensemble sizes. Note: due to the extensive computation time needed in the trivial approach, $mBD_{600,3}$ was not computed.

Table 2: Comparison of the mean±std.dev times for computing mBD with $k = 2$ and $k = 3$ using the new approach and a trivial approach. Times are given in seconds.

	$k = 2$ mean±std.dev. times in s		$k = 3$ mean±std.dev. times in s	
n	new approach	trivial approach	new approach	trivial approach
50	0.09±0.00	0.24±0.01	0.08±0.00	0.51±0.02
100	0.18±0.00	1.92±0.01	0.17±0.00	8.52±0.22
200	0.37±0.00	15.89±0.07	0.35±0.01	140.65±2.66
300	0.58±0.01	22.12±0.70	0.58±0.04	735.91±18.11
400	0.8±0.00	133.93±0.43	0.72±0.02	2273.5±11.57
500	1.03±0.01	261.73±3.50	0.95±0.01	5605.4±81.51
600	1.28±0.03	461.52±16.46	1.20±0.01	-

8.1.2 Band depth

The comparison between times needed to compute the $BD_{n,2}$ for every function in the ensemble with the new approach and the trivial approach is listed in Table 3. Again, mean and std.dev are recorded with a sample size of 100. As can be seen in the table, with the new approach, computation times stay under a second in almost all cases, whereas $BD_{n,2}$ computation using the trivial approach takes over a minute in ensembles with $n > 600$.

Table 3: Comparison of the mean±std.dev times for computing $BD_{n,2}$ using the new approach and a trivial approach. Times are given in seconds.

	$BD_{n,2}$ mean±std.dev. times in s	
n	new approach	trivial approach
50	0.01±0.00	0.03±0.00
100	0.03±0.00	0.26±0.02
200	0.08±0.00	1.95±0.03
300	0.16±0.00	6.71±0.15
400	0.30±0.02	16.01±0.32
500	0.47±0.03	31.32±0.54
600	0.67±0.03	53.96±0.68
700	0.88±0.23	85.06±1.14
800	1.14±0.05	141.48± 2.39

8.2 Radiofrequency ablation simulation ensemble visualization

Methods The reference system is a personal computer running Windows 8.1 64 bit with 16GB of RAM, an Intel-Core i7-4771 CPU with 3.5Ghz and the NVIDIA GeForce 770GTX GPU. All tests are performed on a simulation ensemble comprising 25 simulated lesions. The parameter space consists of the tissue perfusion and tumor perfusion parameter, both sampled in equal-sized steps from $26ml/100/min$ to $46ml/100/min$, where 5 samples are taken per parameter. The whole ensemble consists of 23 slices in the axial direction, 26 slices in the coronal direction and 33 slices in the sagittal direction, leading to a total of 19,734 distinct combinations of the slices.

All bands are rendered with full opacity. The median is rendered as a solid line and outliers are dashed. Unless otherwise specified, the zoom level remains at default value, and no overlays are rendered on top of the contour boxplot.

8.2.1 Contour boxplot

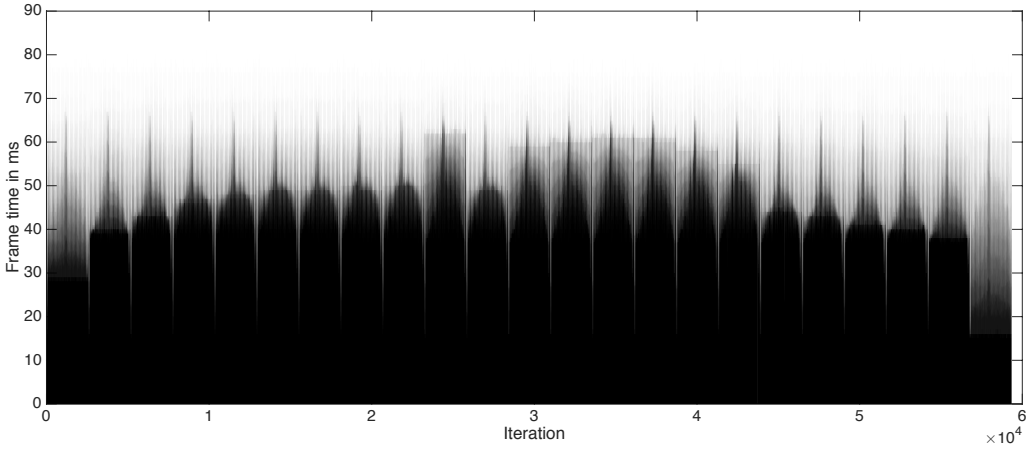
Performance of rendering the data set In order to get an overview of the performance of rendering the contour boxplot, Figure 8.2 shows the frame times for rendering all possible slices of the simulation ensemble in single-threaded and multi-threaded mode. The shape of the figures reflects the selection of the rendered slices: First, the axial slice is set, and for each of the axial slices, all coronal slices are stepped through. For each coronal slice, all of the sagittal slices are selected, thus resulting in all possible slice combinations for the viewing directions. Each combination is rendered consecutively in all three views. Due to the fact that the simulated lesions have a larger extent in central slices, the frame time pattern has a wave-like shape.

The mean/standard deviation of the frame times are 49.4ms/11.2ms (mean frames per second: 20.2) in the single-threaded case and 38.6ms/12.2ms (mean frames per second: 25.9) in the multi-threaded case respectively, which is a reasonable performance for real time rendering the data set in 2D.

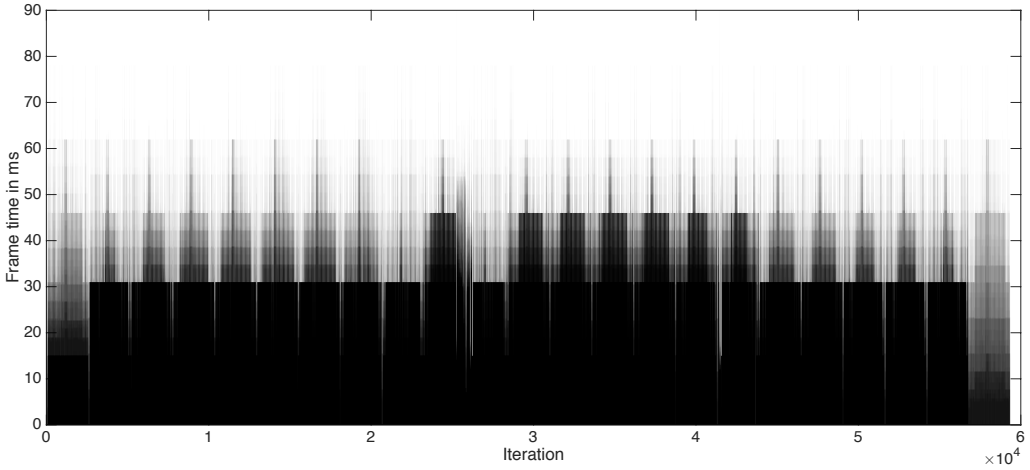
The frame time dependency on the vertex count is further elaborated in Figure 8.3, where the "plateauish" behavior of the frame times in the multi-threaded case becomes more apparent. While in the single-threaded case, the frame time to render the ensemble increases steadily with the number of vertices. Only the maximum frame times in the multi-threaded case seem to increase. This implies that the performance in the multi-threaded case is not as much affected by the number of vertices to process as it is in the single-threaded case; instead, other factors (e.g., vessels, overall shape) are likely to have a significant influence on the rendering performance as the relevance of the vertex count is decreased. In order to get the complete picture, the histograms of the frame times are shown in Figure 8.4. As can be seen in the figure, the frame time distribution of the multi-threaded case is more narrowed with high frequencies around the central frame times. In contrast, the single-threaded case shows more evenly distribution frame times, while still having its maximum at the distribution center.

Performance of particular processing steps Visualizing the contour boxplot requires several distinct steps. Concluding with this section, the performance of executing these steps is analyzed - again in the single-threaded and multi-threaded case. The analyzed steps are described in detail in sections 7.4.1-7.4.5. The single-threaded case is shown in Figure 8.5a), and the multi-threaded case is depicted in Figure 8.5b).

As expected, the multi-threaded version shows lower (or equal) median



(a) Single-threaded case



(b) Multi-threaded case

Figure 8.2: Comparison of the frame times in the single-threaded case (a) and the multi-threaded case (b) that are needed for rendering the whole simulation ensemble in all possible slice combinations. The wave-like nature of both figures resembles the method of slice selection for rendering. Due to the fact that contours in central slices contain more vertices to be processed, the propensity towards higher frame times increases globally as well as locally in central regions. While the frame times in the single-threaded case show a continuous approach towards the high peaks, the frame times in the multi-threaded case show the tendency to remain at certain plateaus.

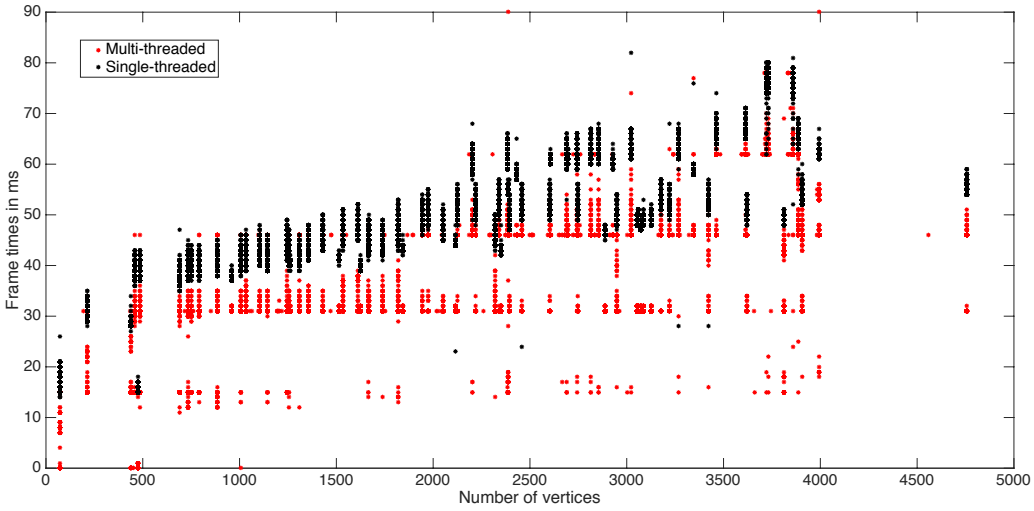
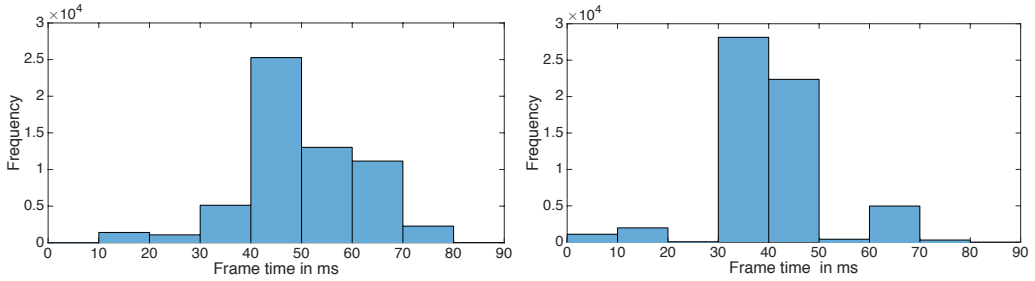


Figure 8.3: Comparison of frame times growth with vertex count. In the single-threaded case, the frame times for each vertex count seems to steadily increase. In the multi-threaded case on the other side, the minima of the frame times largely remain at a certain plateau while only the maxima increase with increasing vertex count, thus leading to an enhanced influence of other factors.

frame times in almost all tasks. The only exception is "band depth calculation", where the multi-threaded version is actually slower, but both in the single-threaded version and in the multi-threaded version, the time needed to complete the operation is negligible. Nevertheless, since the single-threaded case is about three times faster than the multi-threaded case, the task of calculating the BD is executed in a single thread by default. The biggest difference in performance is observable in generating the 2D data, where the median time to complete the task is about three times lower in the multi-threaded case than in the single-threaded case. This difference also explains the overall median frame time difference of about 10ms in the single-threaded and multi-threaded case for rendering the boxplot, refer to Figure 8.2.

8.2.2 Number of contour normals

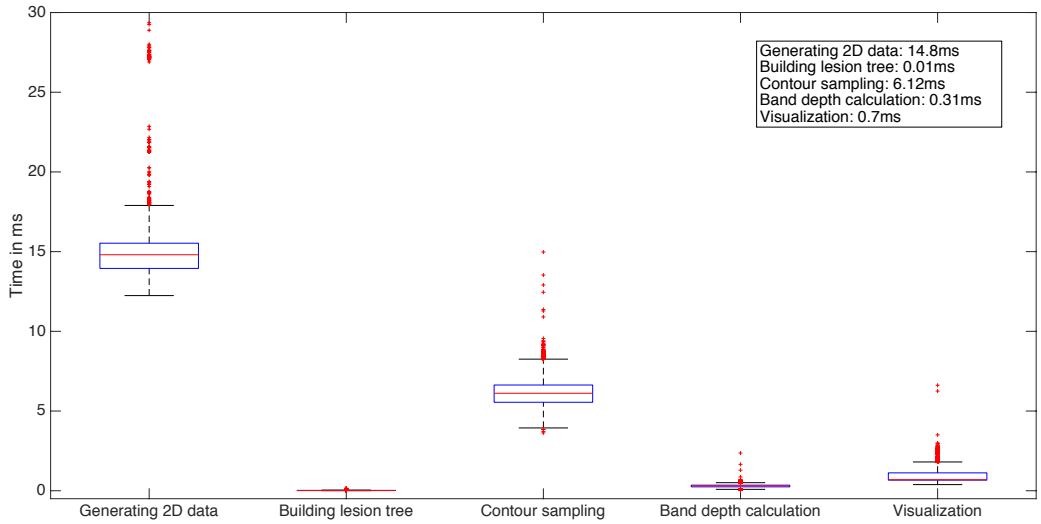
Increasing the number of contour normals effectively decreases the sampling interval and yields a better overall sampling rate of the contours. Especially in cases where the overall shape of the simulation ensemble is convex, the sampling rate decreases towards the outside of the ensemble due to the nature



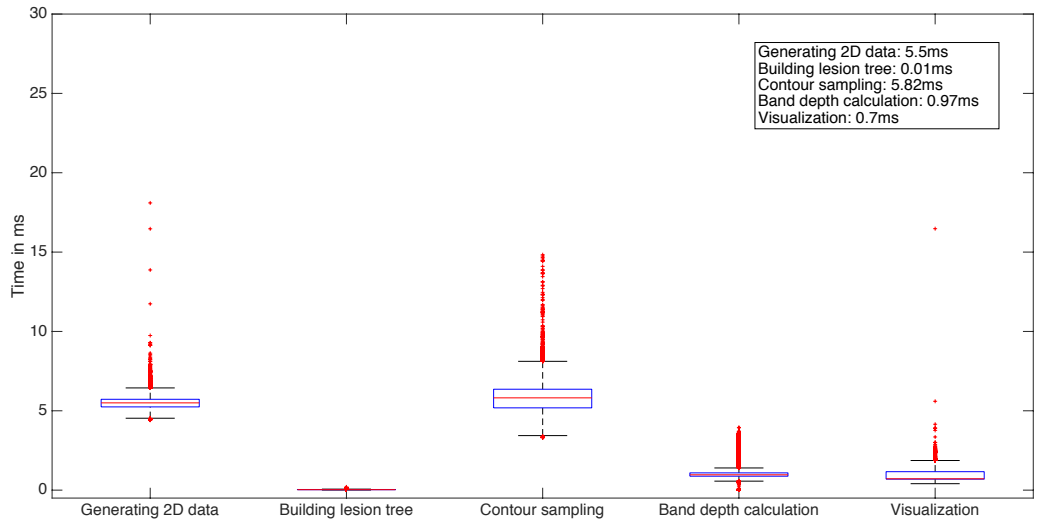
(a) Single-threaded case (b) Multi-threaded case

Figure 8.4: Distribution of the frame times in the single-threaded case (a) and the multi-threaded case (b). Both distributions have their maximum frequency located around their center which is in the 40-50ms region in the single-threaded case and the 30-40ms region in the multi-threaded case. The shape of both distributions differs further in the sense that in the multi-threaded case, frequencies are more narrowed down to the center.

of the contour normals. The opposite is true for non-convex regions, where the sampling rate is lower at the inside of the ensemble. Thus, computing a large number of contour normals seems to intuitively yield a more accurate lBD result. However, a larger number of normals means a larger number of intersections to compute, which consequently may have a performance impact. Figure 8.6 shows the decrease in performance dependent on the number of interpolated normals, as well as the change in lBD values between different interpolation iterations for all of the simulated lesions. The interpolation is done in an exponential manner, where, in the n-th interpolation iteration, $2^n - 1$ normals are generated between two base normals. As in Section 8.2.1, the frame times and lBD values of all possible slices of the test data were taken into account. Multi-threading was enabled.



(a) Single-threaded case



(b) Multi-threaded case

Figure 8.5: Times needed to complete various tasks for rendering the contour boxplot in the single-threaded case (a) and in the multi-threaded case (b). The legend on the top-right shows the median times of the tasks. The task "generating 2D data" constitutes the biggest difference in time between the single-threaded and the multi-threaded case. Interestingly, the BD calculation is actually about three times slower in the multi-threaded case.

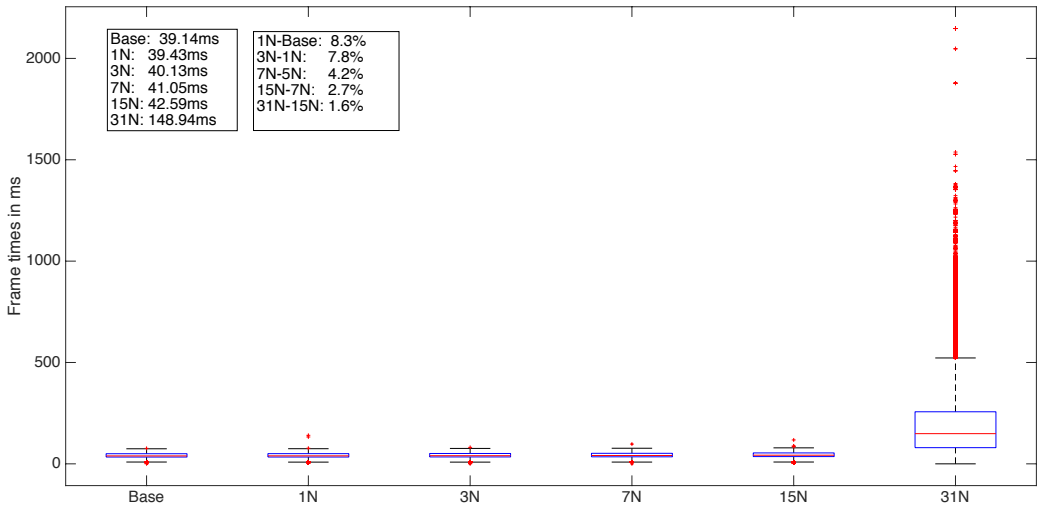


Figure 8.6: Frame times for different numbers of interpolated normals. The abscissa shows the number of interpolated normals, where 3N means three interpolated normals between two base normals. "Base" refers to the standard configuration with no interpolated normals. The median values are shown in the left legend on the upper-left corner. The percentual changes in lBD values is shown in the right legend. As is evident in the plot, interpolating up to 15 normals yields more or less the same frame times. A large increase in frame time is observable when interpolating 31 normals between two base normals. However, going from 15N to 31N only yields a 1.6% change in lBD values.

8.2.3 Overlays

In order to conclude the performance analysis, the time for computing and visualizing the overlays is shown in Figure 9.1. The analysis comprises the processing times of several different vessels and various local contour groups. Processing vessels includes the rendering of the vessel itself, highlighting the median contour of ablating and non-ablating contours and showing the probability of a vessel being ablated. Local contour grouping comprises of computing the groups with the uni-dimensional KDE, visualizing them and displaying the likelihood that a group appears. As can be seen in the figure, the median times for both overlays are 1.7ms, which does not have a great performance impact on the overall rendering times. However, vessel processing yields a higher variability and, consequently, more outliers than local contour grouping, and the maxima for both overlays are in the region

of 3ms. Note that, in this analysis, the comparison between single-threaded and multi-threaded versions is omitted, because both overlays are processed in a single thread only.

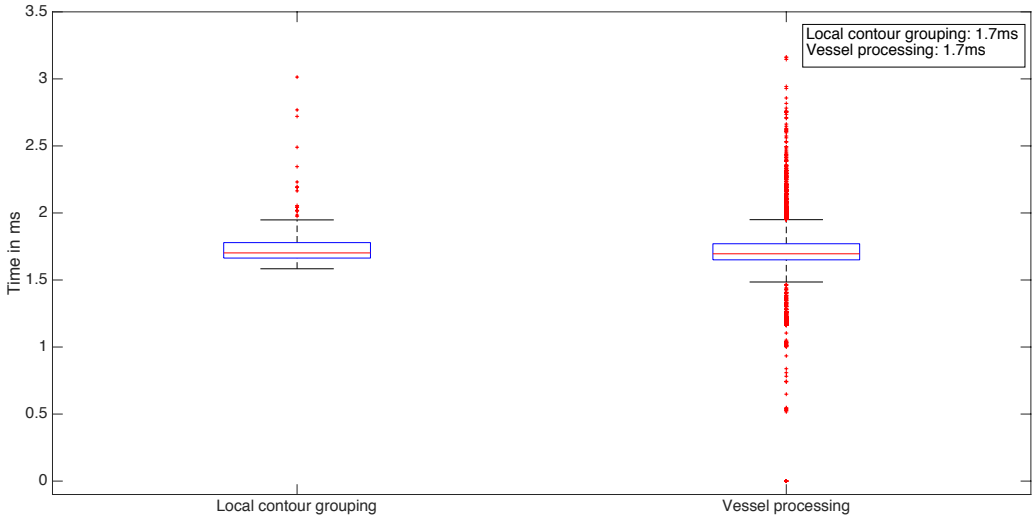


Figure 8.7: Boxplots of the times needed to process both overlays of the Parameter Uncertainty plugin. The legend in the upper-right corner shows the median times for both operations. Coincidently, both medians are equal.

9 Qualitative Results

This section will give a summary of the main features of the Ensemble Visualization module of the Parameter Uncertainty plugin by presenting the concepts in Section 7.

9.1 Contour boxplot representation

When visualizing a simulation ensemble, the users are presented with the boxplot of the ensemble in the three orthogonal 2D views. Using the boxplot, the users can easily locate the median contour and infer the probable extent and shape of the resulting lesion. Moreover, outliers can easily be distinguished. Thus, the boxplot provides a global view of what results to likely expect from treatment. An example of boxplots in the three orthogonal views of the RFA Guardian can be viewed in Figure 9.1. Furthermore, the parameterization can be adjusted dynamically and the visualization will

adapt to the changes. This is especially useful when more knowledge about a certain parameter arises, as variations typically get smaller and the boxplot narrows down. For instance, the users may want to fix the first parameter, and investigate the outcomes using only variations of the second parameter. An example of this case is depicted in Figure 9.2.

Additionally, the users may investigate which parameterizations lead to an ablated area within a certain margin around the median. This can be investigated using the threshold filter. The user may enter a specific margin in mm, and the plugin only visualizes the outcomes that are located within the given margin. The parameter list in the user interface (refer to Section 7.3) changes accordingly, where only parameterizations that lead to contours within the given margin are enabled, Figure 9.3.

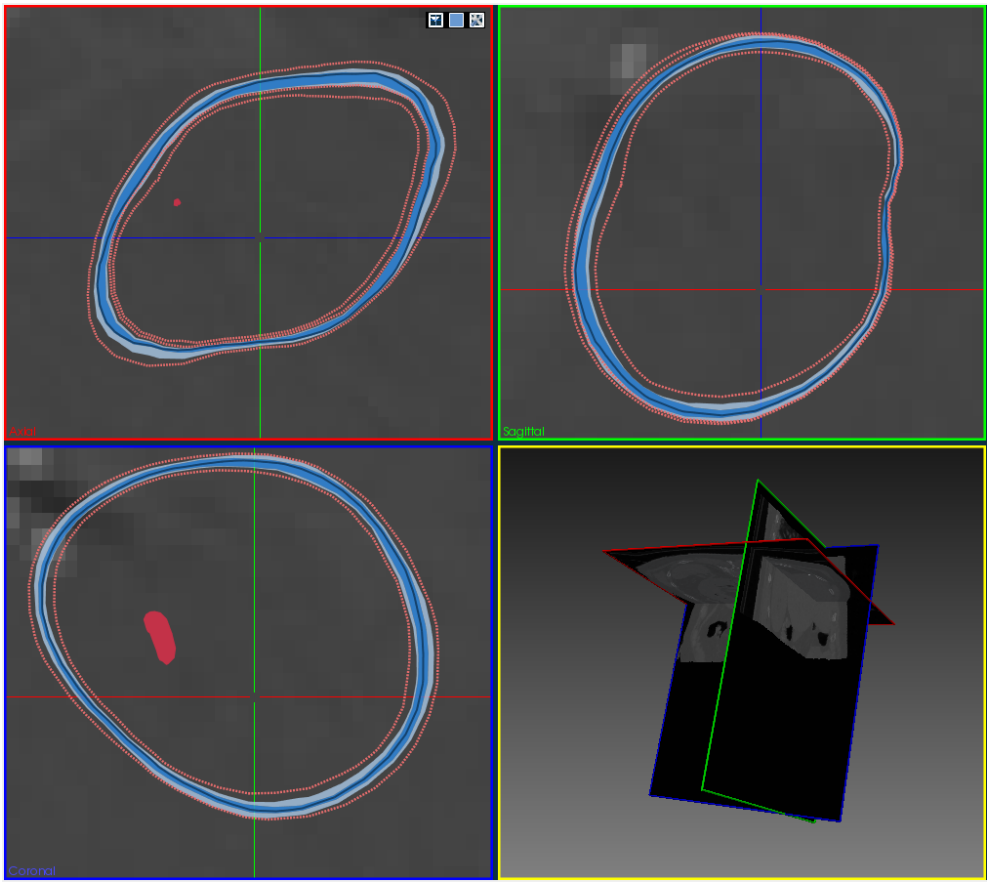
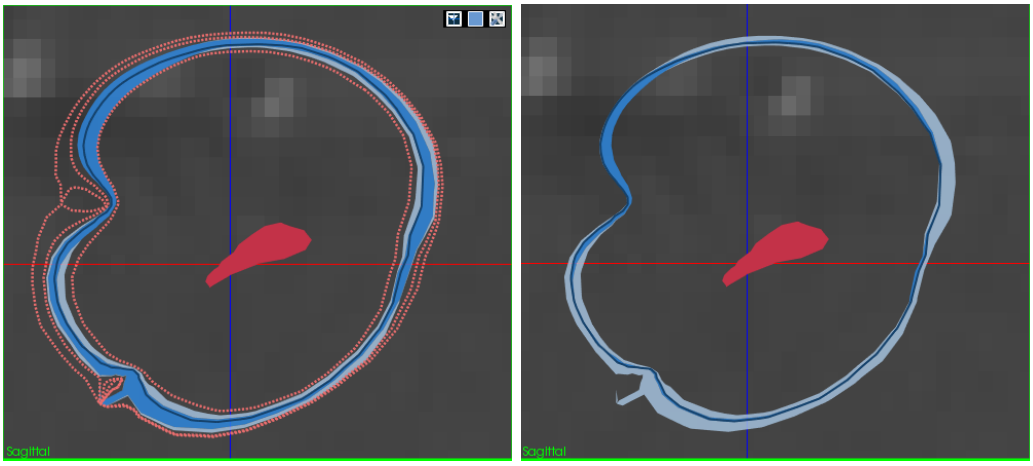


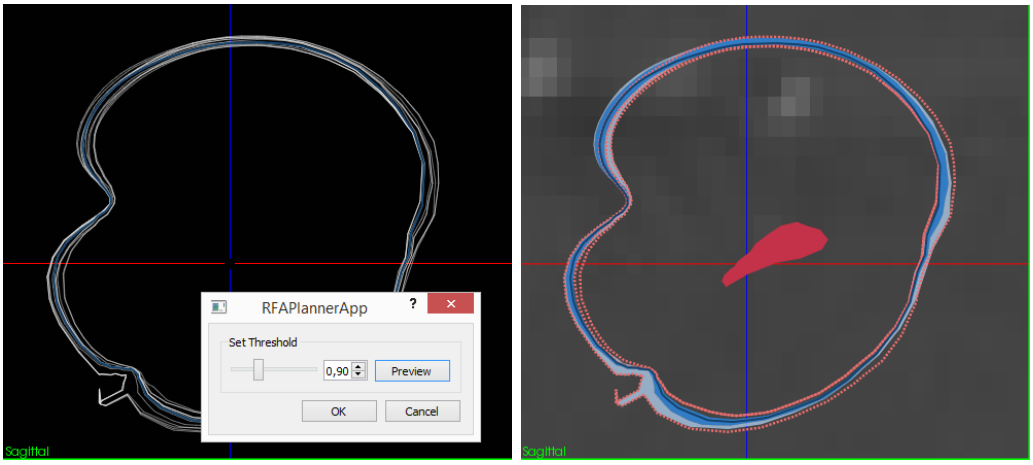
Figure 9.1: The simulation ensemble is visualized using contour boxplots. The median, 50%-region and fence are depicted in shades of blue. Outliers are dashed red. Furthermore, blood vessels are colored in red too.



(a) Initial boxplot

(b) Boxplot using only a subset of the initial parametrization

Figure 9.2: Typically, a boxplot narrows down if a subset of its parameterization is selected due to a decrease of uncertainty. In this example, a single parameter is fixed. Consequently, the initial boxplot in (a) does not contain outliers any more and its 50%-region is significantly less pronounced (b).



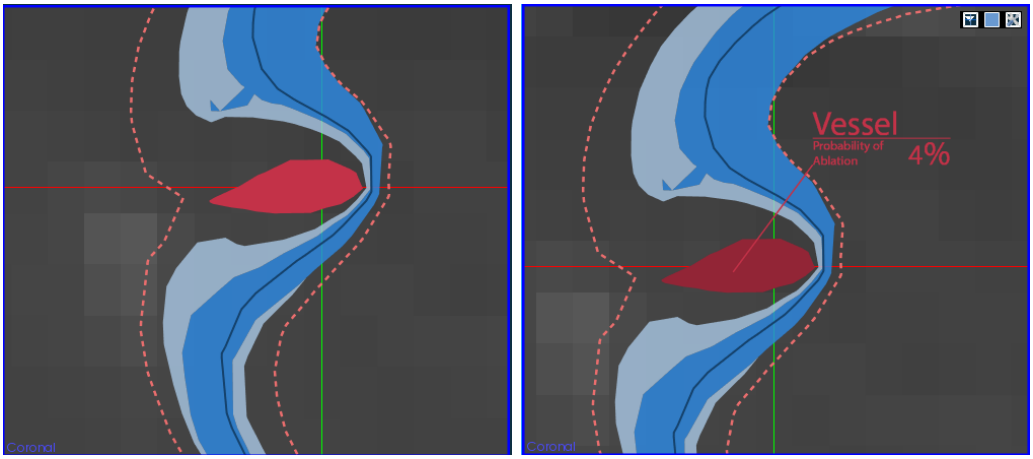
(a) Applying threshold filter

(b) Boxplot using filtered parameterization

Figure 9.3: By applying the threshold filter (a), a margin around the median can be specified in which simulation outcomes have to be located in order to get visualized. The parameter lists adapt to the changes where only parameters that lead to visualized results are enabled. The filtered results are depicted in (b).

9.2 Investigation of vessel ablation

In some situations, it may occur that a large vessel is located near the estimated ablation zone and it is not clear if this vessel will be ablated in the treatment or not. Thus, the radiologist may invoke the Parameter Uncertainty plugin to evaluate whether the vessel will be ablated or not. In the example presented in Figure 9.4a, the investigation of the vessel in the coronal view of the contour boxplot already reveals that the vessel is only ablated by an outlier of the ensemble. For further investigation, the probability that the vessel is ablated, can be displayed, Figure 9.4b. Applying the vessel filter reveals the specific parameterizations that lead to the ablation of this vessel, which, in the the example of Figure 9.4, needs a very low tumor perfusion to be ablated. Hence, by using the additional information gained by the Parameter Uncertainty plugin, the interventional radiologist is able to make inferences about vessel ablation, and to consequently adapt the treatment plan to the newly gained knowledge.



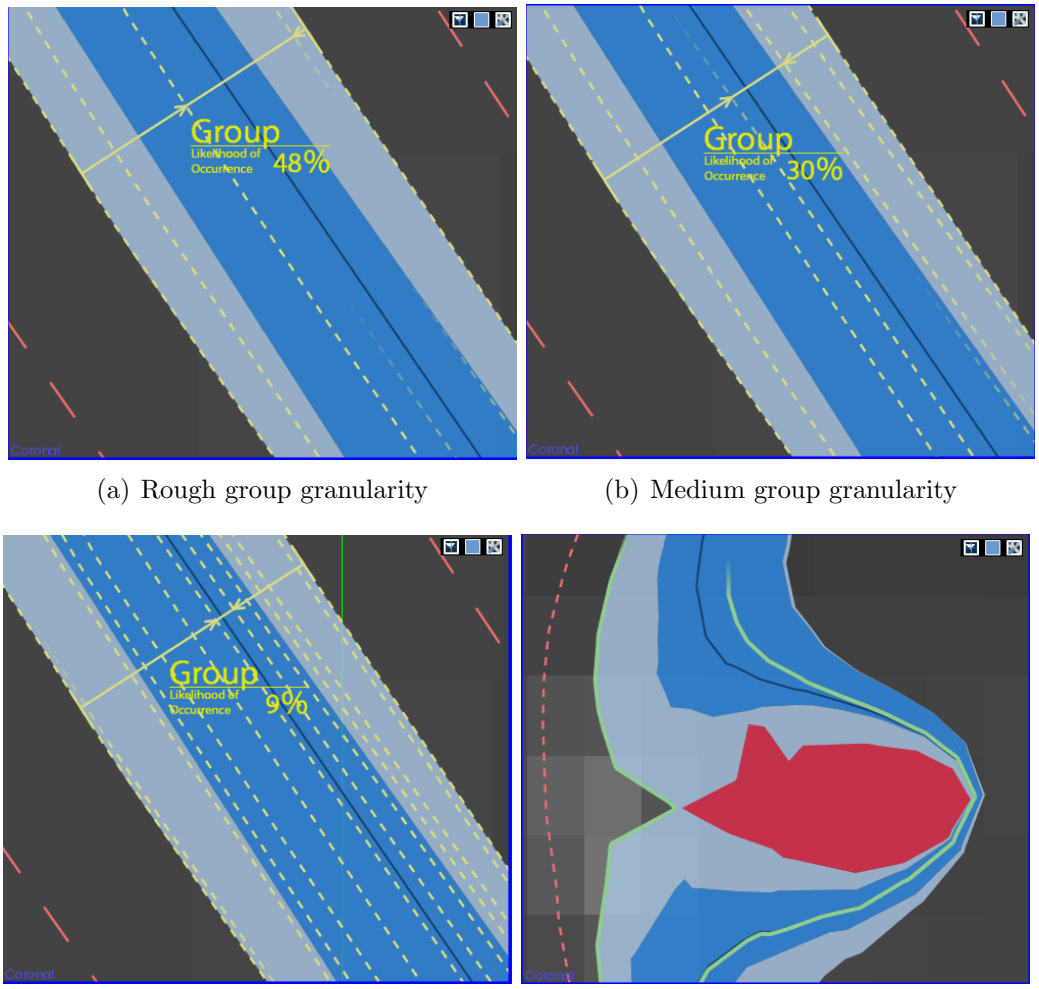
(a) Vessel penetrating the ablation zone (b) Probability of vessel ablation

Figure 9.4: Example of a vessel penetrating the ablation zone. As visible, only an outlier (dashed red contour) ablates the vessel. The fence of the contour boxplot is visualized in light blue and the median is the dark blue contour (a). To get a sense about how likely it is that the vessel will be ablated, the probability of ablation can be displayed for each vessel individually (b).

9.3 Local characteristics

The contour boxplot provides a concise global representation of the simulation ensemble. However, in some cases, knowledge about the local behavior of lesions is required. For instance, the users may want to know how likely it is that a certain local tissue region is ablated. By utilizing the local grouping functionality of the plugin, users are given the ability to investigate how many parameterizations fall within a certain group of simulation outcomes. The granularity of these groups is adjustable; i.e., depending on the extent of the local region of interest, users may fine-tune the kernel width of the KDE to produce group extents based on their preference, Figure 9.5a-c.

Other times, it may be of importance to know the local behavior of contours that ablate a vessel and contours that do not ablate a vessel. In addition to the vessel-related functionalities regarding vessel ablation (Section 9.2) the local median around a certain vessel can be displayed as an overlay upon zooming into the vessel. This local overlay provides a rough overview of how the ensemble median would change, given the vessel is ablated or not, without having to apply the vessel filter. An example is given in Figure 9.5d. In this instance it is obvious that the global median would correspond to a more inlying contour, with respect to the ensemble center.



(a) Rough group granularity

(b) Medium group granularity

(c) Fine group granularity

(d) Local medians around vessel

Figure 9.5: Local visualization features of the Parameter Uncertainty plugin. To investigate how many simulation results fall within a certain local region of the ensemble, the contours can be grouped based on local density with adjustable granularity (a-c). Furthermore, to get a sense of how the ensemble median changes given a vessel is ablated or not, the median of vessel-ablating and non-vessel-ablating contours is shown upon zooming into a vessel (d).

Part V
Conclusion and future work

V Conclusion and future work

10 New approaches to band depth computation

As visible in the performance comparison of the trivial approaches to compute mBD and BD with the approaches introduced in this thesis in Section 8.1, a significant performance gain for ensembles consisting of hundreds of functions can be achieved. The new approaches can be viewed as a generalization and extension of the algorithms developed by Sun et al. [44]. However, as elaborated in the example in Section 3.2, their BD algorithm yields wrong results for certain ensembles, as already pointed out by Kwon and Ouyang [42]. Furthermore, ranking ties are not considered as well, [45]. Both of these disadvantages are circumvented by the approaches introduced in this thesis while still providing reasonable computational performance.

López-Pintado and Romo [41] recommended using second band order for mBD, as results are very stable in k. Another reason they gave was fast computation for second order bands. By using Equation 11, bands of any orders can now be computed with the same computational complexity. Thus, possible advantages by using higher band orders may be investigated in further studies. For the BD measure on the other side, López-Pintado and Romo [41] recommended using bands of third order. To our knowledge, no fast algorithm for fast $BD_{n,3}$ computation or any higher band order exists to date. However, rudiments of Equation 12 may be used in future work to establish a fast algorithm for $BD_{n,3}$.

11 Parameter sampling

The Parameter Sampling module provides the option to sample parameters in three different fashions, two statistical ones and one deterministic one, as described in Section 6.3. While those methods provide a sophisticated sampling scheme dependent on the prior knowledge about the parameters, the actual selection of parameters is limited. At the current stage, the user is able to select from the tissue perfusion and tumor perfusion and the position of the probe tips. Those parameters play an important role in the outcome of a simulation. However, there are other factors to be considered as well, e.g., cell death severity or electrical tissue parameters (refer to Section 1.2.4). Hence, the sampling module would benefit from adding those parameters to the list of parameters to sample from in future iterations of the plugin

giving the user more options to choose from, thus providing an even more sophisticated way of generating the sampling space.

12 Ensemble Visualization

12.1 Contour boxplot and lesion band depth measure

The technique for visualizing a simulated lesion ensemble introduced in this thesis combines the notion of the contour boxplot with a modified version of the mBD method for generating an ordered statistic used in functional boxplots. While, in the classical functional boxplot, the amount for which a contour remains inside a particular band is computed with respect to the abscissa, the "lBD" method samples the overall ensemble with a finite number of so-called contour normals, which embrace the overall shape of the ensemble. Thus, the amount for which a contour stays within a particular band is not measured against an axis but rather the shape of the band itself.

The performance of this method allows for real-time visualization of the simulation ensemble represented as a contour boxplot in three 2D windows with orthogonal directions. This can be achieved through the utilization of the parallel processing capabilities of the GPU and a pre-generated LUT. In the classic case, calculating the contour band depth for each contour entails the computation of a considerable number of bands. E.g., for 30 contours and bands consisting of three contours, 4060 different bands have to be computed. The method for computing the lBD presented in this thesis circumvents this potential bottle neck by reducing the computation to the overall set-theoretic contour intersection for generating the contour normals and their respective intersections with the contours. Once the intersections are obtained, calculating the lBD values only requires performing a lookup to the pre-generated table. The LUT itself is generated using a new approach to compute mBD values using Equation 11. This equation plays an important role in reducing the frame times needed for generating the overall contour boxplot, as the lookup itself is an operation that can be performed in constant time.

Despite the fact that the bands are not computed using exact contour intersections, but rather with the contour normal sampling method, the resulting lBD values converge fast when increasing the number of normals, refer to Figure 8.6. In the performance analysis of the test data set, the difference of the median frame times between no interpolated normals and 15 interpolated normals is only 3.45ms. However, interpolating 31 normals between two base normals would increase the overall frame time by 106.35ms. Due to the fact that the lBD values only change about 1.6% by going from 15 to 31

interpolated normals, the Parameter Uncertainty plugin uses 15 interpolated normals by default.

12.2 Overlays and filters

Vessels Due to the heat sink effect, an important part of the Parameter Uncertainty plugin lies in visualizing the behavior of the simulation ensemble around vessels (Section 7.5). When a vessel is present in the simulation ensemble, the percentage of the lesions that ablate the vessel on the current slice is displayed on mouse-over. While this can be useful for getting an overview of how likely it is for a particular vessel to be ablated on a certain slice, the actual likelihood of the whole vessel being ablated would be additional useful information. However, the Parameter Uncertainty plugin processes the lesions on a per-slice basis, thus the likelihood for other slices being ablated is not known, because vessels on different slices cannot be assigned to one another. In order to address this shortcoming, a segmentation algorithm for the vessel tree could be incorporated into the Parameter Uncertainty plugin. This would make it possible to label vessels on different slices accordingly and to calculate the overall probability of a vessel to be ablated.

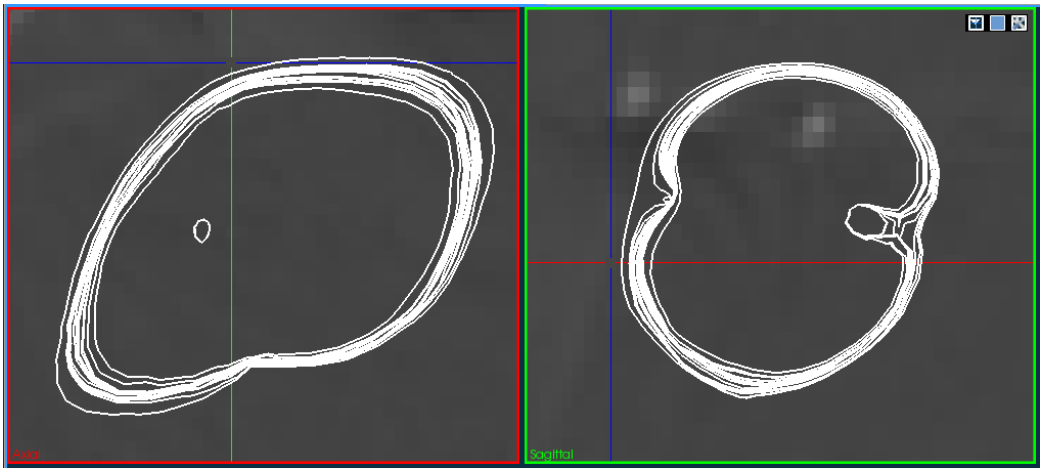
Another important aspect of vessel processing is highlighting the median of ablating contour and non-ablating contour with respect to a particular vessel. This allows the interventional radiologist to further assess the statistical properties of the whole ensemble and to form an opinion in terms of "what is likely going to happen if a certain vessel is ablated/non-ablated". With the help of the vessel filter (Section 7.7), the user may deepen their knowledge about which parameters lead to lesions that fully ablate a vessel and which parameter prevent a vessel ablation. Additionally, the filter helps to get better insight in the shape of the ensemble, provided that a vessel is ablated or not.

Contour grouping The Parameter Uncertainty plugin provides the possibility to perform a local grouping of contours based on their relative density, see Section 7.6. This is realized in a levels-of-detail approach, i.e., groups are only built if the user zooms into the ensemble sufficiently. The action of zooming is interpreted by the Uncertainty Sampling plugin as the user's interest in this local region. By providing the option to specify on how groups should be built, the users can adjust the granularity to their liking. The benefit of the grouping method lies in its locality. While the contour boxplot is a global measure that provides valuable information of the simulation ensemble as a whole, contour grouping gives information about contour properties in the region that is currently viewed. E.g., contours with low lBD value might

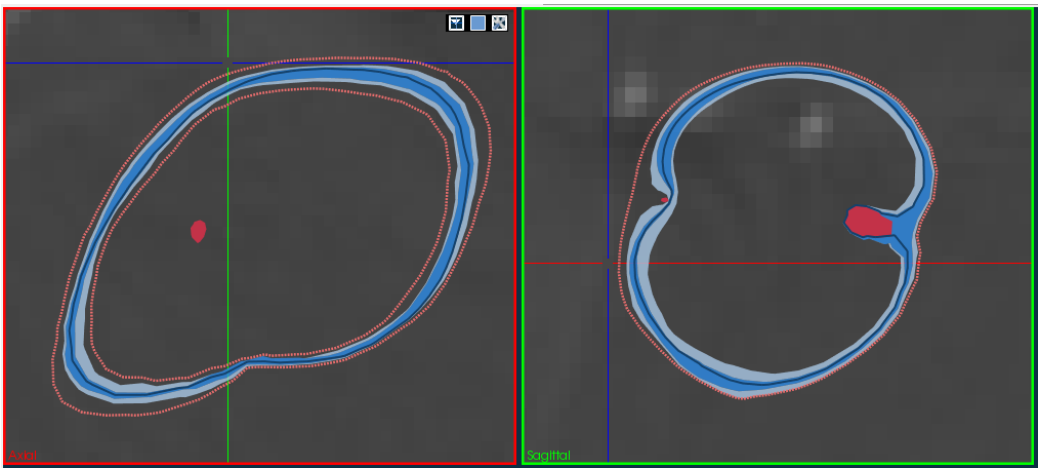
actually be included in a central group in the viewed region. The probability of a contour to fall into a specific group is shown as an overlay on mouse-over in a similar manner as the ablation probability is displayed for a vessel.

12.3 Visualization comparison

In conclusion, a comparison of a standard contour visualization technique and the contour boxplots in two 2D views is shown in Figure 12.1. The Parameter Uncertainty plugin improves the visual representation of the ensemble by not only showing statistical properties through the contour boxplot, but by establishing the connection to the underlying parameterizations as well. While the standard method only visualizes the shape of each contour, the Parameter Uncertainty plugin captures the behavior of contours around vessels and visualizes vessels themselves as well. E.g., when looking on the bottom-right view of the example in Figure 12.1, the user immediately recognizes that the left vessel is likely not to be fully ablated, since the only contour that ablates the vessel is an outlier. In contrast, in the standard technique the same vessel is not even recognizable at first glance, due to the cluttered view. If more information regarding the vessel or contours is needed, the user can make use of the vessel processing techniques, filters, or contour grouping techniques that the plugin offers. Considering these improvements over the standard method, the Parameter Uncertainty plugin may serve as a valuable extension of the RFA Guardian in assisting interventional radiologists in RFA treatment.



(a) Standard visualization



(b) Visualization with the Parameter Uncertainty plugin

Figure 12.1: Comparison of an example ensemble visualized by a standard technique (a) and with the Parameter Uncertainty plugin (b) in two orthogonal 2D views.

References

[1] G. M. Cooper, *Elements of Human Cancer*. Sudbury, Massachuetts: Jones & Bartlett Learning, 1992. Cited on page 3.

[2] W. H. O. Geneva, "Global health estimates 2015: Deaths by cause, age, sex, by country and by region, 2000-2015." 2016. Cited on page 3.

[3] A. Sudhakar, "History of cancer, ancient and modern treatment methods," *J. Cancer Sci. Ther.*, vol. 01, pp. 1–4, dec 2009. Cited on page 3.

[4] A. H. Mahnken, P. Bruners, and R. W. Günther, "Techniques of interventional tumor therapy," *Dtsch. Arztebl. Int.*, vol. 105, pp. 646–653, Sep 2008. Cited on page 3.

[5] A. R. Gillams, "The use of radiofrequency in cancer," *Br. J. Cancer*, vol. 92, pp. 1825–1829, may 2005. Cited on pages 3 and 4.

[6] P. Mariappan, P. Weir, R. Flanagan, P. Voglreiter, T. Alhonnoro, M. Pollari, M. Moche, H. Busse, J. Futterer, H. R. Portugaller, R. B. Sequeiros, and M. Kolesnik, "GPU-based RFA simulation for minimally invasive cancer treatment of liver tumours," *Int. J. Comput. Assist. Radiol. Surg.*, vol. 12, pp. 59–68, jan 2017. Cited on pages 3, 7, and 8.

[7] P. Voglreiter, P. Mariappan, A. Tuomas, H. Busse, P. Weir, M. Pollari, R. Flanagan, H. , D. Seider, P. Brandmaier, M. Van Amerongen, R. Rautio, S. Jenniskens, R. Blanco Sequeiros, R. Portugaller, P. Steigler, J. Futterer, D. Schmalstieg, M. Kolesnik, and M. Moche, "Rfa guardian: Comprehensive simulation of the clinical workflow for patient specific planning, guidance and validation of rfa treatment of liver tumors," *Int. J. Comput. Assist. Radiol. Surg.*, vol. 11, p. 187, jun 2016. Cited on pages 3 and 31.

[8] T. Nguyen, E. Hattery, and V. P. Khatri, "Radiofrequency ablation and breast cancer: a review," *Gland surg.*, vol. 3, pp. 128–135, may 2014. Cited on page 4.

[9] R. J. Bleicher, D. P. Allegra, D. T. Nora, T. F. Wood, L. J. Foshag, and A. J. Bilchik, "Radiofrequency ablation in 447 complex unresectable liver tumors: lessons learned," *Ann. Surg. Oncol.*, vol. 10, pp. 52–58, jan-feb 2003. Cited on page 4.

[10] T. Livraghi, L. Solbiati, M. F. Meloni, G. S. Gazelle, E. F. Halpern, and S. N. Goldberg, "Treatment of focal liver tumors with percutaneous

radio-frequency ablation: Complications encountered in a multicenter study," *Radiology*, vol. 226, pp. 441–451, feb 2003. Cited on page 4.

[11] D. E. Dupuy, R. J. Zagoria, W. Akerley, W. W. Mayo-Smith, P. V. Kavanagh, and H. Safran, "Percutaneous radiofrequency ablation of malignancies in the lung," *AJR Am J Roentgenol.*, vol. 174, pp. 57–59, jan 2000. Cited on page 4.

[12] M. Akeboshi, K. Yamakado, A. Nakatsuka, O. Hataji, O. Taguchi, M. Takao, and K. Takeda, "Percutaneous radiofrequency ablation of lung neoplasms: initial therapeutic response," *J. Vasc. Interv. Radiol.*, vol. 15, pp. 463–470, may 2004. Cited on page 4.

[13] R. J. Zagoria, J. A. Pettus, M. Rogers, D. M. Werle, D. Childs, and J. R. Leyendecker, "Long-term outcomes after percutaneous radiofrequency ablation for renal cell carcinoma," *Urology*, vol. 77, no. 6, pp. 1393–1397, 2011. Cited on page 4.

[14] D. A. Gervais, F. J. McGovern, R. S. Arellano, W. S. McDougal, and P. R. Mueller, "Renal cell carcinoma: Clinical experience and technical success with radio-frequency ablation of 42 tumors," *Radiology*, vol. 226, pp. 417–424, feb 2003. Cited on page 4.

[15] L. Thanos, S. Mylona, P. Galani, D. Tzavoulis, V. Kalioras, S. Tanteles, and M. Pomoni, "Radiofrequency ablation of osseous metastases for the palliation of pain," *Skeletal Radiol.*, vol. 37, pp. 189–194, nov 2008. Cited on page 4.

[16] D. E. Dupuy, D. Liu, D. Hartfeil, L. Hanna, J. D. Blume, K. Ahrar, R. Lopez, H. Safran, and T. DiPetrillo, "Percutaneous radiofrequency ablation of painful osseous metastases," *Cancer*, vol. 116, pp. 989–997, feb 2010. Cited on page 4.

[17] A. Gillams, "Tumour ablation: current role in the kidney, lung and bone," *Cancer Imaging*, vol. 9, no. Special Issue A, pp. 68–70, 2009. Cited on page 4.

[18] S. Tatli, U. Tapan, P. R. Morrison, and S. G. Silverman, "Radiofrequency ablation: technique and clinical applications," *Diagn. Interv. Radiol.*, vol. 18, no. 5, pp. 508–516, 2012. Cited on page 4.

[19] H. Yu and C. T. Burke, "Comparison of percutaneous ablation technologies in the treatment of malignant liver tumors," *Semin. Intervent. Radiol.*, vol. 31, pp. 129–137, jun 2014. Cited on pages 4 and 7.

[20] M. Friedman, I. Mikityansky, A. Kam, S. K. Libutti, M. M. Walther, Z. Neeman, J. K. Locklin, and B. J. Wood, "Radiofrequency ablation of cancer," *Cardiovasc. Intervent. Radiol.*, vol. 27, pp. 427–434, jun 2004. Cited on pages 4 and 5.

[21] S. Maini, "Comparison between thermal ablation techniques for treatment of cancer," *Int. J. Adv. Res. Comput. Commun. Eng.*, vol. 5, sep 2016. Cited on pages 4 and 7.

[22] L. S. Poulou, E. Botsa, I. Thanou, P. D. Ziakas, and L. Thanos, "Percutaneous microwave ablation vs radiofrequency ablation in the treatment of hepatocellular carcinoma," *World J. Hepatol.*, vol. 7, pp. 1054–1063, may 2015. Cited on page 4.

[23] D. Haemmerich, "Biophysics of radiofrequency ablation," *Crit. Rev. Biomed. Eng.*, vol. 38, no. 1, pp. 53–63, 2010. Cited on pages 4 and 5.

[24] J. P. McGahan and G. D. Dodd, "Radiofrequency ablation of the liver: current status," *Am. J. Roentgenol.*, vol. 176, no. 1, pp. 3–16, 2001. Cited on page 5.

[25] M. Ahmed and S. N. Goldberg, *Radiofrequency Tissue Ablation: Principles and Techniques*, pp. 3–28. New York, NY: Springer, 2004. Cited on page 5.

[26] A. Julianov, "Expanding local control rate in liver cancer surgery – the value of radiofrequency ablation," in *Liver Tumors*, ch. 10, InTech, feb 2012. Cited on page 6.

[27] E. J. Patterson, C. H. Scudamore, D. A. Owen, A. G. Nagy, and A. K. Buczkowski, "Radiofrequency ablation of porcine liver in vivo: effects of blood flow and treatment time on lesion size.," *Ann. Surg.*, vol. 227, pp. 559–565, apr 1998. Cited on page 6.

[28] D. S. K. Lu, S. S. Raman, D. J. Vodopich, M. Wang, J. Sayre, and C. Lassman, "Effect of vessel size on creation of hepatic radiofrequency lesions in pigs: assessment of the "heat sink" effect," *Am. J. Roentgenol.*, vol. 178, pp. 47–51, jan 2002. Cited on page 6.

[29] K. Pillai, J. Akhter, T. C. Chua, M. Shehata, N. Alzahrani, I. Al-Alem, and D. L. Morris, "Heat sink effect on tumor ablation characteristics as observed in monopolar radiofrequency, bipolar radiofrequency, and microwave, using ex vivo calf liver model," *Medicine*, vol. 94, p. e580, mar 2015. Cited on page 6.

[30] L. Solbiati, T. Ierace, M. Tonolini, V. Osti, and L. Cova, "Radiofrequency thermal ablation of hepatic metastases," *Eur. J. Ultrasound.*, vol. 13, pp. 149–158, jun 2001. Cited on page 7.

[31] P. Hildebrand, T. Leibecke, M. Kleemann, L. Mirow, M. Birth, H. Bruch, and C. Bürk, "Influence of operator experience in radiofrequency ablation of malignant liver tumours on treatment outcome," *Eur. J. Surg. Oncol.*, vol. 32, pp. 430–434, may 2006. Cited on page 7.

[32] H. H. Pennes, "Analysis of tissue and arterial blood temperatures in the resting human forearm," *J. Appl. Physiol.*, vol. 1, no. 2, pp. 93–122, 1948. Cited on page 8.

[33] S. K. Hall, E. H. Ooi, and S. J. Payne, "Cell death, perfusion and electrical parameters are critical in models of hepatic radiofrequency ablation," *Int. J. Hyperthermia.*, vol. 31, pp. 538–550, may 2015. Cited on page 8.

[34] R. Khlebnikov and J. Muehl, "Effects of needle placement inaccuracies in hepatic radiofrequency tumor ablation," in *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pp. 716–721, IEEE, 2010. Cited on page 8.

[35] E. R. Cosman, J. R. Dolensky, and R. A. Hoffman, "Factors that affect radiofrequency heat lesion size," *Pain Med.*, vol. 15, pp. 2020–2036, dec 2014. Cited on page 8.

[36] P. Voglreiter, M. Hofmann, C. Ebner, R. B. Sequeiros, H. R. Portugaller, J. Ftterer, M. Moche, M. Steinberger, and D. Schmalstieg, "Visualization-Guided Evaluation of Simulated Minimally Invasive Cancer Treatment," in *Eurographics Workshop on Visual Computing for Biology and Medicine*, The Eurographics Association, 2016. Cited on pages 9 and 10.

[37] R. T. Whitaker, M. Mirzargar, and R. M. Kirby, "Contour boxplots: A method for characterizing uncertainty in feature sets from simulation ensembles," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, pp. 2713–2722, dec 2013. Cited on pages 9, 11, 15, and 16.

[38] S. López-Pintado and J. Romo, "Depth-based inference for functional data," *Comput. Stat. Data Anal.*, vol. 51, pp. 4957–4968, jun 2007. Cited on pages 11 and 12.

[39] J. W. Tukey, *Exploratory Data Analysis*. Reading: Addison-Wesley, 1977. Cited on page 11.

[40] Y. Sun and M. G. Genton, "Functional boxplots," *J. Comp. Graph. Stat.*, vol. 20, pp. 316–334, jan 2011. Cited on pages 11, 14, and 15.

[41] S. López-Pintado and J. Romo, "On the concept of depth for functional data," *J. Am. Stat. Assoc.*, vol. 104, pp. 718–734, jun 2009. Cited on pages 11, 13, 14, and 105.

[42] A. Kwon and M. Ouyang, "Clustering of functional data by band depth," in *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies*, BICT'15, pp. 510–515, ICST, 2016. Cited on pages 20, 21, and 105.

[43] S. López-Pintado and R. Jornsten, "Functional analysis via extensions of the band depth," *Inst. Math. Stat.*, vol. 54, pp. 103–120, 2007. Cited on page 20.

[44] Y. Sun, M. G. Genton, and D. W. Nychka, "Exact fast computation of band depth for large functional datasets: How quickly can one million curves be ranked?," *Stat*, vol. 1, no. 1, pp. 68–74, 2012. Cited on pages 21, 22, 23, 25, 27, and 105.

[45] Y. Hong, Y. Gao, M. Niethammer, and S. Bouix, "Shape analysis based on depth-ordering," *Med Image Anal*, vol. 25, pp. 2–10, oct 2015. Cited on pages 21 and 105.

[46] "RFA Guardian - technical workflow." `http://www.clinicimppact.eu/rfa-technicalworkflow.html`. Accessed: 2017-08-20. Cited on page 31.

[47] "MITK Toolkit." `http://mitk.org/wiki/The_Medical_Imaging_Interaction_Toolkit_(MITK)`. Accessed: 2017-08-20. Cited on page 31.

[48] "ITK Toolkit." `https://itk.org`. Accessed: 2017-08-20. Cited on page 31.

[49] "VTK Toolkit." `http://www.vtk.org`. Accessed: 2017-08-20. Cited on page 31.

[50] "VTK 6.1.0 API reference." `http://www.vtk.org/doc/release/6.1/html/`. Accessed: 2017-08-20. Cited on page 31.

[51] Y. Mrabet, "Planes of human anatomy." https://commons.wikimedia.org/wiki/File:Human_anatomy_planes_signatures.svg, sep 2010. Accessed: 2017-08-20. Cited on page 32.

[52] "RFA Guardian - user interface." http://www.clinicimppact.eu/rfa-guardian.html. Accessed: 2017-08-20. Cited on page 33.

[53] L. Papula, *Mathematik für Ingenieure und Naturwissenschaftler Band 3*. Vieweg + Teubner, sixth ed., 2011. Cited on pages 35 and 36.

[54] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*. Springer New York, second ed., 2003. Cited on pages 37 and 38.

[55] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998. Cited on page 37.

[56] A. Jagannatam, "Mersenne twister - a pseudo random number generator and its variants," tech. rep., George Mason University, dec 2010. Cited on page 37.

[57] C. McClanahan, "History and evolution of gpu architecture." http://disi.unal.edu.co/~gjhernandezp/HeterParallComp/GPU/gpu-hist-paper.pdf, 2010. Accessed: 2017-08-20. Cited on page 41.

[58] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Med. Phys.*, vol. 38, pp. 2685–2697, may 2011. Cited on page 41.

[59] S. Hasan, S. M. Shamsuddin, and N. Lopes, "Machine learning big data framework and analytics for big data problems," *Int. J. Advance Soft Compu. Appl.*, vol. 6, 08 2014. Cited on page 41.

[60] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 317–324, Feb 2010. Cited on page 41.

[61] G. Sellers, R. Wright, and N. Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*. OpenGL, Pearson Education, seventh ed., 2015. Cited on pages 41, 42, 43, and 44.

[62] M. Bailey, "How to use and teach opengl compute shaders." https://www.khronos.org/assets/uploads/developers/library/2014-siggraph-bof/KITE-BOF_Aug14.pdf, aug 2014. Accessed: 2017-08-20. Cited on page 44.

[63] C. Hafner, "Compute shaders." https://www.cg.tuwien.ac.at/courses/Realtime/repetitorium/rtr_rep_2016_ComputeShader, 2016. Accessed: 2017-08-20. Cited on page 44.

[64] L. Papula, *Mathematik für Ingenieure und Naturwissenschaftler Band 1.* Vieweg + Teubner, twelfth ed., 2011. Cited on pages 46 and 54.

[65] F. Preparata and M. Shamos, *Computational Geometry: An Introduction.* Monographs in Computer Science, Springer New York, 2012. Cited on page 47.

[66] "A history of linear-time convex hull algorithms for simple polygons." http://cgm.cs.mcgill.ca/~athens/cs601/. Accessed: 2017-08-21. Cited on page 47.

[67] A. Melkman, "On-line construction of the convex hull of a simple polyline," *Inf. Process. Lett.*, vol. 25, pp. 11–12, apr 1987. Cited on pages 47 and 48.

[68] M. Woo, J. Neider, T. Davis, and O. A. R. Board, *Opengl Programming Guide: The Official Guide to Learning Opengl, Version 1.1.* Addison Wesley, second ed., 1997. Cited on pages 63 and 64.

[69] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Prentice Hall, second ed., 2002. Cited on page 66.

[70] B. W. Silverman, *Density Estimation for Statistics and Data Analysis.* Chapman & Hall/CRC Monographs on Statistics & Applied Probability, CRC press, first ed., 1986. Cited on page 81.

[71] B. Turlach, "Bandwidth selection in kernel density estimation: A review," tech. rep., CORE and Institut de Statistique, 1993. Cited on page 81.

[72] M. W. Toews, "Kernel density." https://commons.wikimedia.org/wiki/File:Kernel_density.svg, jul 2007. Accessed: 2017-08-20. Cited on page 82.