Jül-4395

Jülich Supercomputing Centre (JSC)

Optimierung und Parallelisierung der Software simPLI zur Simulation von 3D-Polarized-Light-Imaging Messungen

Sonja Lucksch

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Optimierung und Parallelisierung der Software simPLI zur Simulation von 3D-Polarized-Light-Imaging Messungen

Sonja Lucksch

Berichte des Forschungszentrums Jülich; 4395
ISSN 0944-2952
Jülich Supercomputing Centre (JSC)
Jül-4395

D A96 (Master, FH Aachen/Campus Jülich, 2016)

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH
Zentralbibliothek, Verlag
52425 Jülich
Tel.: +49 2461 61-5220
Fax: +49 2461 61-6103
E-Mail: zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

Abstract

Die Methode *Three-dimensional Polarized Light Imaging* (3D-PLI) wurde am Institut für Neurowissenschaften und Medizin am Forschungszentrum Jülich entwickelt. Damit ist es möglich den Verlauf von Nervenfasern im menschlichen Gehirn am Rechner zu rekonstruieren.
Das Gehirn wird bei dieser Methode post mortem in 60 µm dicke Scheiben geschnitten und anschließend mit polarisiertem Licht durchleuchtet. Die transmittierte Lichtintensität wird hierbei mit einer CCD-Kamera aufgenommen. Aufgrund der hohen benötigten Bildauflösung ist es nicht möglich einen Gehirnschnitt im Ganzen aufzunehmen. Daher wird dieser in Kacheln aufgeteilt, die später im PLI-Rekonstruktionsworkflow zu einer gesamten 3D-Karte des Gehirns zusammengeführt werden.

Die im Rahmen dieser Arbeit entwickelte Software *simPLI* dient der Simulation von *3D-Polarized Light Imaging* Messungen und damit vor allem der Verifizierung der Ergebnisse des Workflows. Mit Hilfe der Software ist es möglich, beliebige Faserverläufe zu simulieren. Dazu gibt der Benutzer die zu erzeugenden Fasern als mathematische Funktionen an, die anschließend an diskreten Stellen eines Gitters ausgewertet werden. Dadurch wird intern eine Faserstruktur in Form eines Hohlrohres aufgebaut.
Weiterhin ist es möglich, eine Kippung des definierten Faserverlaufs in eine zuvor festgelegte Richtung zu simulieren. Schließlich wurde das Simulationsprogramm parallelisiert, um den Zeitaufwand der Simulation zu senken und vor allem größere Ergebnisdaten erzeugen zu können.

Abstract

The method *Three-dimensional Polarized Light Imaging* (3D-PLI) was developed at the institute for neuroscience and medicine at Forschungszentrum Jülich. This technique allows reconstructing the architecture of nerve fibers in the human brain.
For this method the post mortem brain is cut in 60 µm thick slices and afterwards screened with polarized light, where the transmitted light intensity is measured with a CCD-camera. Due to the required high resolution of the image it is not possible to take a picture of the whole brain section. Therefore it is divided into tiles which are later assembled to a 3D brain map using the PLI-workflow.

The software *simPLI*, which was developed in this thesis, is used for simulation of *3D-Polarized Light Imaging* measurements and therefore especially for verification of the workflow results. With the aid of the developed software any fiber structure can be simulated, where the fibers are specified by the user as mathematical functions which are afterwards evaluated at discrete grid points. Thus a fiber structure in the shape of a hollow tube is created.
Furthermore a simulation of the tilting of the defined fiber scenario in a predetermined direction is possible. Finally the software was parallelized to reduce the execution time of the simulation and especially to generate larger amounts of data.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Struktogrammverzeichnis

1. Einleitung

Das menschliche Gehirn in seiner Funktionsweise zu verstehen und dessen Aufbau zu rekonstruieren ist eine der komplexesten Aufgaben, die es in der heutigen Zeit zu bewältigen gilt. Das *Human Brain Project* (HBP) ist ein von der europäischen Union gefördertes Flagship-Projekt, das im Oktober 2013 startete und dessen geplante Laufzeit 10 Jahre beträgt [1]. Gemeinsam forschen Neurowissenschaftler, Ärzte, Informatiker, Physiker, Mathematiker und Computerspezialisten aus 24 verschiedenen Ländern im Rahmen dieses Projekts. Ziel ist es, das Gehirn in Zukunft in seiner Gesamtheit simulieren zu können und dadurch Abläufe in gesunden und vor allem kranken Gehirnen besser verstehen zu können.
Auch im Forschungszentrum Jülich wird in verschiedenen Instituten an dieser Problematik geforscht, beispielsweise im Jülich Supercomputing Centre (JSC) und im Institut für Neurowissenschaften und Medizin (INM).
Zunächst müssen Erkenntnisse über die Grundlagen des Gehirns gewonnen werden. Wie sind die rund 86 Milliarden Nervenzellen im Gehirn aufgebaut? Wie arbeiten sie und wie sind sie miteinander verbunden? Dies sind nur einige Fragestellungen, die es zu beantworten gilt.
Natürlich gibt es schon bildgebende Modalitäten wie Elektronenmikroskopie (EM) oder Magnetresonanztomographie (MRT), aber diese dienen entweder der Detailaufnahme einzelner Axone oder sind so begrenzt in ihrem Auflösungsvermögen, dass sie nur große Faserbahnen aufnehmen können.
Daher wurde im INM-1 ein neues bildgebendes Verfahren, das *Three-dimensional Polarized Light Imaging* (3D-PLI) entwickelt. Damit ist es möglich, die Nervenfaserrichtungen mit einer Auflösung von wenigen Mikrometern zu rekonstruieren. Somit können zum einen ganze Gehirne, zum anderen aber auch Faserbündel und sogar einzelne Nervenfasern durch 3D-PLI am Rechner dargestellt werden.
Das zu untersuchende Gehirn wird post mortem in 60 µm dicke Scheiben geschnitten und anschließend mit polarisiertem Licht durchleuchtet. Die Änderung des Polarisationszustandes des Lichts nach Austritt aus dem Hirnschnitt wird gemessen. Sie hängt maßgeblich von der dreidimensionalen Ausrichtung der Nervenfasern ab. In Kapitel 2 wird beschrieben, wie aus den 3D-PLI Aufnahmen mit Hilfe des PLI-Workflows die Richtung der Nervenfaser in einem Bildpunkt bestimmt werden kann. Physikalische Grundlagen zur Doppelbrechung des Lichts und optische Eigenschaften des PLI-Versuchsaufbaus sind in Kapitel 3 zu finden.
Die 3D-PLI Messungen sollen simuliert werden können, um damit die Ergebnisse des Workflows zu verifizieren. Daher wurde im Rahmen dieser Arbeit die Simulationssoftware *simPLI* optimiert und implementiert, die es ermöglicht, 3D-PLI Messungen zu nachzubilden. Das Simulationsprogramms und die zugehörigen Algorithmen werden

in Kapitel 4 beschrieben.
Der Verlauf einer oder mehrerer Fasern kann über beliebige mathematische Funktionen in einem dreidimensionalen Simulationsvolumen definiert werden. Das Simulationsprogramm erstellt zu diesen Definitionen ein dreidimensionales Volumen, in dem die Fasern als Hohlrohre dargestellt sind.
Auf dieses Volumen wird der Jones Matrix Formalismus angewendet und das Ergebnis dieser Simulation ist eine Folge von Bildern, die zum Vergleich mit realen Daten mit dem gleichen Workflow analysiert werden können. Da Simulationsdaten vom Benutzer selbst definiert wurden, ist auch das Ergebnis des Workflows bekannt und kann somit überprüft werden. Weiterhin ist es möglich, das Simulationsvolumen zu Kippen, sodass die Lichtstrahlen einen anderen Verlauf durch das Hirngewebe nehmen.
Da die durch das Programm erzeugten Daten mit wachsenden Simulationsvolumina schnell eine Größe von mehreren Gigabyte erreichten, wurde das Programm *simPLI* parallelisiert. Dadurch lassen sich größere Daten simulieren und zusätzlich wird noch eine Verringerung der Laufzeit erreicht. Neben verschiedenen Parallelisierungsparadigmen und -modellen und deren Umsetzung im Programm ist in Kapitel 5 eine Analyse der Laufzeit des Simulationssoftware zu finden.

2. 3D-PLI und PLI-Rekonstruktionsworkflow

Um ein räumliches Fasermodell eines Gehirns mittels 3d-PLI zu erstellen, müssen die Richtungen der Nervenfasern bestimmt werden. Diese können nicht direkt aus den Aufnahmen der Gehirnschnitte ermittelt werden, sondern werden mit Hilfe des *3D-PLI Workflows* bestimmt.

2.1. 3D-Polarized Light Imaging (3D-PLI)

Mit Hilfe der Methode 3D-PLI lassen sich Nervenfasern im Gehirn bis zu einem Skalenbereich von wenigen Mikrometern rekonstruieren [2]. Das Verfahren macht sich eine wichtige optische Eigenschaft des Nervengewebes zunutze: die Doppelbrechung (siehe Abschnitt 3.1). Die Fasern im Gehirn (Axone) sind größtenteils mit einer doppelbrechenden Myelinschicht umgeben, die unter anderem der elektrischen Isolation dient und eine starke negative, uniaxiale Doppelbrechung hervorruft (Abb. 2.1).

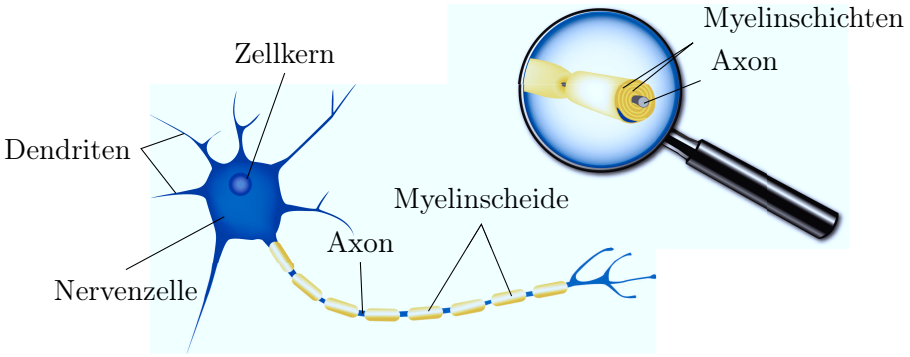


Abbildung 2.1.: Aufbau einer Nervenfaser

Die 60 µm dünnen Gehirnscheiben werden beim 3D-PLI Verfahren in einem Polarisationsmikroskop (Abb. 2.2) platziert und mit polarisiertem Licht der Wellenlänge λ bestrahlt. Dieser Aufbau besteht aus zwei senkrecht zueinander ausgerichteten, linearen Polarisationsfiltern mit einer Probenplatte und einer Verzögerungsplatte dazwischen [3].
Das von der Lichtquelle ausgesandte Licht ist zunächst unpolarisiert, das heißt es

schwingt in alle Richtungen. Die Polarisationsfilter lassen nur Licht durch, das in einer bestimmten Richtung schwingt, sodass das Licht nach dem Passieren des ersten Polarisationsfilters linear polarisiert ist. Die Verzögerungsplatte transformiert dieses anschließend in zirkulär polarisiertes Licht. Durch die doppelbrechende Myelinschicht der Nervenfasern im Hirnschnitt werden die Phasen der Lichtwellen verschoben, sodass das Licht nach Durchleuchten der Probe elliptisch polarisiert ist. Je nach 3D-Ausrichtung der Fasern ist die Phasenverschiebung eine andere. Durch den Anteil des Lichts, das durch den zweiten linearen Polarisationsfilter gelangt, kann man über die Lage der optischen Achsen des doppelbrechenden Gewebes (siehe Abschnitt 3.1) die Richtung der Nervenfasern ermitteln.

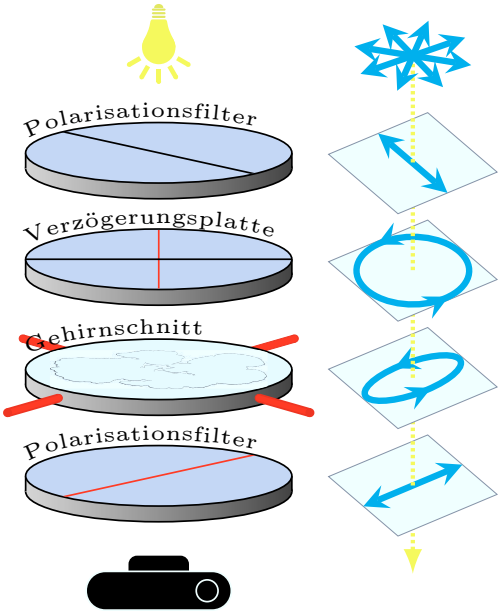


Abbildung 2.2.: 3D-PLI Versuchsaufbau und Schwingungsrichtung des Lichts nach dem Passieren der verschiedenen Teile des Aufbaus

Polarisationsfilter und Verzögerungsplatte rotieren gleichzeitig um das fixierte Hirngewebe von 0° bis 170° in 10° Schritten, wobei nach jedem Schritt ein Bild des Gehirnschnitts aufgenommen wird. Je nach Winkel ändert sich die Intensität des aufgenommenen Lichts. Aus den Helligkeitsänderungen zwischen den Aufnahmen ist es möglich, auf die Lage der Nervenfasern im Gehirnschnitt zu schließen. Die Ergebniswerte der 18 Bilder ergeben einen individuellen sinusförmigen Intensitätsverlauf für jedes Pixel.

2.2. PLI-Rekonstruktionsworkflow

Um den Verlauf der Nervenfasern im Gehirn aus den aufgenommenen Bildern der Hirnschnitte zu rekonstruieren, kommen verschiedenste Verfahren aus der Signal- und Bildverarbeitung zum Einsatz. Anschließend werden die einzelnen Kacheln eines Schnittes zu einem Gesamtbild zusammengesetzt und diese wiederum zu einem dreidimensionalen Bild des Gehirns. Im Folgenden wird dieser Ablauf kurz skizziert.

2.2.1. Vorverarbeitung

Die Faserrichtungen, die man aus den 3D-PLI Bildern berechnet, hängen stark von der Qualität der Aufnahmebilder ab. Durch digitales Rauschen, Staub oder Streuung des Lichts werden die Aufnahmen verfälscht.
Um Fehler durch ungleichmäßige Beleuchtung zu eliminieren, wird ein Bild ohne Gewebeprobe im Versuchsaufbau aufgenommen. Damit können diese Fehlinformationen pixelweise korrigiert werden. Dieser Vorgang nennt sich **Kalibrierung**.
Weiterhin können Objekte Staubpartikel automatisch erkannt und entfernt werden. Dies geschieht mit der *Independent Component Analysis* (ICA) [4].

2.2.2. Efficient PLI Analysis

Der Intensitätsverlauf der 18 Bilder in einem Pixel wird durch eine diskrete Fourier Analyse approximiert [3]. Als Verlauf ergibt sich eine Sinuskurve. Diese kann durch drei Werte charakterisiert werden: Die Verschiebung der Sinuskurve, jeweils in x- und y-Richtung, und die Amplitude der Schwingung (Abb. 2.3).
Die Verschiebung der Sinuskurve in x-Richtung heißt **Direktion** und beschreibt den Winkel der Projektion der Faser in der Schnittebene.
Die **Transmittanz** entspricht der doppelten Verschiebung der Kurve in Richtung der y-Achse. Dieser Wert ist ein Maß für die Lichtauslöschung (Streuung und Absoption) des Gehirnschnitts in diesem Pixel. Zur Vereinheitlichung werden die Transmittanzwerte noch normiert, das heißt die einzelnen Werte werden durch die Transmittanzwerte einer Leeraufnahme dividiert.
Die **Retardierung** entspricht der Amplitude der Sinuskurve, also der Differenz aus maximalem und minimalem Wert. Mit diesem Wert können Aussagen über die Stärke der Myelinisierung der Faser in diesem Pixel getroffen werden. Ein großer Retardierungswert weist auf eine starke Doppelbrechung oder Nervenfaserverläufe in der Schnittebene hin.
Nun wird für jedes Pixel nicht der Intesitätsverlauf über die 18 Bilder gespeichert, sondern die drei Charakteristika der Sinuskurve: **Direktion**, **Transmittanz** und **Retardierung**. Dieser Vorgang geschieht ohne Datenverlust, da davon ausgegangen wird, dass die Abweichungen von der angepassten Sinuskurve durch Messfehler zu begründen sind. Dadurch kann der Speicherbedarf auf ein Sechstel der ursprünglichen Menge reduziert werden.

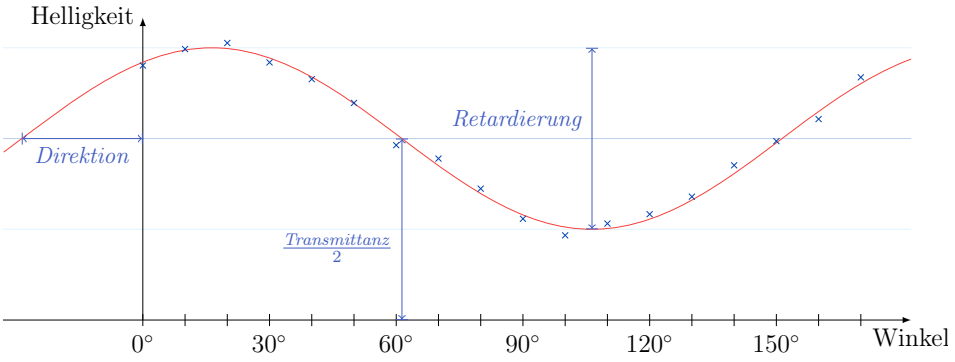


Abbildung 2.3.: Direktion, Transmittanz, Retardierung

2.2.3. Segmentierung

Der Hintergrund der Aufnahmen ist für die spätere räumliche Darstellung nicht von Bedeutung. Daher wird auf der Basis des Transmittanzbildes, welches die klarste Abgrenzung von Gehirn zum Hintergrund besitzt, ein Segmentierungsalgorithmus durchgeführt.
Beim Region-Growing-Algorithmus wird eine Maske erstellt, die die Bildpunkte entweder dem Hintergrund oder dem Gehirn zuordnet. Diese Maske wird mit Hilfe der Histogramme einzelner Kacheln erstellt, die immer einen ähnlichen Verlauf haben (Abb. 2.4).

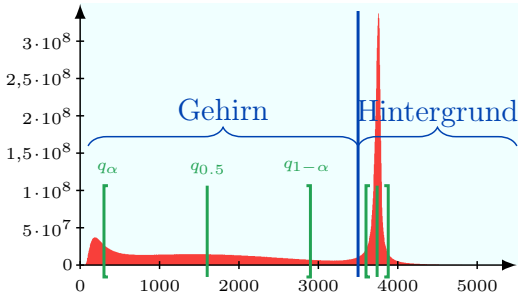


Abbildung 2.4.: Histogramm einer Kachel zur Segmentierung

Durch dieses Vorgehen wird wiederum Arbeitszeit gespart, da Hintergrundpixel in der weiteren Verarbeitung nicht mehr berücksichtigt werden müssen. In einer nächsten Version des Segmentierungsalgorithmus soll ebenfalls eine Unterscheidung zwischen verschiedenen Gehirnregionen, wie weißer und grauer Substanz, möglich sein.

2.2.4. Stiching

Um eine Auflösung von 1,3 µm zu erreichen, lässt sich ein Gehirnschnitt nicht im Gesamten aufnehmen, sondern wird in 2048×2048 Pixel große Teilbereiche (sogenannte Kacheln) unterteilt. Zudem werden die Bilder 30 % überlappend aufgenommen, da so ausgeschlossen werden kann, dass Informationen an den Rändern dieser Kacheln verloren gehen. Die einzelnen Teilbereiche eines Schnitts werden beim Stiching wieder zu einem Gesamtschnitt zusammen gesetzt. Dazu ist die normierte Transmittanz, die ein Maß für die Helligkeit in einem Pixel ist, am besten geeignet.
In den Überlappbereichen werden markante Punkte gesucht. Benachbarte Kacheln

werden anschließend so übereinander geschoben, dass diese markanten Punkte genau übereinander liegen (Abb. 2.5).

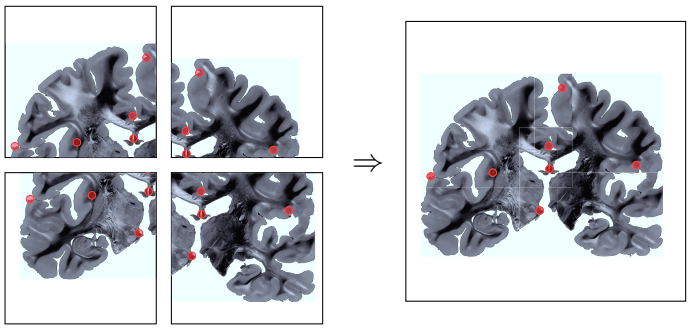


Abbildung 2.5.: Stiching

2.2.5. Inklination

Sind die Kacheln zu einem Schnitt zusammengefügt, so wird die Lage der Nervenfaser im Raum bestimmt. Ein Punkt im dreidimensionalen Raum kann durch seine sphärischen Koordinaten beschrieben werden, das heißt durch seinen Abstand zum Koordinatenursprung und durch zwei Winkel. Die Länge der Nervenfaser spielt keine Rolle, weshalb die Lage einer Nervenfaser über die Direktion δ und die Inklination α festgelegt ist (Abb. 2.6). Die Direktion beschreibt den Winkel der Projektion der Faser in der Schnittebene. Die Inklination entspricht dem Winkel zwischen der Schnittebene und der Nervenfaser.

$$\alpha = \arccos\left(\sqrt{\frac{\arcsin Ret}{\arcsin Ret_{0^\circ}}}\right) \tag{2.1}$$

Der Inklinationswinkel kann über Formel 2.1 berechnet werden. Der Parameter Ret bezeichnet den Retardierungswert in dem Pixel, in dem auch die Inklination berechnet werden soll. Der Parameter Ret_{0° steht für den Retardierungswert, der einer Faser mit einem Winkel von 0° entspricht [5]. Der Verlauf des Histogramms der Retardierungswerte schwankt auf Grund von Streuung und fällt flach ab. Daher wird ein Schwellwert gesucht, ab dem die Histogrammwerte ausschließlich auf Streuung zurückzuführen sind. Es wird ein Fitwert bestimmt und dieser wird für Ret_{0° gewählt, da allen Werten, die größer sind, ein Winkel von 0° zugeordnet wird. Somit ist eine eindeutige Zuordnung von Inklinationswinkeln zu Retardierungswerten möglich.

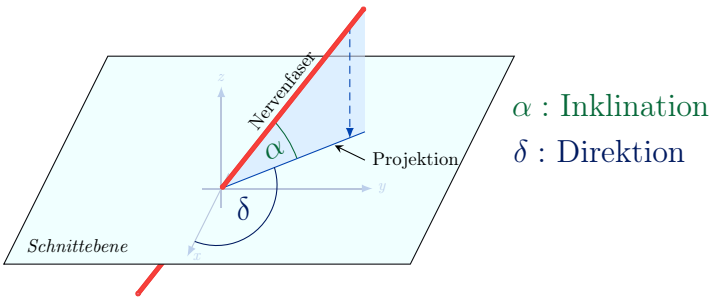


Abbildung 2.6.: Inklination

Da die verschiedenen Gewebetypen, weiße und graue Substanz, unterschiedliche Myelinisierungsdichten aufweisen, ergeben sich je nach Gewebetyp unterschiedliche Intensitätsskalen. In einer Erweiterung werden für beide Gewebetypen unterschiedliche Fitwerte berechnet. Oft ist aber eine strikte Zuordnung zu einer der beiden Substanzen nicht möglich, weshalb ein dynamischer Fitwert eingeführt wurde. Der Transmittanzwert steht in Zusammenhang mit der Dichte der Myelinisierung. Daher werden die Fitwerte für graue und weiße Substanz auf Basis der Transmittanz gewichtet, um einen dynamischen Fitwert zu berechnen.

2.2.6. Globale Registrierung

Abschließend müssen die einzelnen Bilder der Gehirnschnitte zu einem Gesamtvolumen zusammengesetzt werden. Dabei treten einige Probleme auf, denn es reicht nicht aus, die einzelnen Bilder in der richtigen Reihenfolge anzuordnen und zusammenzufügen.

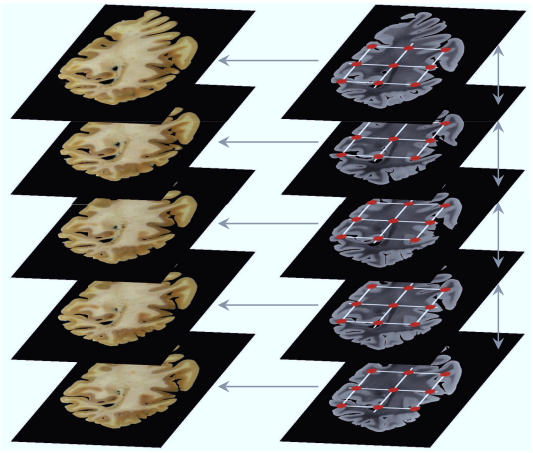


Abbildung 2.7.: Registrierung der einzelnen Gehirnschnitte auf die Blockface Bilder und Zusammensetzung der Schnitte zu einem Gesamtvolumen

Durch den Schneideprozess wurde das Gewebe des Gehirns verformt und somit ist es nicht möglich, durch Zusammensetzen die Originalform zu rekonstruieren. Zudem muss berücksichtigt werden, dass sich die verschiedenen Gehirnregionen, also weiße und graue Substanz, beim Schneiden unterschiedlich verformen.
Vor jedem Schnitt wird zum Vergleich ein Bild der aktuellen ungeschnittenen Gehirnoberfläche, genannt Blockface Bild (links in Abb. 2.7), aufgenommen. Die Schnitte können am Ende des Workflows mit Hilfe des Blockface Bildes mit verschiedenen Transformationen und Algorithmen in ihre ursprüngliche Form transformierts werden.

3. Theoretischer Hintergrund

Eine zentrale Bedeutung bei der Erzeugung der 3D-PLI Bilder hat die doppelbrechende Myelinschicht der Nervenfasern im Gehirn. Um diese simulieren zu können, werden in diesem Kapitel die Prinzipien der Doppelbrechung erläutert. Weiterhin werden zwei Simulationsmodelle der Nervenfasern vorgestellt. Die Doppelbrechung des Myelins kann mit Hilfe des Jones Matrix Calculus abgebildet werden.

3.1. Doppelbrechung

Wechselt ein Lichtstrahl das Medium, so ändert sich seine Geschwindigkeit und seine Wellenlänge. Diese Eigenschaft wird durch den Brechungsindex n des Materials beschrieben. Er ist definiert als Quotient aus der Lichtgeschwindigkeit (oder Wellenlänge) im Vakuum c und der Geschwindigkeit im Medium v [6].

$$n = \frac{c}{v} = \frac{\lambda_0}{\lambda} \tag{3.1}$$

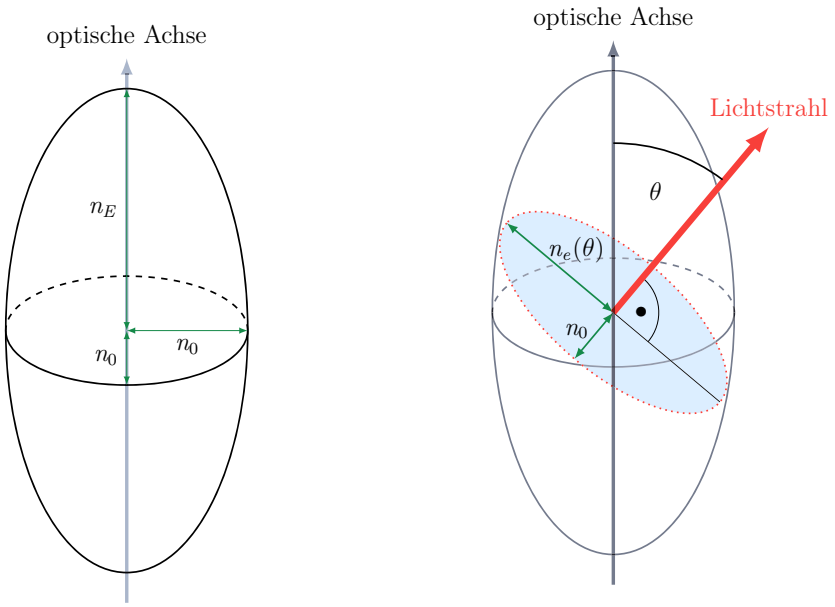
Am Grenzübergang von zwei Medien mit unterschiedlichen Brechungsindizes n_1 und n_2 ändert der Lichtstrahl seine Richtung. Die Beziehung zwischen Eintritts- und Austrittswinkel kann über das (Snelliussche) Brechungsgesetz beschrieben werden [6]:

$$n_1 \sin(\alpha_1) = n_2 \sin(\alpha_2) \tag{3.2}$$

Die Doppelbrechung ist eine Eigenschaft von anisotropen, symmetrisch angeordneten Strukturen. Ein anisotropes Material zeigt richtungsabhängig ein anderes Verhalten oder hat andere Eigenschaften. Bei doppelbrechenden Materialien variiert der Brechungsindex und hängt von der Polarisationsrichtung des Lichts ab.
Um den Brechungsindex eines Lichtstrahls mit beliebiger Richtung durch ein doppelbrechendes Material zu definieren, wird ein sogenannter Indexellipsoid [7] verwendet. Nervenfasern bestehen aus einachsig doppelbrechenden Komponenten und bei dieser Form der Doppelbrechung sind zwei der drei Hauptachsen des Ellipsoiden gleich groß. Mit n_0 wird die Größe der ersten und zweiten Hauptachse bezeichnet und mit n_E die Größe der dritten Hauptachse (Abb. 3.1a). Im doppelbrechenden Medium gibt es ausgezeichnete optische Achsen, die die Symmetrieachsen der Ellipsoiden darstellen.

Die Doppelbrechung eines Materials wird definiert als:

$$\Delta n = n_E - n_0 \tag{3.3}$$



(a) Hauptachsen eines einachsigen Indexellipsoids

(b) Einfallender Lichtstrahl und Brechungsindizes des ordentlichen und außerordentlichen Strahls

Abbildung 3.1.: Indexellipsoid eines einachsigen, positiv doppelbrechenden Materials [8]

Es wird zwischen positiver ($\Delta n > 0,\ n_E > n_0$) und negativer ($\Delta n < 0,\ n_E < n_0$) Doppelbrechung unterschieden.
Der einfallende Lichtstrahl wird in ordentlichen und außerordentlichen Strahl aufgesplittet, die verschiedene Brechungsindizes besitzen. Der ordentliche Strahl hat immer den Brechungsindex n_0, was der Größe der ersten und zweiten Hauptachse des Ellipsoiden entspricht (Abb. 3.1b). Der Brechungsindex des außerordentlichen Strahls $n_e(\theta)$ kann abhängig vom Einfallswinkel θ des Lichtstrahls berechnet werden.
Durch die verschiedenen Brechungsindizes der Strahlen ändert sich deren Wellenlänge und Geschwindigkeit unterschiedlich und es ergibt sich eine Phasenverschiebung der Lichtwelle.

3.2. Zugrundeliegendes Simulationsmodell

Um die 3D-PLI Bilder zu simulieren, wurde ein Modell für die Nervenfasern im Gehirn entwickelt. Dieses stellt die Fasern als dreidimensionale Hohlrohre dar. Es ist möglich, den äußeren und den inneren Radius der Faser zu spezifizieren. Dabei stellt das Rohr die doppelbrechende Myelinschicht dar und der innere Hohlraum simuliert das Axon.

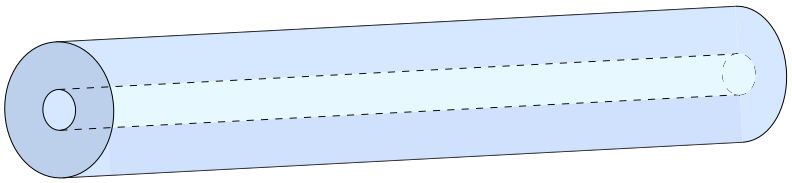
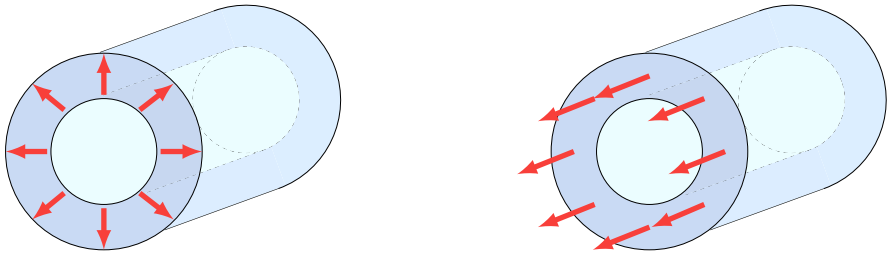


Abbildung 3.2.: Modellvorstellung einer Nervenfaser als Hohlrohr

Die optischen Achsen der doppelbrechenden Myelinschicht können nach zwei unterschiedlichen Modellen [2] simuliert werden:

Mikroskopisches Modell: Im mikroskopischen Ansatz wird die Myelinschicht einer Nervenfaser als einachsig positiv doppelbrechend ($n_E > n_0$) mit radialen optischen Achsen modelliert. Das bedeutet die Achsen verlaufen sternförmig vom Fasermittelpunkt nach außen (Abb. 3.3a).
Die positive Doppelbrechung ist auf die radialsymmetrische Anordnung der Lipidmoleküle in der spiralförmigen Myelinschicht zurückzuführen [8]. Die Moleküle können alle als kleinste Ellipsoiden aufgefasst werden, die eine positive Doppelbrechung besitzen.



(a) Radiale Faserrichtung beim mikroskopischen Modell

(b) Axiale Faserrichtung beim makroskopischen Modell

Abbildung 3.3.: Faserrichtungen der beiden Modelle

Makroskopisches Modell: Die Myelinschicht im makroskopischen Modell wird als einachsig negativ doppelbrechend mit axialen optischen Achsen simuliert. Das bedeutet, dass die optischen Achsen parallel zur Nervenfaser verlaufen(Abb. 3.3b).

Wird eine Scheibe eines Querschnitts durch die Nervenfaser betrachtet, so kann diese wiederum als sehr flacher Ellipsoid betrachtet werden, der insgesamt negativ doppelbrechend ($n_E < n_0$) ist. Viele dieser Schichten hintereinander sorgen für die axiale Richtung entlang der Nervenfaser, die auch durch das 3D-PLI Verfahren ermittelt wird.

In beiden Fällen wird als Näherung angenommen, dass das Axon nicht doppelbrechend ist.

Das gesamte Simulationsvolumen wird diskretisiert, das heißt in Würfel einer vorgegebenen Größe unterteilt. Jeder Würfel wird im Laufe der Simulation entweder Axon, Myelin oder Hintergrund zugeordnet. Außerdem wird jedem Würfel im diskreten Gitter, der zum Myelin gehört, je nach gewähltem Modell, entweder ein radialer oder ein axialer dreidimensionaler Vektor zugeordnet, der im Jones Matrix Formalismus verarbeitet wird.

3.3. Jones Matrix Calculus

Die im 3D-PLI Versuchsaufbau aufgenommene Lichtintensität pro Pixel kann physikalisch durch den Jones Matrix Formalismus (auch Jones Calculus genannt) beschrieben werden. Dieser Formalismus ist nach R. Clark Jones benannt, der diese Bezeichnung im Jahr 1941 einführte [9]. Er dient der Beschreibung linearer optischer Abbildungen unter Berücksichtigung der Polarisation des Lichts. Es lassen sich Systeme analysieren, bei denen ein Lichtstrahl eine Anordnung von optischen Elementen durchläuft, wie es beim 3D-PLI Versuchsaufbau der Fall ist.

Alle Bestandteile des Versuchsaufbaus werden durch sogenannte Jones Matrizen beschrieben (Abb. 3.4) und in der Reihenfolge, in der das Licht durch die optischen Elemente strahlt, miteinander multipliziert. Die Matrizen für die beiden zueinander gekreuzt stehenden linearen Polarisationsfilter sind gegeben durch [10]:

$$P_x = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \qquad P_y = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.4}$$

Die Jones Matrix einer Verzögerungsplatte, die um den Winkel Ψ gegen den Uhrzeigersinn gedreht wird und einen Phasenshift δ bei der Lichtwelle verursacht, kann beschrieben werden durch [10]:

$$M_\delta(\psi) = R(\psi) \cdot M_\delta \cdot R(-\psi) = \begin{pmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{pmatrix} \begin{pmatrix} e^{i\delta/2} & 0 \\ 0 & e^{-i\delta/2} \end{pmatrix} \begin{pmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{pmatrix} \tag{3.5}$$

Die Verzögerungsplatte im Versuchsaufbau ist um ψ = -45° gegenüber der Achse des ersten linearen Polarisationsfilters gedreht. Daher ist die Matrix der Verzögerungsplat-

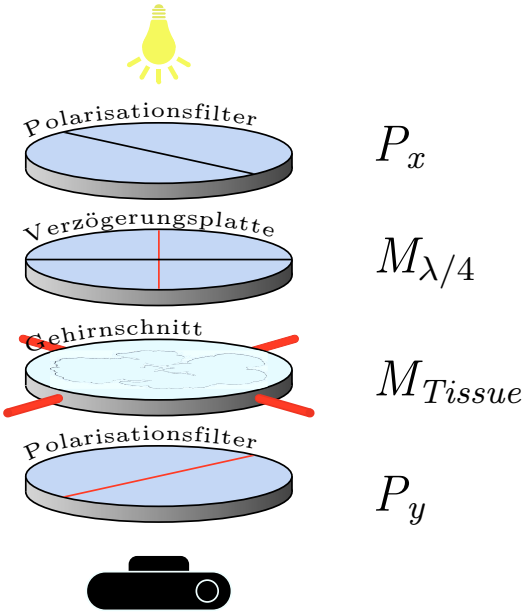


Abbildung 3.4.: Repräsentation der optischen Elemente des 3D-PLI Aufbaus durch die Matrizen P_x, $M_{\lambda/4}$, M_{Tissue} und P_y

te durch die Multiplikation der Matrizen aus 3.5 mit den Parametern $\psi = 45°$ und $\delta = 90°$ gegeben [2].

$$M_{\lambda/4} = M_{90°}(-45°) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \tag{3.6}$$

Bei der Repräsentation des Gehirngewebes ist zu beachten, dass für jeden Würfel im dreidimensionalen Gitter, der zur Myelinschicht gehört, eine Matrix erstellt und multipliziert wird (Abb. 3.5). Diese Matrizen des Gehirngewebes können als Jones Matrizen einer gedrehten Verzögerungsplatte wie in Formel 3.5 aufgefasst werden. Nach der Multiplikation der drei Matrizen mit Phasenshift δ und Rotationswinkel $\psi = \varphi - \rho$ sehen die Jones Matrizen des Gehirngewebes folgendermaßen aus:

$$M_N = \begin{pmatrix} \cos(\delta/2) + i\,\sin(\delta/2)\,\cos(2(\varphi-\rho)) & i\,\sin(\delta/2)\,\sin(2(\varphi-\rho)) \\ i\,\sin(\delta/2)\,\sin(2(\varphi-\rho)) & \cos(\delta/2) - i\,\sin(\delta/2)\,\cos(2(\varphi-\rho)) \end{pmatrix} \tag{3.7}$$

Wie in Kapitel 2.1 beschrieben, werden die Polarisationsfilter simultan um die Gewebeprobe gedreht. Der Parameter ρ in der Jones Matrix 3.7 beschreibt diesen Drehwinkel, der Werte von 0°, 10°, ..., 170° annimmt.
Wenn der polarisierte Lichtstrahl in die doppelbrechende Myelinschicht des Gehirnschnitts eindringt, wird er in einen ordentlichen und einen außerordentlichen Strahl

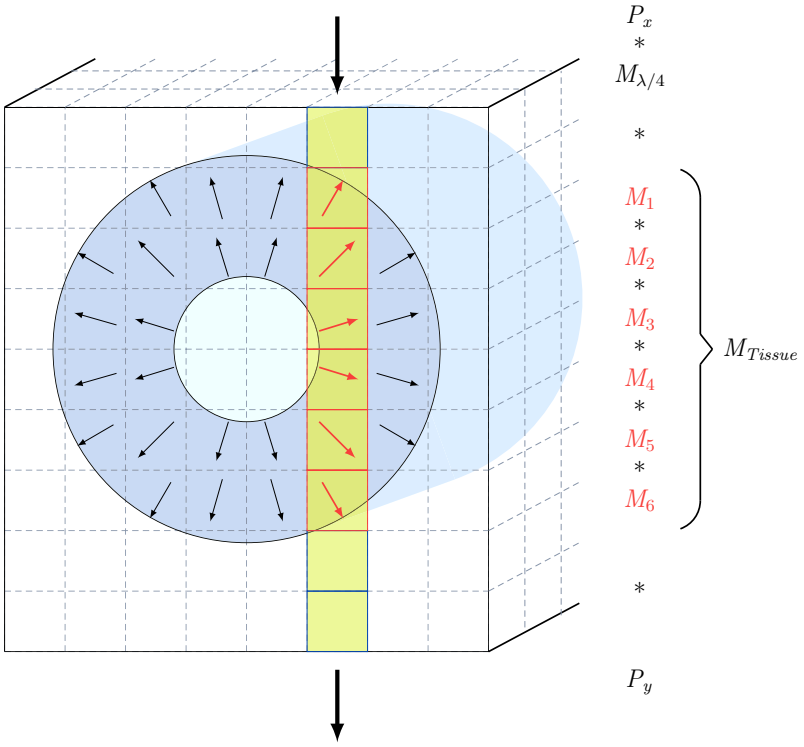


Abbildung 3.5.: Beispiel für den Jones Matrix Calculus anhand einer horizontalen Faser mit radialen optischen Achsen. Für jede der eingefärbten Würfel wird eine Matrix vom Typ 3.7 erstellt und in der abgebildeten Reihenfolge multipliziert. [2]

aufgespalten. Diese erfahren unterschiedliche Brechungsindizes, das bedeutet unterschiedliche Geschwindigkeiten, wodurch ein Phasenshift δ der Lichtwelle zustande kommt.
Der Wert des Brechungsindex des Gewebes ist abhängig von der Richtung der Nervenfaser, da der Lichtstrahl in Richtung der z-Achse einfällt. Über Taylorentwicklungen [2] und $sin^2 + cos^2 = 1$ kann näherungsweise eine Formel für den Brechungsindex hergeleitet werden, wobei $\Delta n = n_E - n_0$ die Materialkonstante der Doppelbrechung von biologischem Gewebe ist:

$$\Delta n(\theta) \approx \Delta n \sin^2(\theta) \tag{3.8}$$

Bekannt ist der Winkel α, der aus dem dreidimensionalen Vektor des aktuellen Myelin-Würfels bestimmt werden kann. φ beschreibt den Direktionswinkel der Projektion der Faser in der xy-Ebene und bei α handelt es sich um den Inklinationswinkel aus dieser Ebene heraus (Abb. 3.6). Die Vektoren, die zunächst als kartesische Koordinaten gespeichert werden, müssen in sphärische Koordinaten konvertiert werden. Dazu gibt es

die folgenden einfachen Umrechnungsvorschriften, wobei x, y und z die Komponenten des Vektors an der aktuellen Position sind:

$$\alpha = \arcsin(z) \qquad \varphi = \arctan2\left(\frac{y}{x}\right) \tag{3.9}$$

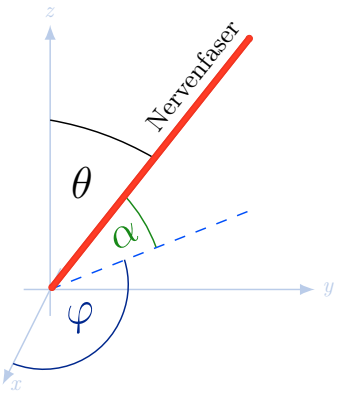


Abbildung 3.6.: Beschreibung der dreidimensionalen Nervenfaser durch φ und α

Über die Beziehung $\theta = 90° - \alpha$ kann die Formel 3.8 folgendermaßen umgeschrieben werden:

$$\Delta n(\theta) \approx \Delta n \cos^2(\alpha) \tag{3.10}$$

Aus diesen Herleitungen ergibt sich folgende Formel für die Phasenverschiebung δ, die durch Doppelbrechung der myelinisierten Nervenfasern im Gehirnschnitt entsteht.

$$\delta \approx \frac{2\pi}{\lambda}\, t\, \Delta n \cos^2(\alpha) \tag{3.11}$$

Mit λ wird die Wellenlänge des einfallenden Lichts bezeichnet und der Wert t beschreibt die Dicke des durchquerten Myelin-Würfels in z-Richtung, der Richtung des einfallenden Lichts. Diese Parameter sind der Eingabedatei zu entnehmen.
Ein Lichtstrahl, der durch den Vektor E_0 beschrieben wird, durchquert den 3D-PLI Versuchsaufbau. Der resultierende Lichtstrahl E_T, der von der Kamera aufgenommen wird, kann zusammengefasst durch folgende Matrixmulitplikationen beschrieben werden [2]:

$$E_T = P_y \cdot M_{Tissue} \cdot M_{\lambda/4} \cdot P_x \cdot E_0 \tag{3.12}$$

$$\text{mit} \qquad M_{Tissue} = M_N \cdot \ldots \cdot M_2 \cdot M_1 \tag{3.13}$$

4. Das Simulationsprogramm *simPLI*

Um die Ergebnisse des PLI-Workflows verifizieren zu können, werden definierte Eingabewerte benötigt. Hierfür wurde das Programm *simPLI* basierend auf dem gleichnamigen Vorgänger [11] in der Programmiersprache C++ entwickelt, das die 3D-PLI Messungen simuliert. Das Programm besteht aus einigen grundlegenden Schritten, die im Folgenden kurz erläutert werden:

1.a) Erzeugung der Nervenfasern
Die in externen Dateien definierten mathematischen Funktionen werden eingelesen und in einem dreidimensionalen Würfelgitter (Voxel) generiert. Die Fasern werden als hohle Rohre modelliert, die die Myelinschicht der Nervenfasern darstellen. Es wird zwischen Hintergrund, Myelin und Axon unterschieden.

1.b) Generierung eines dreidimensionalen Vektorfeldes
Jedem Voxel im Grid, das zur Myelinschicht gehört, wird ein dreidimensionaler Vektor zugeordnet. In einer Konfigurationsdatei kann definiert werden, ob die Vektoren radial oder axial zum Fasermittelpunkt verlaufen.

2. Erzeugung der 3D-PLI Bilderserie mittels Jones Matrix Calculus
Beim Matrix Calculus werden alle optischen Elemente des 3D-PLI Versuchsaufbaus durch Matrizen repräsentiert (Abschnitt 3.3). Diese werden entlang des Verlauf des Lichtstrahls, der den Versuchsaufbau durchleuchtet, miteinander multipliziert. Das Ergebnis ist ein Intensitätswert für jeden Punkt der xy-Ebene. Da sich die Polarisationsfilter und die Verzögerungsplatte von 0° bis 170° in 10° Schritten um das Gewebe drehen, wird so eine Folge von 18 3D-PLI Bildern, deren Intensitätsverlauf eine Sinuskurve approximiert, simuliert.

3. Optionale Kippung des Gehirngewebes
Mit Hilfe der Software ist es ist möglich, das Kippen des Gehirngewebes in der Versuchsanordnung zu simulieren. Da der Lichtstrahl nun durch modifizierte Gewebeteile verläuft, müssen beim Jones Matrix Calculus andere Matrizen für die Gewebevoxel miteinander multipliziert werden.

4.1. Eingabe für *simPLI*

Die Eingabeparameter des Simulationsprogramms *simPLI* werden in einer vom Benutzer zu erzeugenden Textdatei gespeichert und beim Start des Programms eingelesen. Informationen zu den einzelnen Eingabedaten und ein Beispiel einer Konfigurationsdatei ist in Anhang A.1 zu finden.

4.1.1. Funktionsübergabe mittels Python

Bei der Implementation von *simPLI* wurde eine Eingabeschnittstelle von C++ zu Python für die Definition der Nervenfasern entwickelt. Zum einen sollen beliebig viele Fasern definiert werden können, zum anderen soll deren Verlauf für jede Faser unabhängig von den anderen festgelegt werden können.
Die gewünschten Fasern müssen vom Benutzer als Python-Funktionen definiert werden und werden anschließend vom Simulationsprogramm über den Datei- und den Funktionsnamen importiert [12]. Der Vorteil von Python als Interpretersprache liegt darin, dass die extern definierten Funktionen nicht vorher compiliert werden müssen und somit direkt in das C++-Programm eingebettet werden können.

Die Fasern werden durch hohle Rohre im dreidimensionalen Simulationsvolumen modelliert, weshalb die Python-Funktionen vom Benutzer in Parameterform zu implementieren sind. Jeweils für die x-, y- und z-Komponente muss eine beliebige mathematische Funktion definiert werden. Alle drei werden an einer beliebigen Stelle, die der Python-Funktion übergeben wird, ausgewertet. Die drei Koordinatenwerte werden anschließend in einer Liste gespeichert, die an das Simulationsprogramm zurückgeliefert wird. Ein einfaches Beispiel für die Implementation einer Faserfunktion ist im Anhang A.2 zu finden. Das Koordinatensystem wird intern folgendermaßen festgelegt:

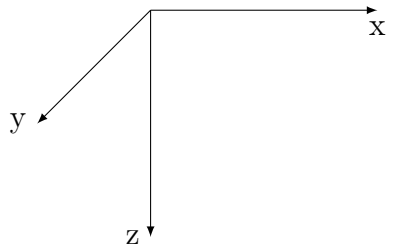


Abbildung 4.1.: Internes Koordinatensystem

Weiterhin ist zu beachten, dass die Funktionen relativ zur physikalischen Größe des Simulationsvolumens definiert werden müssen. Die Pixelgröße aus der Konfigurationsdatei definiert die Größe der Rasterung des zugrundeliegenden Voxelgitters. Der Benutzer ist eigenständig dafür verantwortlich, Funktionsdefinitionen, zugehörige De-

finitionsbereiche und die Größe des Simulationsvolumens und der Pixel so zu wählen, dass die Funktionen im Simulationsvolumen liegen.

4.2. Erzeugung der Nervenfasern

4.2.1. Vorgehen zur Erzeugung

Als Datenstruktur zur Speicherung der Faserstruktur wird ein dreidimensionales Integer Array verwendet. Somit kann zwischen Myelinschicht, Axon und Hintergrund unterschieden werden. Der Wert 0 kennzeichnet den Hintergrund, 1 die Myelinschicht und 2 das Axon.
Da in einem Simulationsvolumen mehrere Fasern definiert werden können, werden die nun folgenden Schritte für jede dieser Definitionen durchgeführt. Der Definitionsbereich, der für jede Faser angegeben werden muss, bestimmt den Verlauf der Faser im Simulationsvolumen.

Algorithmus

Im ersten Schritt wird eine Laufvariable t mit der unteren Grenze des Definitionsintervalls initialisiert. Nun werden die x-, y- und z-Koordinaten der Faserfunktion an dieser Position t bestimmt, indem die Python-Funktion, die zur aktuellen Faser gehört, dort ausgewertet wird. Von dieser Routine wird eine Python-Liste mit drei Elementen zurückgeliefert, die in einem selbst definierten Datentyp *point* gespeichert werden.
Liegen die Ergebniskoordinaten des Punktes innerhalb des Simulationsvolumens, so werden die Voxel, die um diesen Faserpunkt liegen, im Array mit den entsprechenden Werten gefüllt. Die vom Benutzer definierte Python-Funktion liefert eine Liste reeller Zahlen zurück. Da das Voxelgitter, auf dem die Fasern erzeugt werden, aber diskret ist, müssen die Ergebniskoordinaten korrekt gerundet werden, das heißt zum Integerwert, der am nächsten liegt.
Im nächsten Schritt wird für jede Dimension der Abstand in Voxeln vom Fasermittelpunkt bis zum äußeren Radius der Faser bestimmt. In einem Quader (*Sliding Window*) um den aktuellen Faserpunkt mit den zuvor errechneten Abständen als Seitenlängen werden anschließend alle Voxel dahingehend untersucht, ob sie zum Myelin, Axon oder zum Hintergrund gehören (Abb. 4.2). Dieses Fenster wird im Verlauf der Erzeugung entlang der Faserfunktion verschoben, wodurch die gesamte Faserstruktur aufgebaut wird.
Beim Füllen des Simulationsvolumens mit Werten wird zwischen drei verschiedenen Situationen unterschieden:

Fall 1

Ist der Abstand eines Voxels aus dem Quader zum aktuellen Punkt auf der Faserfunktion kleiner oder gleich dem äußeren Faserradius und größer als der innere Radius der Faser, so gehört dieses Voxel zur Myelinschicht und wird im Integer Array mit einer 1 markiert.

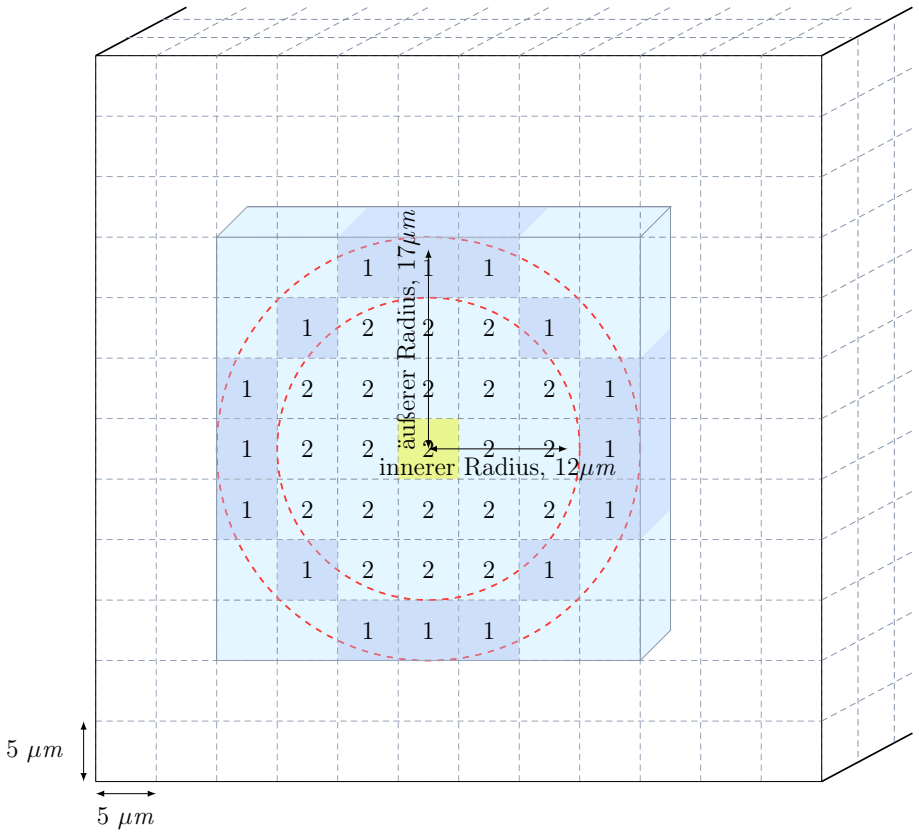


Abbildung 4.2.: Erzeugung einer geraden, horizontalen Faser. Alle Voxel im grauen Quader werden untersucht. Die Myelinschicht wird mit dem Wert 1 gekennzeichnet, das Axon mit dem Wert 2. Alle anderen Voxel gehören zum Hintergrund, der per Default den Wert 0 hat.

Fall 2

Ist der Abstand eines Voxels aus dem Fenster zum Fasermittelpunkt kleiner oder gleich dem inneren Radius, so liegt das Voxel im Axon und ihm wird im Array der Wert 2 zugewiesen.

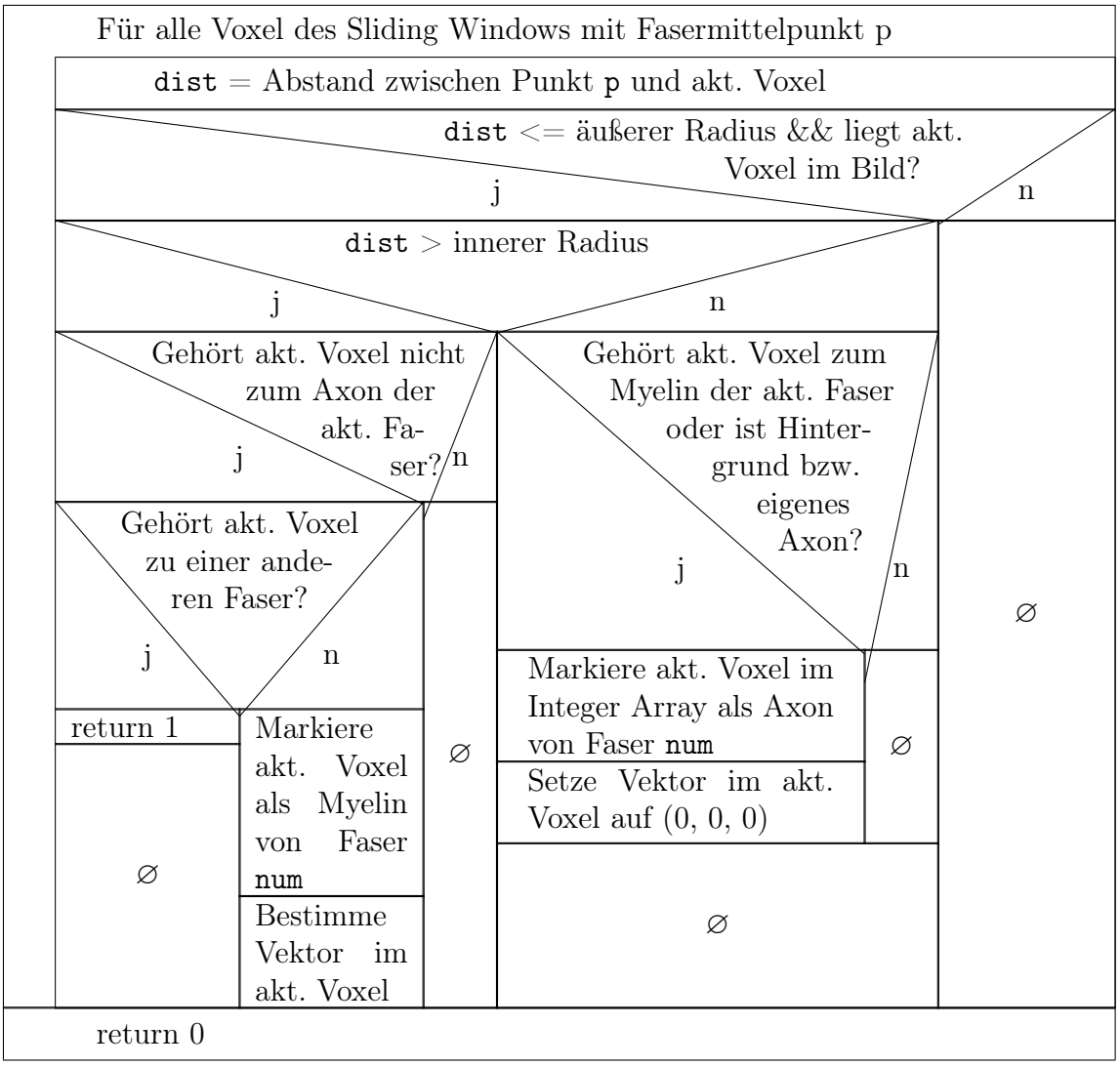
Fall 3

Mit einem Abstand größer als der äußere Radius, gehört das Voxel zum Hintergrund.

Besonderheiten

Da sich die Fasern nicht schneiden dürfen, müssen sich die im Integer Array eingetragenen Werte der unterschiedlichen Fasern voneinander unterscheiden lassen. Daher werden die Identifikatoren für Myelin und Axon mit der Nummer der aktuellen Faser multipliziert, um diese später den Fasern zuordnen zu können.
Mit dieser Methode werden alle Voxel in einem Kugelring um den aktuellen Faserpunkt als Myelin gekennzeichnet. Da die Faser jedoch als Hohlrohr modelliert wird,

int **fillVoxelsAroundFibers(point p, int num)**



Nassi 4.1: Erzeugung der Faserverläufe

wurden einige Voxel als Myelin markiert, die eigentlich dem Axon zuzuordnen sind. Deswegen darf der Identifikationswert des Axons den des Myelins überschreiben, die Umkehrung ist allerdings nicht erlaubt. Wo zuvor ein Hintergrundvoxel war, darf jeder andere Wert abgespeichert werden.
Damit der Eintritt und der Austritt der Funktionen in das Simulationsvolumen korrekt erzeugt werden, wird ein sogenannter *Ghostbereich* um das eigentliche Simulationsvolumen definiert. Dieser hat die Breite des äußeren Faserradius (Abb. 4.3). Bereits in diesem Bereich werden die Voxel im Sliding Window um den aktuellen Faserpunkt untersucht und, falls sie innerhalb des Volumens liegen, im Integer Array markiert. So ist es auch möglich, Fasern mit großem Radius, die teilweise im Simulationsvolu-

men liegen, deren Mittelpunkt aber außerhalb liegt, zu erzeugen. Diese Eigenschaft ist später bei der Parallelisierung des Programms sehr wichtig.

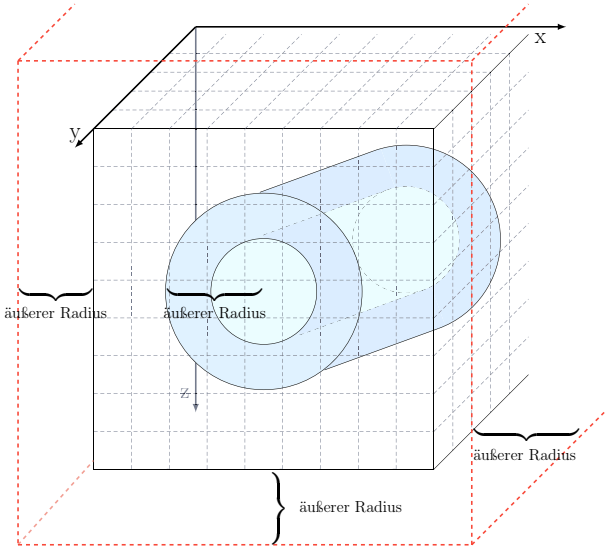


Abbildung 4.3.: Der *Ghostbereich* eines Simulationsvolumens hat immer die Größe des äußeren Faserradius der aktuell erzeugten Nervenfaser.

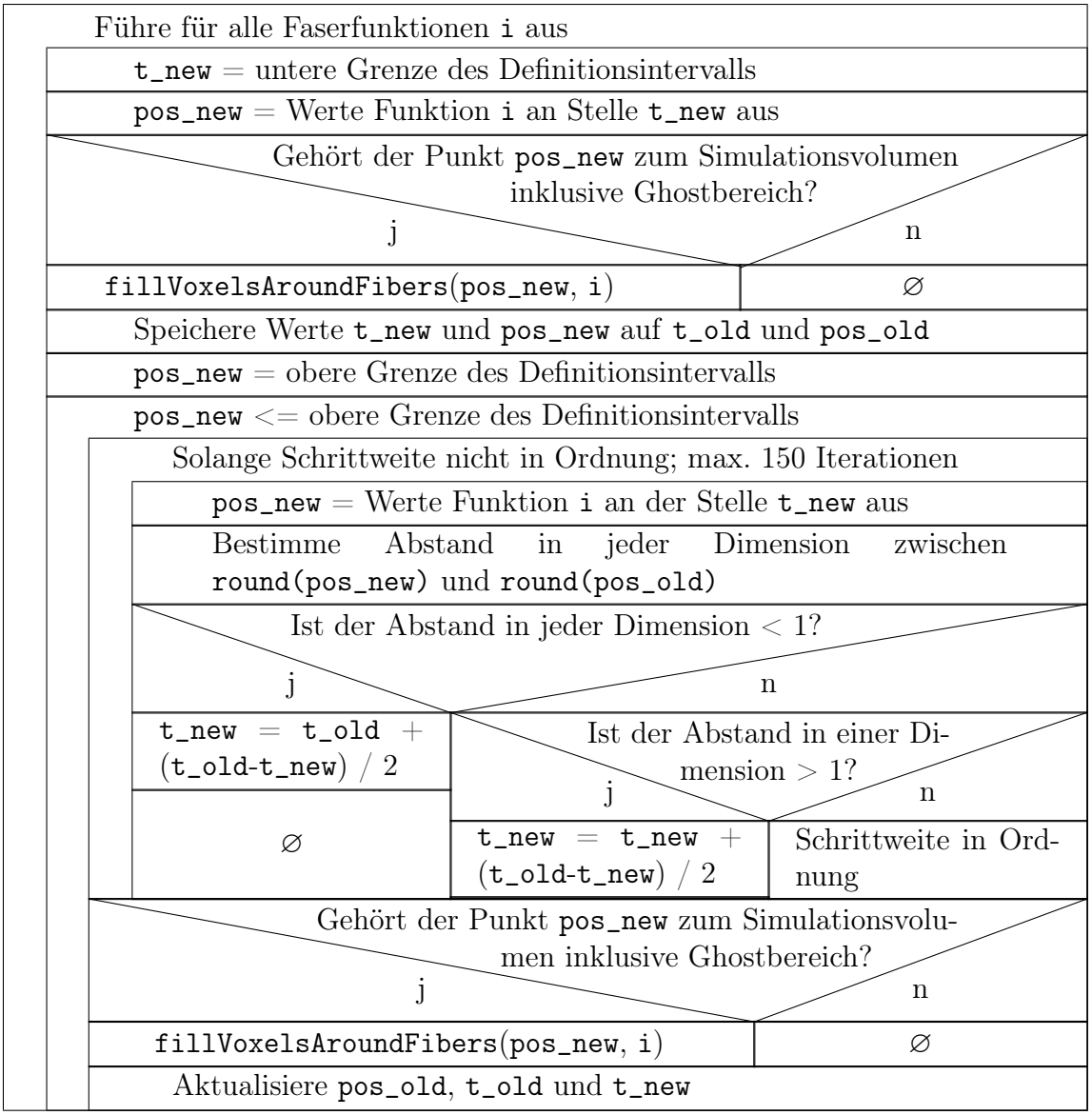
Die gesamte Funktion wird nach und nach an diskreten Stellen t innerhalb des Definitionsbereichs mit einer optimalen, variablen Schrittweite abgetastet und die umliegenden Voxel werden gefüllt. Wie diese Schrittweite zu wählen ist, wird im folgenden Abschnitt beschrieben.

4.2.2. Schrittweitensteuerung

Die Fasern werden durch kontinuierliche mathematische Funktionen in Parameterform beschrieben. Da diese auf einem diskreten Gitter dargestellt werden, muss die Schrittweite, mit der die Funktionen abgetastet werden, optimal bestimmt werden. Die hier beschriebene Routine wird für jede vom Benutzer definierte Faserfunktion durchgeführt.
Im ersten Schritt wird eine Laufvariable t mit der unteren Grenze des Definitionsintervalls initialisiert. Die Schrittweite entspricht zunächst der Länge des gesamten Intervalls. Die nächste Position zur Auswertung entspricht somit der oberen Intervallgrenze.
Anschließend wird die Faserfunktion an der aktuellen Stelle t ausgewertet und geprüft, ob die Ergebniskoordinaten innerhalb des um den *Ghostbereich* erweiterten Simulationsvolumens liegen. Ist dies der Fall, so werden nach der in Kapitel 4.2.1 beschriebenen Methode die umliegenden Voxel im Integer Array mit den richtigen Werten gefüllt.
Die Faserfunktion wird nun an der neuen Position t ausgewertet und der Abstand in

x-, y- und z-Richtung zwischen den vorherigen gerundeten und den neuen gerundeten Ergebniskoordinaten berechnet.

void **generate3DFibers()**



Nassi 4.2: Ablauf der Schrittweitensteuerung

Fall 1

Ist einer dieser gerundeten Abstände größer als ein Pixel, so wird die Schrittweite um die Hälfte verringert, da ansonsten bei der Fasererzeugung Voxel im Faserverlauf übersprungen werden. Die neue Position zur Auswertung wird verworfen und mit Hilfe der neuen Schrittweite erneut bestimmt.

Fall 2

Falls der gerundete Abstand in jeder Dimension kleiner als ein Pixel ist, so entspricht das neue gerundete Voxel dem Alten und die aktuelle Schrittweite ist zu klein gewählt. In diesem Fall wird die Schrittweite um die Hälfte vergrößert und die aktuelle Auswertungsposition erneut bestimmt.

Nach diesem Verfahren wird so lange vorgegangen, bis eine geeignete Schrittweite gefunden ist. Es ist möglich, dass Definitionsbereich und Faserdefinition unpassend gewählt wurden und im Ergebnisbild nichts von den Fasern zu sehen ist. In diesem Fall sollte der Benutzer seine Eingaben in der Konfigurationsdatei und seine definierten Python-Funktionen überprüfen.

4.3. Simulation der 3D-PLI Aufnahmen

4.3.1. Generierung des dreidimensionalen Vektorfeldes

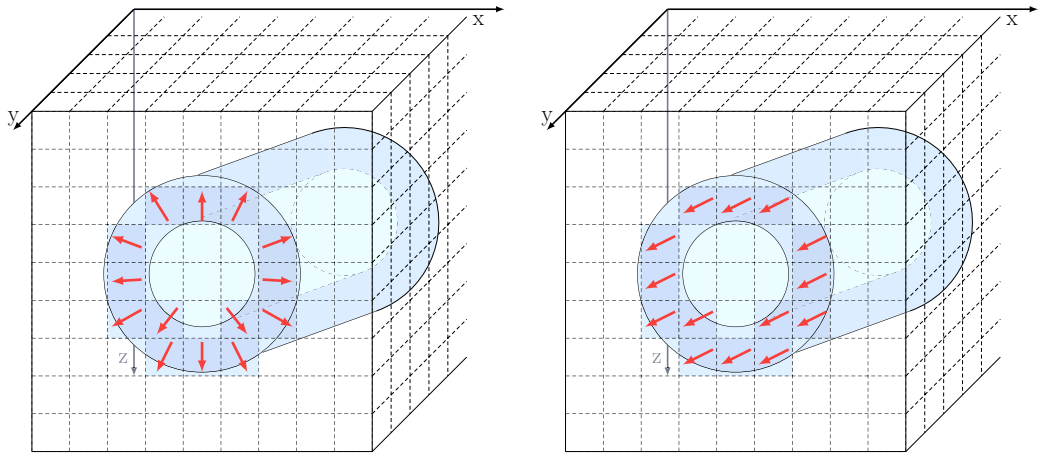
Während der Erzeugung der Faserverläufe wird jedem Voxel, das zur Myelinschicht gehört, ein dreidimensionaler Vektor zugeordnet. Diese Vektoren werden an der Position des aktuellen Myelinvoxels in einem Vektorfeld gespeichert.
Vom Benutzer wird definiert, ob ein axiales oder ein radiales Vektorfeld erzeugt werden soll (Abschnitt 3.2). Beim axialen Vektorfeld verläuft die Richtung der Vektoren parallel zur Faser. Radial bedeutet, dass die Vektoren sternförmig zum Mittelpunkt der Faser gerichtet sind.

Algorithmus

Beim Verfahren zur Fasererzeugung werden alle Voxel in einem Quader um den aktuellen Punkt auf der Faserfunktion markiert und anschließend werden sofort die Vektoren in den als Myelin gekennzeichneten Voxeln berechnet. Daher kommt es vor, dass schon ein Vektor im aktuellen Voxel bestimmt wurde, es aber eine bessere Approximation für diesen Vektor gibt, da der aktuelle Punkt auf der Faserfunktion näher an diesem Voxel liegt. Aus diesem Grund wird der Abstand zwischen dem aktuellen Punkt auf der Faserfunktion und dem Punkt, in dem der Vektor bestimmt werden soll, berechnet und gespeichert. Sollte schon ein Vektor vorhanden sein und der aktuell berechnete Abstand ist kleiner als der bereits gespeicherte, so wird der Vektor ersetzt. Andererseits wird keine neue Vektorberechnung durchgeführt, da die bisherige Approximation genauer ist.

Axiale Vektorerzeugung

Für das axiale Vektorfeld wird an jedes Myelinvoxel ein Tangentenvektor im aktuellen Punkt angenähert. Da die Steigung im aktuellen Faserpunkt unbekannt ist, wird die Tangente als Sekante approximiert. Für die Bestimmung des Sekantenvektors wird die aktuelle Faserfunktion an zwei Stellen in der Nähe des aktuellen Faserpunktes ausgewertet. Um eine möglichst gute Approximation zu erhalten, wird der Wert der



(a) Radiale Vektorerzeugung (b) Axiale Vektorerzeugung

Abbildung 4.4.: Vektorerzeugung in den Myelinvoxeln

aktuellen Laufvariablen t um die Hälfte der Schrittweite vergrößert und verkleinert. Die Faserfunktion wird an diesen beiden Stellen ausgewertet und der Differenzvektor dieser Punkte im dreidimensionalen Raum ergibt eine Näherung für den Tangentenvektor im aktuellen Faserpunkt.

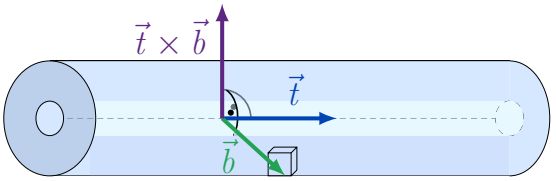
Radiale Vektorerzeugung

Im anderen Fall, der radialen Orientierung, muss der genäherte Tangentenvektor ebenfalls, wie gerade beschrieben, berechnet werden. Anschließend wird der Vektor bestimmt, der senkrecht auf diesem Tangentenvektor steht und durch den Mittelpunkt des Faserrohrs verläuft. Über zweimalige Kreuzproduktbildung wird dieser radiale Vektor berechnet (4.5):

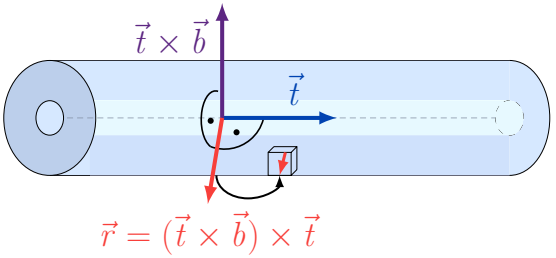
- $\vec{t}$: genäherter Tangentenvektor im aktuellen Faserpunkt
- $\vec{b}$: Differenzvektor aus aktuellem Faserpunkt und Punkt auf der Faserfunktion
- $\vec{r}$: Vektor der senkrecht zu $\vec{t}$ steht und durch aktuellen Faserpunkt verläuft

$$\vec{r} = (\vec{t} \times \vec{b}) \times \vec{t} = \begin{pmatrix} (t_2 b_0 - t_0 b_2) t_2 - (t_0 b_1 - t_1 b_0) t_1 \\ (t_0 b_1 - t_1 b_0) t_0 - (t_1 b_2 - t_2 b_1) t_2 \\ (t_1 b_2 - t_2 b_1) t_1 - (t_2 b_0 - t_0 b_2) t_0 \end{pmatrix} \tag{4.1}$$

In beiden Fällen wird der Vektor normiert, bevor er im Vektorfeld abgespeichert wird.



(a) Der lila Vektor steht senkrecht auf den Vektoren $\vec{t}$ und $\vec{b}$.



(b) Der radiale, rote Vektor $\vec{r}$ steht senkrecht auf dem lila Vektor und dem Vektor $\vec{t}$. Dieser wird normiert und im aktuellen Faserpunkt gespeichert.

Abbildung 4.5.: Schematische Darstellung der Erzeugung eines radialen Vektors durch doppelte Kreuzproduktbildung

void **calcVektorOrientation(point pFunktion, point p, int num)**

<table>
<tr><td colspan="3">dist = Abstand zwischen pFunktion und p</td></tr>
<tr><td colspan="2">Ist bisher kein Vektor gespeichert oder ist dist kleiner als bisher gespeicherter Abstand?
j</td><td>n</td></tr>
<tr><td colspan="2">prev = Werte Funktion num an der Stelle t-0.5 · sw aus</td><td rowspan="8">∅</td></tr>
<tr><td colspan="2">next = Werte Funktion num an der Stelle t+0.5 · sw aus</td></tr>
<tr><td colspan="2">t = Differenzvektor aus next und prev</td></tr>
<tr><td>Ist Vektororientierung radial?
j</td><td>n</td></tr>
<tr><td>b = Differenzvektor aus p und pFunktion</td><td rowspan="3">Speichere normierten Vektor t im Vektorfeld an Position p</td></tr>
<tr><td>r = (t × b) × t</td></tr>
<tr><td>Speichere normierten Vektor r im Vektorfeld an Position p</td></tr>
<tr><td colspan="2">∅</td></tr>
<tr><td colspan="2">Speichere dist im Abstandsarray an Position p</td></tr>
</table>

Nassi 4.3: Bestimmung eines radialen bzw. axialen Vektors

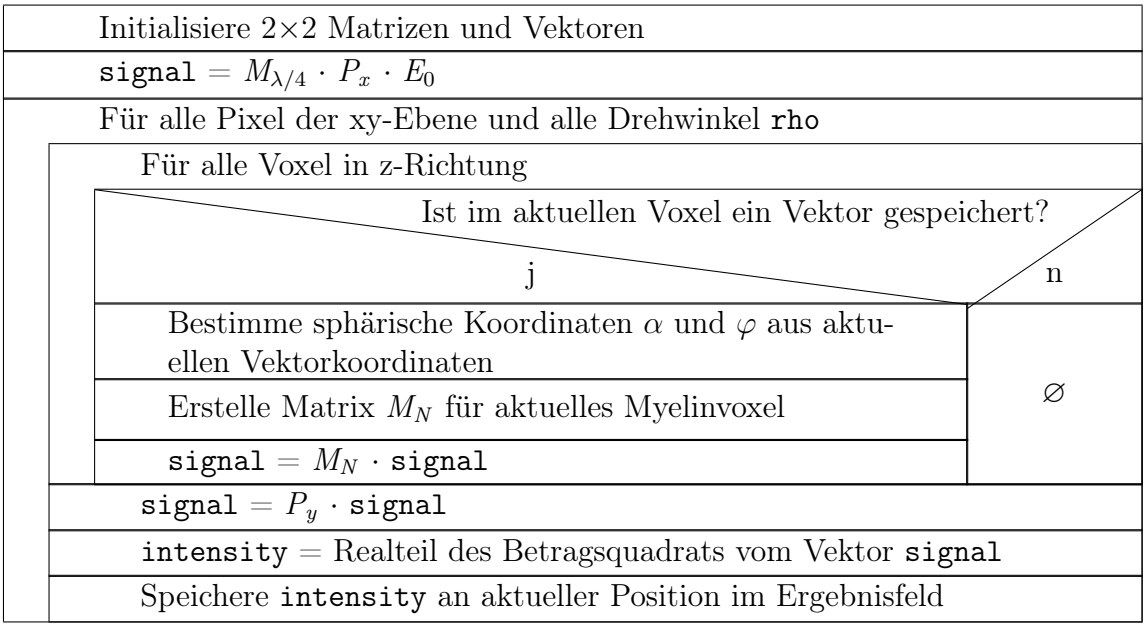
4.3.2. Matrix Calculus

Beim Jones Matrix Calculus werden die optischen Elemente des 3D-PLI Versuchsaufbaus durch Jones-Matrizen beschrieben. Diese werden entlang des Verlaufs des Lichtstrahls miteinander multipliziert. Das Ergebnis dieser Routine ist eine Folge von 18 Bildern, die die simulierten Intensitätswerte der 3D-PLI Aufnahmen darstellen. Das Aussehen der einzelnen Matrizen wird hier nicht erneut beschrieben, ist aber in Kapitel 3.3 nachzulesen.

Algorithmus

Zunächst wird der Vektor des einfallenden Lichts E_0 mit den beiden konstanten Matrizen P_x und $M_{\lambda/4}$ multipliziert. Das Ergebnis ist ein Vektor und wird im Folgenden mit *signal* bezeichnet.
Anschließend wird für jedes Pixel der xy-Ebene und jeden Drehwinkel ρ der Verlauf des Lichtstrahls in z-Richtung durch das Simulationsvolumen verfolgt. Das heißt bei festem x, y und ρ Wert wird die z-Richtung mit einer Schleife durchlaufen und in jedem Voxel wird im Vektorfeld überprüft, ob ein Vektor gespeichert wurde. Ist dies der Fall, so bedeutet das, dass dieses Voxel zur Myelinschicht gehört und es wird eine Matrix M_N erzeugt. Die kartesischen Koordinaten des Vektors werden zuvor in sphärische Koordinaten umgerechnet. Diese Matrix wird nun von links mit dem *signal* Vektor multipliziert. Sollte an der aktuellen Stelle im Gitter kein Vektor gespeichert sein, so wird die Multiplikation mit der Einheitsmatrix nicht durchgeführt.

void **simulateJonesAndMixing()**



Nassi 4.4: Durchführung des Jones-Matrix Calculus

Ist für feste x, y und ρ Werte das Simulationsvolumen in z-Richtung durchlaufen worden, so wird abschließend noch die Matrix P_y mit dem *signal* Vektor multipliziert. Der Intensitätswert, der im Ergebnisfeld gespeichert wird, ist der Realteil des Betragsquadrats des *signal* Vektors.

4.4. Kippung der Faserverläufe

Im 3D-PLI Versuchsaufbau fällt der Lichtstrahl senkrecht auf die Probenplatte mit dem Gehirnschnitt. Die Probenplatte kann jedoch um bis zu 8° in jede Richtung gekippt werden. Diese Kippung wurde im Programm *simPLI* ebenfalls simuliert. Jedoch wird in der Simulation der Einfallswinkel des Lichtstrahls geändert, anstatt das Simulationsvolumen zu kippen, was den gleichen Effekt hat.
Da die beiden Kippwinkel bekannt sind, kann der Eintritts- und der Austrittspunkt des Lichtstrahls im Simulationsvolumen berechnet werden. Anschließend wird eine Gerade bestimmt, die durch diese beiden Punkte verläuft. Dazu wird ein Algorithmus zur Rasterung von Linien auf einem Gitter im Dreidimensionalen verwendet. Die vom Lichtstrahl durchquerten Voxel werden in einer Liste gespeichert, um die zugehörigen Vektoren später im Matrix Calculus zu multiplizieren.

4.4.1. Kippwinkel und Bestimmung des Austrittspunktes

In der Konfigurationsdatei müssen zwei Kippwinkel angegeben werden. Der Winkel θ beschreibt den Winkel zwischen dem senkrechten, ungekippten und dem modifizierten Lichtstrahl (Abb. 4.6). Da die Probenplatte in der Realität nur um einen kleinen Winkel gekippt wird, sind als Eingabewert für θ nur Winkel bis maximal 8° erlaubt. Hat θ den Wert 0°, so findet keine Kippung statt. Der zweite Winkel φ, der angegeben werden muss, beschreibt den Winkel zwischen der Projektion des gekippten Lichtstrahls auf die xy-Ebene und der x-Achse. Dieser Winkel kann Werte von 0° bis 360° annehmen.
Wird der Lichtstrahl gekippt, so nimmt der Strahl einen anderen Verlauf durch das Gehirngewebe. Daher müssen beim Matrix Calculus andere Matrizen entlang des Verlaufs des Lichtstrahls miteinander multipliziert werden als im Fall ohne Kippung.
Ist der Ein- und Austrittspunkt des Lichtstrahls in das Simulationsvolumen bekannt, dann kann die Gerade zwischen diesen beiden Punkten berechnet werden. Als Startpunkt wird der Punkt genau in der Mitte der xy-Ebene gewählt. Sowohl der Endpunkt als auch der Geradenverlauf wird nur für diesen einen Punkt bestimmt und die Verschiebungen in x- und y-Richtung werden im Laufe der Simulation auf alle anderen Punkte der xy-Ebene umgerechnet. Es wird zwischen drei verschiedenen Szenarien unterschieden. Die z-Koordinate des Austrittspunktes ist in allen Fällen der maximal mögliche Wert in z-Richtung.

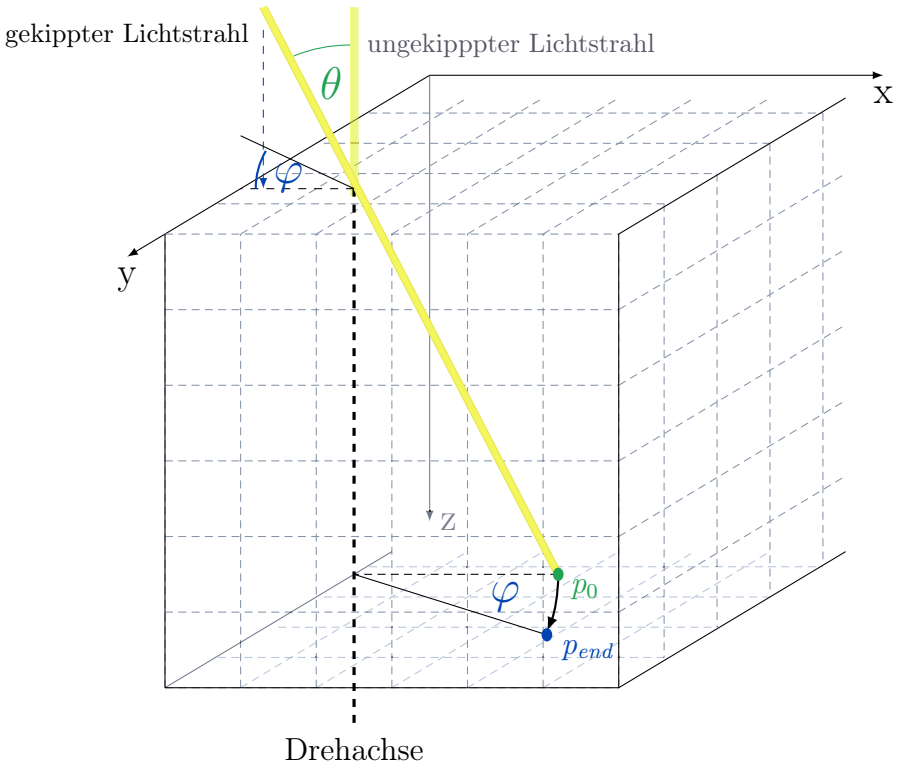


Abbildung 4.6.: Darstellung der Kippung des Lichtstrahls um die Winkel θ und φ. Zunächst wird mit Winkel θ in x-Richtung gekippt, dann wird der resultierende Punkt p_0 um den Winkel φ um die Drehachse zum Punkt p_{end} gedreht.

Fall 1

Hat θ den Wert 0°, so wird der Lichtstrahl nicht gekippt und verläuft in Richtung der z-Achse. Der Endpunkt liegt genau unter dem Startpunkt am unteren Rand des Volumens in z-Richtung.

Fall 2

Ist der Winkel φ gleich Null und hat θ einen Wert, der von Null abweicht, so findet nur eine Kippung des Lichtstrahls in x-Richtung statt. Somit ändert sich auch nur der x-Wert des Endpunktes. Über die Formel

$$m = \tan(90 - \theta) = \frac{\Delta z}{\Delta x} \tag{4.2}$$

kann die Steigung der Geraden berechnet werden. Da für θ maximal Winkel bis 8° zugelassen sind, ist die Steigung relativ groß (zum Vergleich der kleinstmögliche Wert: $\tan(90 - 8) \approx 7.115$). Die x-Koordinate des Austrittspunktes kann bestimmt werden, indem zur x-Koordinate des Startpunktes der Quotient aus Breite des Simulationsvo-

lumens in z-Richtung und Steigung addiert wird (Formel 4.3). Die y-Koordinate des Endpunktes bleibt die des Startpunktes, da der Parameter φ den Wert Null hat.

$$x_{end} = x_{start} + \left(\frac{zDim}{m}\right) \tag{4.3}$$

Fall 3

Sind beide Winkelangaben von Null verschieden, so wird zunächst, wie oben beschrieben, die Kippung in x-Richtung p_0 durch den Winkel θ bestimmt. Anschließend wird dieser vorläufige Endpunkt p_0 um den Winkel φ gedreht und man erhält den endgültigen Austrittspunkt des Lichtstrahls (Abb. 4.6). Die Drehung wird durch die folgende Multiplikation mit einer Rotationsmatrix durchgeführt:

$$\begin{pmatrix} x_{end} \\ y_{end} \\ z_{end} \end{pmatrix} = \begin{pmatrix} \cos(\varphi) & -\sin(\varphi) & 0 \\ \sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 - x_{start} \\ y_0 - y_{start} \\ z_0 - z_{start} \end{pmatrix} + \begin{pmatrix} x_{start} \\ y_{start} \\ z_{start} \end{pmatrix} \tag{4.4}$$

Durch die Multiplikation eines Vektors mit der Rotationsmatrix aus 4.4 wird dieser Punkt um den Winkel φ um die z-Achse gedreht. In der Simulation soll der bereits in x-Richtung gekippte Austrittspunkt p_0 jedoch nicht um die z-Achse, sondern um die Achse des ungekippten Lichtstrahls rotieren. Daher wird p_0 zunächst durch die Subtraktion des Startpunktes zur z-Achse verschoben, dann mittels obiger Matrix gedreht und anschließend durch die Addition wieder zurück verschoben.

4.4.2. Rasterung der Geraden zwischen Ein- und Austrittspunkt

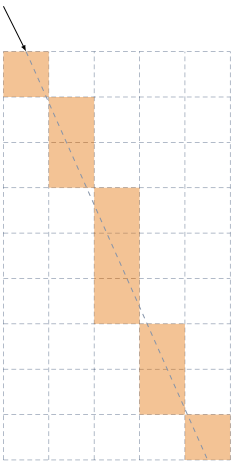


Abbildung 4.7.: Rasterung einer Geraden im Zweidimensionalen

Um eine Gerade auf dem Voxelgitter zwischen Ein- und Austrittspunkt zu bestimmen, wird zunächst der Abstand zwischen den beiden Punkten in jeder Dimension berechnet. Für jede der drei Dimensionen wird eine Schrittweite berechnet. Diese ergibt sich aus dem Quotienten des Abstandes der jeweiligen Dimension und dem Maximum der drei Abstände und ist in der Regel eine Fließkommazahl (Nassi 4.5). Die Koordinate mit dem maximalen Abstand zwischen Ein- und Austrittspunkt, in der Simulation immer die z-Komponente, hat natürlich die Schrittweite 1.
Die Koordinaten der Voxel, die zur Geraden gehören, angefangen beim Startpunkt, werden in einer Liste gespeichert. Zum zuvor gespeicherten Punkt wird dann in jeder Dimension die entsprechende berechnete Schrittweite addiert, die Koordinaten werden gerundet und anschließend zur Punkte-Liste hinzugefügt.

Da zuvor die maximale Dimension bestimmt wurde und die Schrittweite dieser Richtung immer 1 ist, ist bekannt, nach wie vielen Schritten der Austrittspunkt der Geraden erreicht wird. Durch dieses Vorgehen wird im Fall der Simulation immer ein Schritt in z-Richtung gegangen und, abhängig von der Schrittweite in x- und y-Richtung, nach einer gewissen Anzahl von Schritten ein Schritt in diese Richtungen.

void **create3DLines(point start, point end)**

Initialisiere Punkteliste `pList`
`d1` = `end[0] - start[0]` (ebenso für `d2` und `d3`)
`N` = Maximum von `d1`, `d2` und `d3` (hier immer `d3`)
`s1` = `d1` / `N` (ebenso für `s2` und `s3`)
`p` = `start`
Füge `p` zu `pList` hinzu
for `i`=0; `i`<`N`; `i`++
`p[0]` = round(`p[0]` + `s1`) (ebenso für `p[1]` und `p[2]`)
Füge p zu `pList` hinzu

Nassi 4.5: Bestimmung des Geradenverlaufs

4.4.3. Vektorinterpolation zwischen benachbarten Voxeln

Nach der Erzeugung der Punkteliste muss anhand dieser Liste entschieden werden, welche Vektoren anschließend im Matrix Calculus für die Erstellung und die Multiplikation der Jones-Matrizen verwendet werden.
Die Punkte der Liste müssen in der Regel auf ganzzahlige Werte gerundet werden, um diese einem Voxel im Gitter und damit einem Vektor zuordnen zu können. Um eine bessere Approximation des Vektors zu erhalten, wird nicht der Vektor des gerundeten Voxels verwendet, sondern es wird zwischen benachbarten Vektoren interpoliert. Da die Schrittweite zwischen zwei Listenpunkten in z-Richtung immer 1 ist und die z-Werte der gespeicherten Punkte somit immer ganzzahlig sind, wird nur über benachbarte Voxel der xy-Ebene gemittelt. Es wird zwischen drei Fällen unterschieden:

Fall 1

Sind alle Koordinaten des in der Liste gespeicherten Punkts bereits ganzzahlig, so muss nicht interpoliert werden. Zu jedem Vektor wird zusätzlich ein Gewichtungsfaktor bestimmt, der in diesem Fall den Wert 1 hat.

Fall 2

Falls entweder die x- oder die y-Koordinate eines Punkts eine ganze Zahl ist, wird zwischen dem Vektor des Voxels, das man durch Rundung des aktuellen Listenpunktes erhält, und dem Vektor des Voxels, das sich durch Ab- bzw. Aufrundung der nicht

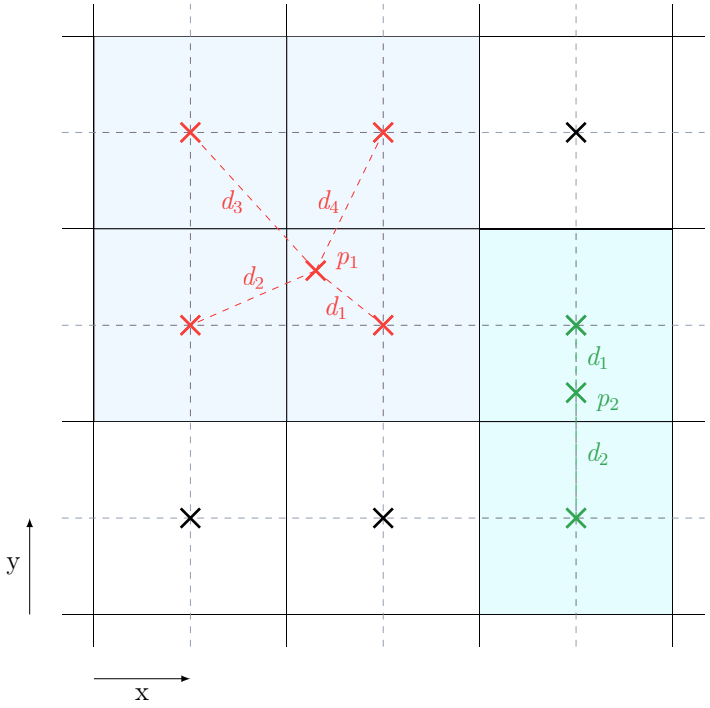


Abbildung 4.8.: Vektorinterpolation in zwei unterschiedlichen Fällen. Im grünen Beispiel wird der Vektor zum Listenpunkt p_2 aus den Vektoren der beiden grün markierten Voxeln zusammengesetzt. Im roten Beispiel ist die Vektorinterpolation bei vier Vektoren für den Punkt p_1 darstellt. Die Gewichtungsfaktoren berechnen sich aus den Abständen der umliegenden Voxel zu p_1.

ganzzahligen Koordinate des aktuellen Punktes ergibt, interpoliert (Abb. 4.8). Die Abstände des ungerundeten Listenpunkts zu den beiden gerundeten Punkten, deren Vektoren interpoliert werden, dienen als Faktoren zur Gewichtung der Vektoren. Allerdings erhält der Vektor des Voxels, dessen Abstand zum ungerundeten Listenpunkt kleiner ist, den größeren Abstand als Faktor, da dieser Vektor aufgrund seiner Nähe höher gewichtet in die Berechnung eingeht. Die Summe der beiden Abstände ergibt 1, da dies der Abstand zwischen zwei benachbarten Voxeln ist.

Fall 3

Ist sowohl die x- als auch die y-Koordinate des gespeicherten Listenpunkts eine Fließkommazahl, so setzt sich der Vektor, der im Matrix Calculus verwendet wird, aus vier benachbarten Vektoren zusammen. Welche vier Voxel in Frage kommen, ist von der Lage des ungerundeten Listenpunktes abhängig (Abb. 4.8). Für die Voxel der Vektoren, die in die Berechnung einfließen, wird der Abstand zum ungerundeten Listenpunkt bestimmt. Zu beachten ist, dass der Vektor des Voxels, das den kleinsten Abstand zum

ungerundeten Listenpunkt hat, den größten Gewichtungsfaktor besitzt. Die Faktoren sollen in der Summe 1 ergeben und werden über die folgende Berechnung bestimmt:

$$
\begin{aligned}
\text{sum} &= d_1 + d_2 + d_3 + d_4 \\
\text{sumInv} &= \frac{\text{sum}}{d_1} + \frac{\text{sum}}{d_2} + \frac{\text{sum}}{d_3} + \frac{\text{sum}}{d_4} \\
\text{fak}_1 &= \frac{\text{sum} : d_1}{\text{sumInv}} \\
\text{fak}_2 &= \frac{\text{sum} : d_2}{\text{sumInv}} \\
&\ldots
\end{aligned}
$$

In allen Fällen wird der interpolierte Vektor anschließend normiert.
Es kann zu Problemen bei der Vektorinterpolation kommen, wenn zwei Fasern im Simulationsvolumen genau nebeneinander liegen und die Vektoren radial nach außen verlaufen. In diesem Fall kann es vorkommen, dass Vektoren miteinander interpoliert werden, die nicht zur gleichen Faser gehören und in entgegengesetzte Richtungen weisen. Die Richtung des resultierenden Vektors ist dann nicht korrekt. Um diese Problematik zu vermeiden, ist der Benutzer dafür verantwortlich, die Fasern im Simulationsvolumen so zu definieren, dass mindestens ein Voxel Abstand zwischen zwei Fasern liegt, beispielsweise indem die Pixelgröße verkleinert wird.

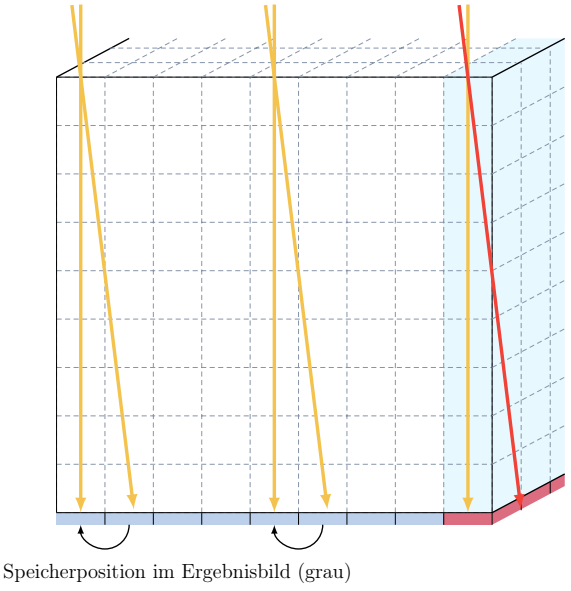


Abbildung 4.9.: Die hellgrauen Randbereiche des Volumens sind im Matrix Calculus nicht mehr abbildbar, das Ergebnisbild ist somit kleiner. Die Intensitätswerte werden im Ergebnisbild an der Position des Eintrittspunkts des Strahls auf der xy-Ebene gespeichert.

In gewissen Randbereichen kann, abhängig von der Kippung des Lichtstrahls, die Simulation der Punkte der xy-Ebene nicht durchgeführt werden. Der gekippte Lichtstrahl verläuft in diesen Bereichen nicht durch die gleiche Anzahl von Voxeln des Simulationsgebiets, sondern tritt aufgrund des Kippwinkels schon vorher aus dem Volumen aus. Da in diesen Punkten der xy-Ebene der Matrix Calculus nicht durchgeführt werden kann, ist das Ergebnisbild des gekippten Volumens kleiner als im ungekippten Fall. Die Randbereiche, in denen die Simulation nicht möglich ist, werden farblich gekennzeichnet, um sie von den simulierten Werten zu unterscheiden (Abb. 4.9).
Weiterhin muss festgelegt werden, an welcher Position die Intensitätswerte des Matrix Calculus des gekippten Simulationsvolumens gespeichert werden. Durch die Kippung ändern sich die x- und y-Komponente zwischen Ein- und Austrittspunkt des Lichtstrahls. Es wurde festgelegt, dass die Intensitätswerte immer an der Position des Eintrittspunkts des Lichtstrahls in der xy-Ebene gespeichert werden.

4.5. Ausgabe der Ergebnisse als HDF5-Datei

4.5.1. Ausgabeformat

Das zur Speicherung der Ergebnisse gewählte Datenformat ist das *Hierarchical Data Format 5* (HDF5). Hauptsächlich werden mit HDF5 große Datenmengen aus wissenschaftlichen Anwendungen gespeichert. Die Datenstruktur ist wie ein Dateisystem hierarchisch organisiert. Jedes Objekt innerhalb einer HDF5-Datei hat einen eindeutigen Pfad unter dem es erreichbar ist. Der Dateipfad innerhalb der HDF5-Datei beginnt mit dem Wurzelverzeichnis „/“. Im Folgenden werden die wesentlichen Elemente der HDF5-Struktur kurz erläutert.

Gruppen

Gruppen sind Strukturen, die wiederum mehrere oder auch gar keine HDF5-Objekte enthalten. Sie sind vergleichbar mit Ordnern bei einem Dateisystem. Sie können auch weitere Gruppen aufnehmen und beliebig tief geschachtelt werden.

Datensätze

Die konkreten Daten, die in HDF5 gespeichert werden sollen, werden in sogenannten Datensätzen verwaltet. Die Dimension und die Größe der Daten ist beliebig wählbar und beides wird bei der Erzeugung des Datensatzes festgelegt. Über den vergebenen Namen des Datensatzes ist der Zugriff auf das Objekt möglich.
Neben dem Namen des Datensatzes muss auch der Datentyp festgelegt werden. Je nachdem welcher Typ von Daten gespeichert werden soll, muss die entsprechende HDF5 Bezeichnung beim Schreib- bzw. Lesevorgang angegeben werden [13].

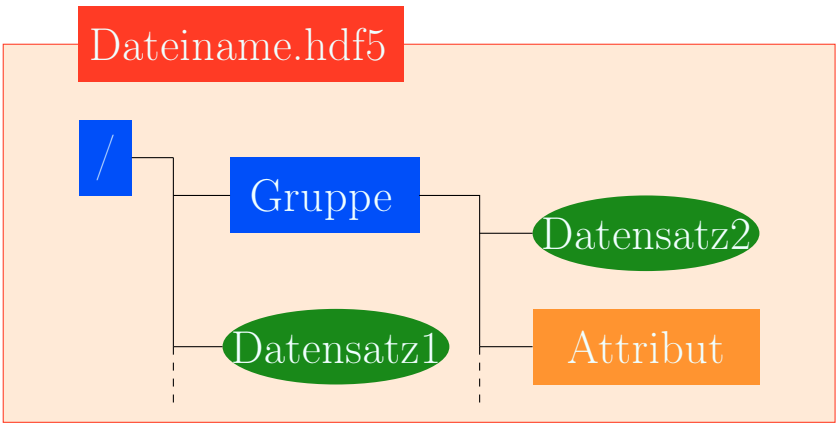


Abbildung 4.10.: Struktur einer HDF5-Datei

Attribute

Es ist möglich, allen HDF5-Objekten, außer dem Wurzelverzeichnis, Attribute hinzuzufügen. Darin können zusätzliche Metadaten zum Objekt gespeichert werden, wie beispielsweise Informationen über die Verwendung der Datensätze oder die ID der Person, die die Daten erstellt hat. Attribute bestehen aus zwei Teilen: dem Namen und einem Wert. Auch Attribute sind genau genommen kleine Datensätze, bieten aber weniger Funktionalität.

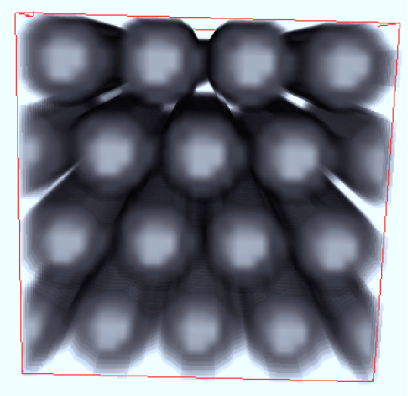
Ausgabestruktur *simPLI*

Bei der Ausgabe der Ergebnisse des Simulationsprogramms *simPLI* wird eine Ausgabedatei erstellt, die alle Daten enthält. Der Name dieser Datei kann beim Aufruf des Programms als zweiter Kommandozeilenparameter angegeben werden. Sollte dies nicht der Fall sein, wird eine Datei mit einem Default-Namen erstellt.
Das Simulationsprogramm schreibt folgende drei Datensätze in die übergebene HDF5-Datei:

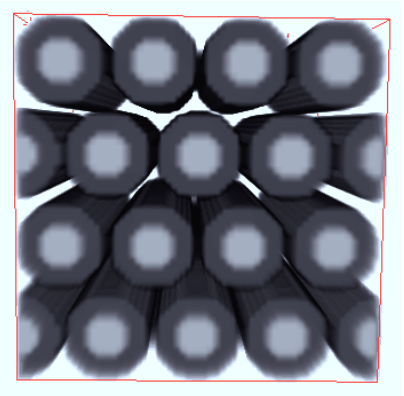
- **tissue:** Das dreidimensionale Feld, das den Verlauf der definierten Nervenfasern im Simulationsvolumen darstellt.
- **stack:** Das dreidimensionale Feld, das die simulierte Folge der 18 Bilder des 3D-PLI Versuchsaufbaus enthält.
- **vectorfield:** Das vierdimensionale Feld, das die Vektoren der Myelinschicht enthält.

4.5.2. Ergebnisse

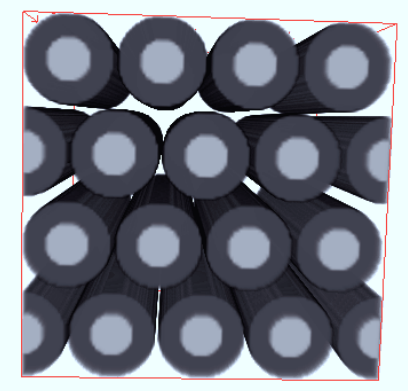
Der Benutzer des Simulationsprogramms *simPLI* kann beliebige Faserszenarien erzeugen und simulieren. Durch die Vergrößerung oder die Verkleinerung der physikalischen Größe eines Pixels bei gleichbleibender Größe des Gesamtvolumens ist es möglich, die Auslösung der Ergebnisbilder zu verschlechtern oder zu verbessern. Als Beispiel ist in Abbildung 4.11 ein hexagonales Faserbündel in verschiedenen Auflösungen dargestellt. Außer der Pixelgröße wurden bei der Erzeugung der verschiedenen Bilder keine anderen Parameter in der Konfigurations- oder der Python-Datei geändert.



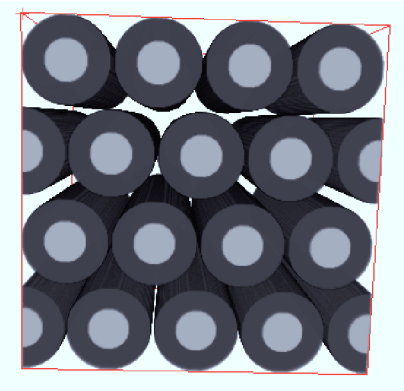
(a) Physikalische Größe eines Pixels: 0.004 mm



(b) Physikalische Größe eines Pixels: 0.002 mm



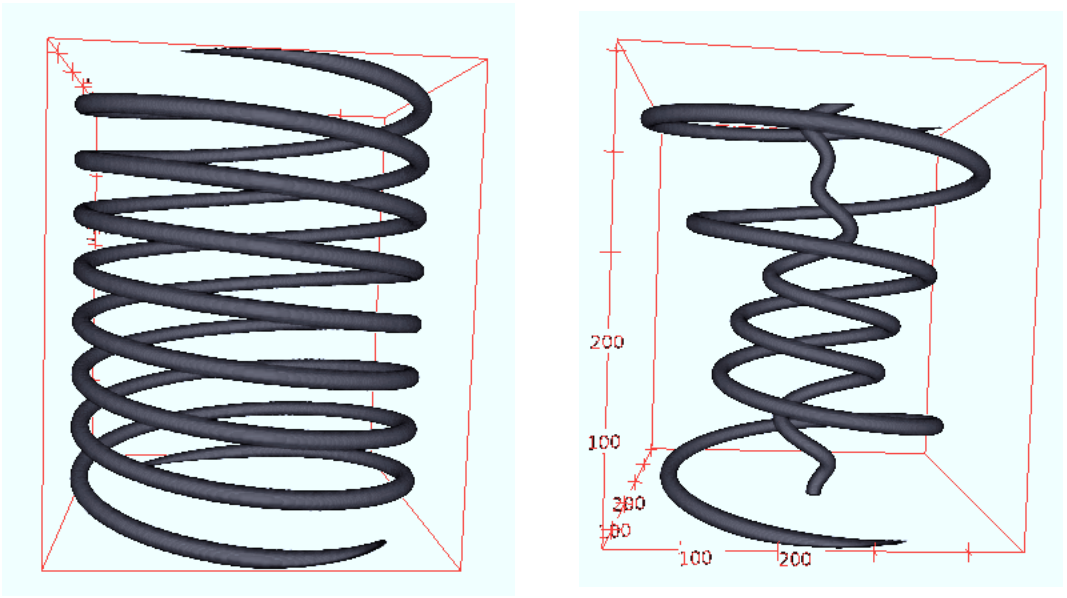
(c) Physikalische Größe eines Pixels: 0.001 mm



(d) Physikalische Größe eines Pixels: 0.0005 mm

Abbildung 4.11.: Hexagonales Faserbündel aus geraden, horizontalen Fasern in verschiedenen Auflösungen. Die physikalische Größe des Simulationsvolumens beträgt in allen Beispielen 0.256 mm×0.256 mm×0.256 mm.

Außer geraden Fasern können ebenfalls Kurven oder Spiralen über Trigonometrische Funktionen definiert werden (Abb 4.12). Auch abschnittsweise definierte Funktionen können innerhalb der Python-Funktionen vom Benutzer implementiert werden. Hier ein Beispiel für zwei ineinander gedrehte Spiralfasern:

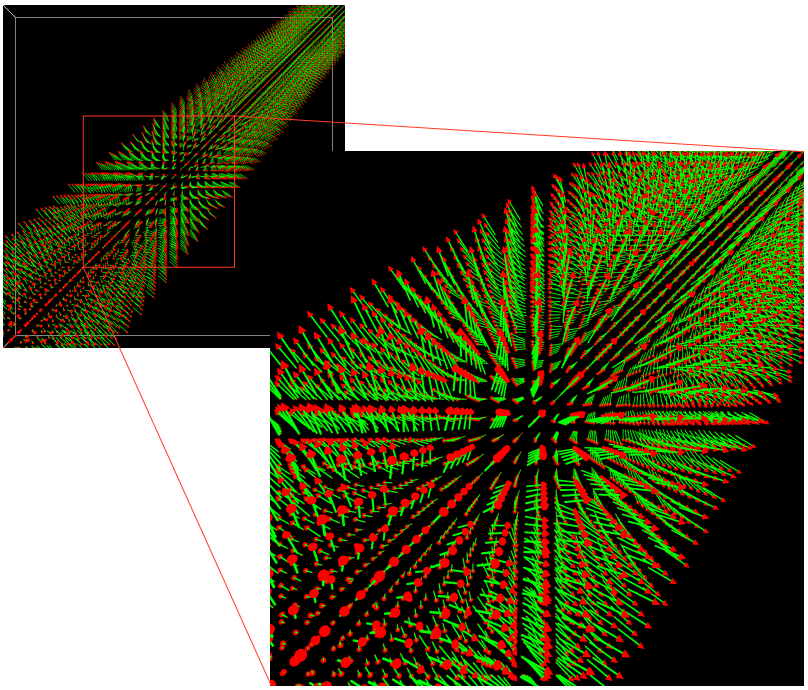


(a) Die Radien der Spiralen bleiben konstant.

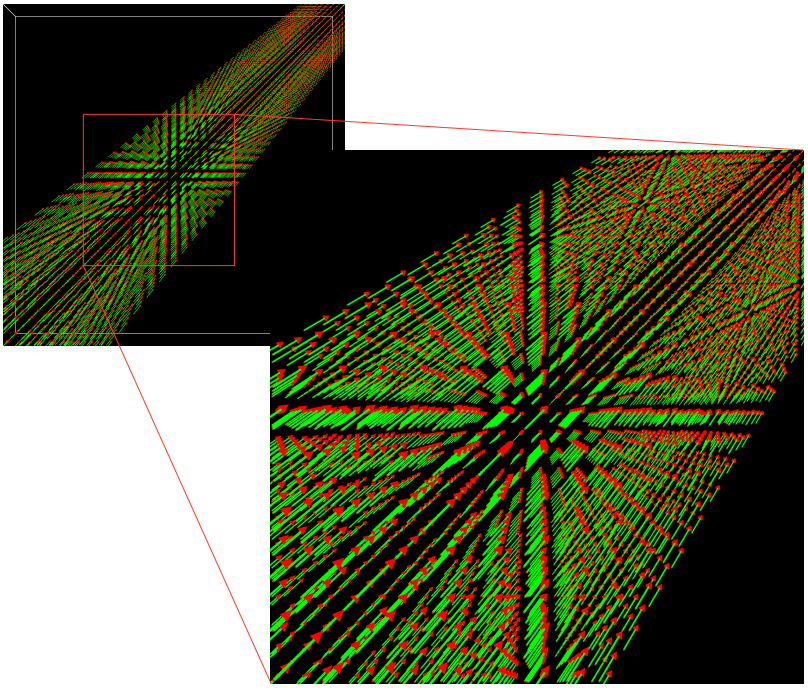
(b) Der Radius einer Spirale wird kleiner, der andere größer.

Abbildung 4.12.: Zwei ineinander gedrehte Spiralfasern. Die physikalische Größe des Simulationsvolumens beträgt 0.512 mm×0.512 mm×0.512 mm und die Seiten eines Pixels sind 0.001 mm lang.

Neben dem dreidimensionalen Datensatz mit den modellierten Fasern im Simulationsvolumen wird auch das Vektorfeld zu Kontrollzwecken ausgegeben. In der Myelinschicht werden entweder radiale oder axiale Vektoren definiert. In Abbildung 4.13 ist das Vektorfeld einer geraden, quer durch das Simulationsvolumen verlaufenden Faser dargestellt. Die roten Spitzen der Vektoren weisen einmal radial nach außen (Abb. 4.13a) und im anderen Fall in Richtung der Faser (Abb. 4.13b).



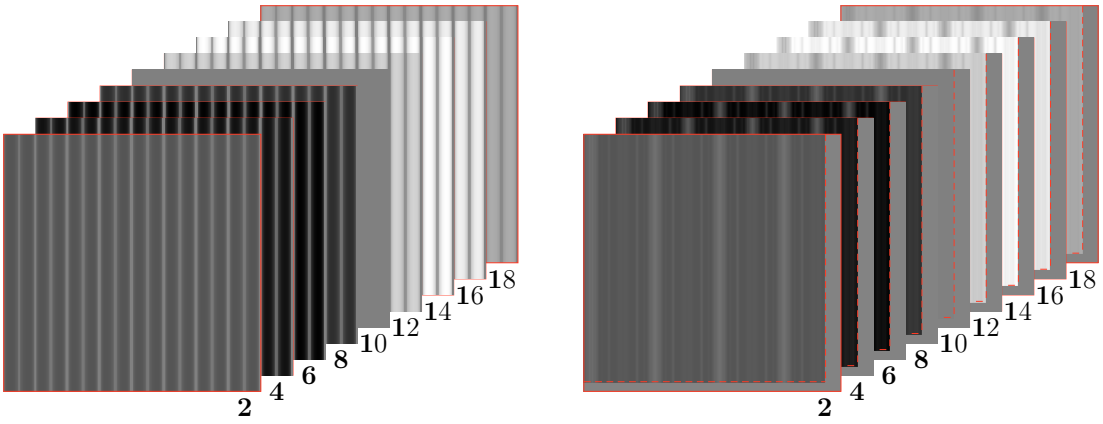
(a) Radiales Vektorfeld einer simulierten Faser



(b) Axiales Vektorfeld einer simulierten Faser

Abbildung 4.13.: Eine gerade, quer durch das Simulationsvolumen verlaufende Faser. Die Faser wurde jeweils mit radialen und axialen Vektoren in der Myelinschicht simuliert.

Der Datensatz, der beim Simulationsprogramm *simPLI* von größter Bedeutung ist, ist die Folge der 18 simulierten 3D-PLI Bilder. Im Folgenden sind die Simulationsergebnisse für das hexagonale Faserbündel mit einer Simulationsgröße von 256×256×256 Pixeln dargestellt. In Abbildung 4.14a fand keine Kippung des Gewebes statt, in Abbildung 4.14b wurde die Probe um die Winkel $\theta = 4°$ und $\varphi = 30°$ gekippt.



(a) Simulierte Bildfolge ohne Kippung

(b) Simulierte Bildfolge mit Kippung

Abbildung 4.14.: Simulierte Bildfolge des hexagonalen Faserbündels bestehend aus 4×4 Fasern mit und ohne Kippung. In beiden Fällen wird nur jedes zweite Bild dargestellt. In der gekippten Bildfolge sind die Bereiche erkennbar, in denen eine Simulation nicht möglich war.

Wie in Kapitel 2.2.2 beschrieben kann der Intensitätsverlauf eines Pixels über die Folge der 18 Bilder hinweg durch eine Sinuskurve approximiert werden. In Abbildung 4.15 ist dieser sinusförmige Verlauf beispielhaft für zwei unterschiedliche Pixel der Ergebnisbildfolge dargestellt. Es ist zu erkennen, dass die Amplitude (Retardierung) und die Verschiebung der Kurven in y-Richtung (Transmittanz) unterschiedlich sind.

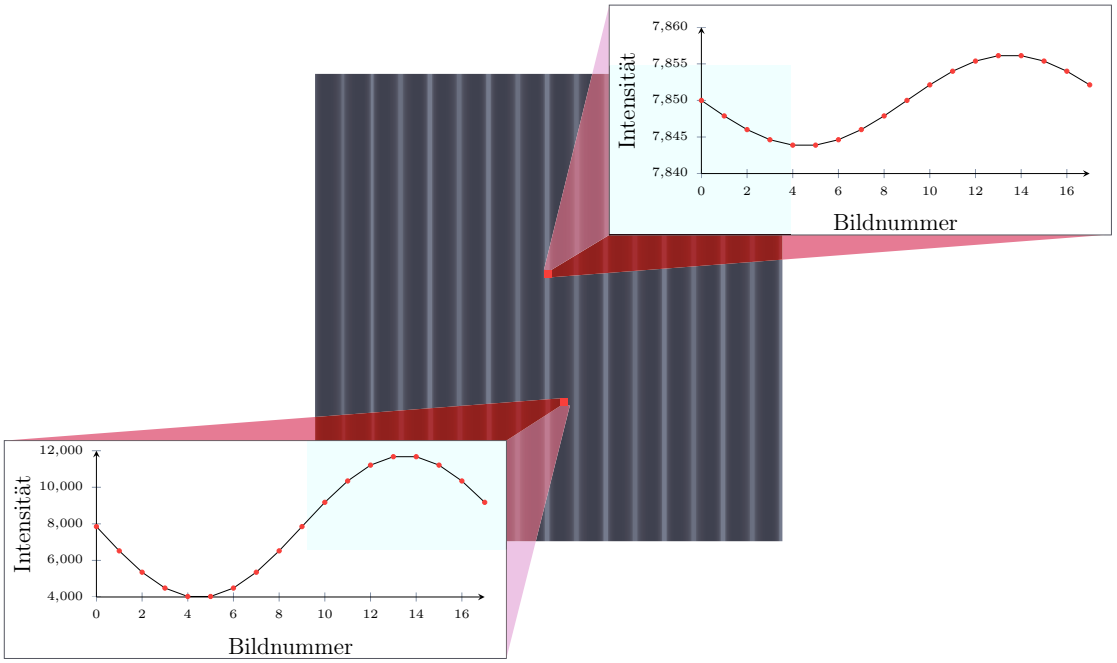


Abbildung 4.15.: Für die zwei im Bild markierten Pixel ist der Intensitätsverlauf über die 18 simulierten Bilder in einem Koordinatensystem dargestellt. Der sinusförmige Verlauf ist deutlich zu erkennen.

5. Parallelisierung *simPLI*

Die zu speichernden Daten erreichen bei größeren Simulationsvolumen schnell mehrere Gigabyte (bei einer Simulationsgröße von 1024×1024×1024 Pixeln werden ungefähr 20 GB Speicherplatz benötigt). Daher wurde das Simulationsprogramm *simPLI* parallelisiert. Der Hauptspeicher eines Rechners und damit auch die Größe der darauf zu speichernden Daten ist begrenzt. Durch die Hinzunahme von mehreren Prozessoren ist nun die Simulation größerer Daten möglich. Außerdem wird durch die Parallelisierung die Laufzeit des Programms reduziert, da sich sowohl die Erzeugung der Nervenfasern (Kapitel 4.2.1) als auch der Jones Formalismus (Kapitel 4.3.2) auf mehrere Prozessoren verteilt durchführen lässt.
Dieses Kapitel verschafft einen Überblick über parallele Rechnerarchitekturen, verschiedene Parallelisierungsstrategien und -modelle. Anschließend wird erläutert wie bei der Parallelisierung des Simulationsprogramms *simPLI* vorgegangen wurde. Im letzten Abschnitt erfolgt eine Analyse der Laufzeit des parallelen Programms.

5.1. Parallelisierungsparadigmen

Michael Flynn klassifizierte im Jahr 1966 parallele Rechnerarchitekturen anhand ihres Befehls- und Datenstroms [14]. Dadurch ergeben sich vier verschiedene Formen der Parallelität:

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Tabelle 5.1.: Klassifikation der Architekturen SISD (Single Instruction, Single Data), SIMD (Single Instruction, Mutiple Data), MISD (Multiple Instruction, Single Data) und MIMD (Mutiple Instruction, Mutiple Data)

SISD (Single Instruction, Single Data)

Konventionelle Einprozessorsysteme, die Anweisungen sequentiell abarbeiten, sind als SISD-Rechner bekannt. Beispiele dafür sind PCs oder Workstations, die nach der Von-Neumann-Architektur aufgebaut sind.

SIMD (Single Instruction, Multiple Data)

Zu dieser Kategorie zählen Systeme, die eine Rechenoperation auf mehreren zur Verfügung stehenden Datenströmen gleichzeitig ausführen. SIMD-Rechner werden auch Vektorprozessoren genannt.
Zum Einsatz kommen sie hauptsächlich bei der Verarbeitung von Bild-, Ton- und Videodaten. Die Anwendung ist dort sinnvoll, da in diesem Gebiet die zu verarbeitenden Daten meist sehr gut parallelisierbar sind. Bei einem Bild sind beispielsweise die Operationen, die auf den einzelnen Pixeln ausgeführt werden, immer die gleichen.

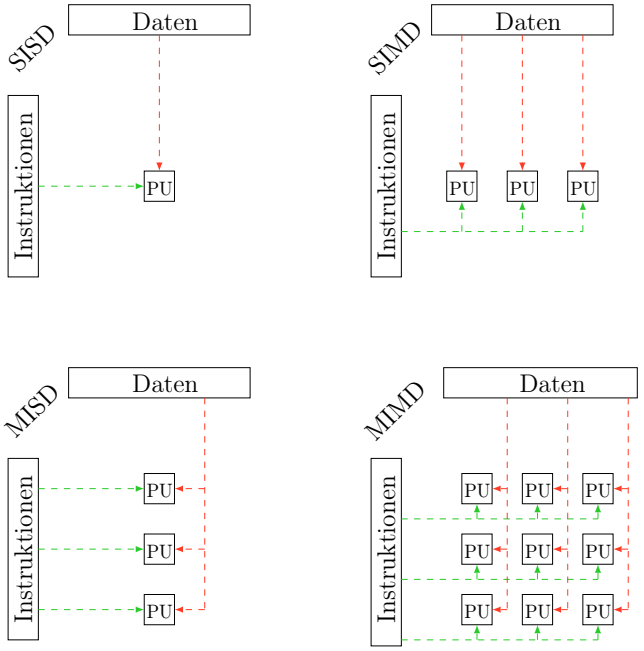


Abbildung 5.1.: Schematische Darstellung der Architekturen SISD, SIMD, MISD und MIMD

MISD (Multiple Instruction, Single Data)

Da der praktische Nutzen der Verarbeitung eines Datenstroms durch mehrere Instruktionsströme begrenzt ist, gibt es eigentlich keine Rechner, die dieser Klasse zuzuordnen sind. Es ist aber möglich,fehlertolerante Systeme, die redundante Berechnungen ausführen, zu dieser Klasse zu zählen. Weiterhin kann dieses System bei der Analyse von Daten, die auf verschiedene Aspekte hin untersucht werden sollen, angewendet werden.

MIMD (Multiple Instruction, Multiple Data)

In diese Kategorie fallen die meisten der heutigen Multiprozessorsysteme. MIMD-Rechner bestehen aus mehreren eigenständigen Prozessoren und können gleichzeitig verschiedene Operationen auf unterschiedlichen Datenströmen ausführen. Einer oder mehrere Prozessoren des Systems verteilen meist zur Laufzeit die anstehenden Aufgaben auf die zur Verfügung stehenden Ressourcen. Dabei kann jeder Prozessor auf die

Daten der anderen Prozessoren zugreifen.
Man unterscheidet zwischen eng gekoppelten Systemen, den Multiprozessorsystemen, und lose gekoppelten Systemen, wie Multicomputersysteme.

5.2. Parallelisierungsstrategien

Es gibt verschiedene Strategien, wie die Arbeit eines sequentiellen Programms auf mehrere Prozessoren aufgeteilt werden kann. Im folgenden Abschnitt werden vier verschiedene Ansätze vorgestellt.

Cloning/Farming

Bei dieser Strategie wird dasselbe Problem auf mehreren Prozessoren bearbeitet, um eine größere Anzahl von Ergebnisdaten zu erhalten. Alle Ergebnisse werden anschließend gesammelt und gemeinsam verarbeitet.
Vorteile beim Cloning sind die einfache Implementierung und die Tatsache, dass keine Kommunikation zwischen den einzelnen Prozessoren nötig ist. Allerdings ist diese Strategie nur für statistische Probleme, die Zufallszahlen nutzen, von Interesse.

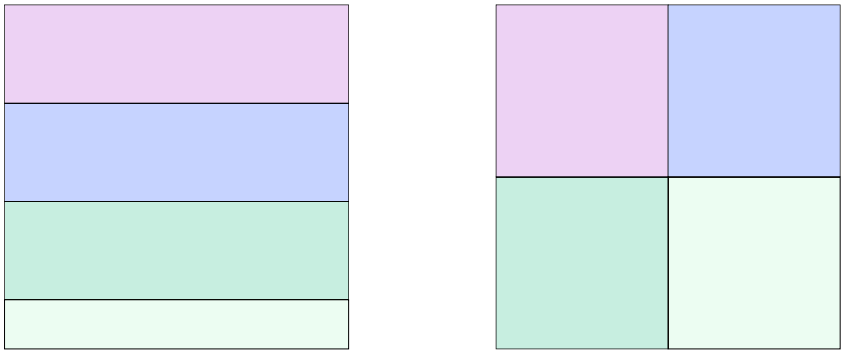
Master - Slave

Hierbei übernimmt ein Prozessor die Rolle des Masters und verteilt die gesamte Arbeit dynamisch auf die anderen Prozessoren, die als Arbeitsprozessoren fungieren. Immer wenn die Arbeitsprozessoren ihren bisherigen Arbeitsteil abgeschlossen haben, bekommen sie vom Master einen neuen Teil zugewiesen. Der Master-Prozessor selbst übernimmt keine Arbeit, sondern ist lediglich für die Zuteilung zuständig. Diese Strategie ist dynamisch, das heißt sie kann angewendet werden, wenn die Größe der Gesamtarbeit vorher nicht bekannt ist. Der Nachteil ist jedoch, dass der Master-Prozess selbst keine Berechnungen durchführt und diese Strategie somit weniger effizient ist. Außerdem läuft die gesamte Datenkommunikation über den Master-Prozess, was die Anwendung verlangsamt und für einen Flaschenhals bei der Kommunikation sorgen kann.

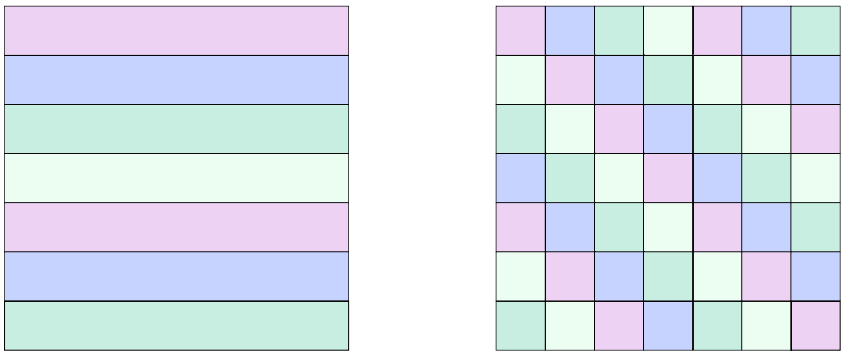
Gebietszerlegung (Domain Decomposition)

Der Datensatz des auszuführenden Programms wird bei der Gebietszerlegung in einzelne, möglichst gleich große Teilgebiete zerlegt und von verschiedenen Prozessoren bearbeitet.
Die Daten können zum einen **blockweise** verteilt werden. Dieses Vorgehen ist statisch, da die Zuteilung der Daten auf die Prozessoren beim Start des Programms bestimmt werden kann, und weist jedem Prozessor einen zusammenhängenden Block des Datengebiets einer zuvor berechneten Größe zu (Abb. 5.2a). Um hierbei eine möglichst gleiche Rechenlast zu erzeugen, sollte darauf geachtet werden, dass die Größe der Teilblöcke jedes Prozessors zumindest annähernd

gleich groß ist.
Eine weitere Möglichkeit die Programmdaten statisch aufzuteilen ist die **zyklische** Zerlegung. Dabei werden die Daten der Reihe nach an die Prozessoren verteilt, beispielsweise bei einem Array zyklisch über die Zeilen oder Spalten (Abb. 5.2b).
Weiterhin kann sowohl die blockweise als auch die zyklische Gebietsaufteilung in nur einer Dimension oder im Zweidimensionalen erfolgen. Die Wahl der Strategie hängt immer von der Art des Problems und des Datensatzes ab.
Die blockweise Zerlegung hat den Vorteil, dass der Arbeitsbereich jedes Prozessors zusammenhängend ist, was bei der zyklischen Verteilung nicht der Fall ist. Beim 2D Fall haben die Prozessoren im Durchschnitt weniger Grenzbereiche mit anderen Prozessoren im Vergleich zum 1D Fall. Dies ist von Vorteil, wenn die Daten in den Randbereichen zwischen den Prozessoren ausgetauscht werden müssen.
Wird das Gebiet zyklisch zerlegt, so sind die Daten unter Umständen homogener auf die Prozessoren verteilt und die Auslastung ist somit besser. Dies ist beispielsweise bei der Bearbeitung von Bildern der Fall.



(a) Blockweise Gebietszerlegung in 1D und 2D



(b) Zyklische Gebietszerlegung in 1D und 2D

Abbildung 5.2.: Gebietszerlegung eines zweidimensionalen Datensatzes auf 4 Prozessoren

Funktionsaufteilung (Functional Decomposition)

Bei diesem Ansatz werden nicht die Daten des Programms auf verschiedene Prozessoren verteilt, sondern die Module oder Funktionen. Der Algorithmus, der auf die gesamten Daten angewandt werden soll, wird in unabhängige Teile zerlegt, die auf die Prozessoren verteilt und parallel ausgeführt werden.

Durch Parallelisierung wird also die Leistung des Systems durch das Hinzufügen von mehr Ressourcen verbessert. Dies wird auch Skalierung genannt. Beim *strong scaling* variiert die Zeit für die Lösung abhängig von der Anzahl der Prozessoren bei fest gegebener **Gesamtgröße** des Problems. Bleibt hingegen bei verschiedener Anzahl von Prozessoren die Problemgröße **pro Prozess** fest, so spricht man von *weak scaling*.

5.3. Parallelisierungsmodelle

Ein Parallelisierungsmodell beschreibt die Sicht des Programmierers auf die Maschine und legt fest wie sie angesprochen wird. Die Sicht ist von verschiedenen Komponenten wie z.B. Betriebssystem, Hardware oder installierten Bibliotheken abhängig.
Es gibt verschiedene Kriterien, mit denen man parallele Modelle kategorisieren kann:

- Wie findet der Informationsaustausch zwischen den Prozessoren statt?
- Muss der Programmierer die Parallelität explizit steuern?
- Welche Synchronisationsmöglichkeiten gibt es?

Dies sind nur einige wichtige Kriterien, durch deren Kombination sich eine Vielzahl von verschiedenen Modellen ergibt. Die folgenden drei Einteilungen stützen sich zum einen auf die unterschiedliche Art des Informationsaustauschs, zum anderen auf das unterschiedliche Eingreifen des Programmierers in die Parallelisierung.

Datenparallelität

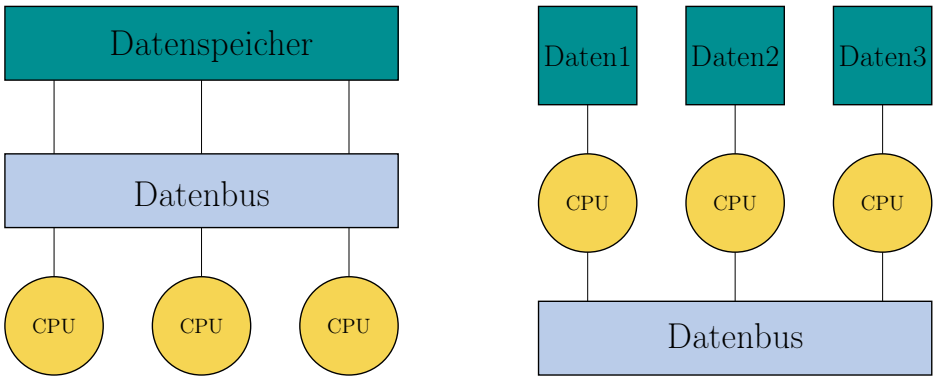
Häufig wird die gleiche Operation auf vielen verschiedenen Elementen einer Datenstruktur ausgeführt, beispielsweise auf allen Elementen eines Arrays oder einer Datenbank. Sind die einzelnen Daten auch noch unabhängig voneinander, dann kann das Prinzip der Datenparallelität angewendet werden. Dabei werden die zu verarbeitenden Daten gleichmäßig auf die Prozessoren aufgeteilt. Dieses Parallelisierungsmodell wird auch analog zum Architekturmodell als SIMD-Modell (Kapitel 5.1) bezeichnet.
Viele Programmiersprachen und Bibliotheken bieten bereits fertige Konstrukte an, um ein sequentielles Programm in ein datenparalleles Programm zu überführen, sodass sich der Programmierer nicht um die Aufteilung der Daten oder die Kommunikation kümmern muss.

Shared Memory

Beim *Shared Memory* Programmiermodell teilen sich alle Prozesse einen gemeinsamen Speicher (Abb. 5.3a). Allen Prozessoren ist der lesende und der schreibende Zugriff auf die gemeinsamen Daten erlaubt.
Die Kommunikation zwischen den Prozessen ist beim Shared Memory Modell einfach, da alle die gleiche Sicht auf die Daten haben. Allerdings kann es schnell zu Konflikten beim Lesen der Daten kommen, wenn zu viele Prozessoren gleichzeitig auf die gleichen Daten über nur einen Datenbus zugreifen wollen.

Distributed Memory

Im Gegensatz zum *Shared Memory* Modell besitzt jeder Prozessor beim *Distributed Memory* Modell einen eigenen, privaten Speicher (Abb 5.3b). Für den Informationsaustausch wird häufig das Prinzip *Message Passing* [14] angewendet. Dabei werden explizite Kommunikationsanweisungen zwischen den an der Kommunikation beteiligten Prozessoren verschickt. Da jeder Prozessor seine Berechnungen auf anderen Daten ausführt, muss der Programmierer sich zuvor um die Verteilung der Gesamtdaten auf die Prozessoren kümmern und dies explizit implementieren.



(a) Shared Memory (b) Distributed Memory

Abbildung 5.3.: Schematische Darstellung der Modelle *Shared Memory* und *Distributed Memory*

5.4. Parallelisierung des Simulationsprogramms

Beim Simulationsprogramm *simPLI* können zwei wesentliche Module parallelisiert werden. Zum einen wird die Erzeugung der Nervenfasern auf mehrere Prozesse verteilt, sodass jeder Einzelne nur einen Teil des Simulationsvolumens mit Gewebewerten füllt. Andererseits eignet sich die Simulation der 3D-PLI Bilder, der Matrix Calculus,

sehr gut für eine Parallelisierung. Da für alle Punkte der xy-Ebene die Matrizen der Myelinschicht in z-Richtung miteinander multipliziert werden, kann auch hier das gesamte Simulationsvolumen in Blöcken auf die Prozesse verteilt werden.
Es wird die Strategie der Gebietszerlegung zur Parallelisierung genutzt und das dreidimensionale Simulationsvolumen wird dazu **blockweise** auf die Prozessoren verteilt. Die y-Dimension ist die, die aufgeteilt wird, während die x- und z-Dimensionen bei der ursprünglichen Größe bleiben (Abb. 5.4). So ist das Teilvolumen jedes Prozessors, anders als bei der zyklischen Zerlegung, zusammenhängend. Das bietet einen erheblichen Vorteil bei der Erzeugung der Nervenfasern, da die auf Seite 23 beschriebenen *Ghostbereiche* jedes Blockes nur einmal für jeden Prozessor berücksichtigt werden müssen. Außerdem ist die parallele Ausgabe der Ergebnisse mit HDF5 einfacher mit einem zusammenhängendem Datenblock, in dem die Daten fortlaufend abgespeichert sind (siehe S. 53).

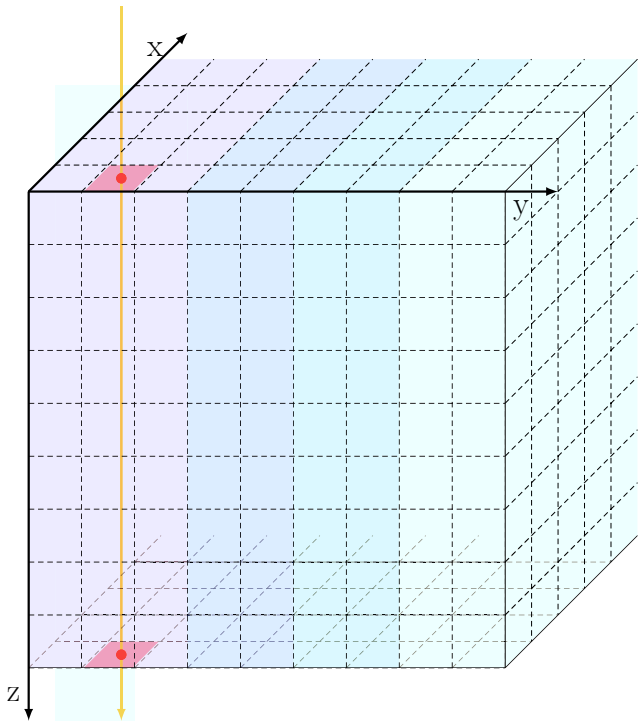


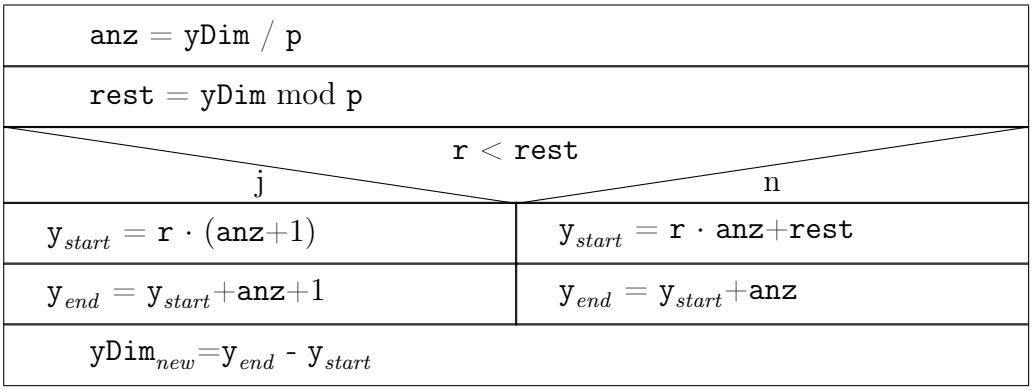
Abbildung 5.4.: Gebietszerlegung des Simulationsvolumens der Größe 9×9×9 auf 4 Prozessoren

Da die Größe des Simulationsvolumens und die Anzahl der Prozessoren bei Start des Programms bekannt sind, kann die Start- und Endposition des Teilvolumens von jedem Prozess unabhängig von den anderen berechnet werden. Eine Master-Slave Strategie ist nicht sinnvoll, weil dadurch unnötiger Kommunikationsaufwand entsteht.
Es wurde ein **Distributed Memory** Modell gewählt, um zu vermeiden, dass jeder Prozess das gesamte Simulationsvolumen speichern muss, was bei der Größe der Daten

nicht umzusetzen wäre. Jeder Prozessor speichert nur das Teilvolumen, in dem er die Fasern erzeugt und die Ergebniswerte simuliert. Da die Berechnungen unabhängig voneinander sind, ist auch keine Kommunikation zwischen den Prozessoren nötig.
Bei der Parallelisierung in Kombination mit der Kippung der Faserverläufe muss auf Verschiedenes geachtet werden. Daher wird zunächst der Fall ohne Kippung betrachtet.

Parallelisierung ohne Kippung

Im Folgenden bezeichnet r die Nummer eines Prozessors von insgesamt p Prozessoren. Die Aufteilung der y-Dimension *yDim* mit Angabe von Start- und Endzeile lässt sich folgendermaßen umsetzen:



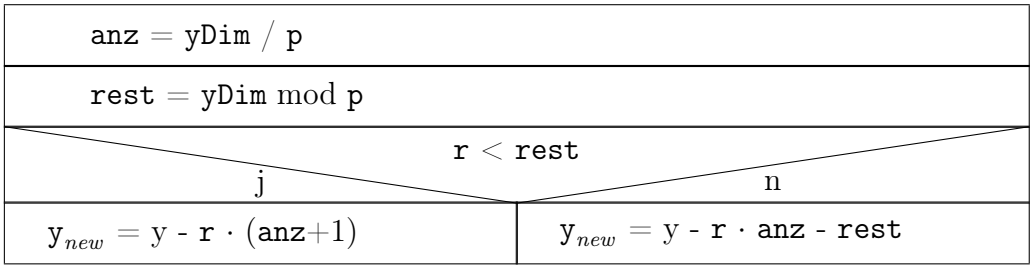
Nassi 5.1: Bestimmung der Start- und Endposition der Gebietszerlegung in y-Dimension

Prozessornummer	y_{start}	y_{end}	Teilgröße
0	$0 \cdot (2+1) = 0$	$0+2+1 = 3$	3
1	$1 \cdot 2+1 = 3$	$3+2 = 5$	2
2	$2 \cdot 2+1 = 5$	$5+2 = 7$	2
3	$3 \cdot 2+1 = 7$	$7+2 = 9$	2

Tabelle 5.2.: Blockweise Zerlegung des Simulationsvolumens der Größe 9×9×9 Pixel (Abbildung 5.4) auf 4 Prozessoren. Für die y-Dimension wird Start- und Endzeile berechnet, die anderen beiden Dimensionen bleiben bei der ursprünglichen Größe. Die Startzeile gehört immer zum aktuellen Prozessor, die Endzeile ist die Startzeile des nächsten Prozessors.

Die eigenen Grenzen sind jedem Prozessor bekannt und somit legt jeder Prozessor intern die drei benötigten Arrays zur Datenspeicherung nicht in der Größe des gesamten Simulationsvolumens, sondern nur in der Größe seines Teilvolumens an. Somit können durch die Erhöhung der Prozessoranzahl bei gleichbleibendem Hauptspeicher pro

Prozessor größere Datensätze simuliert werden (*weak scaling*).
Bei der parallelen Fasererzeugung sind somit nur geringe Veränderungen im Vergleich zum seriellen Vorgehen nötig. Es muss lediglich darauf geachtet werden, dass die Identifikationswerte für Myelin und Axon und die zu den Myelinvoxeln gehörenden Vektoren in den beiden Arrays an der richtigen Position gespeichert werden. Über die Nummer des aktuellen Prozessors kann die Speicheradresse bezogen auf das Gesamtvolumen auf die Speicheradresse des Teilvolumens umgerechnet werden. Abhängig davon, ob sich das Gesamtvolumen gleichmäßig auf die Prozesse aufteilen lässt oder nicht, erfolgt, wie in Nassi 5.2 beschrieben, die Umrechnung der Speicherposition `y`. Ein Zahlenbeispiel dazu ist in Tabelle 5.3 aufgeführt.



Nassi 5.2: Bestimmung der neuen Speicherposition im Teilvolumen

	Prozessor-nummer	x	y	z
Speicherposition im Gesamtvolumen	3	175	248	35
Speicherposition im Teilvolumen	3	175	248-3 · (256/4) = 56	35

Tabelle 5.3.: Umrechnung der Speicherposition bei einem Gesamtvolumen von 256×256×256 Pixeln und 4 Prozessoren (nummeriert von 0-3)

Die Simulation der 3D-PLI Bildfolge durch den Matrix Calculus ist ebenfalls sehr einfach zu parallelisieren. Jeder Prozessor führt die Matrixmultiplikationen in Richtung des Lichtstrahls nur im eigenen Teilvolumen durch, welches zuvor mit Faserwerten gefüllt wurde. Lediglich die Rasterung des Lichtstrahls durch das Simulationsvolumen muss von allen Prozessoren einzeln berechnet werden.

Parallelisierung mit Kippung

Bei der Parallelisierung des gekippten Gehirngewebes verläuft der Lichtstrahl nicht mehr in z-Richtung, sondern schräg durch das Simulationsvolumen. Je nach Wahl der Kippwinkel kann der Matrix Calculus daher an ein oder zwei Randgebieten der Teilvolumina der Prozessoren nicht ausgeführt werden. In diesen Randbereichen liegt das Ende des Lichtstrahls bereits im benachbarten Teilvolumen (Abb. 5.5).

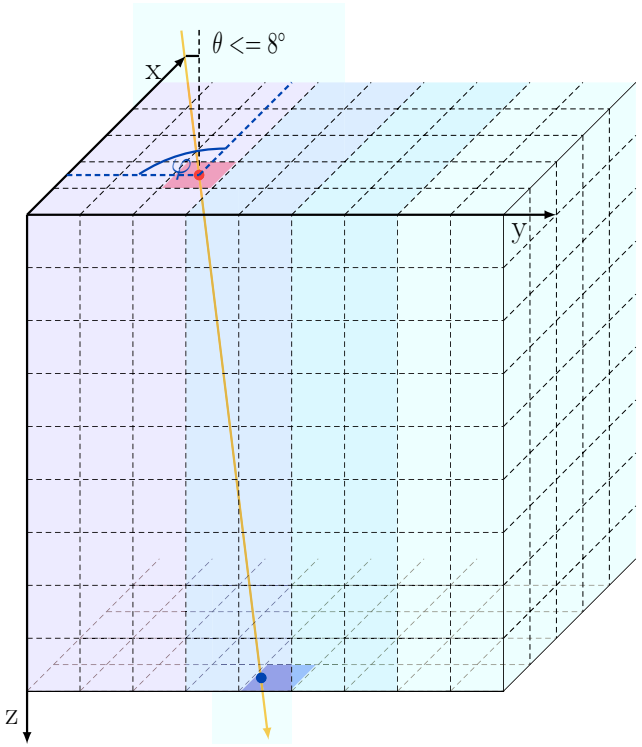


Abbildung 5.5.: Je nach Wahl der Kippwinkel befindet sich der Austrittspunkt des Lichtstrahls nicht mehr im Teilvolumen des Prozessors, in dem der Strahl eingetreten ist. Die Teilbereiche müssen daher überlappend definiert werden.

Die Lösung dieses Problems besteht darin, die Teilvolumina der einzelnen Prozessoren überlappend zu definieren. Die Verschiebung in y-Richtung zwischen Ein- und Austrittspunkt des Lichtstrahls in das Simulationsvolumen kann beim Start des Programms berechnet werden. Die eigentlich berechnete Größe des Teilvolumens wird nun, in Abhängigkeit von der Kipprichtung, um die Differenz zwischen Ein- und Austrittspunkt vergrößert. Dadurch können alle Punkte der xy-Ebene des Teilvolumens im Matrix Calculus vollständig simuliert werden. Der Verlauf der Nervenfasern und die zur Myelinschicht gehörenden Vektoren werden auf dem vergrößerten Teilvolumen erzeugt. Auch beim gekippten Faserverlauf müssen die Speicherpositionen im Teilvolumen, wie zuvor beschrieben, korrekt aus den Speicheradressen im Gesamtvolumen

berechnet werden.
Durch diese Vergrößerung ist anschließend der Matrix Calculus auf den Teilvolumina ohne weiteres durchführbar. Lediglich am Rand des Gesamtvolumens ist die Berechnung der gekippten PLI-Bilder nicht möglich, da der Lichtstrahl dort aufgrund der schrägen Lage nicht vollständig durch das Simulationsvolumen verläuft und die Ergebniswerte am Rand somit verfälscht sind. Diese nicht berechenbaren Randwerte werden im Ergebnisbild in einem einheitlichen Farbton gekennzeichnet.

Parallele Ausgabe

Die Daten liegen bei der Ausgabe auf allen Prozessoren verteilt und müssen parallel in die Ergebnisdatei an die richtige Stelle geschrieben werden. Der Aufbau der Ausgabedatei wurde bereits in Kapitel 4.5.1 beschrieben.
In HDF5 können verschiedene Unterbereiche, sogenannte **Hyperslabs**, eines Datensatzes angewählt werden. Sie legen fest, an welche Stelle Daten geschrieben werden oder von wo Daten gelesen werden. Es können verschiedene Parameter angegeben werden [15]:

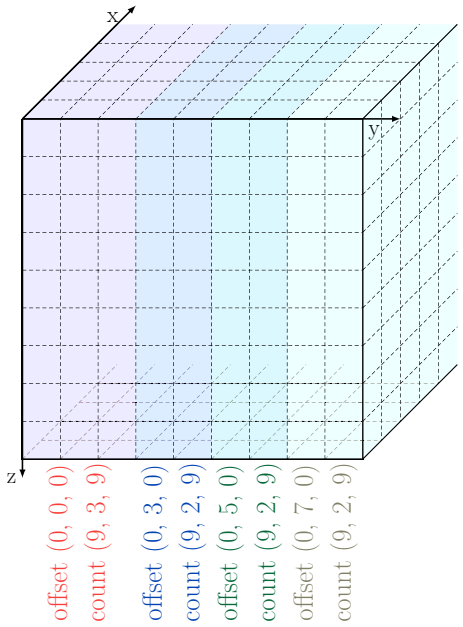


Abbildung 5.6.: Werte für Offset und Count für die parallele Ausgabe eines Simulationsvolumens der Größe 9×9×9 Pixel mit 4 Prozessen

- **Offset:** Verschiebung des Startpunktes zum Schreiben in jeder Dimension.
- **Count:** Anzahl der zu schreibenden Blöcke in jeder Dimension. Ist der Parameter Block für jede Dimension 1, so bestimmt der Count die Anzahl der Elemente in jeder Dimension.

- **Stride:** Anzahl pro Dimension nach der ein weiterer Block geschrieben wird.
- **Block:** Anzahl der Elemente einer Dimension eines Datenblocks. Ist die Block-Variable 1 oder NULL, so besteht der Block in dieser Dimension aus einem einzigen Element.

Bei der parallelen Datenausgabe des Simulationsprogramms *simPLI* müssen lediglich die Parameter Offset und Count für jeden Prozessor spezifiziert werden (Abb. 5.6), da der Datenblock jedes Prozessors bereits zusammenhängend ist. Die Parameter können über die obere und untere Grenze jeder Dimension und die Verschiebung in y-Richtung durch die Kippung berechnet werden. Bei der Ausgabe des gekippten Faserverlaufs muss außerdem darauf geachtet werden, dass die überlappenden Speicherbereiche der Prozessoren richtig zusammengesetzt werden.

5.5. Laufzeitverhalten

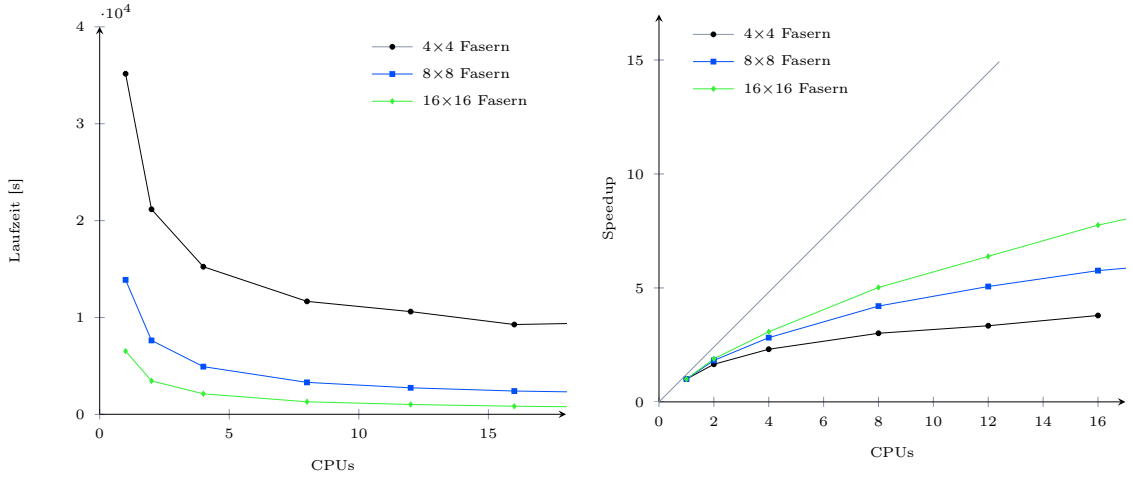
Die Laufzeit eines Programmlaufs hängt von verschiedenen Faktoren ab, die im Folgenden kurz erläutert werden:

- **Größe des Simulationsvolumens:** Je größer das Simulationsvolumen, desto länger dauert auch die Erzeugung der Fasern und die Simulation der 3D-PLI Bilder. Wird beispielsweise die Größe eines Pixels in jeder Dimension halbiert, bedeutet dies eine Verachtfachung des Volumens.
- **Anteil des gefüllten Simulationsvolumens:** Die Laufzeit hängt bei großen Volumina stark davon ab wie viel Prozent des gesamten Volumens mit Fasern gefüllt sind.
- **Äußerer Radius der Fasern:** Je größer das Verhältnis von äußerem Faserradius zur Größe des Simulationsvolumens ist, desto schlechter ist dies für die Laufzeit des Programms. Das Simulationsvolumen wird rundherum um einen *Ghostbereich*, der die Größe des äußeren Radius hat, erweitert. Je größer der Faserradius, desto schlechter ist somit die Laufzeit.
- **Kippung:** Ein Simulationslauf mit gekippten Gehirnfasern dauert länger als der Ungekippte, da das ungekippte Bild ebenfalls zu Vergleichszwecken simuliert wird und die Bestimmung des Lichtstrahlverlaufs durch das Gewebe komplizierter ist.

Zunächst wurde die Laufzeit eines komplett gefüllten Simulationsvolumens der Größe $1024 \times 1024 \times 1024$ Pixel ohne Kippung analysiert. Das Volumen ist mit einem geraden, horizontal verlaufenden Faserbündel gefüllt (Abb. 5.7). Die Anzahl der Fasern wurde variiert, somit kann die Laufzeit mit verschiedenen Faserradien verglichen werden.
Alle Beispiele wurden mit 1, 2, 4, 8, 12 und 16 Prozessoren gerechnet und die verschiedenen Laufzeiten verglichen. Es wurde zwischen der Laufzeit der Fasererzeugung und der Simulation der 3D-PLI Bilder (Matrix Calculus) unterschieden. Der Ein- und Ausgabevorgang ist mit ungefähr einer Sekunde Laufzeit zu vernachlässigen.

500 500

Abbildung 5.7.: Komplett gefülltes Simulationsvolumen mit einem hexagonalen, geraden Faserbündel bestehend aus 4×4 Fasern



(a) Laufzeit der Fasererzeugung

(b) Speedup der Fasererzeugung

Abbildung 5.8.: Laufzeitanalyse der Fasererzeugung eines 1024×1024×1024 Pixel großen, komplett gefüllten Simulationsvolumens mit unterschiedlicher Anzahl an horizontalen, geraden Fasern (hexagonales Faserbündel) ohne Kippung

Fasererzeugung

Am Beispiel des Faserbündels mit unterschiedlicher Anzahl an Fasern sieht man deutlich, dass die Laufzeit der Fasererzeugung von der Dicke der Fasern abhängig ist. Das Simulationsvolumen ist in jedem Fall komplett von Fasern gefüllt, nur mit einer unterschiedlichen Anzahl an Fasern mit verschiedenen Radien. Der innere Radius beträgt immer die Hälfte des Äußeren.
Je kleiner der Faserradius, desto kleiner ist auch der *Ghostbereich*, der das Simulati-

onsvolumen rundherum erweitert und dem äußeren Radius entspricht. Der Nachteil bei diesem Vorgehen ist, dass bei Erhöhung der Prozessoranzahl jedes Teilvolumen um diesen *Ghostbereich* einer festen Größe erweitert wird. Ein kleines Verhältnis von Radius zu Simulationsvolumen ist daher entscheidend für eine geringe Laufzeit.
Das Verhältnis der Breite des Teilvolumens in y-Dimension bei einer Verdopplung der Prozessoranzahl kann über Formel 5.1 berechnet werden. Dabei beschreibt p die Prozessoranzahl, *yDim* die Größe des Simulationsvolumens in y-Richtung und r die Breite des *Ghostbereichs*.

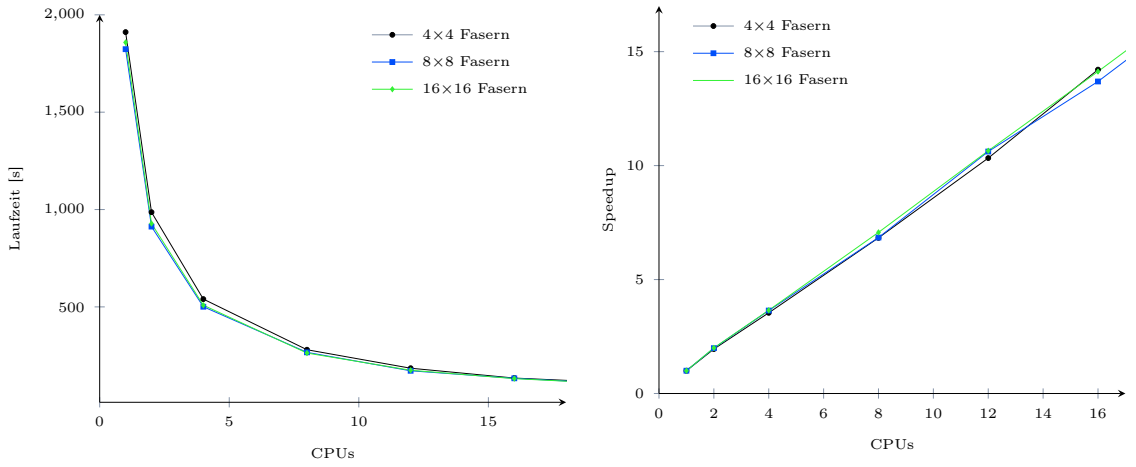
$$\frac{Breite_{2p}}{Breite_p} = \frac{\frac{yDim}{2p} + 2 \cdot r}{\frac{yDim}{p} + 2 \cdot r} \tag{5.1}$$

Dieses Verhältnis ist in jedem Fall größer als 0.5, was man bei einer Halbierung des Volumens zunächst erwarten würde und erklärt teilweise die nicht ideal verlaufende Laufzeitkurve der Fasererzeugung. Das Verhältnis aus Formel 5.1 wird mit Erhöhung der Prozessoranzahl immer größer, da der fixe Anteil $2 \cdot r$ immer mehr Gewicht bekommt.
Außerdem tastet jeder Prozessor die Faserfunktion immer im gesamten Definitionsbereich ab, d.h. auch außerhalb seines Teilvolumens. Da aber außerhalb des Teilvolumens lediglich eine Funktionsauswertung stattfindet, um zu überprüfen, ob die Faser außer- oder innerhalb des eigenen Volumens liegt, geht die Abtastung schnell im Vergleich zum Füllen der Voxel mit Werten. Natürlich hängt die Länge der Auswertung auch von der Komplexität der Faserfunktion ab.
Obwohl der Speedup der Fasererzeugung nicht ideal ist, liefert die Berechnung mit mehreren Prozessoren doch eine deutliche Reduzierung der Laufzeit.

Simulation

Die Laufzeit der Simulation ist nicht abhängig von der Faserbreite, sondern nur von der Anzahl der Myelinvoxel entlang des Lichtstrahlverlaufs (Abb. 5.9). Da in allen drei Beispielen das Simulationsvolumen komplett mit Fasern gefüllt ist und der innere Radius immer der Hälfte des Äußeren entspricht, ist die Anzahl der Myelinvoxel immer gleich. Außerdem ist der Speedup der Simulation deutlich besser als der Speedup der Fasererzeugung, da die Teilbereiche, auf denen der Matrix Calculus ausgeführt wird, auf die Prozessoren aufgeteilt wird. Der Verlauf des Lichtstrahls durch das Gewebe muss jedoch von jedem Prozessor separat bestimmt werden. In Abbildung 5.10 sind die Laufzeitanteile der Fasererzeugung, der Simulation und der Ein- und Ausgabe der Daten an der Gesamtlaufzeit dargestellt. Die Laufzeiten beziehen sich auf das komplett gefüllte, hexagonale Faserbündel bestehend aus 8×8 Fasern und einer Simulationsgröße von 1024×1024×1024 Pixeln.
Die Abbildung zeigt, dass ein Großteil der Laufzeit für die Erzeugung der Nervenfasern benötigt wird. Die Zeit für die Ein- und Ausgabe der Daten kann gegenüber der hohen restlichen Laufzeit vernachlässigt werden. Aus Abbildung 5.8a geht hervor, dass

die Laufzeit von dem Verhältnis der Faserdicke zur Größe des Simulationsvolumens abhängt. Daher ist der Laufzeitanteil der Fasererzeugung bei dem 4×4 Faserbündel noch größer, beim 16×16 Faserbündel jedoch kleiner als in Diagramm 5.10.



(a) Laufzeit der Simulation

(b) Speedup der Simulation

Abbildung 5.9.: Laufzeitanalyse der Simulation eines 1024×1024×1024 Pixel großen, komplett gefüllten Simulationsvolumens mit verschiedener Anzahl von horizontalen, geraden Fasern ohne Kippung

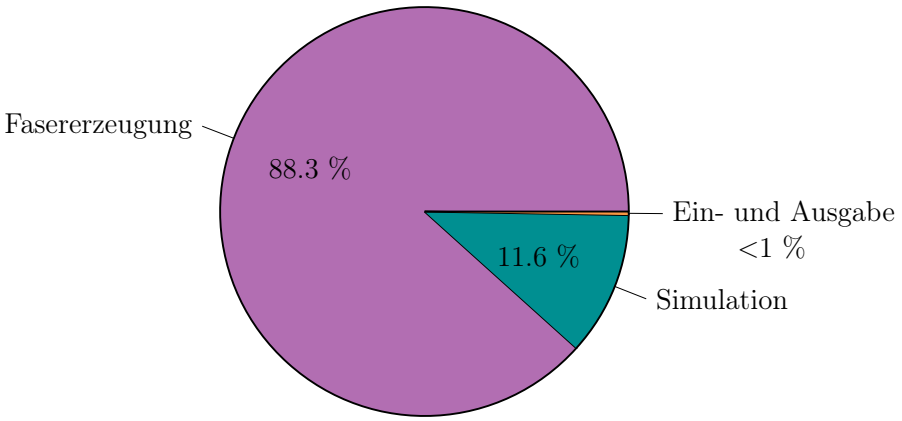


Abbildung 5.10.: Prozentuale Anteile der Laufzeit der Fasererzeugung, der Simulation und der Ein-/Ausgabe der Daten an der Gesamtlaufzeit. Simuliert wurde das hexagonale Faserbündel mit 8×8 Fasern bei einer Größe von 1024×1024×1024 Pixel auf einem Prozessor.

Laufzeitvergleich gekippte und ungekippte Simulation

Abschließend wird kurz die Laufzeit eines gekippten Volumens mit der Standartsimulation verglichen. Dazu wurde ein Volumen mit der Größe von 1024×1024×1024 Pixeln mit vier geraden, horizontalen Fasern definiert (Abb. 5.11). Der äußere Radius beträgt 60 Pixel, der innere Radius die Hälfte davon. Der Kippwinkel θ hat den maximal möglichen Wert 8°, der Winkel φ beträgt 20°.

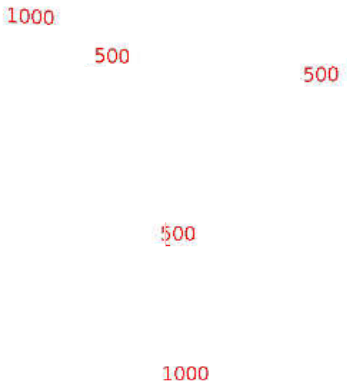
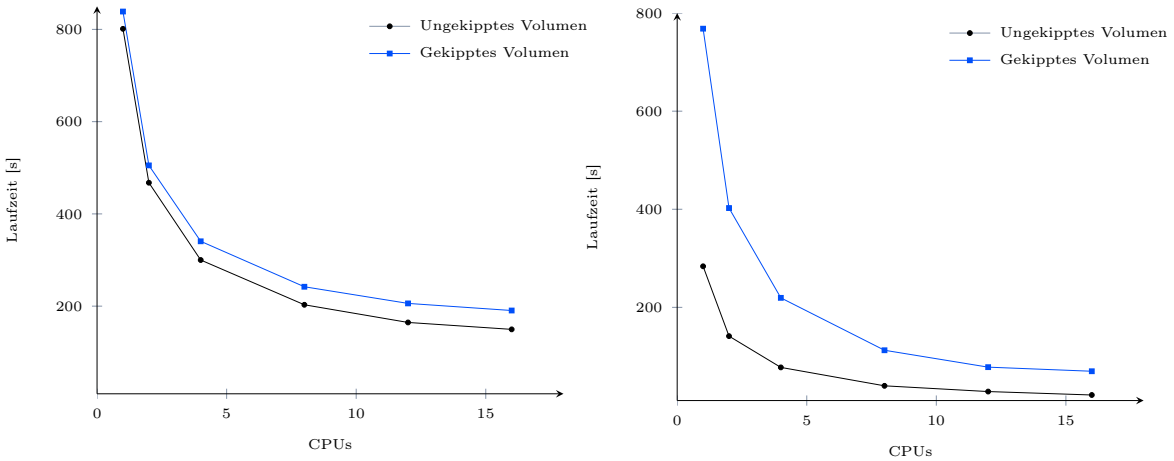


Abbildung 5.11.: 4 gerade horizontale Fasern für die die gekippte und ungekippte Simulation verglichen wurde.



(a) Laufzeiten der Fasererzeugung

(b) Laufzeiten der Simulation

Abbildung 5.12.: Vergleich der Laufzeiten der Fasererzeugung und der Simulation von 4 geraden, horizontalen Fasern im gekippten und im ungekippten Fall. Das Simulationsvolumen hat eine Größe von 1024×1024×1024 Pixel.

Bei diesem Beispiel fällt auf, dass die Erzeugung der Nervenfasern bei der gekippten Simulation bei jeder Prozessoranzahl ungefähr 40s länger dauert als im ungekippten Fall (Abb. 5.12a). Dies ist dadurch zu erklären, dass das Teilvolumen jedes Prozessors vergrößert werden muss, um alle Werte des gekippten Ergebnisbildes beim Matrix Calculus berechnen zu können. Die Zeit, die die gekippte Fasererzeugung länger dauert, hängt von der Wahl der Kippwinkel ab, die für die Größe des Überlappbereichs verantwortlich sind.
Die gekippte Simulation der 3D-PLI Bilder dauert deutlich länger, da die Bildfolge sowohl für den gekippten als auch für den ungekippten Fall zum Vergleich erzeugt wird. Zudem ist die Simulation des gekippten Volumens etwas aufwändiger, da die Erzeugung des Lichtverlaufs und die zusätzliche Interpolation der Myelinvektoren mehr Zeit in Anspruch nimmt.

6. Zusammenfassung und Ausblick

In Rahmen dieser Arbeit wurde das Programm *simPLI* entwickelt, das seinen gleichnamigen Vorgänger ersetzt [11]. Im Gegensatz zum Vorgängermodell, mit dem nur eine beschränkte Anzahl von vordefinierten Faserszenarien gerechnet werden konnte, kann der Benutzer beliebige Fasern über mathematische Funktionen für das neu entwickelte Programm definieren, die mit einer flexiblen Schrittweitensteuerung erzeugt werden. Weiterhin wurde die Simulation der Gewebekippung implementiert, die zuvor noch nicht vorhanden war. Da das Vorgängerprogramm nur Simulationsdaten einer begrenzten Größe rechnen konnte, stand fest, dass das neu entwickelte Programm parallelisiert werden sollte.
Die Software *simPLI* simuliert zu einem zuvor definierten Faserverlauf eine Folge von 3D-PLI Aufnahmen. 3D-PLI Bilder sind durch die Technik *Three-dimensional Polarized Light Imaging* aufgenommene Bilder von Gehirnschnitten. Durch den PLI-Rekonstruktionsworkflow können Richtungsinformationen der Nervenfasern berechnet werden und somit ist es möglich, ein räumliches Modell des Gehirns zu rekonstruieren (Kapitel 2). Mit Hilfe von *simPLI* können Phantome von 3D-PLI Ergebnisbildern erstellt werden, die der Verifizierung des Workflows und der neuroanatomischen Interpretation der realen Daten dienen.
Die Nervenfasern sind von einer Myelinschicht umgeben, die doppelbrechende Eigenschaften hat. Diese Doppelbrechung hat eine zentrale Bedeutung bei der Aufnahme der 3D-PLI Bilder und somit auch für die Simulation der Ergebnisbilder. Im Jones Matrix Formalismus werden die optischen Elemente des 3D-PLI Versuchsaufbaus durch Matrizen dargestellt (Kapitel 3). Diese werden für die Simulation der Ergebnisbilder entlang des Lichtstrahlverlaufs miteinander multipliziert.
Die Fasern werden vom Benutzer als mathematische Funktionen definiert und auf einem dreidimensionalen diskreten Gitter erzeugt. Um die Funktionen möglichst gut abbilden zu können, wurde eine flexible Schrittweitensteuerung entwickelt. Um die doppelbrechende Eigenschaft der Myelinschicht abbilden zu können, wird ein Vektorfeld erzeugt, das anschließend im Jones Matrix Formalismus für die Simulation der 3D-PLI Messungen benötigt wird (Kapitel 4).
Weiterhin ist es möglich die im Versuchsaufbau fixierte Probenplatte mit dem Gehirnschnitt um einen Winkel von bis zu 8° in eine beliebige Richtung zu kippen. Die optionale Kippung der definierten Fasern kann mit *simPLI* ebenfalls simuliert werden.
Je nach Größe des definierten Simulationsvolumens können die zu speichernden Ergebnisdaten schnell eine Größe von mehreren Gigabyte erreichen. Um größere Daten simulieren zu können, wurde das Gesamtvolumen mit einen *Distributed Memory* Ansatz parallelisiert (Kapitel 5). Das bedeutet jeder Prozessor erzeugt und simuliert nur einen Teil der Daten und gibt diese anschließend über das speziell für große Daten-

mengen entwickelte Format HDF5 aus. Durch die Parallelisierung konnte neben dem Hauptziel, der Simulation größerer Daten, ebenfalls die Laufzeit des Programms verringert werden.

Bisher wurde nur die starke Doppelbrechung der Myelinschicht simuliert. Das Faserinnere, Axon genannt, besitzt aber ebenfalls schwache doppelbrechende Eigenschaften. Daher könnte das Programm in Zukunft erweitert werden, indem im Axon ebenfalls radiale oder axiale Vektoren erzeugt werden. Im Matrix Calculus müsste dann für jedes Axon Voxel ebenfalls eine Matrix erstellt und multipliziert werden. Allerdings dürfte bei der Vektorinterpolation im Fall der Kippung nur über Vektoren des gleiches Gewebetyps gemittelt werden, da die Vektoren bei unterschiedlicher Orientierung ansonsten fehlerhaft sind.
Weiterhin könnte ein komplexer Brechungsindex ($n' = n + i\kappa$) für alle möglichen Werte (Hintergrund, Myelin und Axon) und ein komplexer Parameter $\Delta n'$ aus Formel 3.11 (S. 17) jeweils für Myelin und Axon angegeben werden. Dadurch wird neben der Änderung der Wellenlänge beim Eintritt in ein doppelbrechendes Medium auch die Absorption innerhalb des Mediums berücksichtigt. Die beiden Diagonaleinträge der Matrix 3.5 (S. 14) müssten dann folgendermaßen ersetzt werden:

$$
\begin{aligned}
& e^{i\,\frac{2\pi}{\lambda}\,t\,(n'+/-\Delta n')\cos^2(\alpha)/2} \\
& \text{mit} \quad n' = n + i\kappa \\
\text{und} \quad & \Delta n' = \Delta n + i\Delta\kappa
\end{aligned}
$$

Diese Formel würde für Myelin, Axon und Hintergrund gelten, wobei die entsprechenden Werte für n', $\Delta n'$ und α gewählt werden müssen. Dadurch würden weit mehr Matrixmultiplikationen beim Matrix Calculus zustande kommen.
Sollten sich zwei Fasern schneiden, wird bisher eine Fehlermeldung ausgegeben und das Programm beendet. Die bisherigen Ergebnisse der Fasererzeugung werden allerdings dennoch in die Ergebnisdatei geschrieben, um zu sehen, an welcher Stelle es zu einer Kollision gekommen ist. Eine weitere mögliche Erweiterung wäre es, die Fasern bei einer Kollision umeinander wachsen zu lassen. So könnte dennoch eine Simulation ausgeführt werden und die unter Umständen bereits fortgeschrittene und zeitaufwändige Fasererzeugung war nicht umsonst.

A. Beispiele zur Eingabe

A.1. Konfigurationsdatei

Die Eingabedatei muss vor Programmstart vom Benutzer erzeugt werden und der Dateipfad wird relativ zur Lage des ausführbaren Programms beim Aufruf als Kommandozeilenparameter an erster Stelle übergeben. Als zweiter Übergabeparameter folgt der relative Pfad zur Ausgabedatei. Die Daten aus der Konfigurationsdatei werden eingelesen, auf Korrektheit überprüft und in die richtigen Datentypen konvertiert. Bei fehlenden Werten oder falschen Datentypen wird eine entsprechende Fehlermeldung ausgegeben und das Programm beendet.
In Tabelle A.1 sind alle Eingabeparameter und deren Bedeutung aufgelistet. Die Bezeichnungen der ersten Spalte müssen in beliebiger Reihenfolge in der übergebenen Eingabedatei angegeben sein, gefolgt von der korrekten Anzahl an Werten vom richtigen Datentyp. Die Zeile *Fiberdefinition* darf mehrmals angegeben werden, alle anderen nur einmal. Sollen mehrere Fasern im gleichen Volumen simuliert werden, so spezifiziert jede *Fiberdefinition* eine Faser. Es ist ausreichend, wenn von den drei Parametern *pixelNumber*, *pixelSize* und *simulationSize* zwei angegeben werden, da der Fehlende aus den anderen beiden berechnet werden kann. Sollte ein Parameter nicht vorhanden sein, wird er entsprechend ergänzt, sind alle drei angegeben, müssen die Werte fehlerfrei sein und zusammenpassen. Es folgt ein Beispiel für eine Eingabedatei mit zwei definierten Fasern.

```
# CONFIGURATION FILE
#
# in pixels
#pixelNumber 512 512 512
#
# in mm
pixelSize 0.0005 0.0005 0.0005
#
# in mm
simulationSize 0.064 0.064 0.07
#
# Filename  Functionname Fiberradius outer/inner Definitionarea
(tmin, tmax)
Fiberdefinition function_singleFiber fiberfunction1 0.0075 0.005
```

```
-0.01 0.07
Fiberdefinition function_singleFiber fiberfunction2 0.006 0.003
-0.01 0.07
#
# vector orientation (radial(-1)/axial(1))
vectorOrientation -1
# in mm
lambda 0.000525
# in gray values
incidentLight 15700
# properties birefrigence
deltaN 0.001875
# Tilting angels in degrees(theta=0 -> no tilting)
phi 0
theta 0
```

Tabelle A.1.: Eingabeparameter, deren Bedeutung und Datentypen

Bezeichnung in der Eingabedatei	Bedeutung	Datentyp
pixelNumber	Größe des Bildes in Pixeln jeweils in x-, y- und z-Richtung	3 Integer
pixelSize	Physikalische Größe eines Pixels in mm jeweils in x-, y- und z-Richtung	3 Double
simulationSize	Physikalische Größe des Simulationsvolumens in mm jeweils in x-, y- und z-Richtung	3 Double
Fiberdefinition	Definition einer Faserfunktion durch: Dateipfad zur Python-Datei, Name der darin enthaltenen Funktion, innerer und äußerer Faserradius in mm, Minimum und Maximum des Definitionsbereichs	2 Strings, 4 Double
VectorOrientation	Richtung des Vektorfelds der Myelinschicht: 1 für axiale Ausrichtung, -1 für radiale Ausrichtung	1 Integer (1 oder -1)
lambda	Wellenlänge des einfallendes Lichts in mm	1 Double
incidentLight	Intensitätswert des einfallenden Lichts angegeben in Grauwerten	1 Double
deltaN	Lokale Doppelbrechung	1 Double
theta	Winkel zur Kippung des Gehirngewebes in Grad	1 Double (zw. 0 und 8)
phi	Winkel zur Kippung des Gehirngewebes in Grad	1 Double (zw. 0 und 360)

A.2. Python-Datei zum Funktionsimport

An die definierte Python-Funktion wird ein Parameter *t_value* übergeben. Für die x-, y- und z-Komponente wird eine beliebige mathematische Funktion definiert, die an der übergebenen Stelle ausgewertet wird. Die drei Ergebniswerte werden in einer Liste zurückgegeben. Die Funktionen müssen relativ zur physikalischen Größe des Simulationsvolumens angegeben werden. Passend zur Faserfunktion muss ebenfalls der Definitionsbereich in der Konfigurationsdatei gewählt werden.
Im Folgenden ist ein einfaches Beispiel einer Python-Funktion zur Definition einer Geraden im Dreidimensionalen abgebildet. Über trigonometrische Funktionen können ebenfalls Kurven oder Spiralen definiert werden.

```
def fiberfunction(t_value):
    x_value = 0.02+t_value;
    y_value = 0.0315;
    z_value = 0.01-0.05*t_value;
    results = [ x_value,  y_value,  z_value];
    return results;
```

B. Konfiguration JURECA

Die folgende Konfiguration des Supercomputers JURECA (Juelich Research on Exascale Cluster Architectures) wurde für die Simulation genutzt [16]:

Hardware-Eigenschaften:

- 1872 Rechenknoten
  - Pro Knoten: Zwei Intel Xeon E5-2680 v3 Haswell 12-Kern Prozessoren (2.5 GHz)
  - 1605 Knoten mit 128 GiB Hauptspeicher, 128 Knoten mit 256 GiB Hauptspeicher, 64 Knoten mit 512 GiB Hauptspeicher
  - 75 GPU Knoten: Zwei NVIDIA K80 GPUs, 2 × 4992 CUDA Kerne, 2 × 24 GiB GDDR5 Hauptspeicher
- 12 Visualisierungsknoten
  - Zwei Intel Xeon E5-2680 v3 Haswell CPUs pro Knoten
  - Zwei NVIDIA K40 GPUs pro Knoten, 2 x 12 GiB GDDR5 Hauptspeicher
  - 10 Knoten 512 GiB Hauptspeicher, 2 Knoten mit 1024 GiB Hauptspeicher
- Login Knoten mit 256 GiB Speicher pro Knoten
- 45216 CPU Kerne
- 1.8 (CPU) + 0.44 (GPU) Petaflop pro Sekunde Peak Performance

Literaturverzeichnis

[1] https://www.humanbrainproject.eu/2016-overview. Letzer Zugriff: 29.07.2016, 2016.

[2] Menzel M, Michielsen K, De Raedt H, Reckfort J, Amunts K, and Axer M. A Jones matrix formalism for simulating three-dimensional polarized light imaging of brain tissue. *J. R. Soc. Interface 12: 20150734*, 2015.

[3] Markus Axer, Katrin Amunts, David Grässel, Christoph Palm, Jürgen Dammers, Hubertus Axer, Uwe Pietrzyk, and Karl Zilles. A novel approach to the human connectome: Ultra-high resolution mapping of fiber tracts in the brain. *NeuroImage*, 54:1091–1101, 2011.

[4] Jürgen Dammers, Markus Axer, David Gräßel, Christoph Palm, Karl Zilles, Katrin Amunts, and Uwe Pietrzyk. Signal enhancement in polarized light imaging by means of independent component analysis. *NeuroImage*, 49:1241–1248, 2010.

[5] Andreas Müller. Optimierung und Parallelisierung der Inklinationsberechnung von Nervenfaser im PLI-Rekonstruktionsworkflow. 2014.

[6] Eugene Hecht. *Optik*, volume 5. 2009.

[7] Wolfgang Demtröder. *Experimentalphysik 2 - Elektrizität und Optik*, volume 6. Springer Spektrum, 2013.

[8] Miriam Menzel. Simulation and Modeling for the Reconstruction of Nerve Fibers in the Brain by 3D Polarized Light Imaging. Master's thesis, August 2014.

[9] R.C. Jones. A new calculus for the tratment of optical systems. *Journal of the Optical Society of America*, 1941.

[10] Edward Collet. *Field Guide to Polarization*. SPIE Press, 2005.

[11] Melanie Dohmen, Miriam Menzel, Hendrik Wiese, Julia Reckfort, Frederike Hanke, Uwe Pietrzyk, Karl Zilles, Katrin Amunts, and Markus Axer. Understanding fiber mixture by simulation in 3D polarized light imaging. *NeuroImage*, 111:464–475, 2015.

[12] https://docs.python.org/2/c-api/index.html#c-api-index. Letzer Zugriff: 29.07.2016, 2016.

[13] https://www.hdfgroup.org/HDF5/doc/H5.intro.html. Letzer Zugriff: 29.07.2016, 2006.

[14] Barry Wilkinson and Michael Allen. *Parallel Programming*. Pearson Education, 2005.

[15] https://www.hdfgroup.org/HDF5/Tutor/selectsimple.html. Letzter Zugriff: 29.07.2016.

[16] https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/Configuration/Configuration_node.html. Letzer Zugriff: 29.07.2016, 2016.

Jül-4395
September 2016
ISSN 0944-2952