Fachbereich 08
Institut für Informatik
Arbeitsgruppe Computational Geometry

Arbeit zur Erlangung des akademischen Grades Bachelor of Science

Optimierung des adiabatischen Lichtleiters für das Wavelength-shifting Optical Module

Ronja Schnur

1. Gutachter Prof. Dr. Elmar Schömer
Institut für Informatik
Johannes Gutenberg-Universität Mainz

2. Gutachter Prof. Dr. Sebastian Böser
Institut für Physik
Johannes Gutenberg-Universität Mainz

Betreuer Prof. Dr. Elmar Schömer und
Prof. Dr. Sebastian Böser

12. Februar 2020

Ronja Schnur

Optimierung des adiabatischen Lichtleiters für das Wavelength-shifting Optical Module

Arbeit zur Erlangung des akademischen Grades Bachelor of Science, 12. Februar 2020

Gutachter: Prof. Dr. Elmar Schömer und Prof. Dr. Sebastian Böser

Betreuer: Prof. Dr. Elmar Schömer und Prof. Dr. Sebastian Böser

Johannes Gutenberg-Universität Mainz

Arbeitsgruppe Computational Geometry

Institut für Informatik

Fachbereich 08

Staudingerweg 9

55128 Mainz

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Mainz, 12. Februar 2020

Ronja Schnur

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Arbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn Prof. Schömer, der meine Arbeit betreut und begutachtet hat. Für den Vorschlag des Themas, die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Ein besonderer Dank gilt allen Mitgliedern der regelmäßigen WOM-Meetings, ohne die diese Arbeit nicht hätte entstehen können, allen voran Herrn Prof. Böser. Mein Dank gilt eurer Informationsbereitschaft, interessanten Beiträgen und Antworten auf meine Fragen.

Ebenfalls möchte ich mich bei M. Sc. Florian Thomas für die hilfreichen Rückmeldungen zu meinen unzähligen Fragen bedanken. Insbesondere da seine gute Vorarbeit diese Arbeit erst ermöglichte.

Außerdem möchte ich mich bei meinen Kommilitonen und Freunden bedanken, die mir mit viel Geduld, Interesse und Hilfsbereitschaft zur Seite standen. Bedanken möchte ich mich für die zahlreichen interessanten Debatten, Ideen und das Korrekturlesen meiner Bachelorarbeit.

Abschließend möchte ich mich bei meiner Famile und meinen Eltern bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht und stets ein offenes Ohr für mich gefunden haben.

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit einer weiterführenden Optimierung des Wavelength-shifting Optical Module (WOM). Dieses besteht aus einem mit Wellenlängenverschiebender Farbe beschichtetem Hohlzylinder aus Quarz, dem adiabatischen Lichtleiter (ALG) und Photomultipliern zusammengefasst in einem Druckgasbehälter. Ziel des Moduls ist die Erkennung von hochenergetischen Neutrinos im Rahmen des IceCube-Gen2 Projekts.
Speziell behandelt wird ein numerisch simulatives Optimierungsverfahren des ALG auf Basis des Nelder-Mead Simplex Algorithmus unter Verwendung verschiedener, teils eigens konzipierter Splineinterpolationen. Als simulative Grundlage dient dabei der WOM Ray-Tracer (WOMRaT), wenn auch nicht in seiner ursprünglichen Form, sondern als neu konzipierte Software (ALGO).
Die Simulationsergebnisse zeigten, dass eine gleichmäßige Kurve mit optimaler Kathodeneffizienz nicht zu existieren scheint. Stattdessen zeigte sich, dass eine Kurve mit gleichmäßiger Krümmung stets eine gute Lösung liefert. Daher ist eine kostengünstige Profilkurve die bessere Wahl für entsprechende Fertigungsmaßnahmen.

Abstract

This thesis deals with the ongoing optimization of the wavelength-shifting optical module (WOM) . It consists of a quartz hollow cylinder colored wirh wavelength-shifting paint, the adiabatic light guide (ALG) and photomultiplier tubes as bundles in a compressed-gas container. The purpose of the module is the detection of high-energy neutrinos in the context of the IceCube-Gen2 project.
In particular, a numeric simulative optimization method is presented wich is based on the Nelder-Mead simplex algorithm and a paritially self-conceived spline interpolation. The WOM Ray-Tracer (WOMRaT) serves as a simulation basis, even though it is not used in its original form, but in a newly developed software called ALGO.
The simulation results show that there is not the one ideal curve with optimal efficiency at the cathode. Instead, it becomed obvious that a curve with consistent curvature always

provides a good solution. Therefore a cost-efficient profile curve constitutes the better choice for manufactoring.

Inhaltsverzeichnis

Abkürzungsverzeichnis/Glossar

Abkürzungen

ALG Adiabatic Light Guide (dt. Adiabatischer Lichtleiter). vii, 3, 4, 7–9, 13, 15–17, 19, 21, 22, 24, 27, 28, 46, 49, 81

ALGO Adiabatic Light Guide Optimizer [19]. 19

API Application Programming Interface (dt. Programmierschnittstelle). 22

CUDA Compute Unified Device Architecture. 18, 22

GPU Graphics Processing Unit (dt. Grafikkarte). ix, 17, 19–21, 23, 24, 27

WOM Wavelength-shifting Optical Module (dt. Wellenlängenverschiebendes optisches Modul). vii, 3, 9, 15–17, 22, 28, 33, 64, 77, 81

WOMRaT Wavelength-shifting Optical Module Ray-Tracer. vii, 4, 7, 19, 24, 25, 49, 75

Glossar

Étendue Die Étendue ist eine Erhaltungsgröße der geometrischen Optik. Innerhalb eines Lichtkonzentrators ist diese, sofern keine Photonen verloren werden, der Erhaltung der Querschnittsfläche äquivalent. [5] . 27

Falkekurve Die Falkekurve bezeichnet die bisher optimal angenomme Profilkurve

$$r(x) = r_0 + \frac{r_0}{\epsilon^2 - 1} - \sqrt{\left(\frac{r_0}{\epsilon^2 - 1}\right)^2 + \frac{x^2}{\epsilon^2 - 1}}$$

aus der Arbeit von Peter Falke [5]. Die numerische Exzentrizität ϵ ist gegeben mit $\epsilon = 1.4502$. r_0 bezeichnet den Startradius für $x = 0$ des ALG . 4, 11, 12, 27–29, 33–49, 59–63, 65–70, 75–77

Parabelkurve Der Begriff Parabelkurve bezeichnet hier die folgende Profilkurve unter Vorgabe des Startradius y_0, Endradius y_e und Länge l

$$r(x) = \frac{(r_e - r_0)}{l^2} \cdot x^2 + r_0$$

mit $r'(0) = 0$. 33–35, 38, 39, 41, 42, 44–46, 59–63, 65–68, 77

1 Einleitung

> *Der Weg ist das Ziel.*
>
> — **Konfuzius**
> 551 - 479 v. Chr.

1.1 Motivation

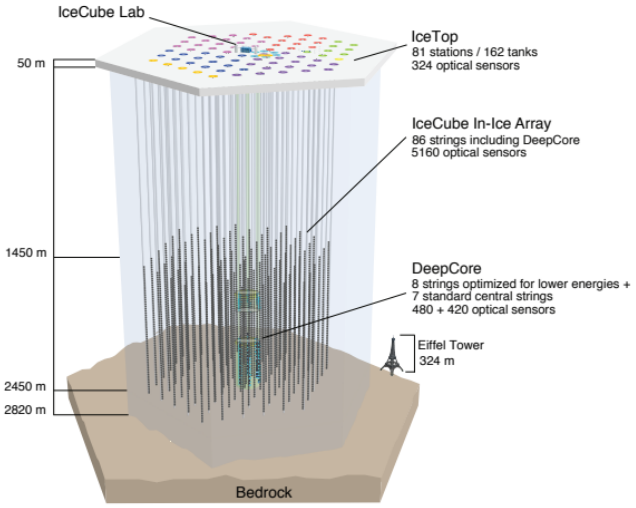
Der adiabatische Lichtleiter (ALG) ist Bestandteil des Wavelength-shifting Optical Module (WOM) [6], einem UV-sensitivem Photonendetektor. Dieses Modul befindet sich bereits seit Jahren in Entwicklung und ist Teil des IceCube-Gen2 Projektes [3], der nächsten Generation des IceCube - einem Hochenergie-Neutrino-Observatorium in der Antarktis (siehe Abbildung 1.1a). Das Ziel von IceCube ist es Neutrinos in einer aktuell 1 km^3 großen dielektrischen Eismasse zu detektieren.

Neutrinos sind elektrisch neutrale Teilchen im Standardmodell der Elementarteilchenphysik (EM), die bis zum heutigen Tage noch in vielen Bereichen unerforscht sind. So ist beispielsweise nicht bekannt, welche Masse diese aufweisen, noch wie diese innerhalb der verschiedenen Neutrinoarten (Elektron, Myon, Tau) anzuordnen ist [4]. Dies ist im Besonderen ein Forschungsbereich des genannten Antarktisobservatoriums.

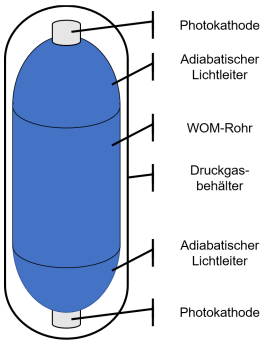
Da Neutrinos nur unter schwacher Wechselwirkung(1) mit Materie interagieren, müssen diese eine hohe Energie mit sich führen, wodurch dies selten zu beobachten ist. In diesen Fällen tritt Tscherenkow-Strahlung auf, welche in Form von UV-Strahlung vom WOM gefangen werden soll. Dazu ist dieser mit einer Wellenlängen-schiebenden Schicht bemalt, um diese Strahlung in das sichtbare Spektrum zu übertragen. Um so wichtiger ist es also, eine maximale Effizienz des Moduls zu erzielen.

Das WOM besteht im Wesentlichen aus einem Hohlzylinder (WOM-Rohr), dem ALG (Adiabatischer Lichtleiter) und einer Photokathode (siehe Abbildung 1.1b). Die Funktion des Zylinders ist dabei das Einfangen von Photonen. Das Ziel des ALG ist, den Radius des Rohres zu verringern, um das Signal-Rausch-Verhältnis der Kathode ausreichend groß zu halten.

(1) Eine der vier Grundkräfte der Physik.



(a) Übersicht über das IceCube-Neutrino Observatorium (Quelle: [1])



(b) Schematische Darstellung des Wavelength-shifting Optical Module

Abb. 1.1.: IceCube Observatorium und WOM

Im Rahmen seiner Masterthesis [21] entwickelte Florian Thomas den WOM-Ray-Tracer (WOMRaT) [22] an der JGU Mainz. Dabei wurde der ALG mit quadratischen Splines in seiner aktuell vermuteten optimalen Form (Falkekurve) [5] approximiert und auf Effizienz[(2)] getestet. Diese liegt im Mittel bei ≈ 33% (siehe Abschnitt 2.3 und Kapitel 4). Die Hauptursache dieser durchaus geringen Effizienz liegt darin, dass ≈ 60% (siehe Kapitel 4) der Photonen sich innerhalb des ALG zurück in das WOM-Rohr bewegen, wodurch im Gesamtdurchlauf die meisten Photonen vom Material absorbiert werden oder durch Streuung verloren gehen.

Ziel dieser Arbeit ist es nun, mittels Variation der Profilkurve des ALG eine effizientere Geometrie zu finden oder die Falkekurve als Optimum zu bestätigen. Dazu wurde der bestehende Quelltext des WOMRaT [22] überarbeitet und an die neu entstandenen Anforderungen angepasst (siehe Kapitel 3).

1.2 Aufbau der Arbeit

Kapitel 2 umfasst sämtliche theoretische Grundlagen und Konzepte, welche im neuen Quelltext verwendet wurden, insbesondere auch die eigens konzipierten Splineinterpolationen. In **Kapitel 3** werden die Modifizierungen der Softwarearchitektur am bestehenden Quelltext erläutert, motiviert und validiert.
Eine Zusammenfassung der durchgeführten Simulationen und deren Ergebnisse finden sich in **Kapitel 4**. Das letzte **Kapitel 5** zieht eine Schlussfolgerung aus den gefundenen

[(2)] Anteil von der Photokathode erkannter Photonen gegenüber der Gesamtzahl an simulierten Photonen.

Ergebnissen und gibt einen kurzen Ausblick über eine mögliche weiterführende Vorgehensweise.

1.3 Notationshinweise

Hier aufgelistet finden sich einige Hinweise zur verwendeten Notation:

- Mit "$*$" markierte Gleichungen verweisen auf eine detaillierte Ausführung und/oder Herleitung im Anhang.
- Für die Menge der natürlichen Zahlen $\mathbb{N}$ gilt: $0 \notin \mathbb{N}$.
- Die Menge $\{1, ..., k\} \subset \mathbb{N}$ sei bezeichnet durch $\underline{k}$.
- Die Einheitsmatrix sei bezeichnet mit $\mathbb{1}_n \in \mathbb{R}^{n \times n}$.

1.4 Verfügbarkeit des Quelltextes

Der Quelltext der hier vorgestellten Software ALGO findet sich als git-Repository unter `https://gitlab.rlp.net/wom/ALGO.git`, als auch auf der beigefügten CD, sofern es sich um die Druckfassung handelt. Enthalten sind unter anderem Python-Beispielskripte als auch eine Dokumentation über die Funktionalität einzelner Klassen.

2 Theoretische Grundlagen

> *Das Ergebnis habe ich schon, jetzt brauche ich nur noch den Weg, der zu ihm führt.*
>
> — **Carl Friedrich Gauß**
> 1777 - 1855

In diesem Kapitel werden die theoretischen Grundlagen und Konzepte erläutert, welche im Rahmen der numerischen Optimierung des ALG und ergänzend bezüglich der bestehenden Softwarebasis WOMRaT [22] benötigt werden.

2.1 Optimierungsverfahren

Um eine lokale numerische Optimierung des ALG durchzuführen, wird das Resultat der Ray-Tracing-Simulation als Energiefunktion aufgefasst und diese anhand formgebender Parameter optimiert. Mathematisch betrachtet wird also das Parameterset P gesucht, welches das Simulationsergebnis optimiert:

$$\arg\max_{p_i \in P} f(\text{simulation}(p_0, \ldots, p_n, a_0, \ldots, a_m))$$

Dabei ist die Menge $A = \{a_0, \ldots, a_m\}$ eine zusätzliche Argumentliste, welche über den Optimierungszeitraum konstant bleibt. Diese beinhaltet beispielweise die Brechungsindizes des Materials. Die Zielfunktion f ist jene, nach welcher optimiert wird. Diese Funktion bildet dabei auf $\mathbb{R}$ ab und erzeugt so die Energielandschaft, in welcher das Optimum gesucht wird. Insbesondere kann diese die Anzahl erkannter Photonen, Winkel der Photonen oder auch den Endradius des ALG berücksichtigen. Das Parameterset P spezifiziert die Form der Profilkurve durch verschiedene Splineinterpolationen (siehe Abschnitt 2.2). Auch kann hier zusätzlich die Länge des Lichtleiters, sowie der Endradius mitberücksichtigt werden.

Entscheidend ist also, dass unterschieden wird, welche Parameter p_i für den Optimierer die Dimension des Suchraumes bilden und welche Argumente als feste Größen a_i in die Simulationkonfiguration eingehen. Es kann also schlecht von der Optimierung des ALG gesprochen werden, vielmehr handelt es sich um eine Optimierung nach bestimmten Kriterien. In Kapitel 4 werden verschiedene Simulationsergebnisse unter diversen Ansätzen diskutiert.

2.1.1 Wahl des Optimierers

Als lokaler Optimierer wurde der Nelder-Mead Simplex Algorithmus [11] aufgrund seiner einfachen Implementierungsmöglichkeit und der Tatsache, dass es sich hierbei um ein gradientenfreies Abstiegsverfahren handelt, verwendet. Insbesondere ist Letzteres zwingend erforderlich, da die Simulation selbst nicht differenzierbar ist. Im Zusammenhang sei erwähnt, dass eine Zielfunktion f für einen optimalen Kurvenverlauf minimal werden muss. Eine mögliche Lösung liefert gegebenenfalls eine Negation des Funktionsergebnis:

$$\arg\max_{p_i \in P}(f(...)) = \arg\min_{p_i \in P}(-f(...))$$

2.2 Spline Interpolationen

Eine zugleich stimmige als auch performante Approximation der Profilkurve des ALG liefern Splines, intervallweise definierte Funktionen. Aus den einzelnen Segmenten lassen sich deren Rotationskörper mit den Trajektorien der Photonen auf Schnitt überprüfen. Die Koeffizienzen wurden bisher extern über eine Eingabedatei dem Programm übergeben. Um jedoch mehrere Simulationsdurchführungen sequentiell im Rahmen der numerischen Optimierung durchzuführen, wurde aus Performancegründen (siehe Kapitel 4) eine eigene Implementierung durchgeführt. Dabei wurden zugleich verschiedene Ansätze verfolgt, welche zu unterschiedlichen Suchräumen führen. Diese werden im Folgenden besprochen.

Die Abschnitte 2.2.1 und 2.2.2 basieren auf einem Vorlesungsskript der University of Houston [17] und stellen eine Erweiterung dessen dar. Die in den Abschnitten 2.2.3 und 2.2.4 gezeigten Splineinterpolationen wurden nach unserem Wissensstand in dieser Form noch nicht durchgeführt.

Seien im Folgenden stets $n+1$ Stützpunkte $\{(x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\} \subset \mathbb{R} \times \mathbb{R}$ mit $x_i < x_{i+1}$ für alle $i \in \{0, 1, ..., n-1\} =: I$ gegeben.

2.2.1 Linear interpolierende Splines

Linear interpolierende Splines bilden die einfachste Variante der Splines ersten Grades. Sie charakterisieren eine einfache Approximation der Stützpunkte, indem sie diese der Reihe nach miteinander verbinden. Das j-te Polynom eines solchen Splines lässt sich so beispielsweise aufstellen als:

$$s_j(x) = \frac{y_{j+1} - y_j}{x_{j+1} - x_j}(x - x_j) + y_j \quad \forall\ j \in I$$

Hier sei zu beachten, dass stets gilt: $s_j(x_j) = y_j$ und $s_j(x_{j+1}) = y_{j+1}$. Lineare Splines bilden jedoch eine sehr schlechte Approximation des ALG. Insbesondere kann wegen fehlender Ableitung an den Stützstellen keine (eindeutige) Normale für den ALG gefunden werden, wodurch das Ray-Tracing erschwert würde.

2.2.2 Quadratisch interpolierende Splines

Im Gegensatz zu linear interpolierenden Splines, welche die Stützpunkte geradlinig verbinden, fordern quadratisch interpolierende Splines zusätzlich einfache Differenzierbarkeit. Ebenso können die Stützwerte eines solchen als Parameter für den Optimierer verwendet werden. Der entstehende Spline $s(x) : [x_0, x_n[\rightarrow \mathbb{R}$ hat die Form

$$s(x) = \begin{cases} s_0(x) = a_0 x^2 + b_0 x + c_0, & x \in [x_0, x_1[\\ \vdots & \\ s_i(x) = a_i x^2 + b_i x + c_i, & x \in [x_i, x_{i+1}[\\ \vdots & \\ s_{n-1}(x) = a_{n-1} x^2 + b_{n-1} x + c_{n-1}, & x \in [x_{n-1}, x_n[\end{cases}$$

mit der Forderung $\frac{\mathrm{d}}{\mathrm{d}x} s_{i-1}(x_i) = \frac{\mathrm{d}}{\mathrm{d}x} s_i(x_i) \; \forall \; i \in \underline{n-1}$. Daraus folgt unmittelbar, dass $\frac{\mathrm{d}}{\mathrm{d}x} s(x)$ ein linearer Spline sein muss, da die Ableitung einer quadratischen Funktion stets eine Gerade ist. Für $\frac{\mathrm{d}}{\mathrm{d}x} s_i(x)$ gilt gemäß Abschnitt 2.2.1:

$$\frac{\mathrm{d}}{\mathrm{d}x} s_i(x) = \frac{w_{i+1} - w_i}{x_{i+1} - x_i}(x - x_i) + w_i$$

wobei die Stützpunkte des linearen Splines gegeben sind mit $\{(x_0, w_0), ..., (x_n, w_n)\}$ mit $x_i < x_{i+1} \forall \; i \in I$. Nach einmaliger Integration ergibt sich:

$$\int \frac{\mathrm{d}}{\mathrm{d}x} s_i(x) \, \mathrm{d}x = \frac{w_{i+1} - w_i}{2(x_{i+1} - x_i)}(x - x_i)^2 + w_i(x - x_i) + q_i = s_i(x) \tag{2.1}$$

mit q_i als Integrationskonstante. Unter Hinzunahme, dass der Spline die Stützpunkte schneiden muss, also $y_i = s_i(x_i)$ und $y_{i+1} = s_i(x_{i+1})$ gilt, folgt:

$$q_i = s_i(x_i) = y_i. \tag{2.2}$$

Unter Hinzunahme von (2.2) gilt zudem:

$$\begin{aligned} &\frac{w_{i+1} - w_i}{2(x_{i+1} - x_i)}(x_{i+1} - x_i)^2 + w_i(x_{i+1} - x_i) + q_i = y_{i+1} \\ \Leftrightarrow\ &\frac{w_{i+1} - w_i}{2}(x_{i+1} - x_i) + w_i(x_{i+1} - x_i) + q_i = y_{i+1} \\ \Leftrightarrow\ &\frac{w_{i+1} + w_i}{2}(x_{i+1} - x_i) = y_{i+1} - q_i \overset{(2.2)}{=} y_{i+1} - y_i. \end{aligned} \tag{2.3}$$

Dies führt zu einem Gleichungssystem mit $n+1$ Unbekannten und n Gleichungen. Mit einer zusätzlichen Randbedingung:

$$0 =: \frac{\mathrm{d}}{\mathrm{d}x} s_0(x_0) = \frac{w_1 - w_0}{x_1 - x_0}(x_0 - x_0) + w_0, \tag{2.4}$$

welche sich aus der Geometrie (Tangetialer Anschluss ALG, WOM-Rohr) zwingend ergibt, vervollständigt sich das Gleichungssystem.

Für die $3n$ unbekannten Koeffizienten $(a_0, b_0, c_0, \ldots, a_{n-1}, b_{n-1}, c_{n-1})$ existieren also $3n$ Bestimmungsgleichungen: Die Anfangssteigung $\frac{\mathrm{d}}{\mathrm{d}x} s_0(x_0) = 0$, die $n-1$ Stetigkeitsbedingungen erster Ordnung an den Stützstellen $x_1, \ldots, x_{n-1}$ und die $2n$ Interpolationsbedingungen $s_i(x_i) = y_i$ sowie $s_i(x_{i+1}) = y_{i+1}$. Aus den obigen Gleichungen (2.3) und (2.4) lässt sich ein lineares Gleichungssystem folgender Form aufstellen:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & \ldots & 0 \\ 1 & 1 & 0 & \vdots & \vdots \\ 0 & 1 & 1 & \vdots & \vdots \\ \vdots & \ldots & \ddots & \ddots & \vdots \\ 0 & \ldots & \ldots & 1 & 1 \end{pmatrix}}_{M=(m_{i,j})\in\mathbb{R}^{n\times n}} \underbrace{\begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ \vdots \\ w_{n-1} \end{pmatrix}}_{w\in\mathbb{R}^n} = \underbrace{\begin{pmatrix} 0 \\ \frac{2(y_1-y_0)}{x_1-x_0} \\ \vdots \\ \vdots \\ \frac{2(y_n-y_{n-1})}{x_n-x_{n-1}} \end{pmatrix}}_{d\in\mathbb{R}^n}$$

Dabei ist $m_{i,j} = 1$, wenn $i = j \vee i = j+1$, sonst gilt $m_{i,j} = 0$. Um eine schnellere Laufzeit für die Berechnung erzielen zu können, kann $Q := M^{-1}$ direkt bestimmt werden (siehe Lemma 1).

Lemma 1. *Sei $H = \mathbb{N} \cup \{0\}$ und seien Matrizen gegeben $M = (m_{i,j}), Q = (q_{i,j}) \in \mathbb{R}^{n\times n}$ für ein $n \in \mathbb{N}$ mit*

$$m_{i,j} = \begin{cases} 1, \text{ wenn } i = j \vee i = j+1 \\ 0, \text{ sonst,} \end{cases} \quad \text{und } q_{i,j} = \begin{cases} -1, \text{ wenn } i-2l-1 = j \text{ für ein } l \in H \\ 1, \text{ wenn } i-2l = j \text{ für ein } l \in H \\ 0, \text{ sonst} \end{cases}$$

für $i, j \in \underline{n}$, dann gilt: $MQ = \mathbb{1}_n$.

Beweis. Für alle $i, k \in \underline{n}$ gilt:

$$\begin{aligned} ((MQ)_{i,k}) &= \sum_{j=1}^{n} m_{i,j} q_{j,k} \\ &\overset{\substack{\text{Fast alle 0} \\ \text{nach Wahl von } M}}{=} m_{i,i-1} q_{i-1,k} + m_{i,i} q_{i,k} \\ &= \begin{cases} q_{i,k} + q_{i-1,k}, i > 1 \\ q_{i,k}, \text{ sonst.} \end{cases} \end{aligned}$$

(1) Für $i = 1 \Rightarrow q_{i,i} = 1$, denn $-2l = 1$ hat keine Lösung in H und $q_{i,k} = 0$ für $k > 1$, da $-2l < 0 \; \forall \; l \in H$.

(2) Für $i > 1$ und $i < k$ folgt $q_{ik} = 0$ und $q_{i-1k} = 0$, da die Gleichungen $i = k+2l+2$ und $i = k+2l+1$ und $i = k+2l$ unlösbar sind für alle $l \in H$.

(3) Für $i > 1$ und $i \geq k$ mit $i-2r = k$ für ein $r \in H \Rightarrow q_{ik} = q_{i,i-2r} = 1$ und $q_{i-1,k} = q_{i-1,i-2r} \overset{p=i-1}{=} q_{p,p-2r+1} = -1$.

(4) Für $i > 1$ und $i \geq k$ mit $i-2r-1 = k$ für ein $r \in H \Rightarrow q_{ik} = q_{i,i-2r-1} = -1$ und $q_{i-1,i-2r-1} \overset{p=i-1}{=} q_{p,p-2r} = 1$.

Aus (1)-(4) folgt die Behauptung. □

Im Ergebnisvektor $w = Qd = (w_1, \ldots, w_n)^T$ ergibt sich ein kaskadierendes Muster, wodurch die Lösung direkt bestimmt werden kann:

$$w_i = \sum_{k=1}^{n} q_{ik} d_k = \sum_{k=1}^{i} q_{ik} d_k = \sum_{k=1}^{i} (-1)^{((i+k) \bmod 2)} d_k = \begin{cases} 0, \text{ wenn } i = 1, \\ d_i - d_{i-1}, \text{ wenn } i = 2, \\ d_i - d_{i-1} + w_{i-2}, \text{ sonst.} \end{cases}$$

Die abschließende Bestimmung der Koeffizienten erfolgt gemäß Gleichung (2.1):

$$a_i = \frac{w_i - w_{i-1}}{2 \cdot (x_i - x_{i-1})}, \quad b_i = w_{i-1} - 2 a_i x_{i-1}, \quad c_i = a_i x_{i-1}^2 - w_{i-1}^2 + y_{i-1}.$$

Die Implementierung in ALGO [19] ist entsprechend umgesetzt (Listing 1):

```
168 void generate() override
169 {
170     std::vector<value_t> d(this->_knot_x.size()); // size N
171     std::vector<value_t> w(this->_knot_x.size()); // size N
172
173     for (size_t i = 1; i < d.size(); ++i) // d[0] = 0
174     {
175         d[i] = 2 * (this->_knot_y[i] - this->_knot_y[i - 1]) /
176                (this->_knot_x[i] - this->_knot_x[i - 1]);
177     }
178
179     // solve linear equation system
180     for (size_t i = 1; i < d.size(); i++) // z[0] = d[0] = 0
181     {
182         w[i] = d[i];
183         w[i] -= d[i - 1];
184         w[i] += (i > 1) ? w[i - 2] : 0;
185     }
186
187     // compute coefficients
188     for (size_t i = 1; i < this->_knot_x.size(); i++)
189     {
190         auto id = (i - 1) * 3;
191         auto &x{this->_knot_x}, &y{this->_knot_y}, &p{this->_polynom};
192
193         // quadratic coefficient
194         p[id] = 0.5 * (w[i] - w[i - 1]) / (x[i] - x[i - 1]);
195         // linear coefficient
196         p[id + 1] = w[i - 1] - 2 * p[id] * x[i - 1];
197         // constant coefficient
198         p[id + 2] = p[id] * x[i - 1] * x[i - 1] - w[i - 1] * x[i - 1] + y[i - 1];
199     }
200 }
```

Listing 1: Implementierung der *naiven* Splineinterpolation.
Siehe: ALGO/include/My/QuadraticSpline.h [19]

Die obige Implementierung liefert im Vergleich zum naiven Lösen des Gleichungssystems ($\mathcal{O}(n^3)$) eine Gesamtlaufzeit von $3(n+1) + c \in \mathcal{O}(n)$. Insbesondere liefert diese dabei gleiche Abweichungen (≈1:10000) zur Falkekurve wie `scipy` [21] (siehe Abbildung 2.1) unter Verwendung von 10 äquidistanten Stützstellen.

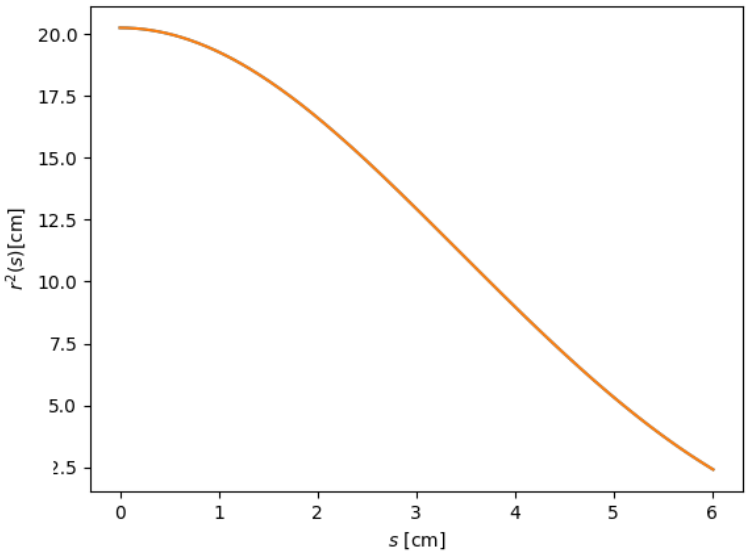


Abb. 2.1.: Vergleich der Spline Interpolation mit 10 Stützpunkten (orange) und der Falkekurve [5] (blau). Beide Kurven sind nahezu deckungsgleich.

2.2.3 Quadratische Splines mit vorgegebenen Krümmungen

Ist eine Folge von Krümmungen $a_0, a_1, a_2, \ldots, a_{n-1} \in \mathbb{R}$ gegeben, so kann ein Spline mit äquidistanten Stützstellen gefunden werden, sodass gilt $\frac{\mathrm{d}^2}{\mathrm{d}x^2} s_i(x) = 2a_i$. Diese Krümmungen können statt der Stützwerte aus Abschnitt 2.2.2 als Parameter für eine numerische Optimierung der Profilkurve verwendet werden (siehe auch Kapitel 4 und Abschnitt 2.1). Seien

$$s_i(x) = a_i x^2 + b_i x + c_i$$

die i-ten Segmente des Splines mit Krümmung a_i, so werden als Interpolationsbedingungen mit dem äquidistanten[(1)] Abstand δ gefordert:

$$\frac{\mathrm{d}}{\mathrm{d}x} s_i(\delta) = \frac{\mathrm{d}}{\mathrm{d}x} s_{i+1}(0) \tag{2.5}$$

$$s_i(\delta) = s_{i+1}(0) \tag{2.6}$$

Wichtig ist hier, dass das Folgesegment $s_{i+1}(x)$ für das Intervall $[0, \delta]$ konstruiert wird und erst durch Verschiebung um $\delta \cdot i$ den Spline vervollständigt. Die Koeffizienten der unverschobenen Segmente können über die folgenden Rekursionsgleichungen gemäß der Interpolationsbedingungen bestimmt werden:

$$b_{i+1} = 2a_i\delta + b_i \tag{2.7}$$

$$c_{i+1} = a_i\delta^2 + b_i\delta + c_i \tag{2.8}$$

mit Rekursionsanfängen b_0 und $c_0 = y_0$ mit y_0 erster Stützwert des Splines.

(1) Der Abstand der Stützstellen und zugleich die Breite eines Splinesegments.

Für den letzten Stützwert y_n ergibt sich somit:

$$\begin{aligned} y_n &= s_{n-1}(\delta) \\ &= a_{n-1}\delta^2 + b_{n-1}\delta + c_{n-1} \end{aligned} \tag{2.9}$$

Dabei gilt für b_{n-1} und c_{n-1}:

$$b_{n-1} = 2\delta \sum_{i=0}^{n-2} a_i + b_0 \tag{2.10*}$$

$$c_{n-1} = \delta^2 \sum_{i=0}^{n-2} a_i + 2\delta^2 \sum_{i=0}^{n-2} \sum_{j=0}^{i-1} a_j + \delta(n-1)b_0 + y_0 \tag{2.11*}$$

Die Gleichungen 2.10* und 2.11* in Gelichung (2.9) eingesetzt:

$$y_n = \delta^2 \left(\sum_{k=0}^{n-1} (2k-1)a_{n-k} \right) + \delta(n-1)b_0 + y_0 \tag{2.12*}$$

Mittels eines zusätzlichen Skalierungsfaktors α, durch Modifikation der Krümmungen zu $a'_i := \alpha a_i$, kann y_n (Endradius ALG) frei gewählt werden:

$$\begin{aligned} y_n &= \delta^2 \left(\sum_{k=0}^{n-1} (2k-1)\alpha a_{n-k} \right) + \delta(n-1)b_0 + y_0 \\ &= \alpha\delta^2 \left(\sum_{k=0}^{n-1} (2k-1)a_{n-k} \right) + \delta(n-1)b_0 + y_0 \\ \Leftrightarrow \alpha &= \frac{y_n - y_0 - \delta(n-1)b_0}{\delta^2 \left(\sum_{k=0}^{n-1} (2k-1)a_{n-k} \right)} \end{aligned} \tag{2.13}$$

Aufgrund der Vereinfachung der Intervalle auf äquidistante Segmentbreiten müssen diese bei der Splineevaluation entsprechend verschoben werden:

$$s_i(x) \rightarrow s_i(x + i\delta) \quad \forall \; i \in I$$

Für die Koeffizienten des zugrunde liegenden quadratischen Splines ergibt sich:

$$\begin{aligned} a_i &= a'_i \\ b_i &= b'_i - 2a'_i(\delta i + x_0) \\ c_i &= c'_i + a'_i(\delta i + x_0)^2 - b'_i(\delta i + x_0) \end{aligned}$$

Dabei bezeichne x_0 den Startwert des gültigen Intervalls. Auf den Plots in Kapitel 4 zeigt sich insbesondere, dass die so entstandene Interpolation unter Verwendung der gleichen Krümmungen eines aus Abschnitt 2.2.2 interpolierten Splines, die gleichen Koeffizienten liefert. Dies bietet den Vorteil lokale Optimierungen ohne zusätzliche Umrechnungen von der gleichen Intialkurve (bzw. Initialsimplex) starten zu können.

2.2.4 Quadratische Splines mit vorgegebenen Steigungen

Werden für $n+1$ Stützpunkte statt der Stützwerte die Steigungen $\frac{\mathrm{d}}{\mathrm{d}x} s_i(x) = \eta_i$ vorgegeben, so kann ebenfalls ein quadratischer Spline gebildet werden. Diese Werte können statt der Stützwerte aus Abschnitt 2.2.2 als Parameter für eine numerische Optimierung der Profilkurve verwendet werden (siehe auch Kapitel 4 und Abschnitt 2.1). Für $s_i(x) = a_i x^2 + b_i x + c_i$, $i \in I$ gilt entsprechend der Stetigkeitsbedingungen:

$$\begin{cases} s_i(x_i) & = y_i \\ \frac{\mathrm{d}}{\mathrm{d}x} s_i(x_i) & = \eta_i \\ s_i(x_i + \delta_i) & = y_{i+1} \\ \frac{\mathrm{d}}{\mathrm{d}x} s_i(x_i + \delta_i) & = \eta_{i+1} \end{cases} \Leftrightarrow \begin{cases} a_i x_i^2 + b_i x_i + c_i & = y_i \\ 2a_i x_i + b_i & = \eta_i \\ a_i (x_i + \delta_i)^2 + b_i (x_i + \delta_i) + c_i & = y_{i+1} \\ 2a_i (x_i + \delta_i) + b_i & = \eta_{i+1} \end{cases}$$

Dabei ist $\delta_i = x_{i+1} - x_i$. Das so entstandene Gleichungssystem besitzt vier Unbekannte und ist damit eindeutig lösbar. Daraus lassen sich Gleichungen für a_i, b_i, c_i, y_i aufstellen:

$$\begin{aligned} a_i &= \frac{\eta_i - \eta_{i-1}}{2\delta_i} \\ b_i &= \frac{x_{i-1}\eta_{i-1} - x_{i-1}\eta_i}{\delta_i} + \eta_{i-1} \\ c_i &= \frac{\eta_i x_{i-1}^2 - \eta_{i-1} x_{i-1}^2}{2\delta_i} - x_{i-1}\eta_{i-1} + y_{i-1} \\ y_i &= \frac{\delta_i \eta_{i-1} + \delta_i \eta_i}{2} + y_{i-1} \end{aligned}$$

Dabei wird als Rekusionsanfang zusätzlich für y_i der Stützwert y_0 gefordert. Als Variation kann für das letzte Segment die einfache Interpolation wie in Abschnitt 2.2.2 verwendet werden, um den Spline auf einen gewünschten Endpunkt (x_n, y_n) (hier: Endradius) abzubilden. Demnach gilt für die Parameter $a_{n-1}, b_{n-1}, c_{n-1}$:

$$\begin{cases} a_{n-1} x_{n-1}^2 + b_{n-1} x_{n-1} + c_{n-1} & = y_{n-1} \\ a_{n-1} x_n^2 + b_{n-1} x_n + c_{n-1} & = y_n \\ 2a_{n-1} x_{n-1} + b_{n-1} & = \eta_{n-1} \end{cases}$$

$$\begin{aligned} a_{n-1} &= \frac{-y_n + y_{n-1} - \delta_{n-1}\eta_{n-1}}{\delta_{n-1}^2} \\ b_{n-1} &= \frac{2x_{n-1}(y_{n-1} - y_n + \delta_{n-1}\eta_{n-1})}{\delta_{n-1}^2} + \eta_{n-1} \\ c_{n-1} &= \frac{y_n x_{n-1}^2 + \delta_{n-1}^2 (y_{n-1} - x_{n-1}\eta_{n-1}) - x_{n-1}^2 (y_{n-1} + \delta_{n-1}\eta_{n-1})}{\delta_{n-1}^2} \end{aligned}$$

Auf den Plots in Kapitel 4 zeigt sich, dass die so entstandene Interpolation unter Verwendung der gleichen Steigungen eines aus Abschnitt 2.2.2 interpolierten Splines, die gleichen Koeffizienten liefert. Das bietet den Vorteil Optimierungen ohne zusätzliche Umrechnungen von der gleichen Intialkurve (bzw. Initialsimplex) starten zu können.

2.3 Anpassungen Ray-Tracing Algorithmus

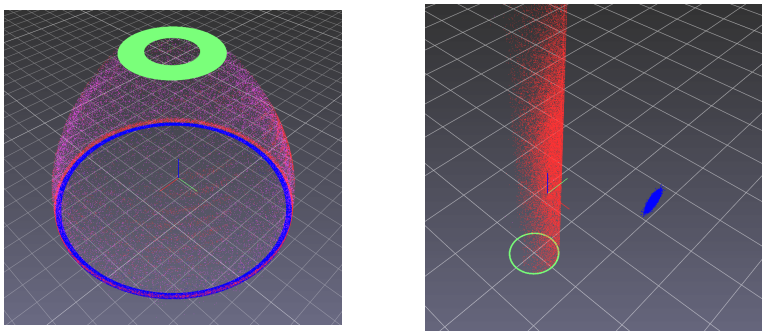
Bisher wurden Photonen anhand ihrer Position nach der Simulation ihrem Status (Kathodentreffer, Verloren, etc.) sortiert. Dies führte jedoch zu einer schlechten Zuordnung aufgrund von Rundungsfehlern bezüglich zurücklaufender[2] Photonen. Deshalb wird zusätzlich ein Statuscode für jedes Photon ermittelt und direkt festgehalten, mit welchem Resultat dieses die Simulation verlässt (siehe Listing 2).

```
res = HitResult.Null  # initial statuscode
scattered = False     # flag if scattering occured for current photon
distance_total = 0    # total distance the photon travelled

# loop until photon is no longer active (break below)
while 1:
    distance_part = 0  # partial distance for current iteration

    # intersect with corresponding model
    res = model.intersect(...)  # returns statuscodes: Null, Success, Continue, Returned
    distance_total += distance_part

    # apply material interactions
    scattered = apply_scattering(...)  # true if scattered
    if apply_absorbed(...):
        res = HitResult.Absorbed
        break # absorbed
    if scattered:
        dir = random_direction(...)
        continue  # don't check for wall interaction

    if (res != HitResult.Continue) break

    if not interact_wall(...):
        res = HitResult.Null  # failed interaction -> Photon Lost
        break

if scattered:
    res += 64
compute_output(...)
```

Listing 2: Pseudo-Python Code für den angepassten Ray-Tracing-Algorithmus

Des weiteren sorgt dies für eine einfache Sortierbarkeit der Photonen für die vollständige WOM-Simulation (siehe Abschnitt 2.4). Der Statuscode kann folgende Werte annehmen: *continue*[3], *returned*, *success*, *lost* oder *absorbed*. Zusätzlich wird die Information gespeichert, ob während des Simulationsdurchlaufes eines Photons Streuung aufgetreten ist[4]. Die Ermittelung eines Kathodentreffers (*success*) wird nun mittels Schnitttest durchgeführt (siehe Abschnitt 2.3.1). Die Kathode wird dabei als Kreisfläche entlang der z-Achse modelliert. In Abbildung 2.2 sind 3D-Plots der Einteilung zu sehen.

[2] Gemeint ist, wenn Photonen den ALG in Richtung des WOM-Rohres verlassen.
[3] Dieser Statuscode ist für den Ray-Tracing-Algorithmus relevant. Er induziert, dass ein Photon seine Bahn noch nicht vollendet hat.
[4] Dies wird durch Repräsentation des Statuscodes als `byte` umgesetzt. Photonen mit Statuscode < 64 weisen keine Streuung auf. Der tatsächliche Ausgangscode lässt sich so mittels modulo berrechnen.



(a) Adiabatischer Lichtleiter (b) WOM-Rohr

Abb. 2.2.: Photonen nach der Simulation im 3D Plot unter Verwendung des pptk-Toolkits [7] mit Statuscodes. Dabei sind die blauen Photonen in 2.2b jene, die das WOM initial nicht treffen.
blau: $returned$; grün: $success$; rot: $lost$; violett: $absorbed$

2.3.1 Achsen-orientierte Kreisfläche – Strahl Schnitttest

Da die Photokathode achsenorientiert und symmetrisch um die z-Achse positioniert ist, lässt sich der Schnitttest mit der korrespondierenden Ebene einfach schreiben als:

$$o_z + \lambda d_z = e_z \Rightarrow \lambda = \frac{e_z - o_z}{d_z}$$

Dabei wird der Strahl (Photon) beschrieben durch: $o + \lambda d$ mit $o, d \in \mathbb{R}^3$.
Im Folgenden gilt also noch zu prüfen, ob $\lambda > 0$ ist und der Schnittpunkt $P = o + \lambda d$ im Kreis mit Radius r liegt.
Dazu ergibt sich aus der Kreisgleichung ($z^2 = x^2 + y^2$) die Bedingung:

$$r^2 \geq P_x^2 + P_y^2$$

2.4 WOM Simulationsalgorithmus

```
1  N = len(PHOTONS)
2  initialIntersection(TUBE, PHOTONS, N)
3  sim_tube = True
4  while(N > 0):
5      if sim_tube:
6          simulate(TUBE, PHOTONS, N)
7          N = partition(PHOTONS, N, SUCCESS)
8      else:
9          simulate(ALG, PHOTONS, N)
10         N = partition(PHOTONS, N, RETURNED)
11     sim_tube = not sim_tube
```

Listing 3: Pseudo-Python Code für den WOM Simulationsalgorithmus

Ziel des WOM Simulationsalgorithmus ist die vollständige Simulation des WOM Moduls von der Lichtquelle zur Photokathode, um die Gesamteffizienz des Moduls zu messen. Dies ist für die ALG Optimierung irrelevant, wurde jedoch im Rahmen der Umstrukturierung implementiert. Dazu wird zu Beginn ein initialer Schnitttest mit dem WOM-Rohr

Modell (Tube) durchgeführt, um die Photonen in das Modul hinein zu bewegen. Im Folgenden wird abwechselnd das WOM-Rohr und der ALG simuliert, bis kein weiteres Photon mehr in diesem aktiv ist. Dazwischen müssen die Photonen nach ihren Statuscodes paritioniert werden, um die noch aktiven Photonen zu identifizieren. In Listing 3 ist der Algorithmus im Pseudocode angegeben.

In N wird dabei stets die aktuelle Zahl an aktiven Photonen gespeichert. Zu beachten ist hier, dass sowohl die Folgesimulationen als auch die Folgepartitionierungen immer auf einer kleiner werdenden Photonenmenge arbeiten. Ebenso muss die Partitionierung jeweils nach anderen Prädikaten agieren, da im `EllipticModel` Photonen, die das WOM-Rohr an den Enden verlassen als *success* eingetragen werden, während beim ALG zurücklaufende Photonen mit Statuscode $returned$ vermerkt werden. In beiden Fällen sind dies die Photonen, welche im Folgedurchlauf weitersimuliert werden sollen. Ein 3D-Plot eines solchen Simulationsdurchlaufes findet sich im Anhang B.3.

2.5 Binärer Divide&Conquer (GPU-)Partitionsalgorithmus

Um die Performance der vollständigen Simulation (Abschnitt 2.4) zu gewährleisten, ist eine möglichst schnelle binäre Partitionierung erforderlich. Glücklicherweise befinden sich die zu sortierenden Photonendaten (Positionen, Richtung, Status, ...) bereits auf der GPU, wodurch zumindest ein zusätzliches Kopieren seine Notwendigkeit verliert. Es existiert zwar bereits eine Implementierung für einen GPU-Partitionierungsalgorithmus in der thrust-Bibliothek [8], jedoch hat diese in Kombination mit dem verwendeten Compiler zu Problemen geführt, weshalb eine eigene Implementierung durchgeführt wurde.

Anforderungen

Der Algorithmus fordert hohe Effizienz in Sachen Geschwindigkeit als auch im Speicherverbrauch. So sollte wenn möglich kein zusätzlicher Speicher alloziert werden müssen, sondern die Partitionierung *in-place* stattfinden. Dies wird insbesondere gefordert, da mehr als ein Array simultan sortiert werden muss, ohne ein komplett neues *getupeltes* Array anzulegen. Auch muss spezifiziert werden können, nach welcher Prädikatfunktion p, welche jedes Element einer der beiden Partitionen $[0,1]$ zuordnet, das Array beziehungsweise die gekoppelten Arrays paritioniert werden.

Ansatz

Sei m die Anzahl der verwendeten Threads, n die Anzahl der zu sortierenden Elemente und a das zu partitionierende Array. Initial sortiert jeder Thread i ein kleines Teilsegment der Größe $s = \frac{n}{m}$, welches dieser dann auf naive Art und Weise mit einer Laufzeit in $\mathcal{O}\left(\left(\frac{n}{m}\right)^2\right)$ partitioniert.

Dieser intiale *Stride s* wird dann im Folgenden stets verdoppelt, wodruch zunehmend die Hälfte der Threads deaktiviert wird. Die nun für jeden aktiven Thread verbleibende doppelt so große Sektion wird mittels des Mergeprinzip partitioniert. Dazu wird der erste Index u ermittelt, auf welchen die Prädikatfunktion innerhalb der ersten Sektion $[i \cdot s, i \cdot s + \frac{s}{2}[$ des Arrays für den i-ten Thread $p(a[u]) = 0$ liefert. Ebenso wird der letzte Index v innerhalb der zweiten Sektion $[i \cdot s + \frac{s}{2} - 1, (i+1) \cdot s[$ des Arrays ermittelt für den gilt: $p(a[v]) = 1$ und $p(a[v+1]) = 0$. Die Einträge der Indizes u, v im Array werden nun paarweise unter fortlaufender Inkrementierung von u und Dekrementierung von v vertauscht. (siehe auch Abbildung 2.3).

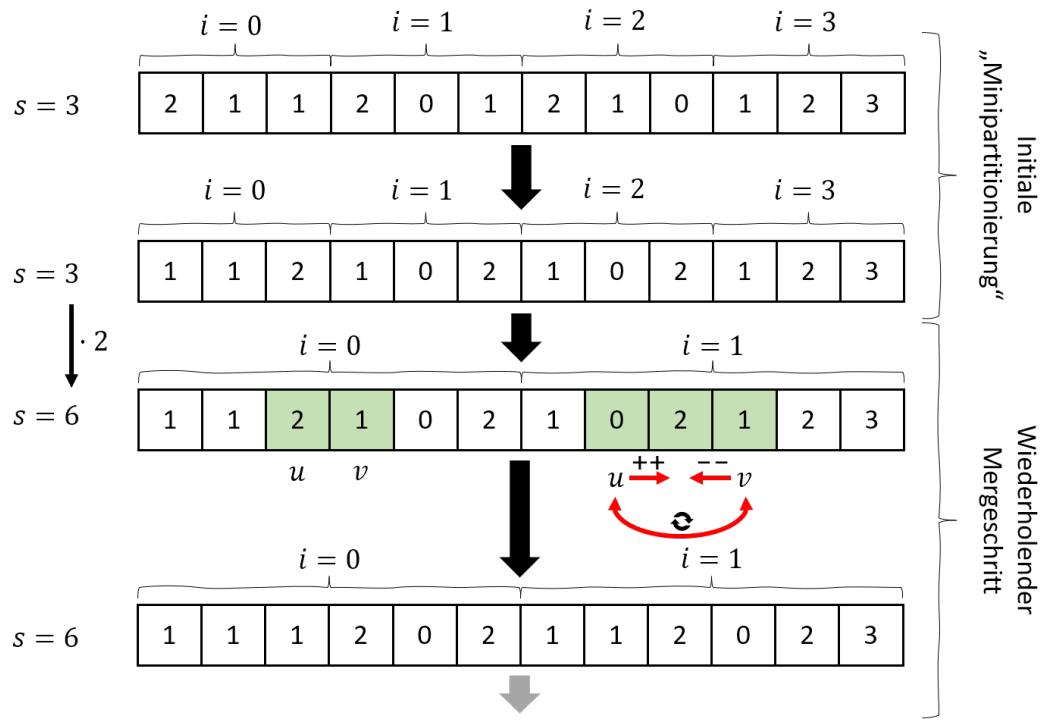


Abb. 2.3.: Schematischer Ablauf des Partitionierungsalgorithmus für den intialen Partitionsschritt und einem Mergeschritt.

Es sei an dieser Stelle erwähnt, dass die Implementierung in ALGO [19] auf CUDA 8 aufsetzt. Die Implementierung muss also in einem Block ausgeführt werden, da blockübergreifende Synchronisierung erst durch Cooperative Groups ermöglicht wird [13] und diese somit nicht unterstützt werden.

Performance

Der Gegebene Algorithmus besitzt eine Gesamtlaufzeit in $\mathcal{O}(\frac{n}{m})^2 + n\log_2(m)$. Er weist demnach eine ausreichende Geschwindigkeit auf, um einen vollständigen Simulationsdurchlauf in angemessener Zeit ausführen zu können.

Validierung

Um die Korrektheit des Partitionsalgorithmus zu verifizieren wurden mehrere Simulationen mit je variabler Zahl an Einträgen durchgeführt und mit der naiven Variante verglichen und erfolgreich verifiziert.

3 Softwarearchitektur

> *640 Kilobyte ought to be enough for anybody.*
>
> — **Bill Gates**

Dieses Kapitel gibt einen Überblick über die Architektur der neu entstandenen Software ALGO [19]. Die Neustrukturierung des WOMRaT [22] wurde durchgeführt, um eine möglichst dynamische Simulationsdurchführung zu ermöglichen. Darüber hinaus wurde eine Schnittstelle für den Python Interpreter [15] hinzugefügt, um Simulationsabläufe schnell anpassen und skripten zu können, ohne das Projekt neu kompilieren zu müssen oder aufwendige und fehleranfällige Konfigurationsdateien zu verwenden. Für weitere Informationen sei auf die Dokumentation (siehe Abschnitt 1.4) verwiesen.

3.1 Splines

Splines werden verwendet, um die Kurvenführung des ALG zu approximieren (siehe Abschnitt 2.2). Für eine gute Austauschbarkeit der unterliegenden Splineinterpolation wird ein Interface `Spline` verwendet.

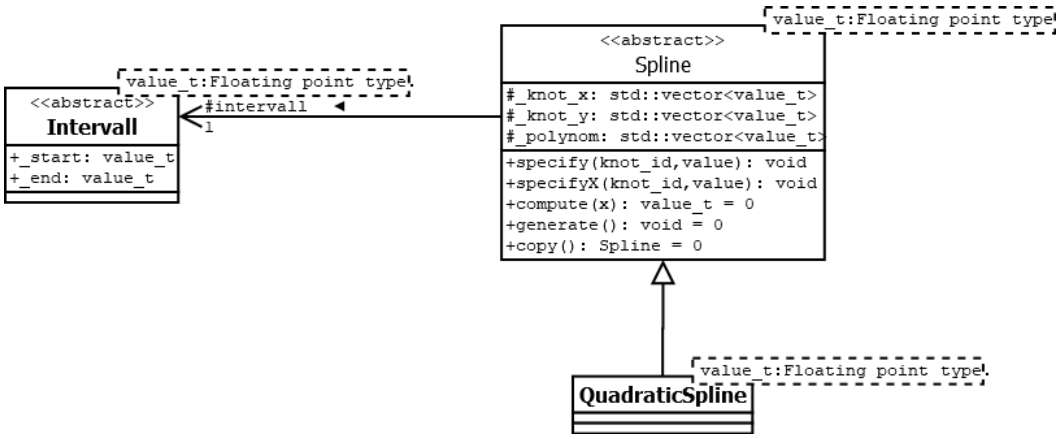


Abb. 3.1.: Überlick über die `Spline` Interface Spezifikation. Hierbei sind Funktionsargumente und Rückgaben stark vereinfacht oder ausgelassen.

Dieses Interface (als abstrakte Klasse in C++ implementiert) spezifiziert die minimal notwendige Schnittstelle, um auf Spline Daten (Stützpunkte und Koeffizienten) zuzugreifen und so einen direkten Zugriff auf die Daten zu haben. Dies ermöglicht das Kopieren der Daten zum GPU-Hauptspeicher, ohne die zugrundeliegende Implementierung[(1)] zu

[(1)] Beispielweise die Klasse `QuadraticSpline` welche von `Spline` ableitet.

kennen, erhält aber zugleich die Option den Interpolationsalgorithmus durch Variation der Kindklasse entsprechend auszutauschen.
Hierbei wird in Kauf genommen, dass eine Interpolation keine Stützwerte verwendet, beziehungsweise nicht auf diesen aufsetzt, so zum Beispiel quadratische Splines mit vorgegebenen Krümmungen (siehe Abschnitt 2.2.3). Das korrespondierende Modell benötigt für den Schnitttest jedoch weiterhin die Intervalle, in welchen ein Spline Segment valide ist, so sind diese Referenzwerte ebenso stets notwendig.

3.2 Photonen Generatoren

Um auch die Generierung von Photonen flexibel zu gestalten, wurde das Interface `PhotonGenerator` hinzugefügt. Durch Variation der Implementierung ermöglicht beispielsweise die Klasse `PhotonLoader` das Laden von Photonen aus einer Datei.
Die Funktion dieser Schnittstelle ist es, die Photonenquelle für die Simulation austauschbar zu machen. Dabei sind auch hier, wie im Interface `Spline`, Daten, welche vom Ray-Tracing Algorithmus auf der GPU benötigt werden, direkt im Interface (abstrakte Klasse) eingebunden, um die Kopien auf den Hauptspeicher der Grafikkarte ohne Typkonvertierungen zu ermöglichen. Diese beinhalten unter anderem die Photonendaten (Startpunkte und Richtungen) sowie die `curand_state`[2] Informationen.

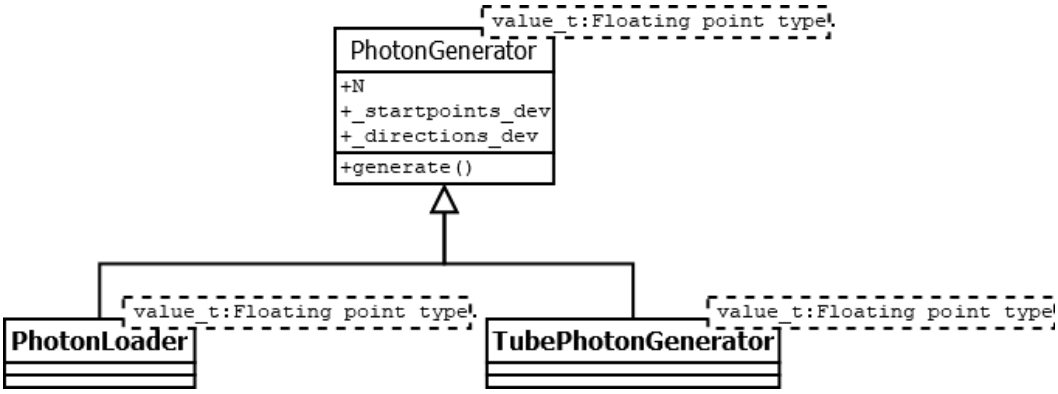


Abb. 3.2.: Schematische Übersicht der Generatoren. Hierbei sind Funktionsargumente und Rückgaben stark vereinfacht und teils ausgelassen.

3.3 Nelder-Mead Simplex Optimierer

Der Downhill Simplex Algorithmus [11] wird als Optimierer verwendet (siehe Abschnitt 2.1). Durch eine objektorientierte Realisierung ist es möglich Funktionsargumente, genauer deren Interaktion untereinander, dynamisch zu verändern. So werden die Splines innerhalb einer `SimplexFunctionArgument` Implementierung gekapselt, um zusätzliches Kopieren zu vermeiden. Dieses Interface spezifiziert dabei die Interaktion der Funktionsparameter in Form von Addition, Subtraktion, Division und Multiplikation[3]. Die

[2] Diese speichern den Zustand des curand Zufallszahlengenerators. [13]
[3] Diese bestimmen wie sich der Simplex durch den Parameterraum bewegt, so unter anderem wie die Berechnung des Schwerpunktes durchgeführt wird. Ferner könnte in einem anderen Kontext durch Überschreiben der Methoden festgelegt werden, dass dieser Parameterraum der Oberfläche einer Kugel entsprechen soll.

Einführung der Schnittstelle bietet den Vorteil besserer Kapselung der Daten, der Möglichkeit einer Variation des Parameterraums durch Überschreiben der arithmetischen Operationen und die Möglichkeit zusätzliche Optionen auf das konkrete Problem hinzuzufügen. So bietet die Implementierung `SplineSimulationArgument` beispielsweise eine Option zur Deaktivierung einzelner Parameter. Das `SimplexFunction` Interface weist neben der `compute(...)` Methode auch eine `preCompute(...)` Methode auf, deren Ziel es ist, rechenaufwendige Prozesse wie die Generierung der Spline Koeffizienten auszulagern, um diese nicht nach jeder arithmetischen Operation, beispielsweise in Form einer Iteration des Simplex, durchführen zu müssen.

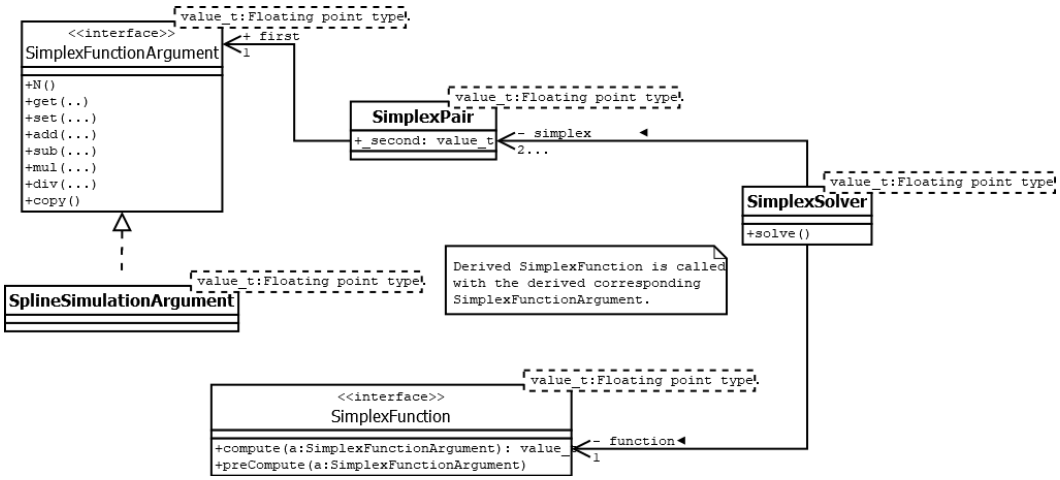


Abb. 3.3.: Überlick über die Simplex Schnittstelle. Hierbei sind Funktionsargumente und Rückgaben stark vereinfacht oder ausgelassen.

Die konkrete Funktion ist im Rahmen der ALG Optimierung die Simulationsklasse (siehe Abschnitt 3.4), welche in `preCompute(...)` die Generierung der Spline Koeffizienten vornimmt und das Kopieren dieser in den Hauptspeicher der GPU ausführt. In `compute(...)` wird dann die Simulation durchgeführt und die definierte Zielfunktion (Python oder C++) auf den Ergebnisdaten ausgewertet.
Die Klasse `SimplexSolver` ist die konkrete Implementierung des Algorithmus, welche anhand der gegebenen Parameter (siehe Abschnitt 4.1.2) eine Optimierung durchführen kann. Die verbleibende Klasse `SimplexPair`[4] (siehe Abbildung 3.3) bündelt die konkreten Implementierungen von `SimplexFunctionArgument` mit dem zugehörigen Funktionswert und ermöglicht einfache Sortierung und Vergleiche innerhalb der `SimplexSolver` Klasse.

3.4 Modelle und Simulation

Die Klasse `Simulation` bildet den Kern der Simulationsausführung. Dabei stellt sie die notwendigen Daten auf der GPU bereit und bietet zugleich eine Möglichkeit, diese nach der Simulation in Arrays zu überführen. Diese Funktionalität ist unter anderem für die Python Schnittstelle erforderlich (siehe dazu Abschnitt 3.5). Da Interfaces auf der

[4] Die Klasse ist eine Abwandlung von `std pair`.

Grafikkarte (CUDA) nicht in vollem Umfang verwendbar sind, erfordert die Klasse einen zusätzlichen Template Parameter, welcher das verwendete Modell (ALG, WOM-Rohr) repräsentiert. Dies bringt einige Nachteile mit sich; so ist es unter anderem nicht möglich, das Modell dynamisch zur Laufzeit zu ändern.

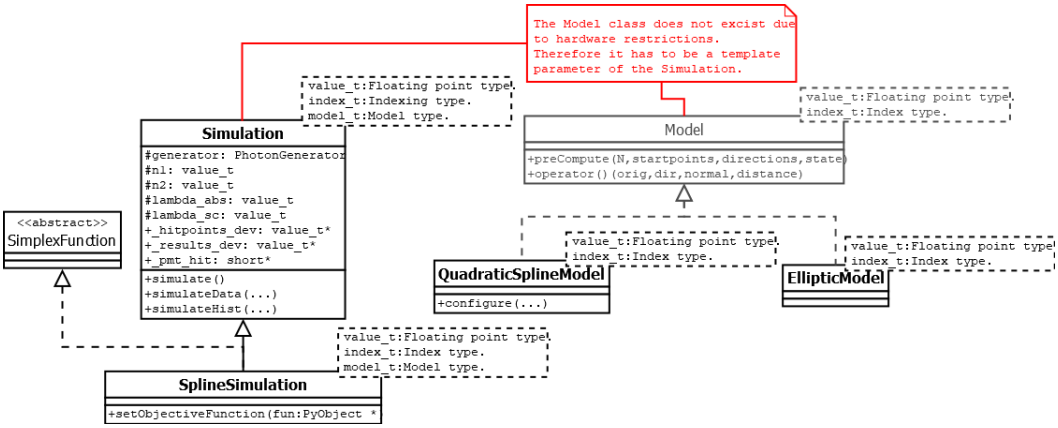


Abb. 3.4.: Schematische Übersicht der Simulationskomponenten. Hierbei sind Funktionsargumente und Rückgaben stark vereinfacht oder ausgelassen.

Es geht ebenfalls die Möglichkeit verloren, bestimmte Funktionen als Schnittstelle für eine Modell Klasse vorauszusetzen. Hier könnten Concepts und Constraints aus dem bald erscheinenden C++20 Standard [10] eine Abhilfe schaffen.

3.5 Python Schnittstelle

Um die existierende Klassenhierachie möglichst beizubehalten, wurde als Framework Cython [2] in Kombination mit der Python C-API [16] verwendet. Im Listing 4 ist beispielsweise ein solcher Wrapper für die `PhotonLoader` Klasse gezeigt.

```
343 # PhotonLoader
344 cdef extern from "include/ALGO/PhotonLoader.h" namespace "ALGO":
345     cdef cppclass FPhotonLoader:        # C++ class declaration
346         FPhotonLoader(string) except +  # C++ constructor declaration
347
348 cdef class PhotonLoader:                        # Python class
349     cdef shared_ptr[FPhotonLoader] gen           # Hold pointer to C++ object
350
351     def __cinit__(self, str path): # Python "constructor" calls C++ constructor
352         cdef string s = path.encode("UTF-8")
353         self.gen = make_shared[FPhotonLoader](s)
```

Listing 4: Cython-Wrapper für `PhotonLoader`
Siehe: ALGO/pyalgo.pyx [19]

Wie dort erkennbar, weist der Code starke Redundanz zu den existierenden C++-Klassen auf, da jede einzelne Klasse im Import deklariert und dann als Python-Klasse implementiert werden muss. Dies ist in Cython so gegeben und entsprechend nur unter Verwendung eines anderen Frameworks vermeidbar, jedoch eine einfache Lösung, insbesondere im Gegensatz zur Verwendung der C-API, welche ein Wrapping von Klassen ohne weiteres nicht unterstützt. Die zusätzliche Verwendung der C-API ermöglicht die direkte

Verwendung des Interpreters vom C++ Code. So kann die Zielfunktion für den Optimierungsalgorithmus in der `SplineSimulation` Klasse dynamisch vom Python Code gewählt werden (siehe Listing 5).

```
683 #ifdef PYTHON
684         if (_objective)
685         {
686             PyObject * result_object = PyObject_CallObject(
687                 _objective, Py_BuildValue("ffff", returned, detected, lost, radius_end));
688             result = value_t(PyFloat_AsDouble(result_object));
689         }
690         else
691         {
692 #endif
693             result = this->_generator->N() - detected;
694 #ifdef PYTHON
695         }
696 #endif
```

Listing 5: Aufruf der Python Funktion aus C++, wodurch Recompilierung vermieden wird. Siehe: ALGO/include/ALGO/SplineSimulation.h [19]

Insbesondere vereinfacht Cython das Einbinden von `numpy` [12] als Basis für die Ausgabe-Arrays, sodass keine Typkonvertierung in Form von neu erstellten Python-Objekten und den C++-Typen notwendig ist[5], sondern lediglich einfaches Typcasting (siehe Listing 6). So werden die Zeiger auf die existierenden `numpy`-Arrays neuinterpretiert und die Ergebnisse direkt über diese von der GPU kopiert.

```
222 void simulateData(value_t * hitpoints,    // incoming numpy float array
223                   value_t * results,      // incoming numpy float array
224                   int32_t * status_codes, // incoming numpy int8_t array
225                   index_t * result)       // incoming uint32
226 {
227     *result = this->simulate();
228
229     cudaMemcpy(reinterpret_cast<Vec3<value_t> *>(hitpoints), _hitpoints_dev,
230                sizeof(Vec3<value_t>) * _generator->N(), D2H);
231     CUERR;
232     cudaMemcpy(reinterpret_cast<Vec3<value_t> *>(results), _results_dev,
233                sizeof(Vec3<value_t>) * _generator->N(), D2H);
234     CUERR;
235
236     std::vector<int8_t> codes(_generator->N());
237     cudaMemcpy(codes.data(), _status_codes_dev, sizeof(int8_t) * _generator->N(), D2H);
238     CUERR;
239
240     for (size_t i = 0; i < codes.size(); ++i) //
241         status_codes[i] = static_cast<int32_t>(codes[i]);
242 }
```

Listing 6: *Kopieren* der Ergebnisdaten in `numpy`-Arrays. Siehe: ALGO/include/ALGO/Simulation.h [19]

[5] Eine Ausnahme bilden hier die Statuscodes, welche in `numpy.int32` konvertiert werden.

In Listing 7 findet sich ein Python Skript, welches die Schnittstelle verwendet, um eine einfache Simulaton des ALG durchzuführen. Dort werden initial Photonen aus einer Datei geladen, die quadrierte[6] Profilkurve (`falke(...)`-Funktion) mittels eines `QuadraticSpline` (siehe Abschnitt 2.2.2) approximiert, der GPU übergeben und in einer Instanz der Klasse `Simulation` simuliert. Die verwendeten Interfaces ermöglichen nun beispielsweise eine Verwendung der Klasse `CurvatureSpline`, welche die Splines mit vorgegebenen Krümmungen (siehe Abschnitt 2.2.3) realisiert, ohne einen Wechsel der Modell- oder Simulationsklasse durchzuführen. Die Nutzung der Python-Schnittstelle sorgt für ein schnelles Wechseln der Simulationsparameter oder Initialkurve, ohne den gesamten Code recompilieren zu müssen.

```
generator = PhotonLoader('test/tube.out')  # Load input data

# Initialize spline
spline = QuadraticSpline(10, Intervall(0, 6.01))
for i in range(spline.numKnots()):
    spline.specify(i, falke(4.5, spline.knotX(i)))
spline.generate()

# Initialize the corresponding (spline)-model
model = QuadraticSplineModel(4.3, 4.5)
model.configure(6.01, spline)

# Initialize the simulation class. ARGS: model, n1, n2, lambda_abs, lambda_sc, generator
simulation = Simulation(model, 1.58, 1.0, 1000, 1000000, generator)
hitpoints, angles, statuscodes, n_success = simulation.simulateData()
```

Listing 7: Beispielumsetzung eines einfachen Simulationsdurchlaufs des ALG in Python (Importe ausgelassen).
Siehe: ALGO/scripts/pyalgo_run.py [19]

3.6 Validierung der Softwareänderungen

Aufgrund der Umstrukturierung des WOMRaT [22] Quelltextes (siehe Kapitel 3) und dem Hinzufügen neuer Funktionalität (siehe Kapitel 2) ist eine Validierung der so entstandenen ALGO Software [19] erforderlich. Insbesondere ist zu prüfen, ob der neu entstandene Code unter gleichen Voraussetzungen gleiche oder zumindest ähnliche[7] Resultate erzielt.

Um diese Änderungen zu prüfen, wurde ein zuvor simuliertes Set von $\approx 1{,}7$ Millionen Photonen mit dem ursprünglichen Code vorsimuliert und im Folgenden in beiden Implementierungen (WOMRaT [22] und ALGO [19]) durch den ALG simuliert und im 3D-Plot (siehe Abbildung 3.5) verglichen.

[6] Nur so entsteht die Rotationsquadric für den Ray-Tracing-Algorithmus [21].
[7] Gleiche Ergebnisse können aufgrund von Randomisierung nicht ohne weitere (zu validierende) Anpassungen erzielt werden.

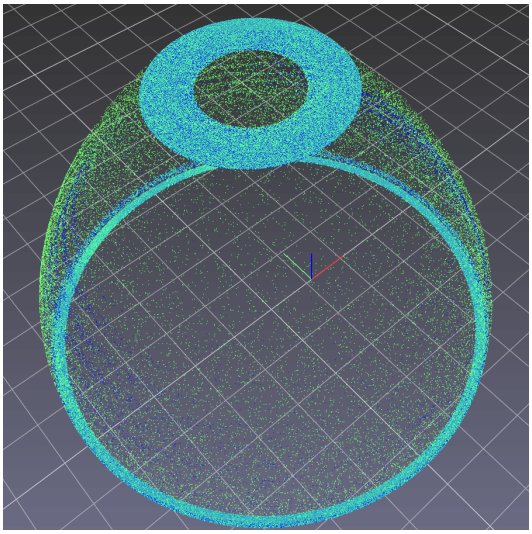


Abb. 3.5.: Positionen der Photonen im $\mathbb{R}^3$ nach der Simulationsdurchführung.
Blau: WOMRaT [22] - Grün: ALGO
Der Plot wurde unter Verwendung des pptk-Tookits [7] erstellt.

Auffällige Unterschiede finden sich nur entlang des Kurvenverlaufs. Hier ist jedoch die Photonendichte besonders gering (siehe auch Kapitel 4), wodurch sich keine Farbmischungen (Kathodenfläche und Startfläche türkisfarben) ausbilden. Es kann also davon ausgegangen werden, dass die Software korrekt agiert.

Um sicherzustellen, dass die verwendete Implementierung des Nelder-Mead Simplex Algorithmus (siehe Abschnitt 3.3) korrekt arbeitet, wurden mehrdimensionale (glatte) Funktionen, deren Minimum bekannt ist, durchsimuliert und die Ergebnisse verglichen. In jedem Durchlauf traten nur minimale Abweichungen[8] auf.

[8] Abweichungen im Bereich der Floating-Point Präzision, sofern die Toleranz entsprechend gewählt wurde.

4 Simulationsdurchführungen

> *Nothing happens until something moves.*
>
> — **Albert Einstein**
> 1879-1955

Die numerisch lokale Optimierung des ALG verfolgt die Idee der Formoptimierung durch Variation der einzelnen Parameter des formgebenden Splines.
Wie in Kapitel 2 besprochen, gibt es verschiedene Möglichkeiten einen Spline zu repräsentieren. Also gilt es, für alle Folgesimulationen einen vergleichbaren Rahmen zu schaffen. Dazu wurde für alle Simulationen ein vorsimuliertes Set [21] an Photonen und in allen Fällen, sofern nicht anders angegeben, folgende Argumente verwendet:

Tab. 4.1.: Übersicht der Simulationsargumente

Anzahl Photonen	1720535
Länge	6.01 cm
Äußerer Startradius	4.5 cm
Innerer Startradius	4.3 cm
Äußerer Endradius	$r(6.01\text{ cm})$ mit Falkekurve $r(x)$ und $r_0 = 4.5$ cm
Innerer Endradius	$\sqrt{r^2(6.01\text{ cm}) - r_0^2 + (4.3\text{ cm})^2}$ mit Falkekurve $r(x)$ und $r_0 = 4.5$ cm
Anzahl Stützpunkte	10
Brechungsindex n_1	1.0
Brechungsindex n_2	1.58
$\lambda_{absorption}$	10^3 cm
$\lambda_{scatter}$	10^5 cm
Verteilung der Stützstellen	äquidistant

$\lambda_{absorption}$ ist hier die Distanz, nach welcher Photonen im Mittel absorbiert werden. Diese ist aufgrund der Existenz von Photonen ohne Vortrieb erforderlich. $\lambda_{scatter}$ ist die Distanz, nach welcher Photonen im Mittel streuen. Diese findet im Durchschnitt mit dem gegebenen Argument unter den ≈ 1.7 Millionen Photonen ≈ 300 Mal statt. Auch sei für den inneren Kurvenverlauf jeweils die querschnittsflächenerhaltende Kurve mit Hinblick auf Erhaltung der Étendue [5] angenommen. Die im Folgenden relevanten Werte für $n_{success}$, $n_{returned}$, *etc.* (Zahl der Photonen mit entsprechendem Statuscode) wurden mittels Atomics unter Verwendung von Warp-Intrinsics [18] direkt auf der GPU berechnet.

Als Vergleichswerte für die folgenden Simulationen sind die Ergebnisse für die Falkekurve gegeben mit: $n_{returned} = 1101329$, $n_{success} = 568975$ und $n_{lost} + n_{absorbed} = 50230$.

4.1 Variation der äußeren Profilkurve

Ziel der in diesem Abschnitt behandelten Simulationen ist es, die Kurve des ALG nur durch Variation der äußeren formgebenden Kurve zu optimieren. Dabei wurde der erste und letzte Stützwert jeweils festgehalten, um einer Degeneration[1] vorerst vorzubeugen.

4.1.1 Batch Simulationen

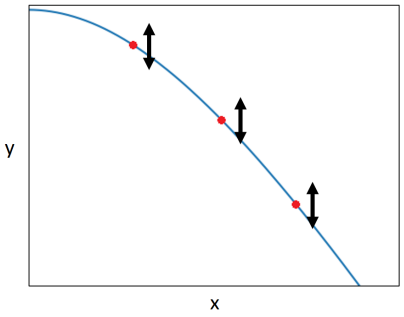
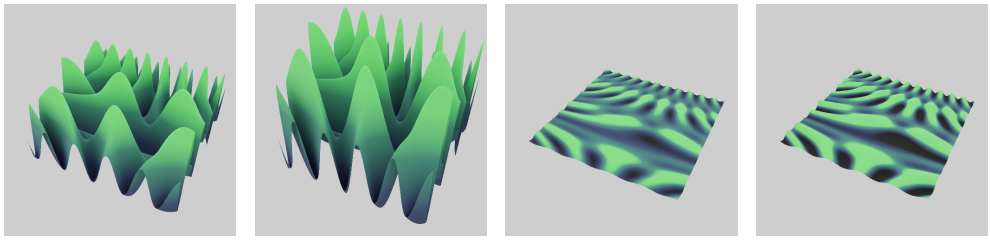


Abb. 4.1.: Skizzierung des naiven Ansatz - Variation der Stützwerte

Zu Beginn wurde für jede betrachtete Spline Interpolation (siehe Abschnitt 2.2) eine Batch Simulation durchgeführt. Dazu wurden die formgebenden Parameter variiert (siehe Abbildung 4.1) und so der Funktionenraum und die Zielfunktion, die Maximierung von $n_{success}$, um die Falkekurve abgetastet. Dies geschah mit der Intention, ein Gefühl die Funktionenräume bezüglich der lokalen Optimierungen zu schaffen. Zusätzlich wurden für die Kurvenverläufe Histogramme angefertigt, die zeigen in welchem Definitionsbereich der Kurve Photonen die Simulation mit Statuscode $lost$ oder $absorbed$ beenden (n_{lost}). Dies wurde betrachtet, um eventuell erste Ansätze für eine weiterführende analytische Optimerung zu finden. In Abbildung 4.2 ist die Zielfunktion auf den vermuteten Funktionenräumen, gegeben der einzelnen Ansätze, schematisch dargestellt. Die Ansätze werden im Folgenden diskutiert.



(a) Naiver Ansatz (b) Relativer Ansatz (c) Ansatz mit Variation der Krümmung (d) Ansatz mit Variation der Steigung

Abb. 4.2.: Schematische Darstellung der Zielfunktion (Maximierung von $n_{success}$) auf den vermuteten Funktionenräumen bezüglich der einzelnen Ansätze

Naiver Ansatz

Die Stützwerte wurden durch Setzen von $y_i := y_{i,initial} + k \cdot \delta$ für ein festes $\delta \in [0.01, 0.1, 0.2]$ mit $k \in [-5, 5] \subset \mathbb{Z}$ variiert. Zugleich wurde die Anzahl der Stützpunkte verringert, um den Simulationsaufwand zu verringern (siehe Tabelle 4.2). In Abbildung 4.3 sind Plots der besten (blau) und schlechtesten (orange) Kurve bezüglich der Zielfunktion zu sehen. Die-

[1] Gemeint ist eine Konvergenz zur konstanten Funktion, welche dem WOM-Rohr gleichzusetzen wäre.

se legen nahe, dass es sich bei der Falkekurve um ein lokales Optimum handelt. Jedoch lässt sich mit Betrachtung der Überlagerung aller Plots (siehe Abbildung 4.4) und dem korrespondierenden Lost-Histogramm erkennen, dass nebenliegende Kurven aufgrund der unregelmäßigen Krümmung ein sehr schlechtes Ergebnis liefern. Durch einfache Variation der Stützwerte lässt sich demnach kein brauchbarer Funktionenraum für eine Optimierung finden, da dieser für die Zielfunktion betragsmäßig große Gradienten um ein lokales Optimum aufweist (siehe Abbildung 4.2a). Die Folge ist, dass lokale Optimierer an der Ausgangsposition verharren, da eine möglicherweise bessere Lösung in allen Richtungen durch umliegende Minima unerreichbar wird. Die Simulationsergebnisse für $\delta = 0.2$ und $\delta = 0.01$ finden sich im Anhang B.1.

Tab. 4.2.: Geänderte Simulationsargumente von Tabelle 4.1 für die Batchsimulation mit naiver Variation der Stützwerte

Anzahl Stützpunkte	5

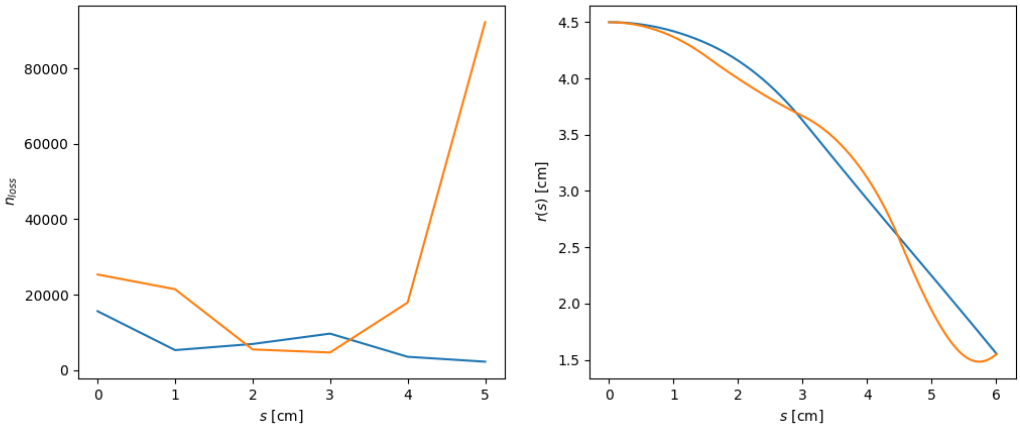


Abb. 4.3.: Plots der besten (blau) und schlechtesten Kurve (orange) mit $\delta := 0.1$ (r) mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen.

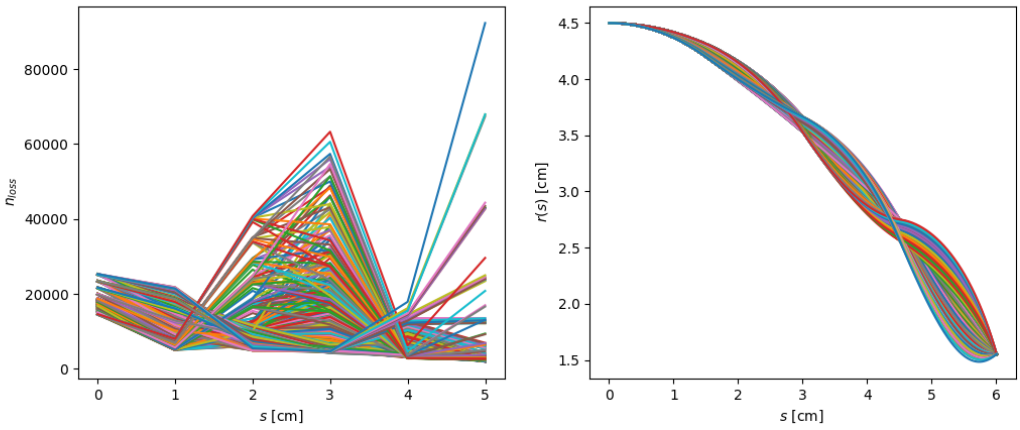


Abb. 4.4.: Plots aller Kurven mit $\delta := 0.1$ (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen.

Ansatz mit relativen Stützwerten

Ein Ansatz den entstandenen Funktionenraum und somit die Zielfunktion zu glätten, bildete die relative Repräsentation der Stützwerte. Dabei wurde jeder Stützwert als Anteil des vorherigen Stützwertes repräsentiert und im Folgenden dieser Anteil variiert.
Der niedrigste Wert jedes Stützwertes ergibt sich aus dem relativen Wert von Startradius zu Endradius und wurde fortlaufend mit 0.1 erhöht, solange dieser einen Wert kleiner eins aufwies. Leider führte diese Herangehensweise zu einem mitunter schlechteren Funktionenraum (siehe auch Abbildung 4.2b) für die Optimierung (siehe Abbildungen 4.5 und 4.6) als im naiven Ansatz.

Tab. 4.3.: Geänderte Simulationsargumente von Tabelle 4.1 für die Batchsimulation mit relativer Variation der Stützwerte

Anzahl Stützpunkte	5

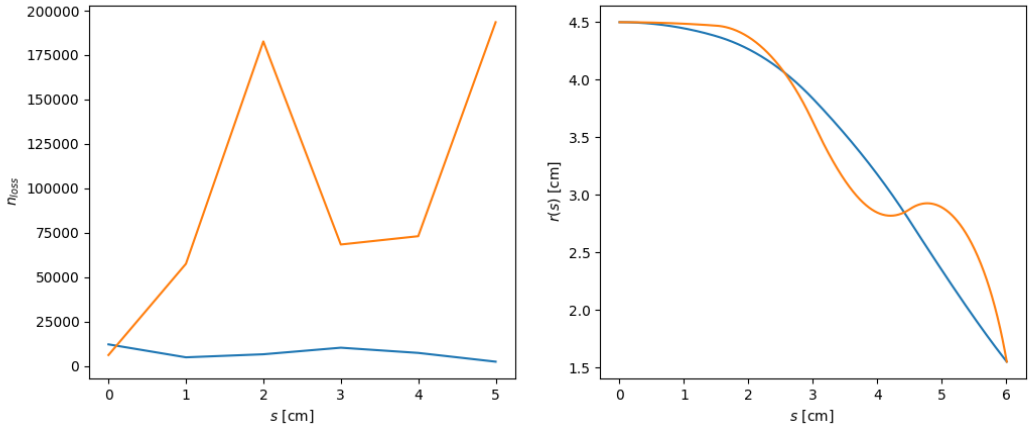


Abb. 4.5.: Plots der besten (blau) und schlechtesten Kurve (orange) (r) mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen.

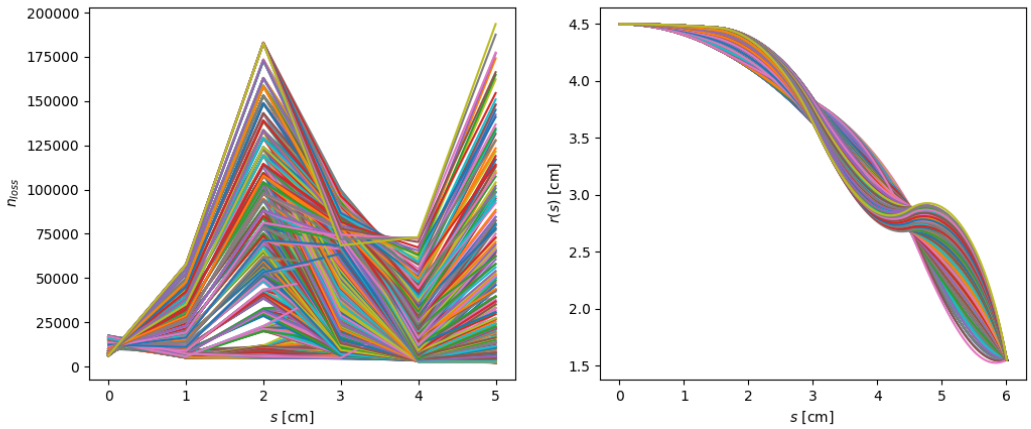


Abb. 4.6.: Plots aller Kurven (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen.

Ansatz mit Variation der Krümmungen

Für diesen Ansatz wurden initiale Krümmungen (siehe Abschnitt 2.2.3) vorgegeben und durch Setzen von $a_i := a_{i,initial} + k \cdot \delta$ für ein festes $\delta \in \{0.01, 0.05, 0.1\}$ mit $k \in [-5, 5] \subset \mathbb{Z}$ variiert. Diese wiesen stets eine gleichmäßige gekrümmte Form auf. Dies hat allerdings zur Folge, dass die Zielfunktion (siehe auch Abbildung 4.2c) auf dem entstandenen Funktionenraum sehr flach wird und keine signifikanten Optima auftauchen. So sind die Werte für $n_{success}$, der Anzahl der Kathodentreffer, für die beste und schlechteste Kurve 572405 und 497335 im Simulationsdurchlauf zwar nicht gleich, jedoch weichen diese im statistischen Sinne nur wenig voneinander ab (siehe Abbildungen 4.7 und 4.8 für Plots der Kurven). Ergebnisse für $\delta \in \{0.1, 0.01\}$ finden sich im Anhang B.1.

Tab. 4.4.: Geänderte Simulationsargumente von Tabelle 4.1 für die Batchsimulation mit Variation der Krümmungen

Anzahl Stützwerte	Keine
Anzahl Krümmungen	5

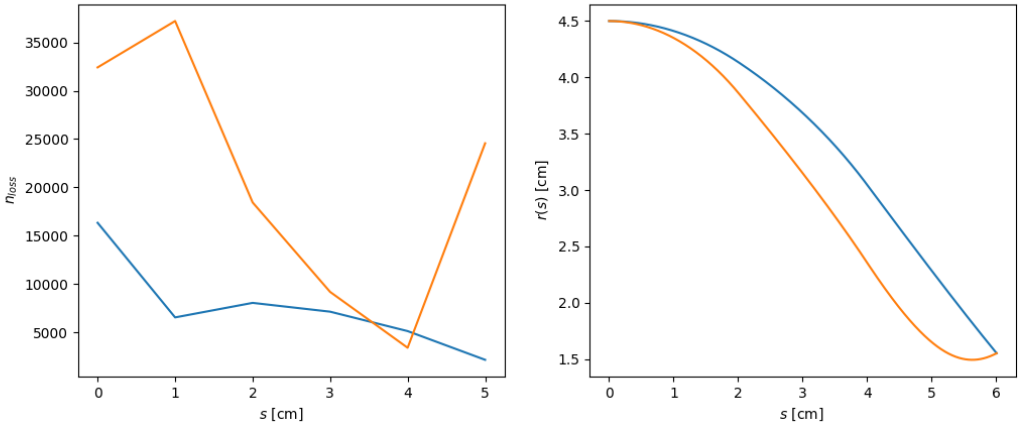


Abb. 4.7.: Plots der besten (blau) und schlechtesten Kurve (orange) (r) mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen. $\delta = 0.05$

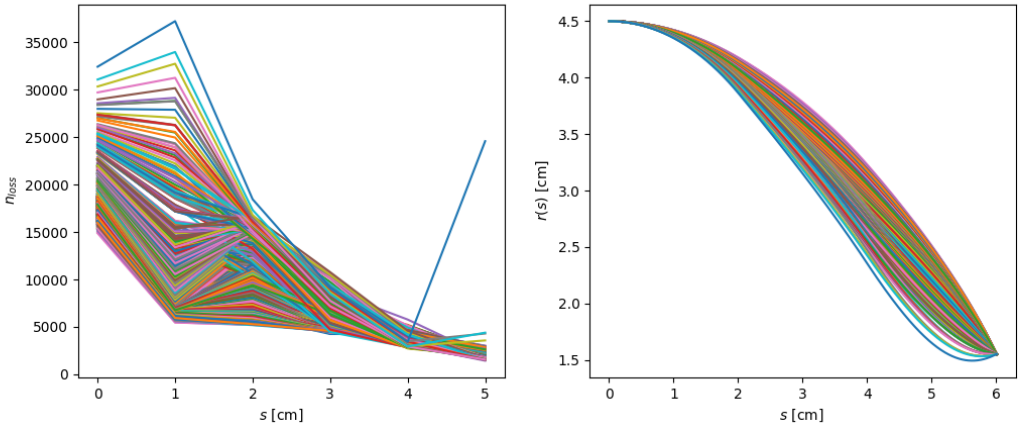


Abb. 4.8.: Plots aller Kurven (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve,welche Zahl an Photonen verloren wird. $\delta = 0.05$

Ansatz mit Variation der Steigungen

Eine ähnlich gute Repräsentation eines Funktionenraumes liefern Splines mit vorgegeben Steigungen (siehe Abschnitt 2.2.4). Diese wurden durch Setzen von $b_i := b_{i,initial} + k \cdot \delta$ für ein festes $\delta = 1$ mit $k \in [-5,5] \subset \mathbb{Z}$ variiert. In Abbildung 4.10 kann jedoch anhand vertikaler Linien erkannt werden, wo sich die Stützstellen des Splines befinden. Es bildeten sich zugleich sich extreme Degenerationen, wie nicht kontinuierliche Profilkurven wie die Schlechteste aus Abbildung 4.9, weshalb diese eine geringere Stabilität im anschließenden Optimierungsverfahren aufwiesen.

Tab. 4.5.: Geänderte Simulationsargumente von Tabelle 4.1 für die Batchsimulation mit Variation der Steigungen

Anzahl Stützpunkte	5
Anzahl Steigungen	3

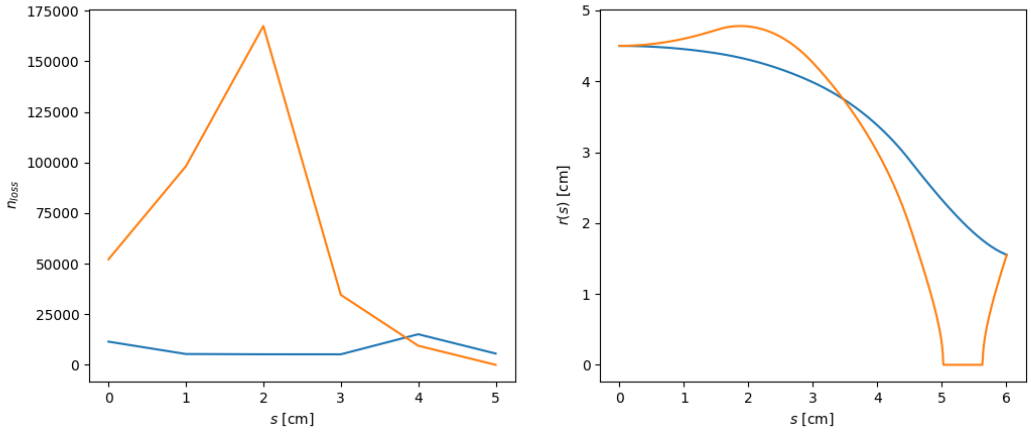


Abb. 4.9.: Plots der besten (blau) und schlechtesten Kurve (orange) (r) mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen. $\delta = 1$

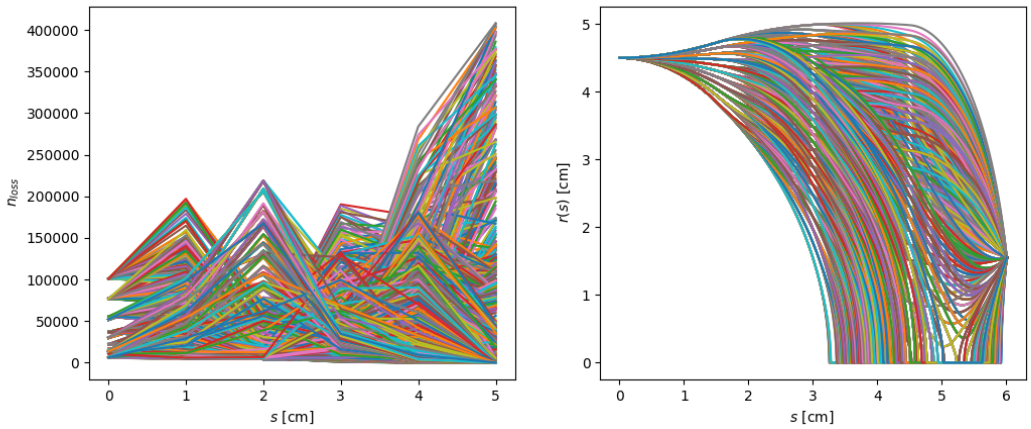


Abb. 4.10.: Plots aller Kurven (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen. $\delta = 1$

4.1.2 Optimierung mittels Simplex Algorithmus

Die Wahl der Zielfunktion kann, wie bereits in Abschnitt 2.1 erwähnt, sehr viele variable Parameter aufweisen, denn ein Simulationsdurchlauf liefert mehrere Informationen: Die Anzahl der erkannten Photonen ($success$), die Anzahl der verlorenen Photonen ($lost$) und die Anzahl der zurücklaufenden Photonen ($returned$). Und selbst diese sind nur ein kleiner Teil der Simulationsausgabe.

Betrachtet wurden die **Zielfunktionen**, die a) n_{lost} minimieren, b) $n_{returned}$ maximieren und c) $n_{success} = n_{detected}$ maximieren.
Nach n_{lost} wurde insbesondere optimiert um eine verlustfreie Kurvenform zu identifizieren. $n_{returned}$ wurde maximiert, um zu prüfen, ob eine Form existiert, welche nahezu alle Photonen zurück in das WOM-Rohr leitet.
Um weiterhin Vergleichbarkeit zu erhalten, wurde jede lokale Optimierung jeweils von der Falkekurve und der Parabelkurve, welche den gleichen Endradius erzeugt und waagerechte Tangente am y-Achsenabschnitt besitzt, gestartet.
Der initiale Simplex wurde für alle Simulationsoptimierungen gemäß einer Ausgangslösung mit den Eingabeparametern $q_0, ..., q_m$ mit Kantenlänge $\lambda_{simplex}$ initialisiert [14]. Demnach gilt für die Parameter p_j mit $j \in \underline{m}$ des t-ten Eintrag des Simplex ($t \in \underline{m+1}$):

$$l\left[p_j := q_j - \lambda_{simplex} \cdot \delta_{jt}\right]_t$$

δ_{jt} ist hier das Kronecker-Delta, welches für $t = j$ eins und sonst null liefert. Das Argument τ aus Tabelle 4.6, welche die nun insgesamt zusätzlichen Argumente auflistet, ist dabei die Toleranz, nach welcher der Algorithmus terminiert. Weist der Simplex für die zwei besten Kurven nur einen Abstand von τ auf, so terminiert er.

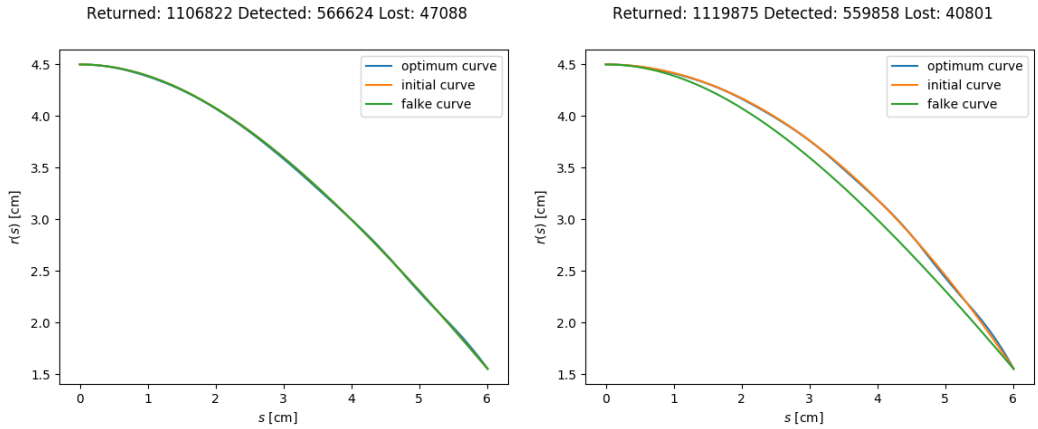
Tab. 4.6.: Übersicht der zusätzlichen Simulationsargumente für die Simplexoptimierung

$\lambda_{simplex}$	0,4
τ	100

In allen gezeigten Abbildungen finden sich die aufsummierten Werte für $n_{success}$ als *Detected*, $n_{returned}$ als *Returned* und n_{lost} mit $n_{absorbed}$ als *Lost*.

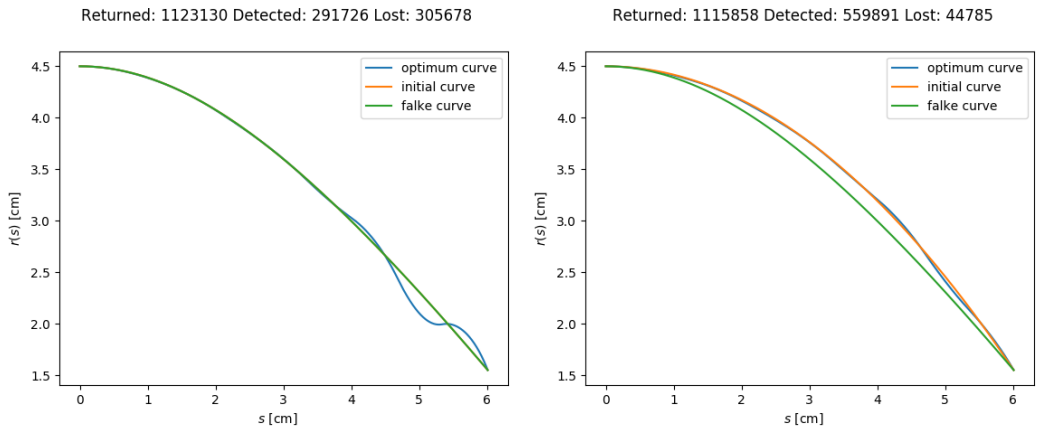
Naiver Ansatz

Für den naiven Ansatz wurden die Stützwerte (ausgenommen des Ersten und Letzten) als freie Simulationsparameter p_i (siehe Abschnitt 2.1) gewählt.
Nach den Beobachtungen in Abschnitt 4.1.1 sollte sich hier als optimale Lösung für die Profilkurve die Ausgangskurve oder eine sehr nah an der Ausgangskurve liegende Funktion aufgrund der Beschaffenheit des Funktionenraumes finden. Unter Betrachtung der Abbildungen 4.11, 4.12 und 4.13 kann dies auch festgestellt werden. Es ist ein weiteres Indiz dafür, dass der Funktionenraum auf den gegebenen Splines für eine lokale Optimierung zu uneben ist.



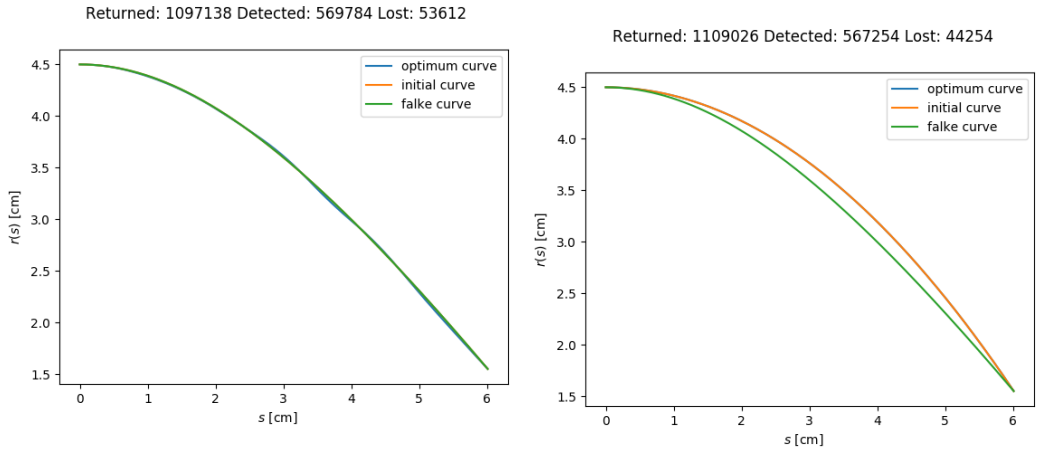
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. 4.11.: Naive Optimierung nach n_{lost}



(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. 4.12.: Naive Optimierung nach $n_{returned}$

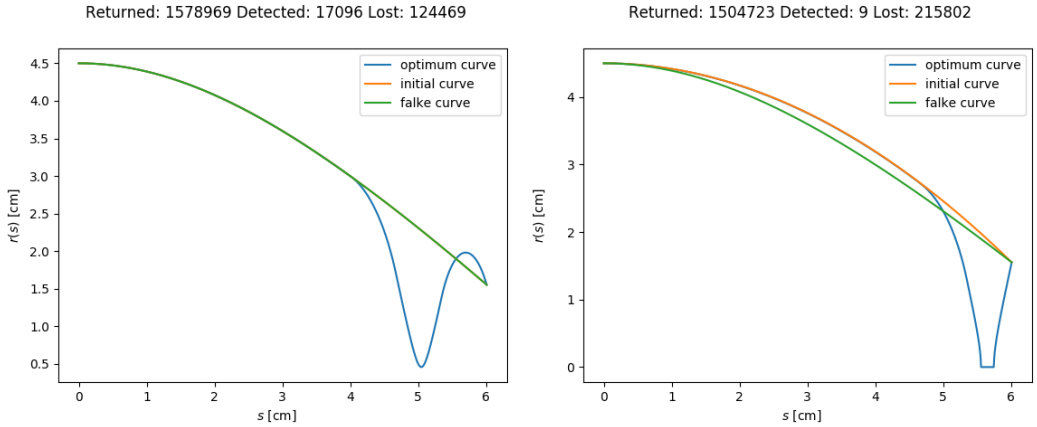


(a) Start von Falkekurve

Abb. 4.13.: Naive Optimierung nach $n_{success}$

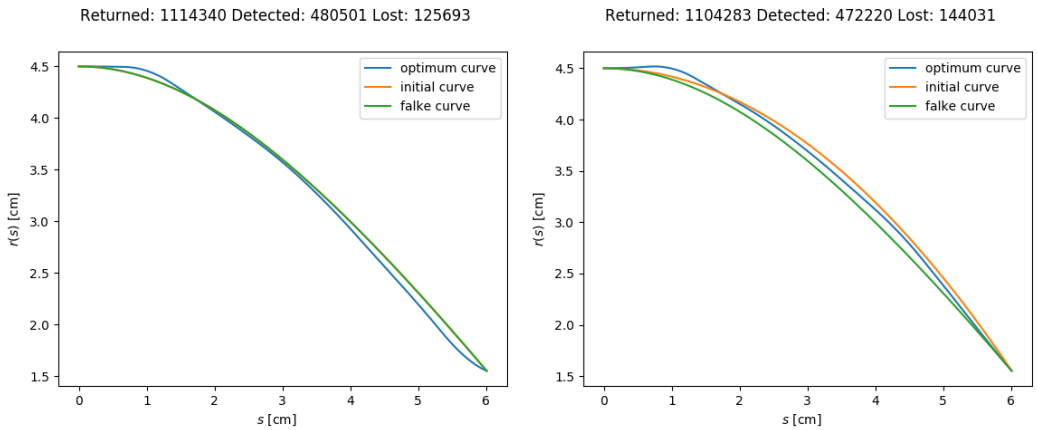
Ansatz mit relativen Stützwerten

Unter Verwendung relativer Stützwerte, also der Repräsentation der freien Simulationsparameter p_i als prozentuale Anteile der vorherigen Stützwerte, stellte sich ein ähnliches Problem wie beim naiven Ansatz ein, nämlich der Terminierung an der Ausgangskurve (siehe Abbildung 4.15). Der Optimierer fand jedoch eine gute Approximation für die Optimierung nach $n_{returned}$, die sich an eine Kreiskurve[(2)] (siehe Abbildung 4.14) anzulehnen scheint. Auf eine Optimierung nach n_{lost} wurde verzichtet.



(a) Start von Falkekurve (b) Start von Parabelkurve

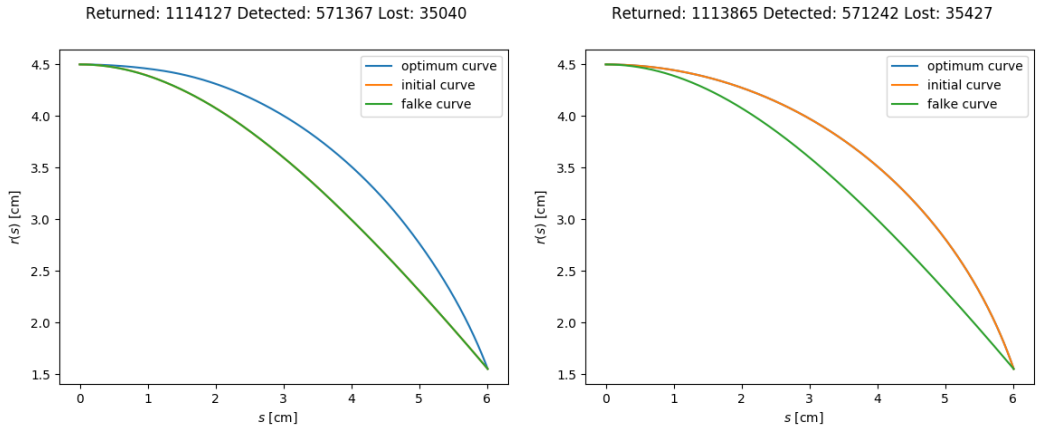
Abb. 4.14.: Optimierung mit relativen Splines nach $n_{returned}$



(a) Start von Falkekurve (b) Start von Parabelkurve

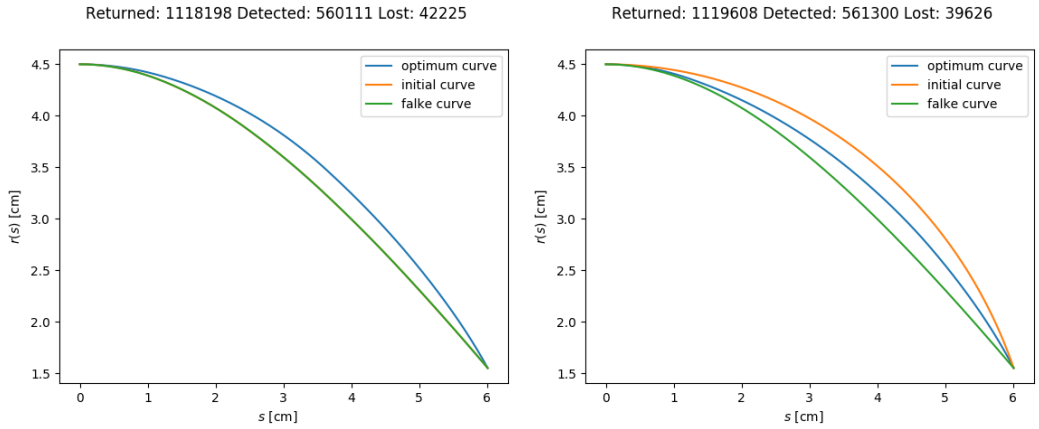
Abb. 4.15.: Optimierung mit relativen Splines nach $n_{success}$

[(2)] Kreis mit Mittelpunkt im Ursprung mit dem äußeren Startradius als Radius.



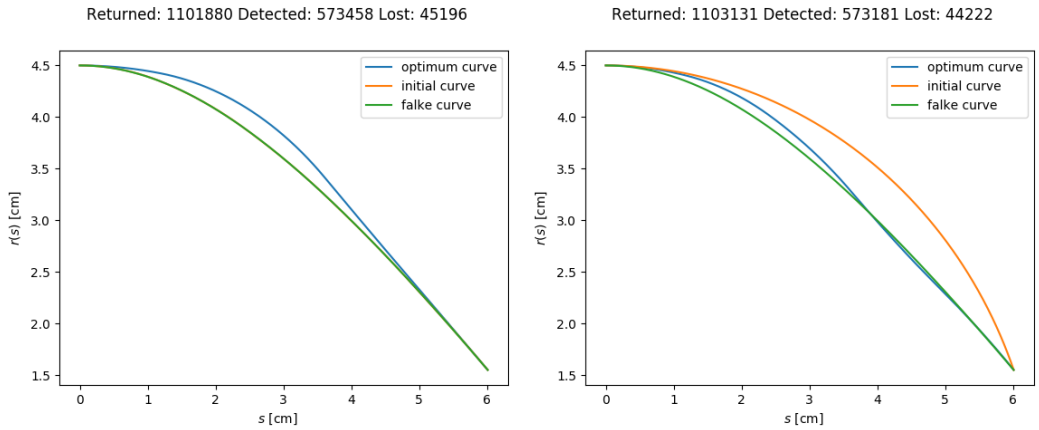
(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung.

Abb. 4.16.: Optimierung mittels Krümmung der Splines nach n_{lost}. Die blaue Kurve ist jeweils deckungsgleich mit der orangefarbenen Kurve.



(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.17.: Optimierung mittels Krümmung der Splines nach $n_{returned}$



(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.18.: Optimierung mittels Krümmung der Splines nach $n_{success}$

Ansatz mit Variation der Krümmung

Als Initialkurven werden zum einen die Krümmungen der quadrierten Falkekurve verwendet, zum anderen wird die konstante Kurve[3] verwendet ($a_i = -0.15$). Unter Verwendung dieser Repräsentation lieferte der Algorithmus mit der Zielfunktion in Abhängigkeit der gegebenen Krümmungen a_i für alle Ausgangskurven ein ähnliches Resultat (siehe Abbildungen 4.16, 4.17 und 4.18). Jedoch muss hier die numerische Stabilität in Frage gestellt werden, da die Zielfunktion auf dem Funktionenraum, wie bereits in Abschnitt 4.1.1 erwähnt, sehr flach ist und kaum Abweichungen der Ergebnisse auftreten. Das Simulationsargument $\lambda_{simplex}$ wurde betragsmäßig vergrößert, da der Algorithmus sonst mit dem initialen Simplex terminierte.

Tab. 4.7.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung mit variablen Krümmungen

Anzahl Stützwerte	Keine
Anzahl Krümmungen	5, variabel 5
$\lambda_{simplex}$	1

Ansatz mit Variation der Steigung

Unter Vorgabe von Steigungen an den äquidistanten Stützstellen zeigen sich deren Vor- und Nachteile gegenüber der Variante mit vorgegebenen Krümmungen. So konvergierten die Ergebnisse in den Abbildungen 4.19 und 4.21 zur gleichen Kurve. Diese weisen jedoch nicht die Glattheit der Kurven im obigen Ansatz (siehe Abschnitt 4.1.2) auf. Jedoch lieferte die Optimierung nach $n_{returned}$ (siehe Abbildung 4.20) eine Konvergenz zu zwei unterschiedlichen Kurven, was sich durch ein lokales Maximum zwischen diesen erklären lässt. Dieses Ergebnis legt nahe, dass die Kurve mit konstanter Krümmung ein möglicherweise besseres Ergebnis als die Falkekurve liefert. Insgesamt stellt sich jedoch auch hier die Frage nach der numerischen Stabilität, da es sich hier auch um einen sehr flachen Funktionenraum handelt und die Ergebnisse der beiden Kurven nur wenig voneinander abweichen.

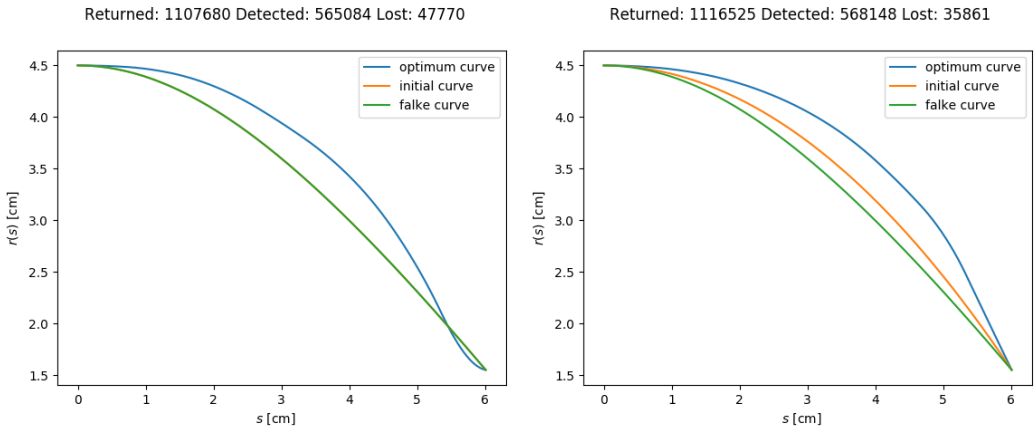
Tab. 4.8.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung mit variablen Steigungen

Anzahl Stützwerte	Keine
Anzahl Steigungen	10, variabel 8
$\lambda_{simplex}$	1

Zwischenfazit

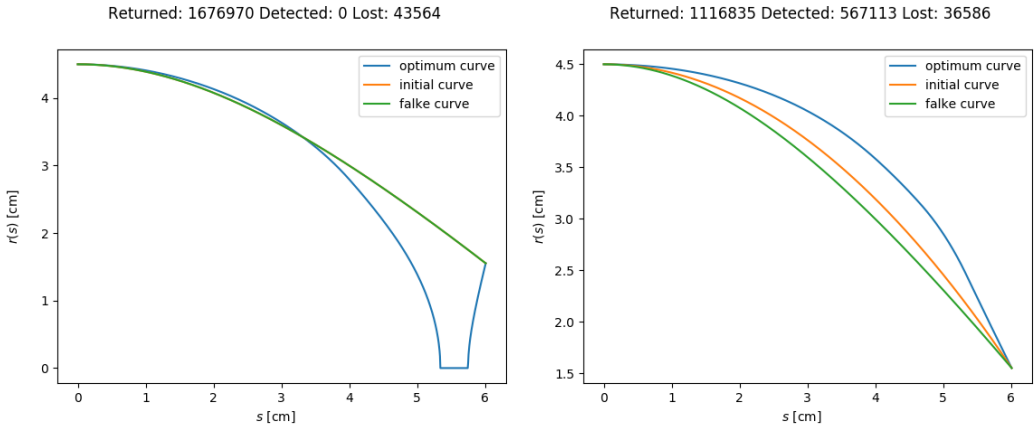
Die Verwendung von Splines mit vorgegebenen Krümmungen haben den glattesten Funktionenraum geliefert, jedoch besteht hier ein Problem des *Overfitting*, da einige Kurven entsprechend ausgeschlossen werden. So sind die Splines mit vorgegebenen Steigungen möglicherweise eine bessere Repräsentation des Suchraumes, allerdings weisen diese ein weniger stabiles Verhalten im gegebenen Algorithmus auf.

[3] Die konstante Kurve ist initial für alle Werte gleich, da das berechnete α diesen ausgleicht. Siehe dazu Anhang C.1.



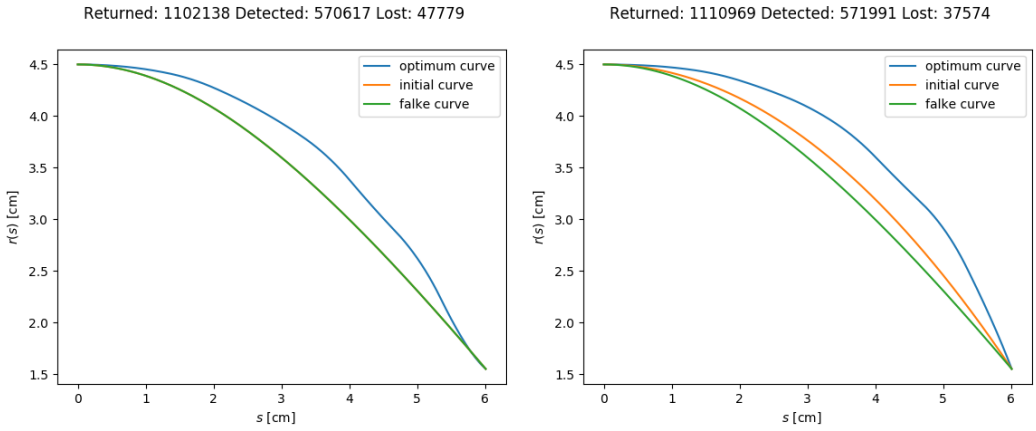
(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. 4.19.: Optimierung mittels Steigung der Splines nach n_{lost}.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. 4.20.: Optimierung mittels Steigung der Splines nach $n_{returned}$



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. 4.21.: Optimierung mittels Steigung der Splines nach $n_{success}$

4.2 Variation der inneren Profilkurve

Eine Variation der inneren Kurve wurde entgegen der geltenden Annahme, dass die Querschnittsflächenerhaltung gelten muss, in Betracht gezogen, um gegebenenfalls eine in der Praxis anwendbarere Lösung zu finden. Insbesondere weisen die verwendeten inneren Kurven stets gleich viele Stützwerte, Krümmungen oder Steigungen wie die korrespondierenden äußeren Kurven auf. Der Ansatz mit relativen Stützwerten wurde nicht verfolgt, da sich diese in den bisherigen Versuchen stets als schlechteste Variante herausgestellt haben.

4.2.1 Optimierung mittels Simplex Algorithmus unter Variation der inneren Kurve

Hier wurden die gleichen Zielfunktionen, der gleiche Intialsimplex und Simulationsargumente (siehe Tabelle 4.6) wie in Abschnitt 4.1.2 verwendet. Als Ausgangskurven dienten für die äußere Profilkurve die Falkekurve und die in Abschnitt 4.1.2 erwähnte Parabelkurve. Die innere Startkurve wurde stets flächenerhaltend [21] zur äußeren Kurve gewählt. Zusätzlich wurden wieder der Start- und Endradius der inneren Kurve und die äußere vollständig festgehalten.

Naiver Ansatz

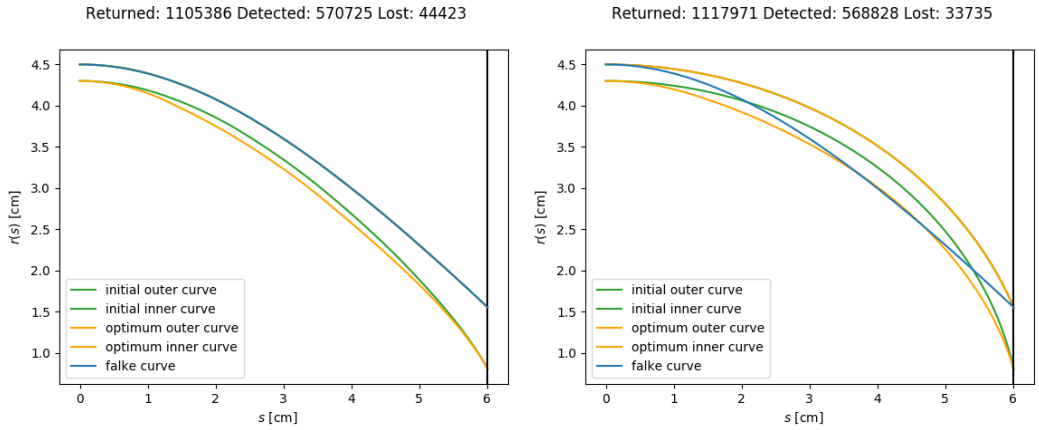
Wie bereits in der Variation der äußeren Profilkurve mit Erhaltung der Querschnittsfläche stellte sich unter Optimierung nach n_{lost}, $n_{returned}$ und $n_{success}$ (siehe Abbildungen B.9, B.10 und B.11 im Anhang) die Ausgangskurve als optimale Lösung ein. Auch hier lässt sich als Ursache die Unebenheit der Zielfunktion auf dem Funktionenraum finden.

Ansatz mit Variation der Krümmungen

Die Ergebnisse wiesen je nach Ausgangskurve eine starke bis zu keiner Abweichung von dieser auf (siehe Abbildungen 4.22, 4.23 und 4.24). Dies kann hier wieder auf die numerische Instabilität des Verfahrens zurückgeführt werden, da der Ergebnissraum sich als deutlich flacher herausgestelle als noch in Abschnitt 4.1.2. Die hier ermittelten Werte von $n_{success}$, $n_{detected}$ und $n_{returned}$ weisen nur Abweichungen von wenigen 1000 Photonen auf.

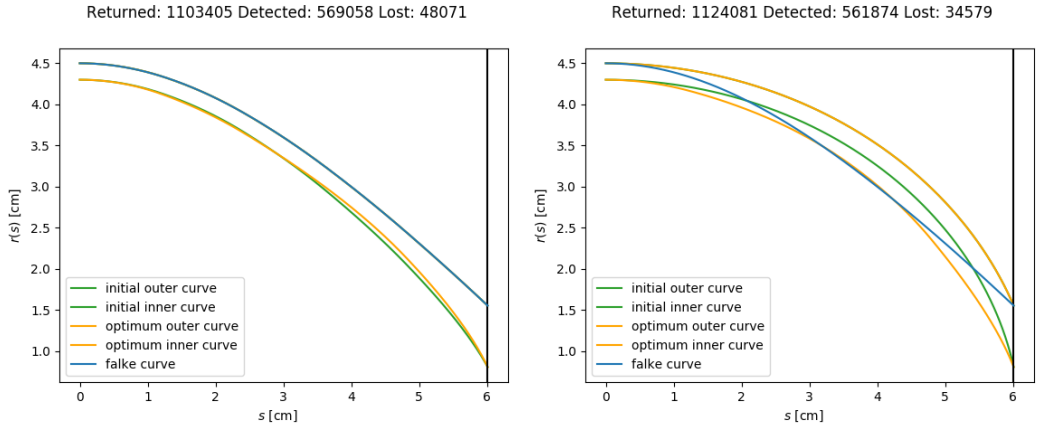
Tab. 4.9.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung (innere Kurve) mit variablen Krümmungen

Anzahl Stützwerte	Keine
Anzahl Krümmungen	5, variabel 5
$\lambda_{simplex}$	1



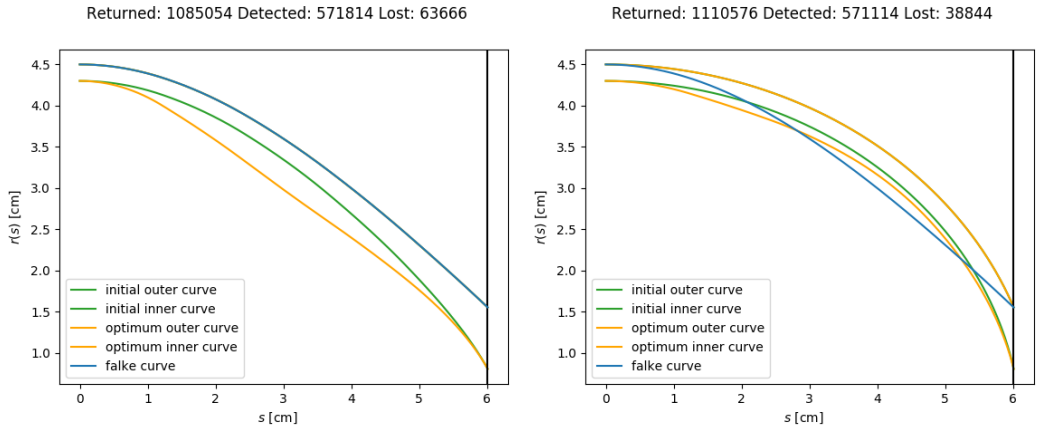
(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.22.: Optimierung mittels Krümmung der Splines nach n_{lost} mit Variation der inneren Kurve.



(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.23.: Optimierung mittels Krümmung der Splines nach $n_{returned}$ mit Variation der inneren Kurve.



(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

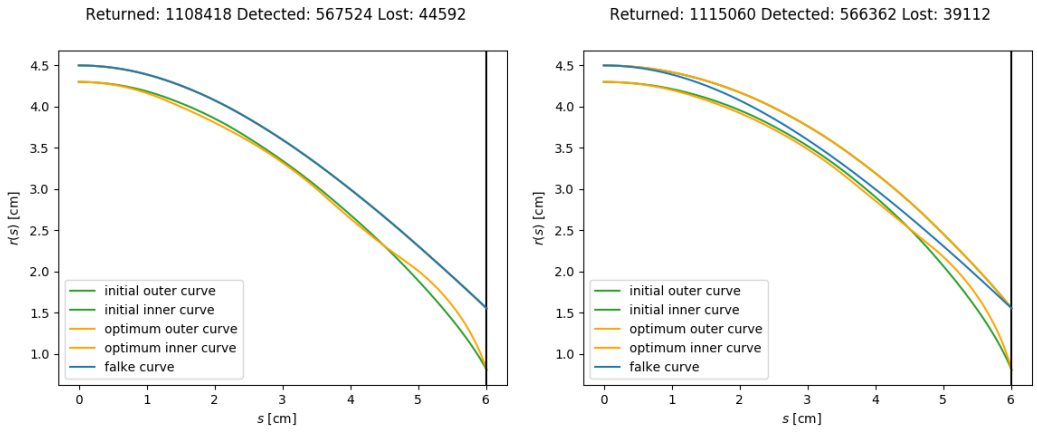
Abb. 4.24.: Optimierung mittels Krümmung der Splines nach $n_{success}$ mit Variation der inneren Kurve.

Ansatz mit Variation der Steigungen

Die ermittelte innere Kurve zeigt sich nahezu deckungsgleich zur Ausgangskurve. Auch ist die numerische Stabilität anzuzweifeln, da die Zielfunktion auf dem Funktionenraum sehr flach ist. Die Resultate könnten lediglich als Indiz gewertet werden, dass die Flächenerhaltung gelten muss (siehe Abbildungen 4.25, B.12 (im Anhang) und 4.26).

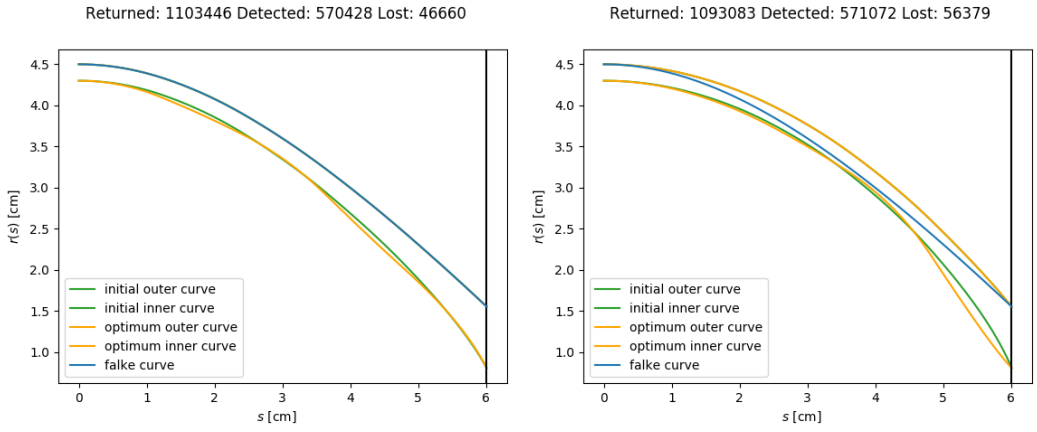
Tab. 4.10.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung (innere Kurve) mit variablen Steigungen

Anzahl Stützpunkte	Keine
Anzahl Steigungen	10, variabel 8
$\lambda_s implex$	1



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. 4.25.: Optimierung mittels Steigung der Splines nach n_{lost} mit Variation der inneren Kurve.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. 4.26.: Optimierung mittels Steigung der Splines nach $n_{success}$ mit Variation der inneren Kurve.

4.2.2 Optimierung mittels Simplex Algorithmus unter Variation der inneren und äußeren Kurve

Hier wurden die gleichen Zielfunktionen und der gleiche Intialsimplex wie in Abschnitt 4.1.2 verwendet. Als Ausgangskurven dienten hier als äußere Profilkurve die Falkekurve und die dort erwähnte Parabelkurve. Die innere Startkurve wurde stets flächenerhaltend [21] zur äußeren Kurve gewählt. Auch hier wurden Start- und Endradius beider Kurven festgehalten.

Naiver Ansatz

Wie zuvor konvergiert der naive Ansatz mit leichten Abweichungen zur Ausgangskurve (siehe Abbildungen B.13, B.14 und B.15 im Anhang). Lediglich unter Optimierung nach $n_{returned}$ stellt sich für den Start von der Falkekurve eine Abweichung ein. Diese kann darauf zurückgeführt werden, dass nur ein Photonenset für die Simulation verwendet wurde und der Algorithmus entsprechend die Kurve auf dieses Set optimierte.

Ansatz mit Variation der Krümmungen

Unter Optimierung nach $n_{returned}$ scheint eine Konvergenz in Richtung der konstanten Kurve aufzutreten (siehe Abbildungen 4.27, 4.28 und 4.29). In den anderen Fällen stoppt die Optimierung in der Nähe der Ausgangskurve. Das kann darauf zurückgeführt werden, dass die Flächenerhaltung durch Variation der inneren Kurve nicht erfüllt ist und so nur minimale Änderung möglich ist. Es entsteht also auch hier eine unebene Zielfunktion auf dem Funktionenraum, welcher den Lösungsraum nicht gut abbildet beziehungsweise enthält.

Tab. 4.11.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung (innere & äußere Kurve) mit variablen Krümmungen

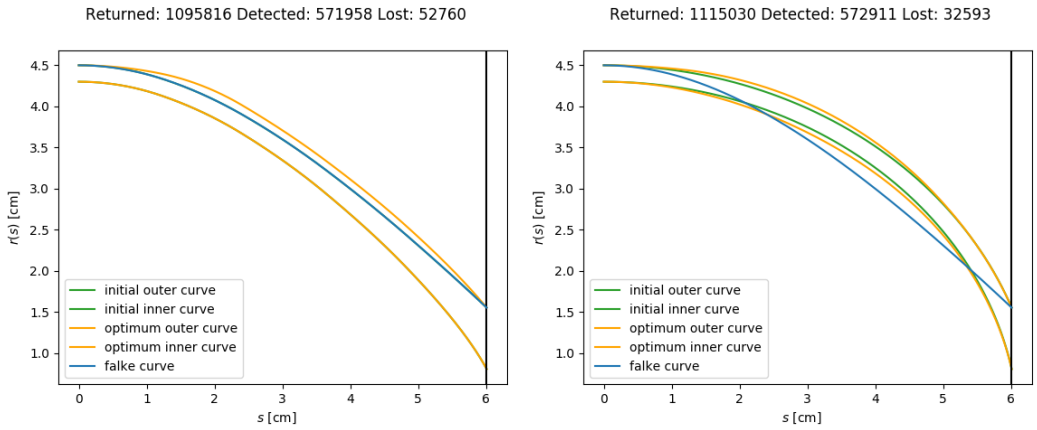
Anzahl Stützwerte	Keine
Anzahl Krümmungen	6 bzw. 12, variabel 6 bzw. 12
τ	1

Ansatz mit Variation der Steigungen

In allen Simulationsdurchläufen haben sich die neugefundenen Kurven im Vergleich zur Ausgangskurve kaum oder nicht verändert (siehe Abbildungen B.16, B.17 und B.18 im Anhang). Dies kann auf die fehlende Flächenerhaltung zurückgeführt werden, da so erneut eine unebene Zielfunktion auf dem Funktionenraum entsteht.

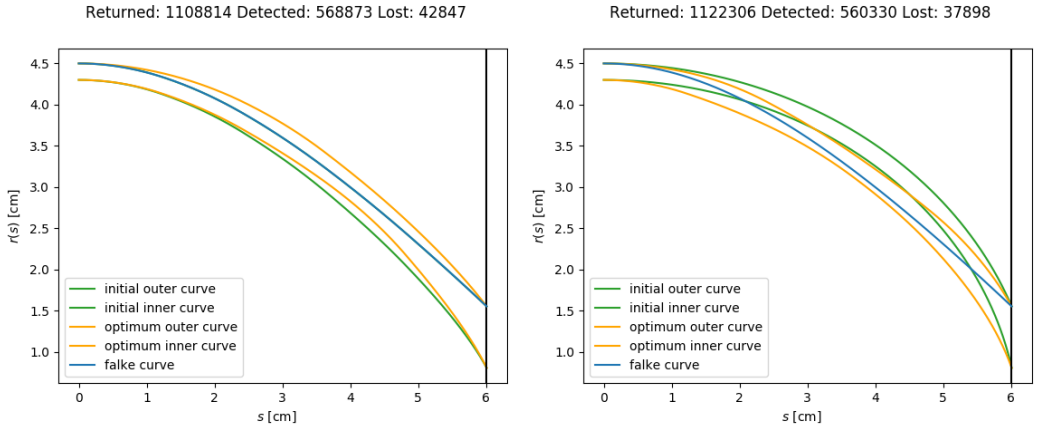
Tab. 4.12.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung (innere & äußere Kurve) mit variablen Steigungen

Anzahl Stützwerte	Keine
Anzahl Steigungen	10 bzw. 20, variabel 8 bzw. 16
τ	1



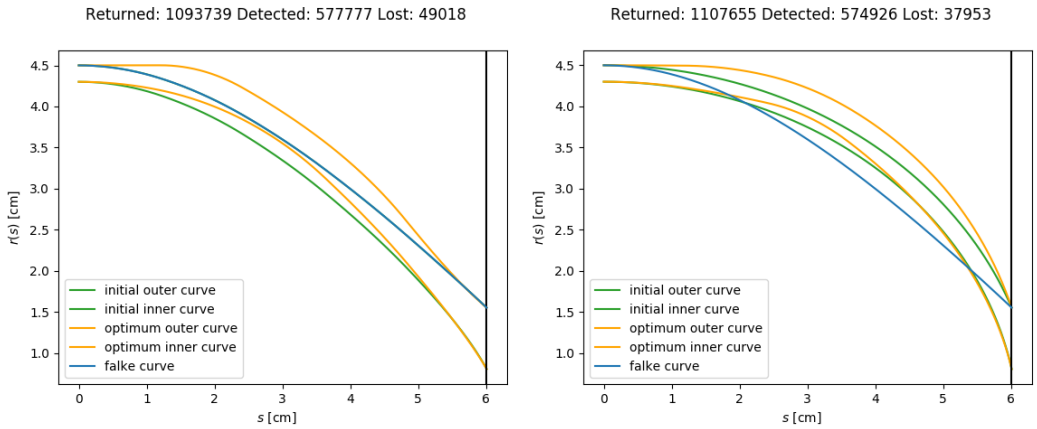
(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.27.: Optimierung mittels Krümmung der Splines nach n_{lost} mit Variation der inneren und äußeren Kurve.



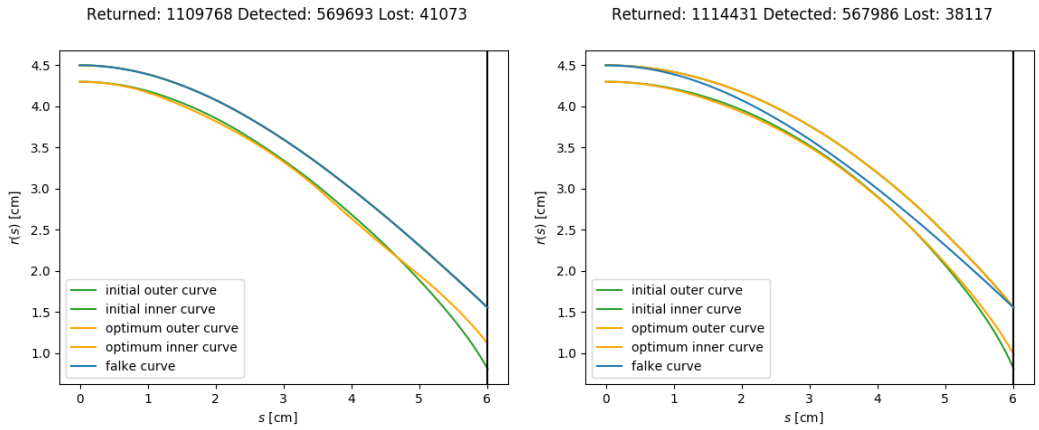
(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.28.: Optimierung mittels Krümmung der Splines nach $n_{returned}$ mit Variation der inneren und äußeren Kurve.



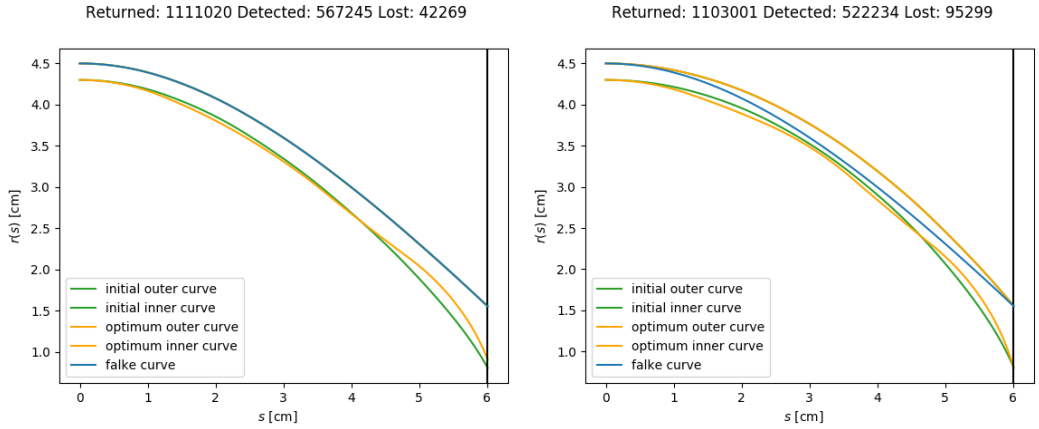
(a) Start von Krümmungen der Falkekurve (b) Start von konstanter Krümmung

Abb. 4.29.: Optimierung mittels Krümmung der Splines nach $n_{success}$ mit Variation der inneren und äußeren Kurve. In (a) ist $n_{success}$ der größte beobachtete Wert.



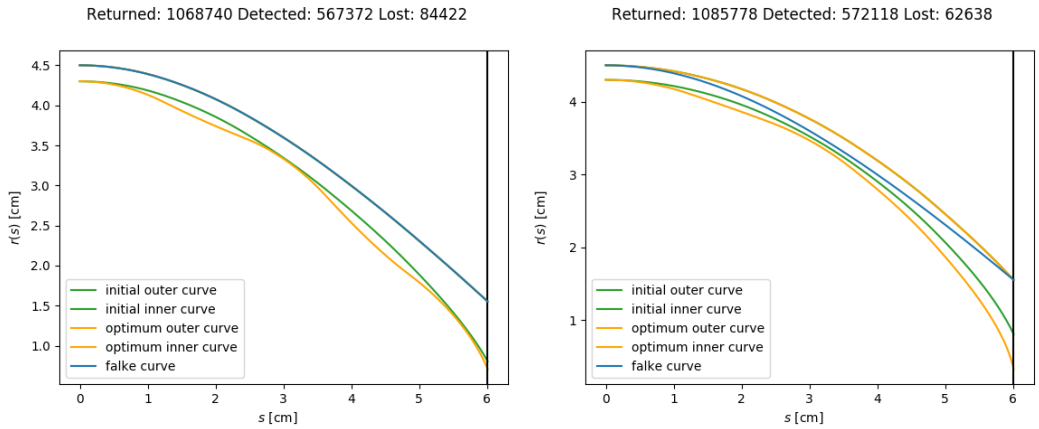
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. 4.30.: Naive Optimierung nach n_{lost} mit Variation der inneren Kurve und des inneren Endradius.



(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. 4.31.: Naive Optimierung nach $n_{returned}$ mit Variation der inneren Kurve und des inneren Endradius.



(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. 4.32.: Naive Optimierung nach $n_{success}$ mit Variation der inneren Kurve und des inneren Endradius.

4.2.3 Optimierung mittels Simplex Algorithmus unter Variation der inneren Kurve und des inneren Endradius

Die gleichen Zielfunktionen und der gleiche Intialsimplex wie in Abschnitt 4.1.2 wurden verwendet. Als Ausgangskurven dienten ebenso als äußere Profilkurve die Falkekurve und die in Abschnitt 4.1.2 erwähnte Parabelkurve. Die innere Startkurve wurde stets flächenerhaltend [21] zur äußeren Kurve gewählt.

Naiver Ansatz

Auffallend ist hier, dass sowohl eine Verengung als auch eine Erweiterung des inneren Endradius auftaucht (siehe Abbildung 4.30, 4.31 und 4.32). Überwiegend stellte sich jedoch keine große Veränderung der initialen Kurve, als auch Veränderung im Simulationsergebnis, ein.

Ansatz mit Variation der Krümmungen

Dieser Ansatz konnte in der bestehenden Implementierung nicht durchgeführt werden, da eine entsprechende Schnittstelle zur Änderung des Endradius (noch) fehlt.

Ansatz mit Variation der Steigungen

Hier traten nur Verengungen des Endradius auf (siehe Abbildungen B.19, B.20 und B.21 im Anhang). Das Ergebnis wich jedoch wie in den bisherigen Versuchen nur minimal von der Ausgangskurve ab.

Tab. 4.13.: Geänderte Simulationsargumente von Tabelle 4.1 und Tabelle 4.6 für die Simplexoptimierung (innere Kurve & innerer Endradius) mit variablen Steigungen

Anzahl Stützwerte	Keine
Anzahl Steigungen	10, variabel 9
τ	1

Zwischenfazit

Aus den Simulationen unter Variation der inneren Kurve ließen sich lediglich Indizien für die Flächenerhaltung nachweisen. Insbesondere ist das Optimierungsverfahren für das gegebene Teilproblem wohl nicht geeignet.

4.3 Photonenbahnen

Unter Betrachtung einiger Photonenbahnen entstand die Annahme, dass Photonen sich stets in einer Ebene bewegen und es so aufgrund der Eingangswinkel (siehe Abschnitt 4.4) für entsprechende Photonen nicht möglich ist, die Kathode zu schneiden, sofern dessen Ebene die Kathode nicht schneidet. Diese Vermutung konnte jedoch widerlegt werden. Es wurden 100 Photonenbahnen nachverfolgt und deren Ebenengleichung ermittelt. Für je 2 disjunkte Positionspaare wurden die Verbindungsvektoren berechnet und mittels Kreuzprodukt die Normale der aufspannenden Ebene ermittelt. Damit konnte mittels der Normalengleichung einer Ebene $(x-p)^T n = 0$ getestet[(4)] werden, ob diese koplanar sind.

[(4)] p ist dabei ein beliebiger Punkt der Photonenbahn.

Das Ergebnis führte jedoch zu großen Abweichungen, wodurch angenommen werden kann, dass sich diese nicht in einer Ebene bewegen.

4.4 Winkelverteilungen

Aufgrund der neu eingeführten Kategorisierung der Photonen nach der Simulation wurden die Verteilungen der Startrichtungen der Photonen partitioniert geplottet (siehe Abbildungen 4.34 und 4.35). Die dortigen Winkel sind die Vortriebswinkel[(5)] θ_F und ϕ_F. Die gesamte Eingabeverteilung ist in Abbildung 4.33 zu erkennen. Beide Simulationen wurden mittels der gleichen Parameter simuliert, welche Tabelle 4.1 zu entnehmen sind. Für den zweiten Simulationsdurchlauf wurde die Länge jedoch auf 3 cm verkürzt, um eine Vergleichbarkeit zu erreichen. Als Profilkurve wurde die Falkekurve und die Parabelkurve aus Abschnitt 4.1.2 verwendet. Die Verteilungen für $lost$, $absorbed$ and $failed$, sowie Verteilungen für die Parabelkurve-Profilkurve finden sich im Anhang B.4.

Auffallend ist, dass für Abbildung 4.34b, welche die Winkelverteilung für $n_{returned}$ zeigt, in der Mitte eine Lücke entsteht, welche von der korrespondierenden Verteilung für $n_{success}$ (siehe Abbildung 4.34a) *gefüllt* werden kann. Welchen Zusammenhang dieser auftauchende Trennwinkel jedoch im Bezug zur Geometrie des ALG aufweist, wurde nicht mehr untersucht. Erwähnenswert ist jedoch, dass der Zusammenhang überwiegend auf ϕ_F zurückzuführen ist. Zwischen den Verteilungen der beiden Profilkurven (Falkekurve, Parabelkurve) fanden sich keine signifikanten Unterschiede.

Fazit

Aus den erzielten Simulationsergebnissen lässt sich schließen, dass die Falkekurve nicht das erwartete Optimum liefert, sondern nur eine vieler *guter* Kurven ist. So zeigten die Batch Simulationen, dass bereits durch Abtasten der Suchräume keine bessere Kurve gefunden werden konnte. Die verschiedenen lokalen Optimierungen lieferten zum Teil andere Lösungskurven, jedoch wichen auch diese im Hinblick auf die Zielfunktion, der Maximierung von $n_{success}$, nur wenig voneinander ab. Für entsprechende Fertigungsmaßnahmen kann also eine kostengünstige Profilkurve gewählt werden. Die Winkelverteilungen zeigen eine visuelle Trennung der erkannten und zurücklaufenden Photonen, welche es in weiterführender Forschung zu untersuchen gilt.

[(5)] ϕ_F und θ_F sind dabei die Polarwinkel im lokalen Koordinatensystem der initialen Photonen [21].

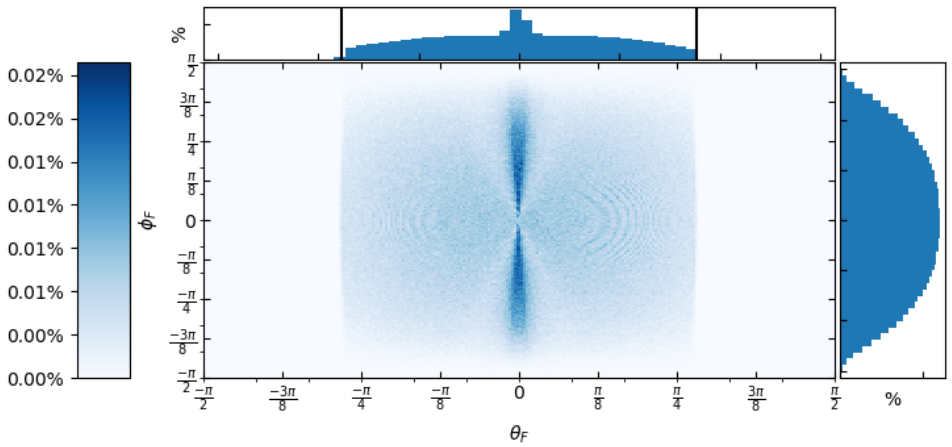
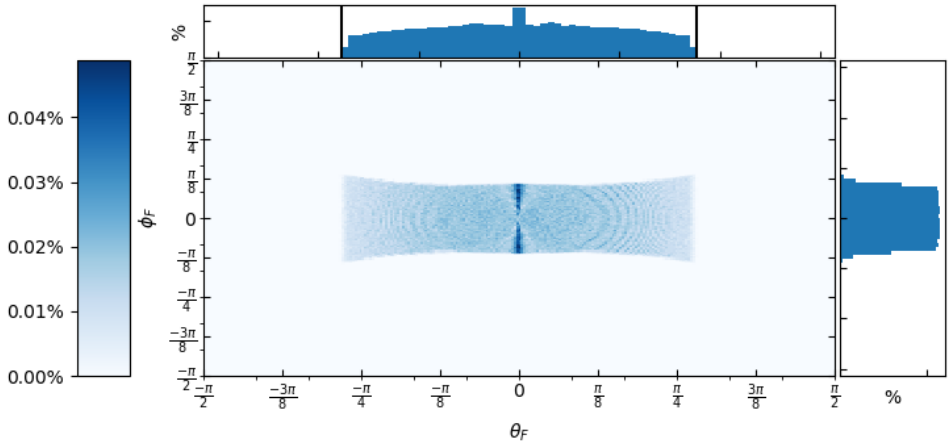
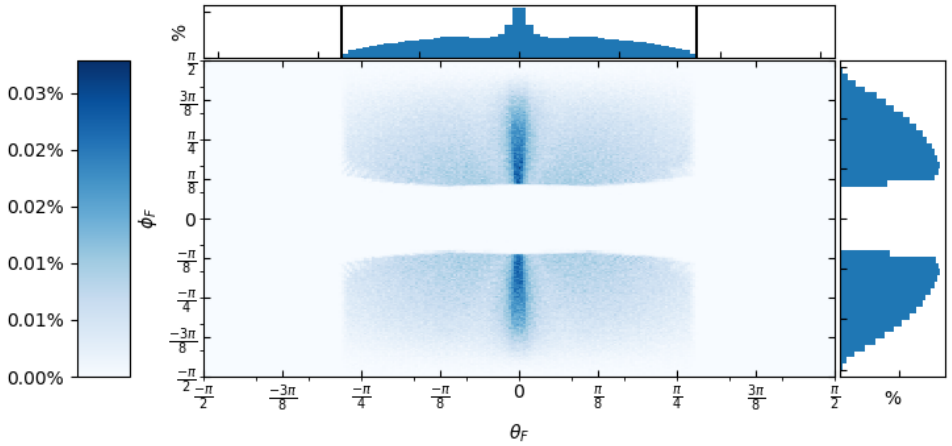


Abb. 4.33.: Gesamteingabeverteilung der Vortriebswinkel
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

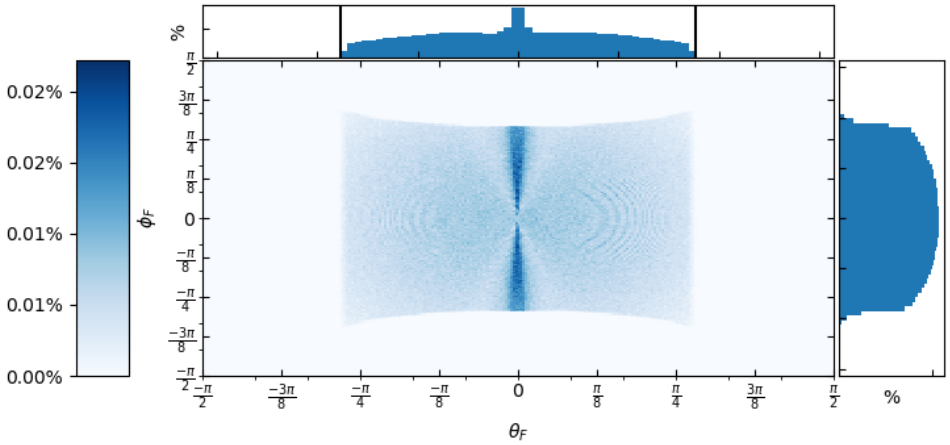


(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *SUCCESS* nach der Simulation durch den ALG mit der Falkekurve als Profil.

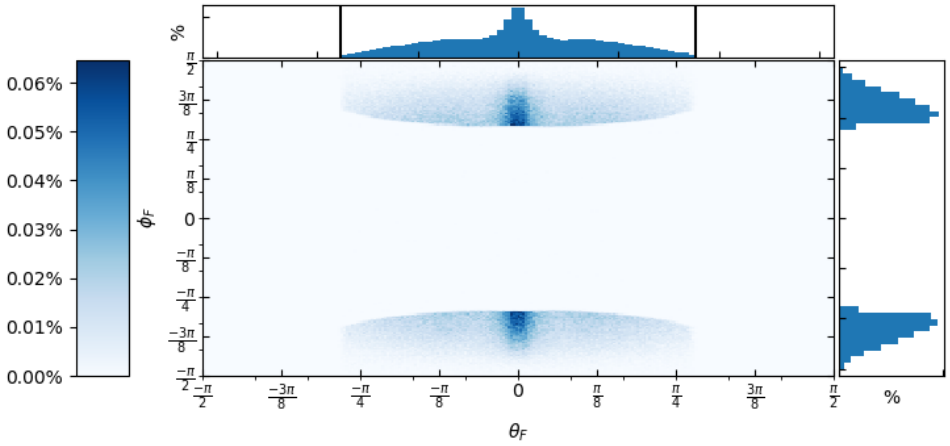


(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *RETURNED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. 4.34.: Winkelverteilungen mit Falkekurve gruppiert nach Statuscode mit $l = 6$ cm
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.



(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *SUCCESS* nach der Simulation durch den ALG mit der Falkekurve als Profil.



(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *RETURNED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. 4.35.: Winkelverteilungen mit Falkekurve gruppiert nach Statuscode mit $l = 3$ cm. Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

5 Schlussfolgerung und Ausblick

> *Selbst aus schlechten Aussichten lassen sich noch gute Einsichten gewinnen.*
>
> **— Ernst Ferstl**

Die in dieser Arbeit aufgezeigten Anpassungen und Erweiterungen des WOMRaT [22] liefern eine erweiterbare neue Softwareplattform (ALGO [19]) für zukünftige simulative Projekte der Art. Insbesondere mittels der zugefügten Python Schnittstelle konnten durch leichte Anpassungen an den Skripten schnell mehrere Simulationen durchgeführt und miteinander verglichen werden.

Die neuartige Spline Interpolation anhand der Vorgabe von Krümmungen liefert die beste bisher bekannte Approximation des Funktionenraumes für rotationssymmetrische Profilkurven des adiabatischen Lichtleiters, wodurch eine numerische Optimierung erst möglich wurde. Es konnte nachgewiesen werden, dass eine numerische Optimierung prinzipiell möglich ist, wenn auch instabil.

Die Simulationsauswertungen lassen im Gesamten nur die Schlussfolgerung zu, dass sich für den adiabatischen Lichtleiter mit Erhaltung von Rotationssymmetrie keine optimale herstellbare Form findet, welche zu 100% Kathodeneffizienz führt. Die bekannte Falkekurve liefert zwar eine gute Geometrie für den Gesamtverlust, aber keine vollständig verlustfreie. Da die meisten Kurven eine gleiche Effizienz aufweisen, kann für Produktionszwecke eine möglichst kostengünstige Form des ALG gewählt werden.

In weiterführender Forschung können andere numerische Optimierer verwendet werden um zu testen, ob das Verfahren, die Ursache für die teilweise auftretende Instabilität ist. Simulationen mit einem größeren Set an Photonen werden empfohlen, um gegebenfalls weitere Unterschiede zwischen den Kurven hervorzuheben.

Sollte ein stabileres Verfahren gefunden werden, so kann auch eine Variation der Zielfunktion, beispielsweise in Abhängigkeit des Signal-zu Rauschverhältnisses, eine mögliche Alternative sein. Ebenso gilt es den Grund der visuellen Trennung von erkannten und zurücklaufenden Photonen in den Winkelverteilungen weiterer zu untersuchen.

Anhänge

Gleichungen

A

A.1 Quadratische Splines mit vorgegebenen Krümmungen

Bestimmung der Formel für b_{n-1}:

$$\begin{aligned} b_{n-1} &= 2a_{n-2}\delta + b_{n-2} \\ &= 2a_{n-2}\delta + (2a_{n-3}\delta + b_{n-3}) \\ &= 2a_{n-2}\delta + 2a_{n-3}\delta + \cdots + 2a_0\delta + b_0 \\ &= 2\delta \sum_{i=0}^{n-2} a_i + b_0 \end{aligned}$$

Bestimmung der Formel für c_{n-1}:

$$\begin{aligned} c_{n-1} &= a_{n-2}\delta^2 + b_{n-2}\delta + c_{n-2} \\ &= a_{n-2}\delta^2 + b_{n-2}\delta + \left(a_{n-3}\delta^2 + b_{n-3}\delta + c_{n-3}\right) \\ &= a_{n-2}\delta^2 + b_{n-2}\delta + a_{n-3}\delta^2 + b_{n-3}\delta + \cdots + a_1\delta^2 + b_1\delta + a_0\delta^2 + b_0\delta + y_0 \\ &= \delta^2 \sum_{i=0}^{n-2} a_i + \delta \sum_{i=0}^{n-2} (b_i) + y_0 \\ &= \delta^2 \sum_{i=0}^{n-2} a_i + \delta \sum_{i=0}^{n-2} \left(2\delta \sum_{j=0}^{i-1} a_j + b_0\right) + y_0 \\ &= \delta^2 \sum_{i=0}^{n-2} a_i + 2\delta^2 \sum_{i=0}^{n-2} \sum_{j=0}^{i-1} a_j + \delta(n-1)b_0 + y_0 \end{aligned}$$

Bestimmung der Formel für y_n:

$$
\begin{aligned}
y_n &= a_{n-1}\delta^2 + \left(2\delta \sum_{i=0}^{n-2} a_i\right)\delta + \left(\delta^2 \sum_{i=0}^{n-2} a_i + 2\delta^2 \sum_{i=0}^{n-2}\sum_{j=0}^{i-1} a_j + \delta(n-1)b_0 + y_0\right) \\
&= a_{n-1}\delta^2 + \left(2\delta \sum_{i=0}^{n-2} a_i\right)\delta + \left(\delta^2 \sum_{i=0}^{n-2} a_i + 2\delta^2 \sum_{i=0}^{n-2}\sum_{j=0}^{i-1} a_j\right) + \delta(n-1)b_0 + y_0 \\
&= a_{n-1}\delta^2 + 2\delta^2 \sum_{i=0}^{n-2} a_i + \delta^2\left(\sum_{i=0}^{n-2} a_i + 2\sum_{i=0}^{n-2}\sum_{j=0}^{i-1} a_j\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + 2\sum_{i=0}^{n-2} a_i + \sum_{i=0}^{n-2} a_i + 2\sum_{i=0}^{n-2}\sum_{j=0}^{i-1} a_j\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + 3\sum_{i=0}^{n-2} a_i + 2\sum_{i=0}^{n-2}\sum_{j=0}^{i-1} a_j\right) + \delta(n-1)b_0 + y_0 \\
&\overset{\text{Vertauschung der Summationsreihenfolge}}{=} \delta^2\left(a_{n-1} + 3\sum_{i=0}^{n-2} a_i + 2\sum_{i=0}^{n-2}(((n-1)-1-i)a_i)\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + 3\sum_{i=0}^{n-2} a_i + 2\sum_{i=0}^{n-2}((n-2-i)a_i)\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + \sum_{i=0}^{n-2}(3+2(n-2-i))a_i\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + \sum_{i=0}^{n-2}(3+2n-4-2i)a_i\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(a_{n-1} + \sum_{i=0}^{n-2}(2(n-i)-1)a_i\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(\sum_{i=0}^{n-1}(2(n-i)-1)a_i\right) + \delta(n-1)b_0 + y_0 \\
&= \delta^2\left(\sum_{k=0}^{n-1}(2k-1)a_{n-k}\right) + \delta(n-1)b_0 + y_0
\end{aligned}
$$

B Zusätzliche Plots und Visualisierungen

B.1 Batch Simulationen

B.1.1 Naiver Ansatz für $\delta = 0.01$

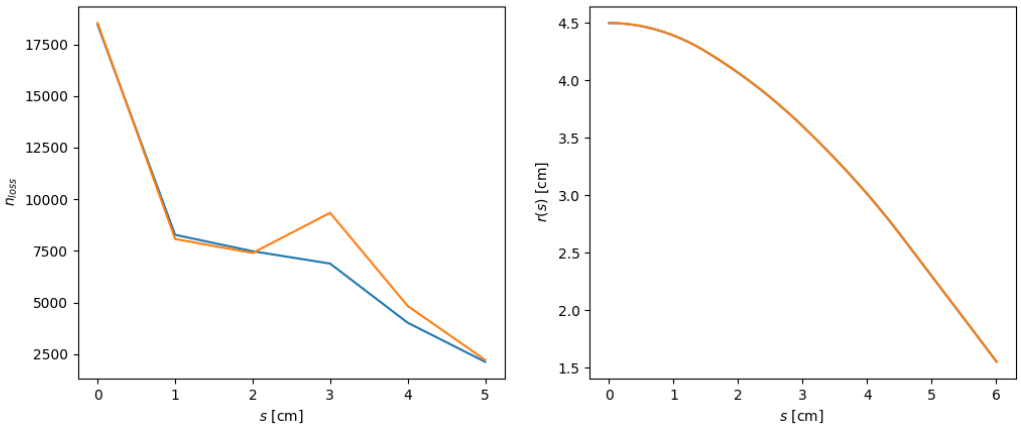


Abb. B.1.: Plots der besten (blau) und schlechtesten Kurve (orange) (r) mit $\delta := 0.01$ mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve, welche Menge an Photonen verloren gehen.

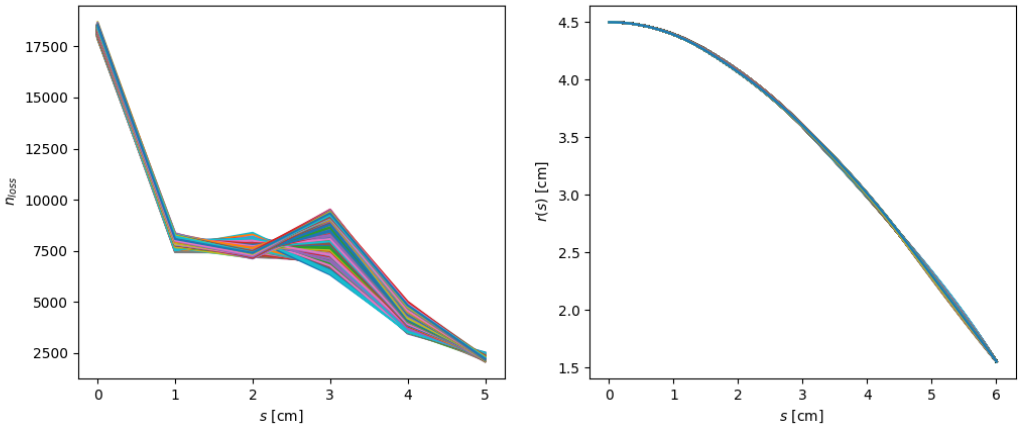


Abb. B.2.: Plots aller Kurven mit $\delta := 0.01$ (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen.

B.1.2 Naiver Ansatz für $\delta = 0.2$

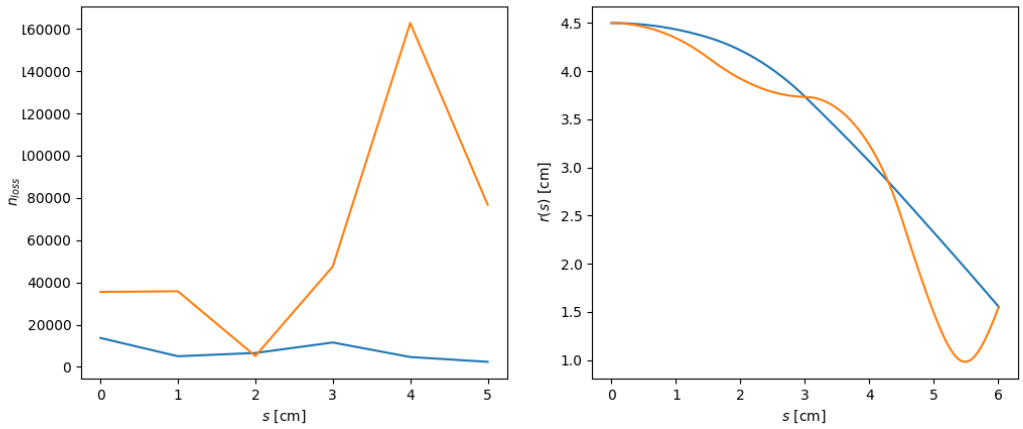


Abb. B.3.: Plots der besten (blau) und schlechtesten Kurve (orange) (r) mit $\delta := 0.2$ mit zugehörigem Lost-Histogramm (l), welches angibt, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen.

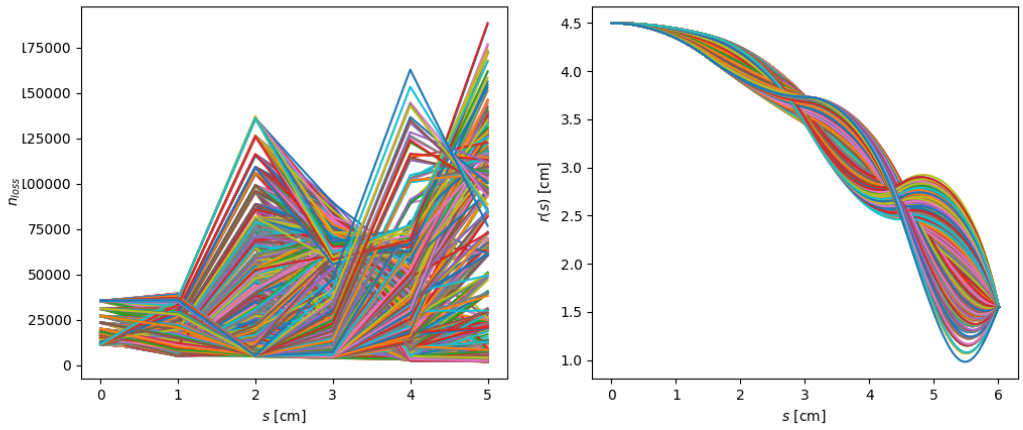


Abb. B.4.: Plots aller Kurven mit $\delta := 0.2$ (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen.

B.1.3 Ansatz mit gekrümmten Splines für $\delta = 0.01$

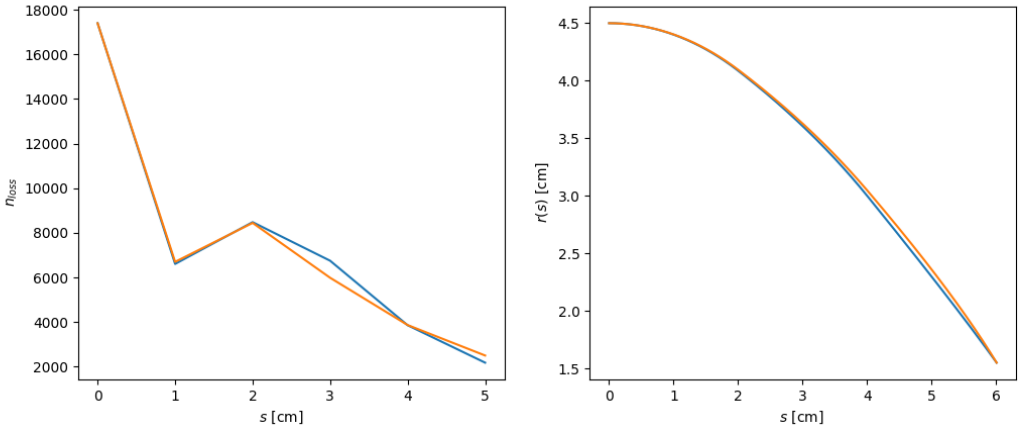


Abb. B.5.: Plots der besten (blau) und schlechtesten Kurve (orange) (l) mit zugehörigem Lost-Histogramm (r), welches angibt, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen. $\delta = 0.01$

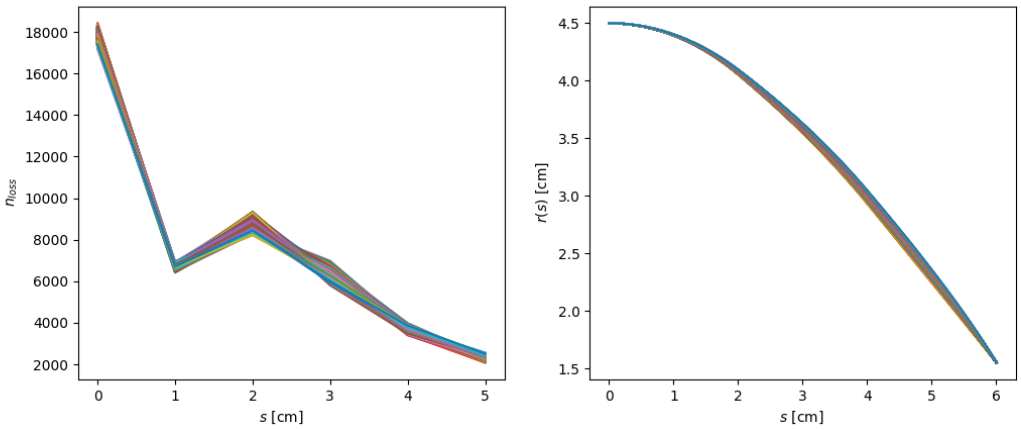


Abb. B.6.: Plots aller Kurven (l) mit zugehörigem Lost-Histogrammen (r), welche angeben, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen. $\delta = 0.01$

B.1.4 Ansatz mit gekrümmten Splines für $\delta = 0.1$

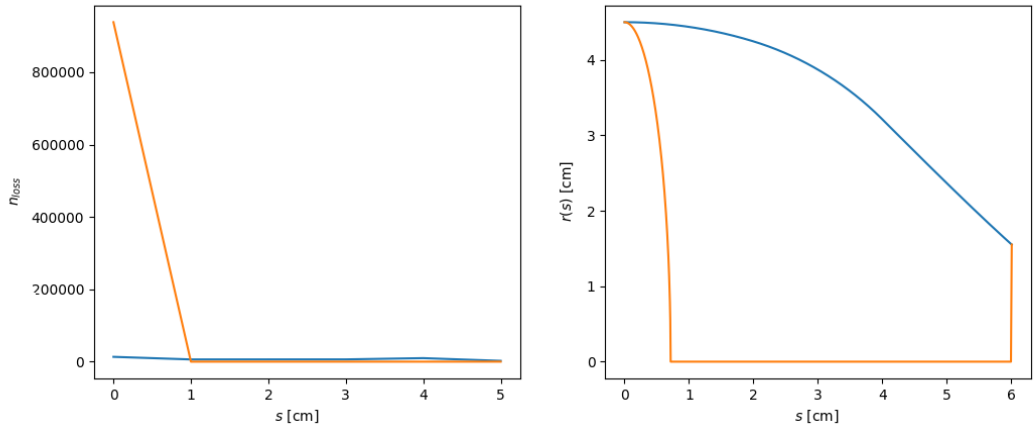


Abb. B.7.: Plots der besten (blau) und schlechtesten Kurve (orange) (l) mit zugehörigem Lost-Histogramm (r), welches angibt, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen. $\delta = 0.1$

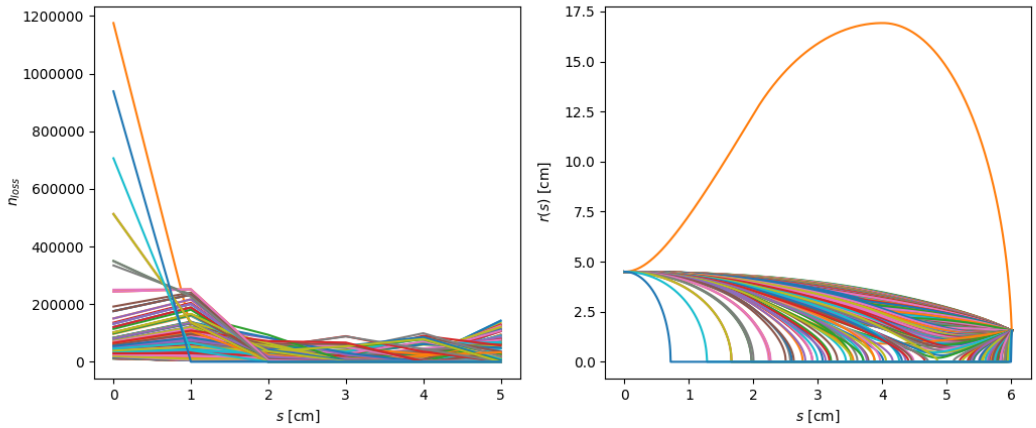
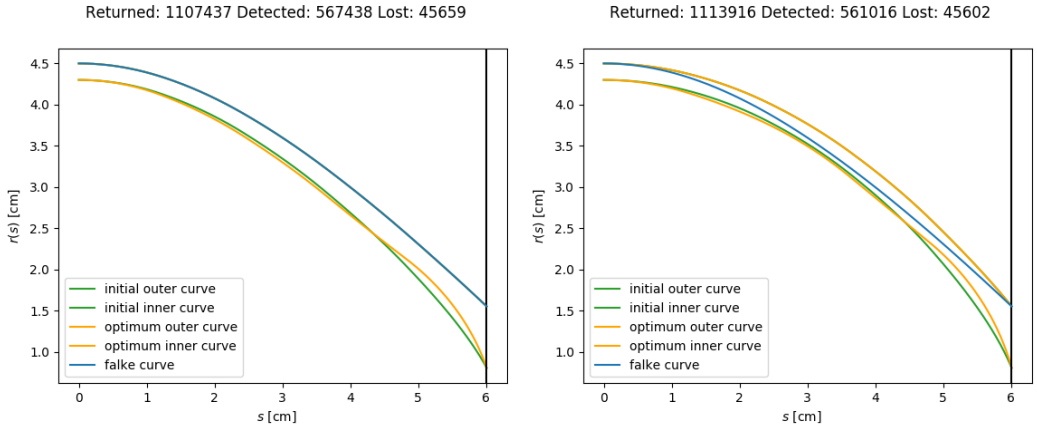


Abb. B.8.: Plots aller Kurven (r) mit zugehörigem Lost-Histogrammen (l), welche angeben, in welchem Bereich der Kurve,welche Menge an Photonen verloren gehen. $\delta = 0.1$

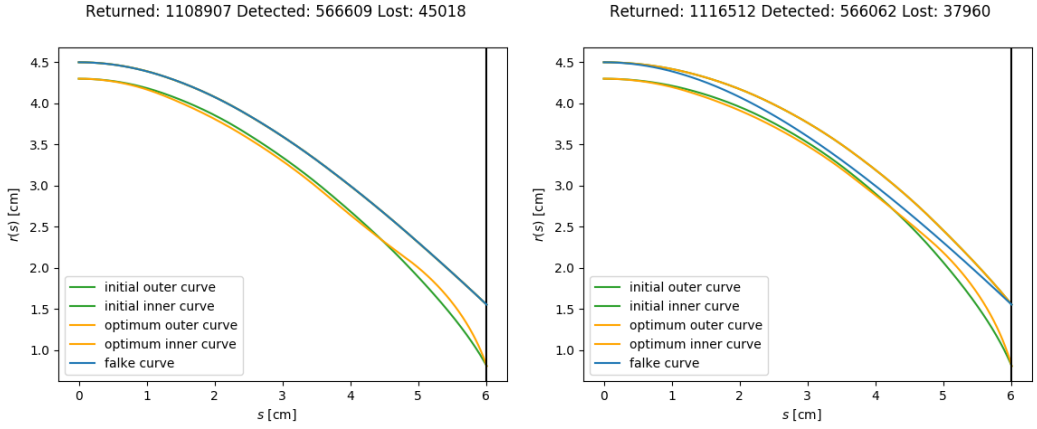
B.2 Simplexoptimierungen

B.2.1 Variation der inneren Kurve



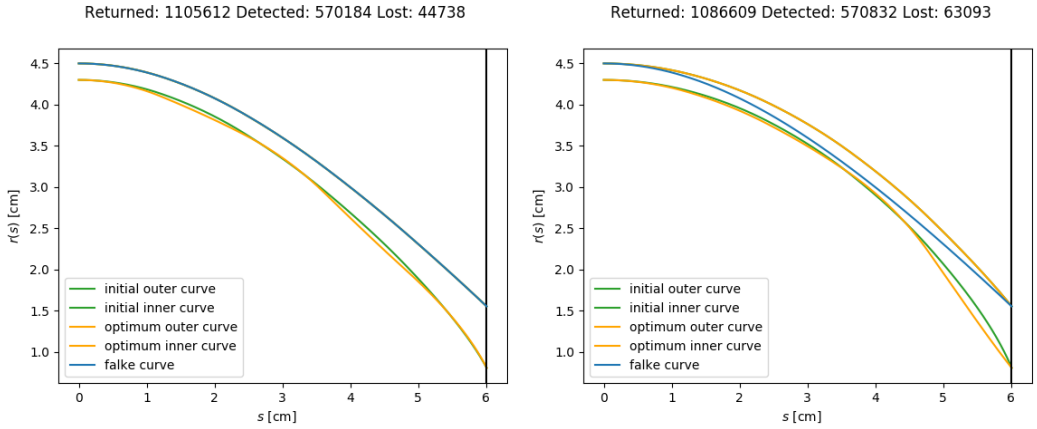
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.9.: Naive Optimierung nach n_{lost} mit Variation der inneren Kurve.



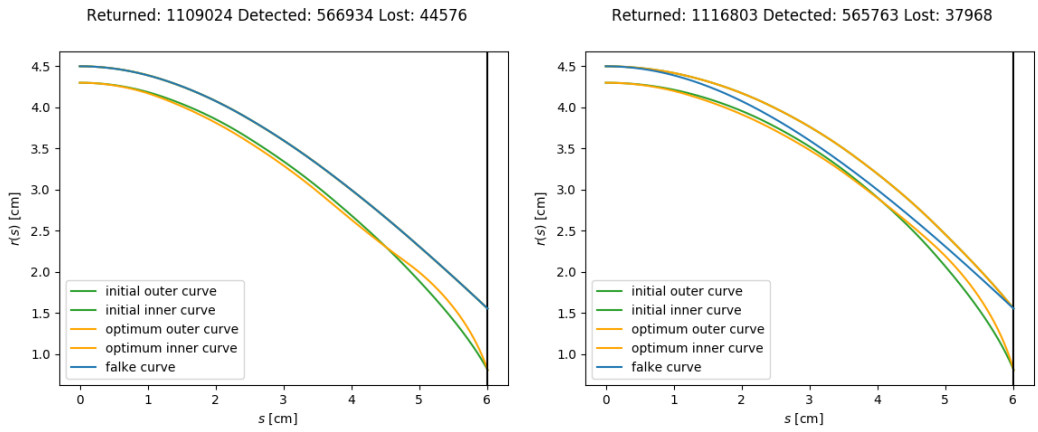
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.10.: Naive Optimierung nach $n_{returned}$ mit Variation der inneren Kurve.



(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.11.: Naive Optimierung nach $n_{success}$ mit Variation der inneren Kurve.

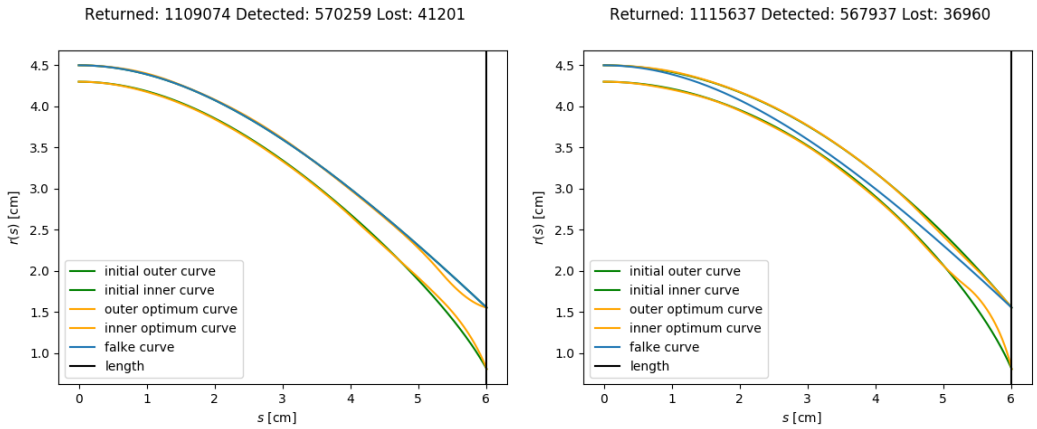


(a) Start von Steigungen der Falkekurve

(b) Start von Steigungen der Parabelkurve

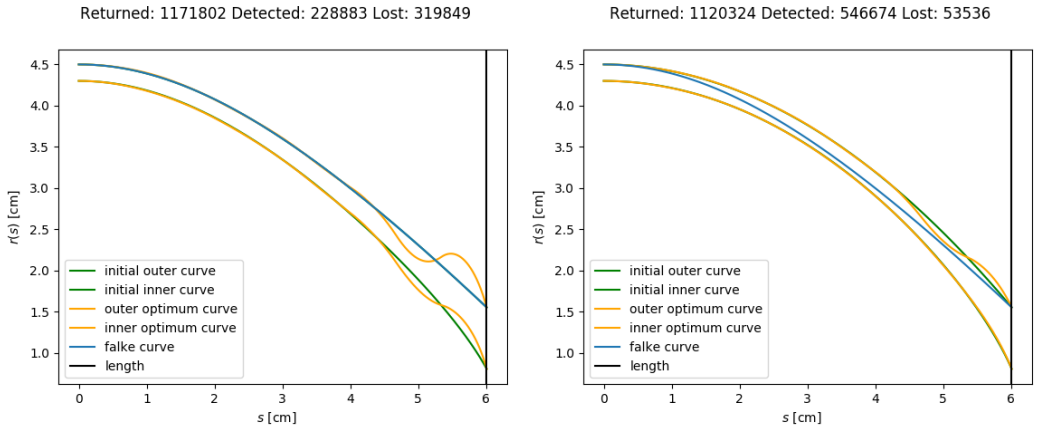
Abb. B.12.: Optimierung mittels Steigung der Splines nach $n_{returned}$ mit Variation der inneren Kurve.

B.2.2 Variation der inneren und äußeren Kurve



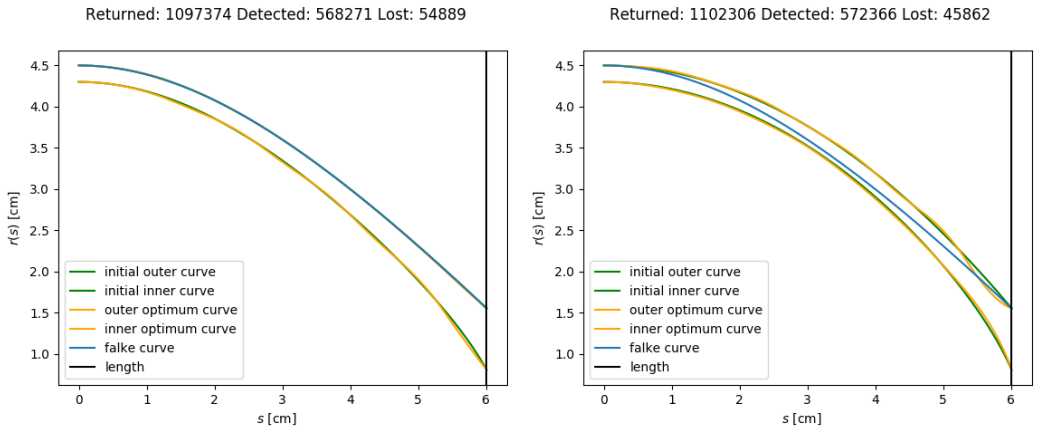
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.13.: Naive Optimierung nach n_{lost} mit Variation der inneren und äußeren Kurve.



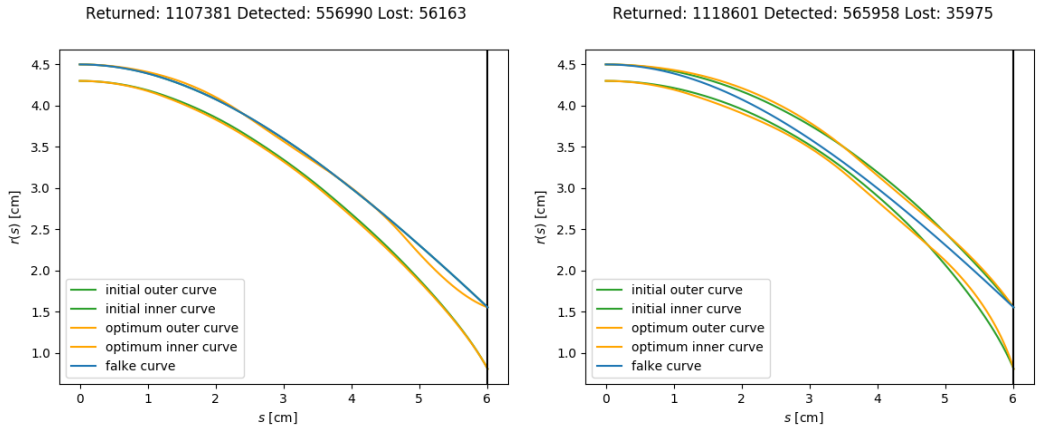
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.14.: Naive Optimierung nach $n_{returned}$ mit Variation der inneren und äußeren Kurve.



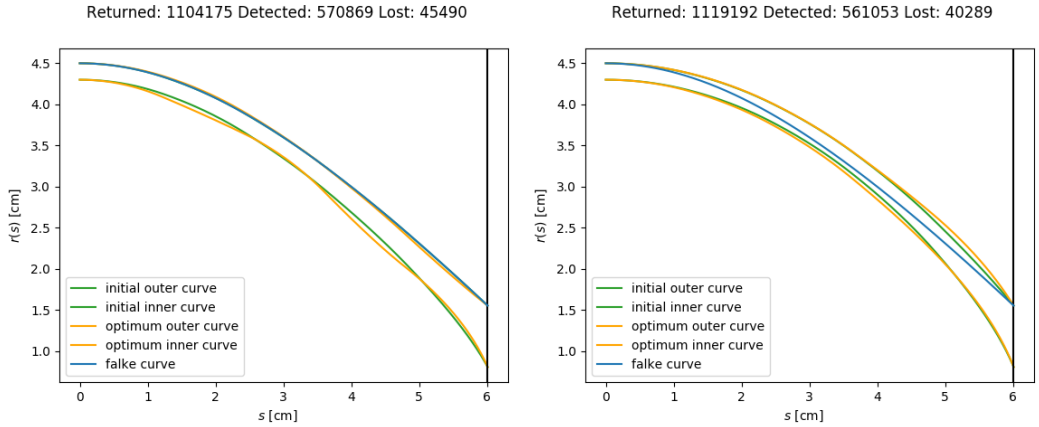
(a) Start von Falkekurve (b) Start von Parabelkurve

Abb. B.15.: Naive Optimierung nach $n_{success}$ mit Variation der inneren und äußeren Kurve.



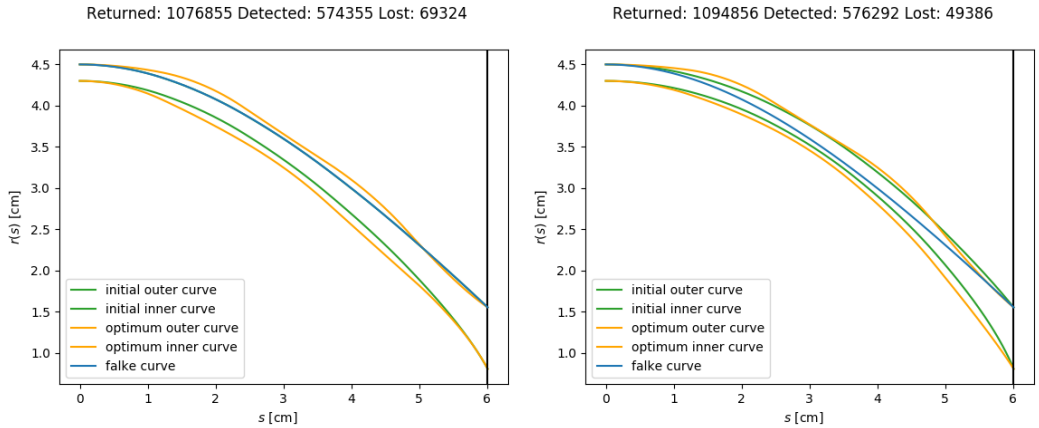
(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. B.16.: Optimierung mittels Steigung der Splines nach n_{lost} mit Variation der inneren und äußeren Kurve.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

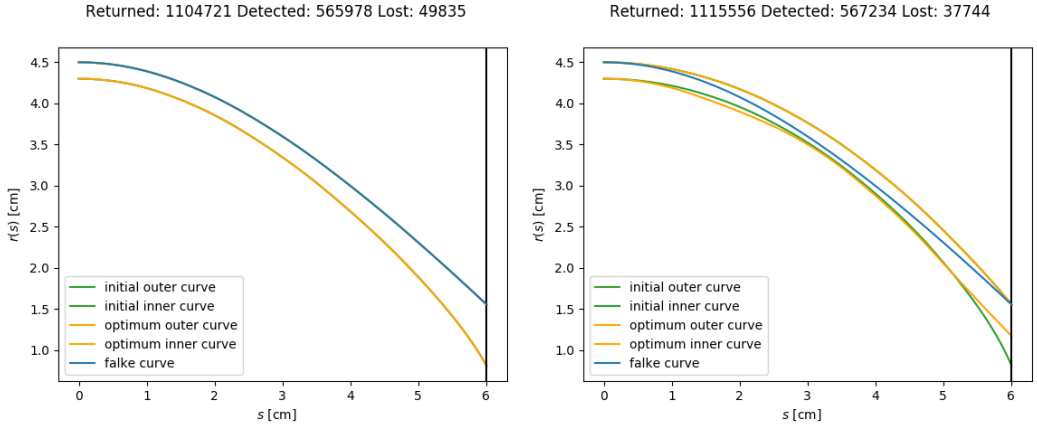
Abb. B.17.: Optimierung mittels Steigung der Splines nach $n_{returned}$ mit Variation der inneren und äußeren Kurve.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

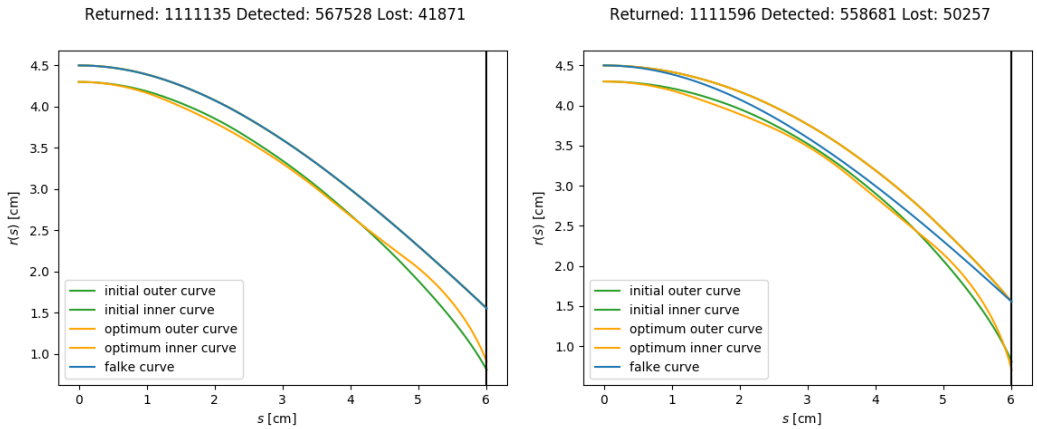
Abb. B.18.: Optimierung mittels Steigung der Splines nach $n_{success}$ mit Variation der inneren und äußeren Kurve.

B.2.3 Variation der inneren Kurve und des inneren Endradius



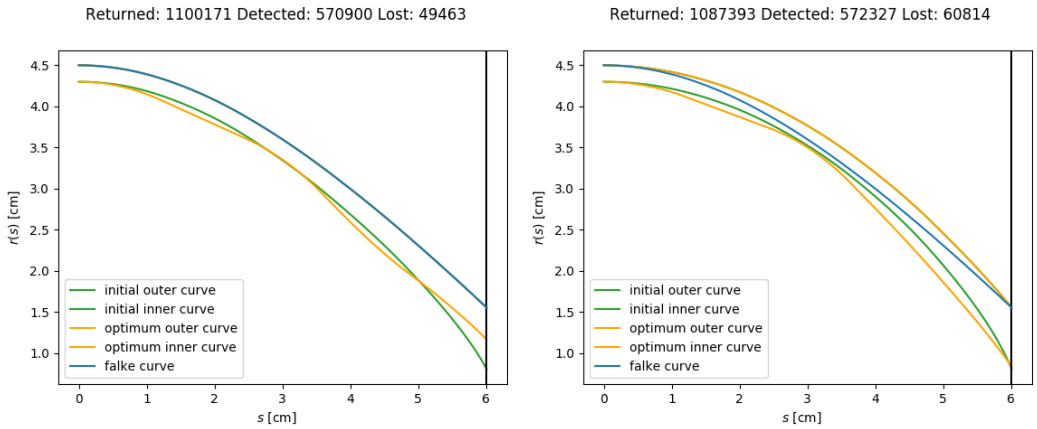
(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. B.19.: Optimierung mittels Steigung der Splines nach n_{lost} mit Variation der inneren Kurve und des inneren Endradius.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. B.20.: Optimierung mittels Steigung der Splines nach $n_{returned}$ mit Variation der inneren Kurve und des inneren Endradius.



(a) Start von Steigungen der Falkekurve (b) Start von Steigungen der Parabelkurve

Abb. B.21.: Optimierung mittels Steigung der Splines nach $n_{success}$ mit Variation der inneren Kurve und des inneren Endradius.

B.3 3D-Plot der Photonen nach einem vollständigen Simulationsdurchlauf

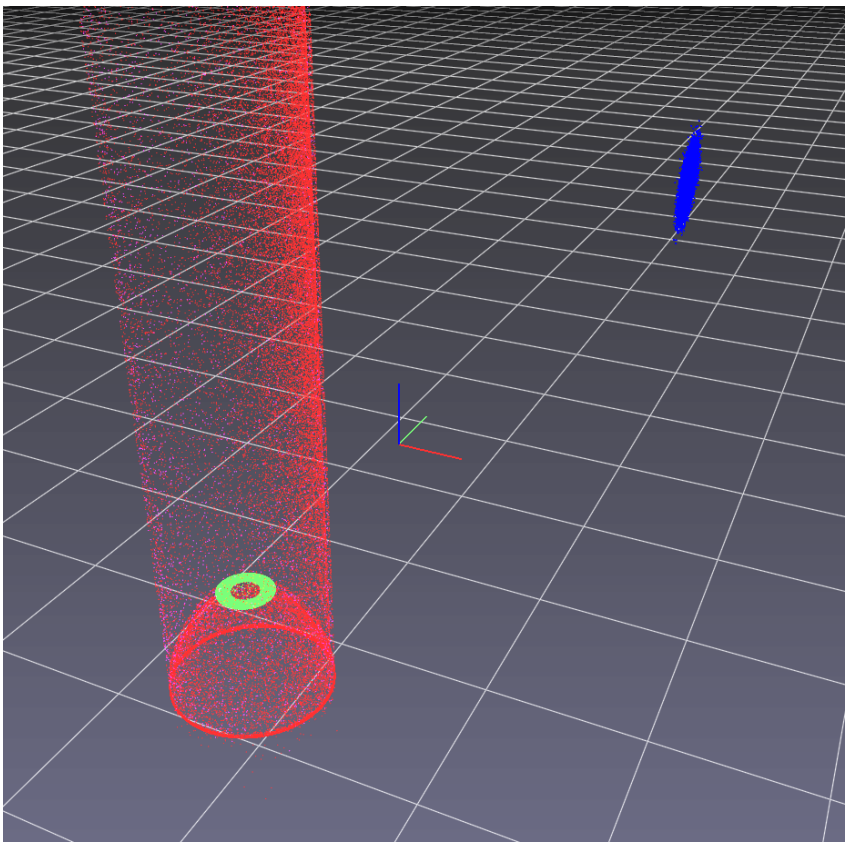
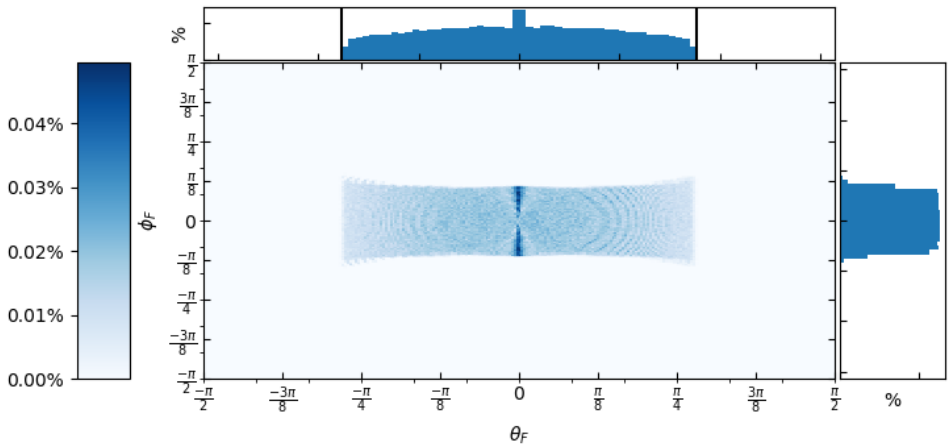


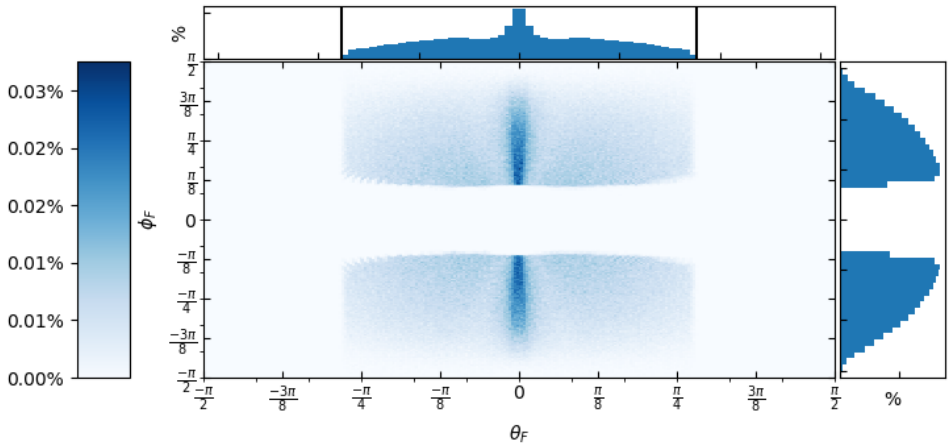
Abb. B.22.: 3D-Plot, erstellt unter Verwendung des pptk-Tookits [7], der WOM Simulation

B.4 Winkelverteilungen

B.4.1 Winkelverteilungen mit Parabelkurve-Profilkurve ($l = 6.01$ cm)

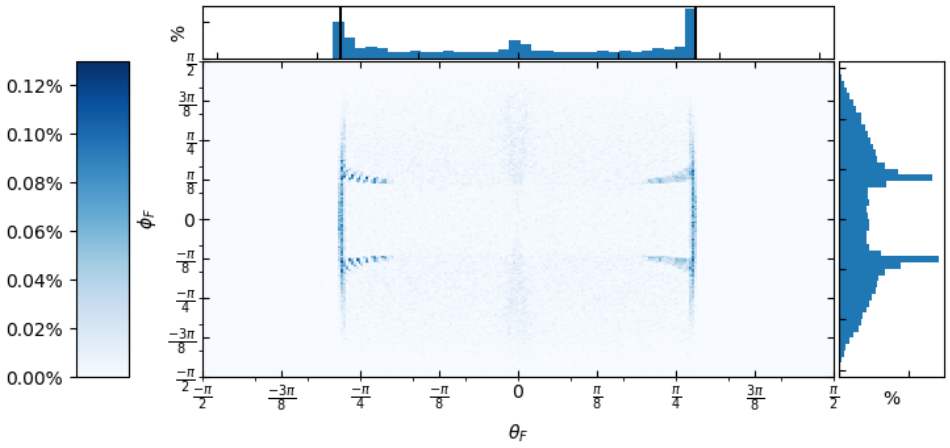


(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *SUCCESS* nach der Simulation durch den ALG mit der Falkekurve als Profil.

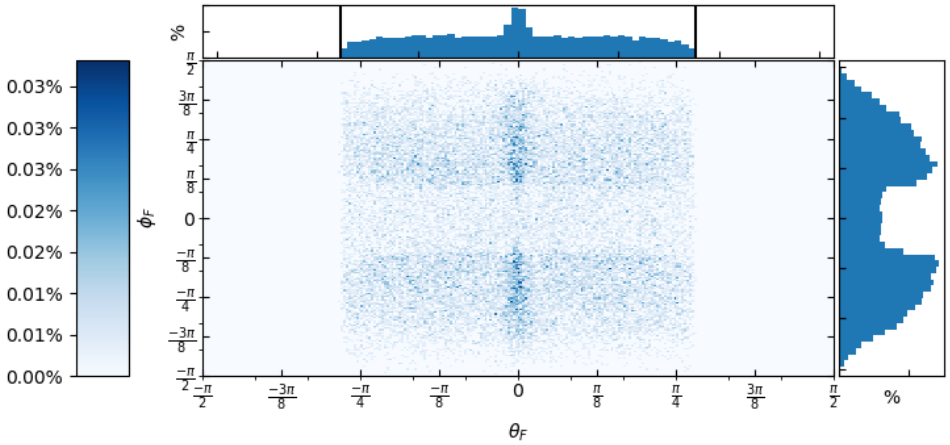


(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *RETURNED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

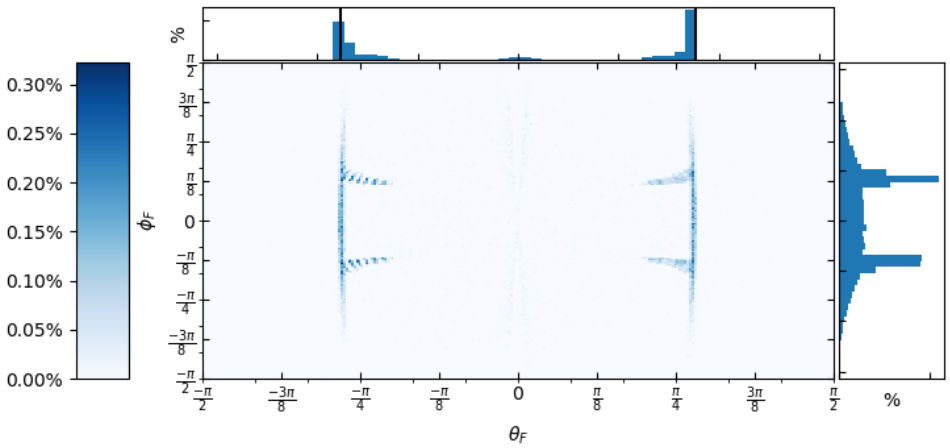
Abb. B.23.: Winkelverteilungen mit Parabelkurve-Profilkurve gruppiert nach Statuscode mit $l = 6.01$ cm.
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.



(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *LOST* nach der Simulation durch den ALG mit der Falkekurve als Profil.



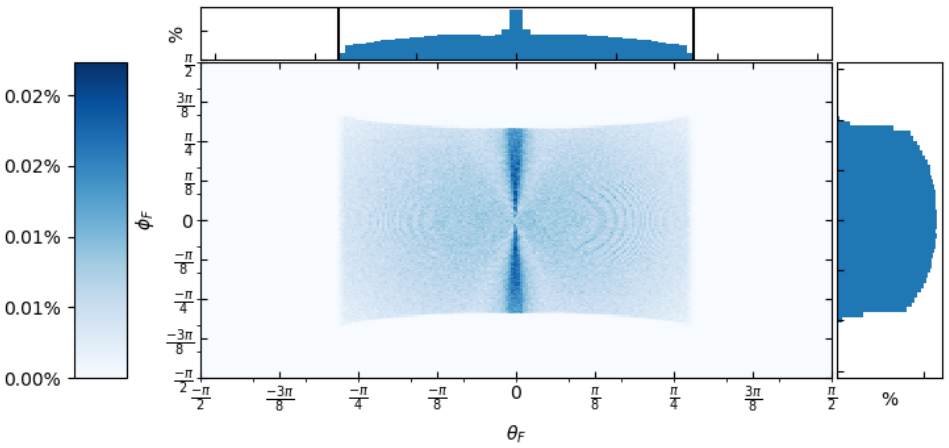
(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *ABSORBED* nach der Simulation durch den ALG mit der Falkekurve als Profil.



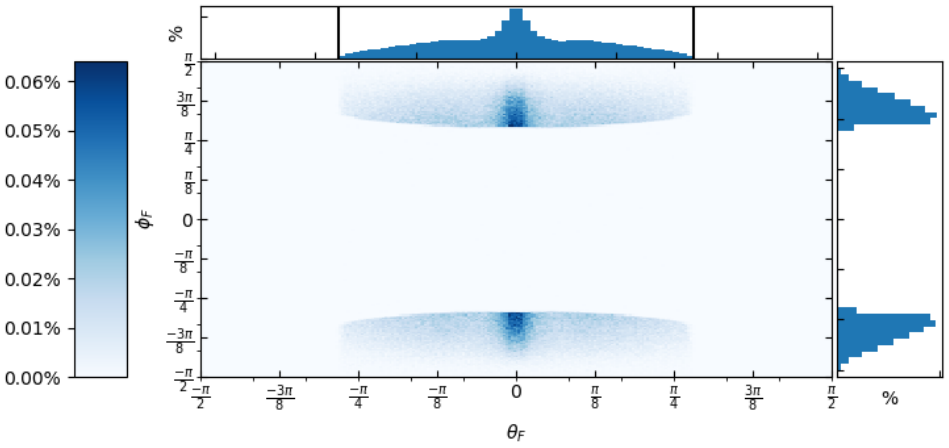
(c) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *FAILED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. B.24.: Winkelverteilungen mit Parabelkurve-Profilkurve gruppiert nach Statuscode mit $l = 6.01$ cm.
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

B.4.2 Winkelverteilungen mit Parabelkurve-Profilkurve ($l = 3$ cm)

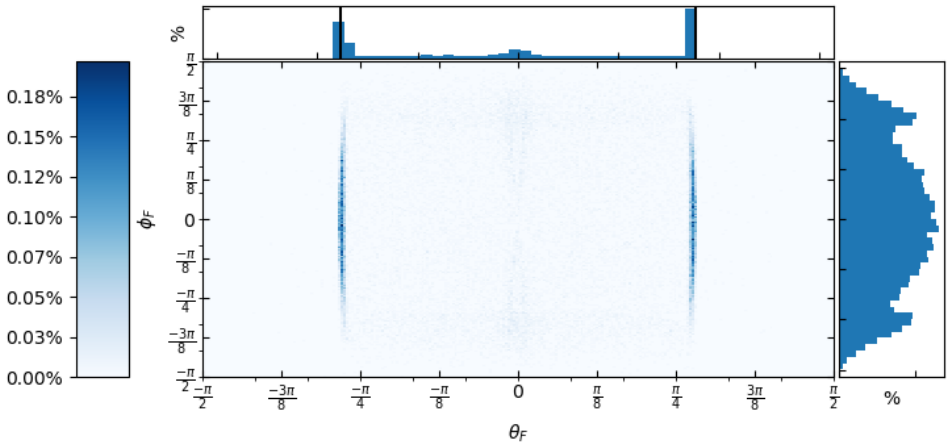


(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *SUCCESS* nach der Simulation durch den ALG mit der Falkekurve als Profil.

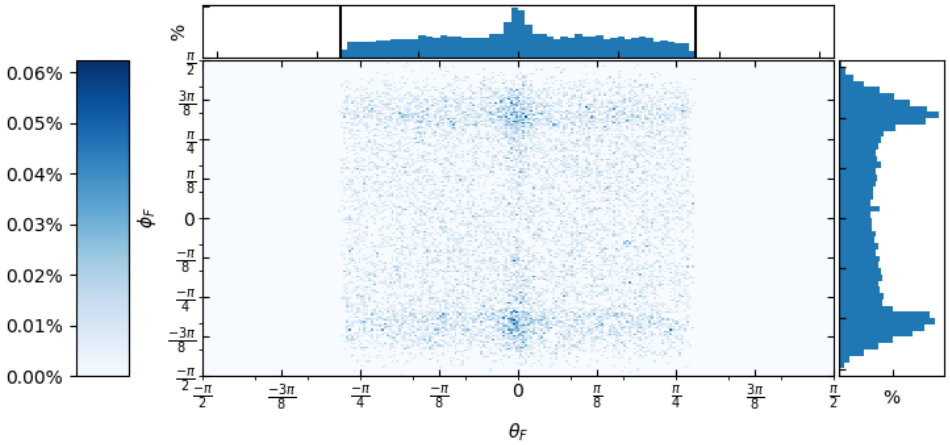


(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *RETURNED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

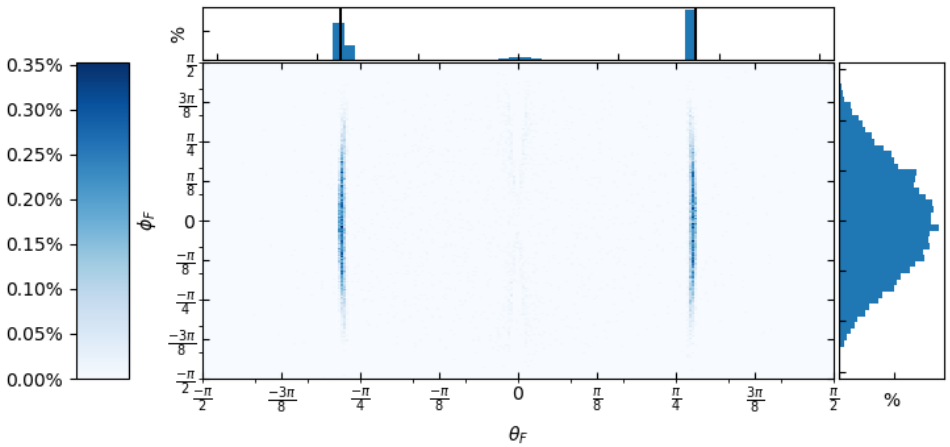
Abb. B.25.: Winkelverteilungen mit Parabelkurve-Profilkurve gruppiert nach Statuscode mit $l = 3$ cm.
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.



(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *LOST* nach der Simulation durch den ALG mit der Falkekurve als Profil.



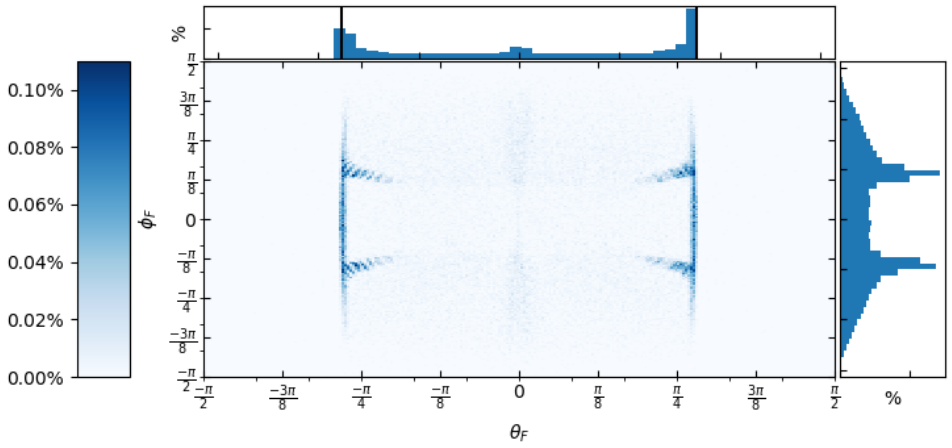
(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *ABSORBED* nach der Simulation durch den ALG mit der Falkekurve als Profil.



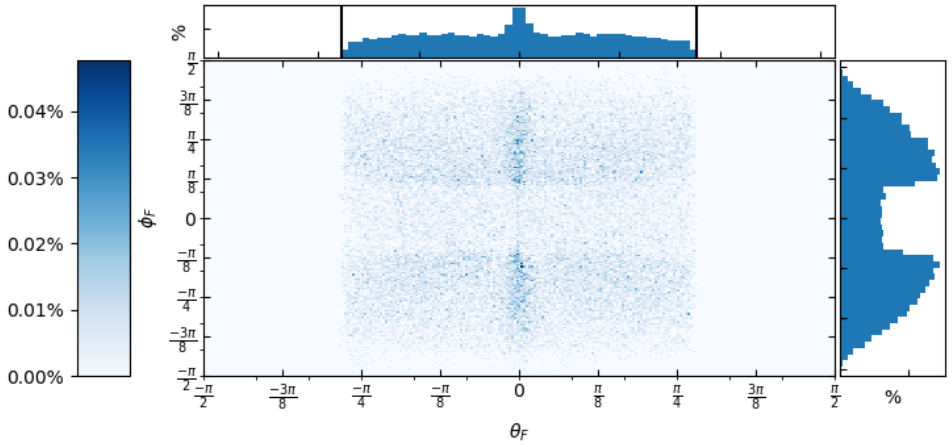
(c) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *FAILED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. B.26.: Winkelverteilungen mit Parabelkurve-Profilkurve gruppiert nach Statuscode mit $l = 3$ cm.
Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

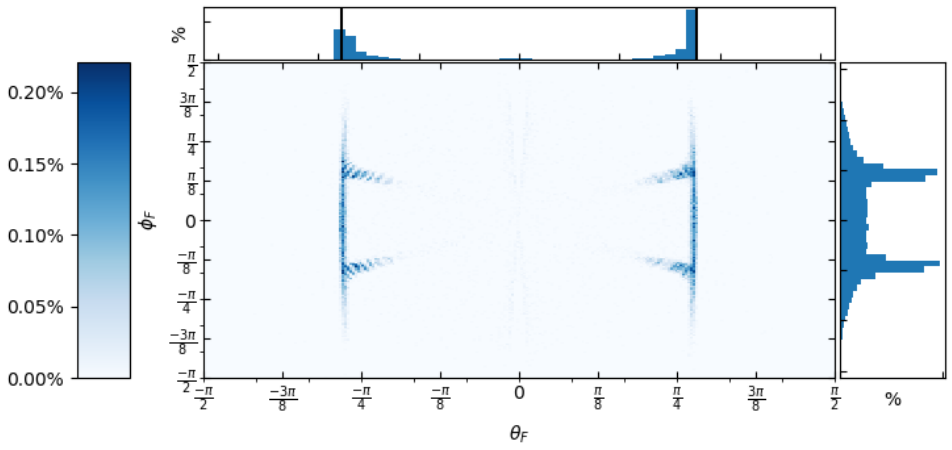
B.4.3 Ergänzende Winkelverteilungen mit Falkekurve ($l = 6.01$ cm)



(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *LOST* nach der Simulation durch den ALG mit der Falkekurve als Profil.



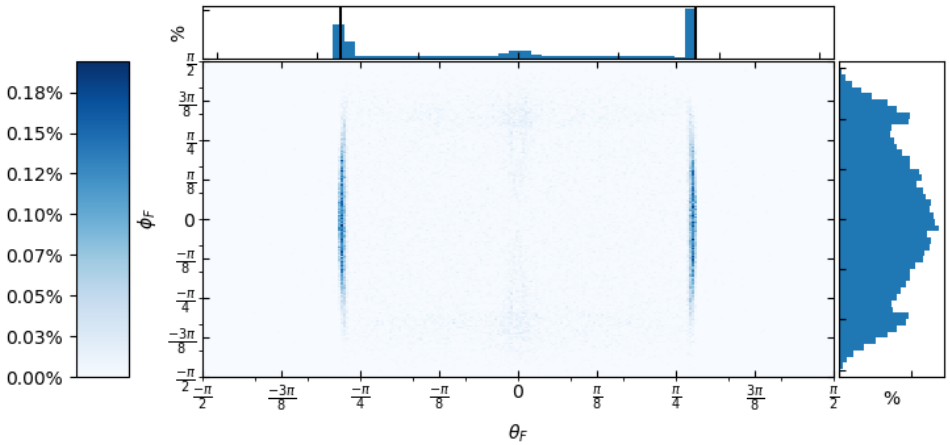
(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *ABSORBED* nach der Simulation durch den ALG mit der Falkekurve als Profil.



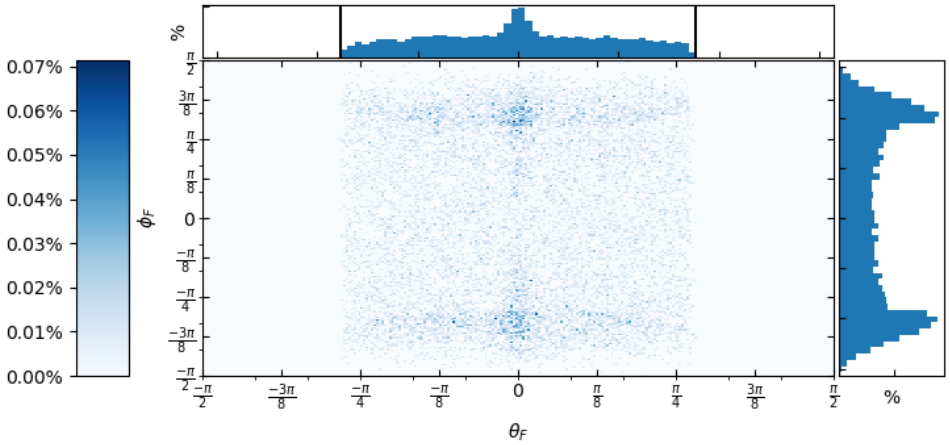
(c) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *FAILED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. B.27.: Winkelverteilungen mit Falkekurve gruppiert nach Statuscode mit $l = 6.01$ cm. Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

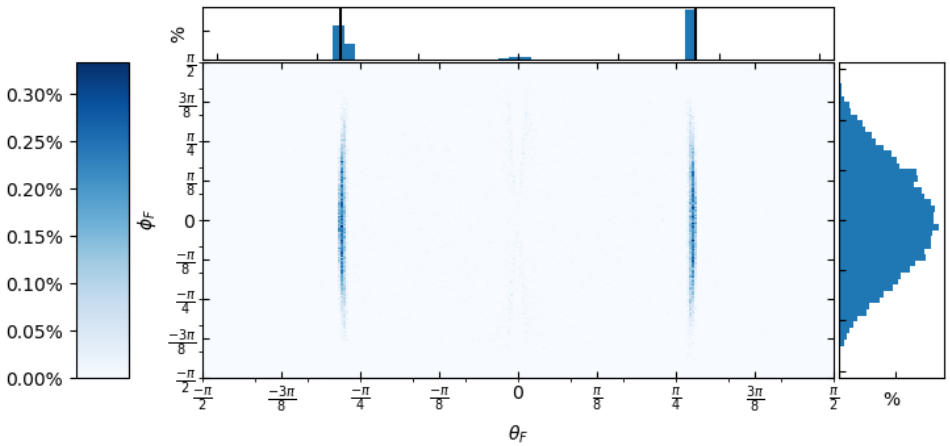
B.4.4 Ergänzende Winkelverteilungen mit Falkekurve ($l = 3$ cm)



(a) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *LOST* nach der Simulation durch den ALG mit der Falkekurve als Profil.



(b) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *ABSORBED* nach der Simulation durch den ALG mit der Falkekurve als Profil.



(c) Eingangsverteilung der Vortriebswinkel der Photonen mit Statuscode *FAILED* nach der Simulation durch den ALG mit der Falkekurve als Profil.

Abb. B.28.: Winkelverteilungen mit Falkekurve gruppiert nach Statuscode mit $l = 3$ cm. Der schwarzfarbene Balken markiert den Totalreflexionswinkel $\theta_c = \frac{\pi}{2} - \alpha_c$.

C Ergänzende Beweise

C.1 Existenz des einen Quadratischen Splines mit konstanter Krümmung

Satz. *Gegeben zwei Quadratische Splines S_p, S_q mit konstanten Krümmungen mit $a_{i\,p} = c_p, a_{i\,q} = c_q \forall\ i \in \{0,1,\ldots,k\, y_n := y_{n\,p} = y_{n\,q}, y_0 := y_{0\,p} = y_{0\,q}$ und $\delta := \delta_p = \delta_q$ dann ist*

$$S_p \Leftrightarrow S_q,$$

d.h. S_p und S_q resultieren in den gleichen Spline Koeffizienten. Ferner also:

$$\alpha_p \cdot c_p = \alpha_q \cdot c_q$$

Beweis. Für α_p gilt:

$$\begin{aligned}
\alpha_p &= \frac{y_n - y_0}{\delta^2\left(c_p + 3\sum_{i=0}^{n-2} c_p + 2\sum_{i=0}^{n-2}\left((n-3-i)c_p\right)\right)} \\
&= \frac{y_n - y_0}{\delta^2\left(c_p + 3c_p\sum_{i=0}^{n-2} 1 + 2c_p\sum_{i=0}^{n-2}(n-3-i)\right)} \\
&= \frac{y_n - y_0}{\delta^2\underbrace{\left(1 + 3\sum_{i=0}^{n-2} 1 + 2\sum_{i=0}^{n-2}(n-3-i)\right)}_{\gamma} c_p}
\end{aligned} \tag{C.1}$$

Für α_q gilt analog:

$$\alpha_q = \frac{y_n - y_0}{\delta^2\underbrace{\left(1 + 3\sum_{i=0}^{n-2} 1 + 2\sum_{i=0}^{n-2}(n-3-i)\right)}_{\gamma} c_q} \tag{C.2}$$

Daraus folgt:

$$\alpha_p \cdot c_p = \frac{y_n - y_0}{\delta^2\gamma c_q} c_q = \frac{y_n - y_0}{\delta^2\gamma} \cdot 1 = \frac{y_n - y_0}{\delta^2\gamma}\frac{c_p}{c_p} = \alpha_q \cdot c_q$$

□

Literatur

[1] M.G. Aartsen, M. Ackermann, J. Adams u. a. „The IceCube Neutrino Observatory: instrumentation and online systems“. In: *Journal of Instrumentation* 12.03 (März 2017), P03012–P03012.

[2] Robert Bradshaw und Stefan Behnel et. al. *Python C++ Extentions for Python*. Aufgerufen: 23.11.2019. URL: https://cython.org/.

[3] IceCube-Gen2 Collaboration, : M. G. Aartsen u. a. *IceCube-Gen2: A Vision for the Future of Neutrino Astronomy in Antarctica*. 2014. arXiv: 1412.5106 [astro-ph.HE].

[4] Wolfgang Demtröder. *Experimentalphysik 4: Kern-, Teilchen- und Astrophysik (Springer-Lehrbuch) (German Edition)*. Springer Spektrum, Sep. 2013.

[5] Peter Johannes Falke. „Entwicklung eines Lichtkonzentrators basierend auf einer Hohlzylinder Geometrie“. Bachelor Thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2014.

[6] Hebecker, Dustin, Archinger, Markus Gerhard, Böser, Sebastian u. a. „A Wavelength-shifting Optical Module (WOM) for in-ice neutrino detectors“. In: *EPJ Web of Conferences* 116 (2016), S. 01006.

[7] HERE Europe B.V. *pptk - Point Processing Toolkit*. URL: https://github.com/heremaps/pptk.

[8] Jared Hoberock und Nathan Bell. *thrust - Parallel Algorithms Library*. Aufgerufen: 7.1.2020. URL: https://thrust.github.io/.

[9] John D. Hunter. *Matplotlib Documentation*. Aufgerufen: 20.12.2019. URL: https://matplotlib.org/3.1.1/contents.html.

[10] ISOC++ Foundation. *C++ Standard Draft Sources*. Aufgerufen: 7.1.2020. URL: https://github.com/cplusplus/draft.

[11] J. A. Nelder und R. Mead. „A Simplex Method for Function Minimization“. In: *The Computer Journal* 7.4 (Jan. 1965), S. 308–313. eprint: http://oup.prod.sis.lan/comjnl/article-pdf/7/4/308/1013182/7-4-308.pdf.

[12] NumPy Developers. *NumPy*. URL: https://numpy.org/.

[13] NVIDIA Corporation. *Cuda Documentation*. Aufgerufen: 23.11.2019. URL: https://docs.nvidia.com/cuda/.

[14] William H. Press, Saul A. Teukolsky, William T. Vetterling und Brian P. Flannery. *Numerical Recipes in C*. Second. Cambridge, USA: Cambridge University Press, 1992.

[15] Python Software Foundation. *Python*. Aufgerufen: 7.1.2020. URL: https://www.python.org/.

[16] Python Software Foundation. *Python C API Introduction*. Aufgerufen: 23.11.2019. URL: https://docs.python.org/3/c-api/intro.html.

[17] Jingmei Qiu. *Interpolation by Splines*. Aufgerufen: 7.1.2020. 2012. URL: https://www.math.uh.edu/~jingqiu/math4364/spline.pdf.

[18] Bertil Schmidt, Christian Hundt, Jorge Gonzalez-Dominguez und Moritz Schlarb. *Parallel programming : concepts and practice*. Cambridge, MA: Morgan Kaufmann Publishers, an imprint of Elsevier, 2018.

[19] Ronja Schnur. *ALGO - Adiabatic Light Guide Optimizer*. 2019. URL: https://gitlab.rlp.net/wom/ALGO.git.

[20] The LLVM Team. *Clang Frontend User Manual*. Aufgerufen: 6.12.2019. URL: https://clang.llvm.org/docs/UsersManual.html.

[21] Florian Thomas. „Light propagation simulation for the Wavelength-shifting Optical Module on CUDA GPUs“. Master Thesis. Johannes Gutenberg-Universität Mainz, 25. Okt. 2019.

[22] Florian Thomas. *WOMRaT - WOM Ray Tracer - A rewritten and enhanced ray tracing simulation for the WOM on CUDA GPUs*. Aufgerufen: 1.11.2019. 2019. URL: https://etap-git.physik.uni-mainz.de/WOM/WOMRaT.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses an adopted version of *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.