Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Implementierung eines Verfahrens zu effizienten Berechnung oszillierender Integrale

Implementation of a method for the efficient calculation of oscillatory integrals

Tobias Arens

Master-Abschlussarbeit

Betreuer: Professor Hans-Peter Beise

Mehren, 23.10.2022

Kurzfassung

In dieser Arbeit wird das in „An analysis of the steepest descent method to efficiently compute the 3D acoustic single-layer operator in the high-frequency regime“ vorgestellte Verfahren implementiert. Basierend auf der Matlab-Implementierung der Autoren wurde eine hinsichtlich der Benutzbarkeit und Laufzeit eine verbesserte C++Implementierung umgesetzt. Die Laufzeit der neuen Implementierung konnte im Vergleich nahezu halbiert werden und bei gleicher Genauigkeit der berechneten Integrale. Diese Verbesserungen wurden mithilfe von CPU-Parallelisierungen erreicht. Um die Implementierung einsetzen zu können werden ein Python-Modul sowie Matlab Mex-Funktionen bereitgestellt.

Abstract

In this paper, the method described in „An analysis of the steepest descent method to efficiently compute the 3D acoustic single-layer operator in the high-frequency regime“. is implemented. Based on the Matlab implementation of the authors, an improved C++ implementation with respect to usability and runtime was implemented. The runtime of the new implementation has been almost halved in comparison and with the same accuracy of the calculated integrals. In order to be able to use this in a meaningful way, a Python module as well as a Matlab mex-functions are provided.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1

Einleitung und Problemstellung

1.1 Zielsetzung

Basierend auf der Arbeit „An analysis of the steepest descent method to efficiently compute the 3D acoustic single-layer operator in the high-frequency regime“[2] soll in dieser Arbeit das daraus resultierende Verfahren neu implementiert werden. Die ursprüngliche Implementierung in Matlab funktioniert zwar, weißt aber Unzulänglichkeiten hinsichtlich der Benutzbarkeit und auch der Performanz auf. So kann diese Lösung lediglich in Matlab verwendet werden und bietet keine Schnittstelle für Nutzer an. Des weiteren wurden keine Maßnahmen zur Optimierung der implementierten Lösung egriffen.

In dieser Arbeit sollen diese beiden Aspekte gelöst werden. Zum einen wird eine Implementierung angestrebt welche Paralellisierungstechniken verwendet um das Laufzeitverhalten zu verbessern, sowie eine API anbietet welche einfach verwendet werden kann. Darüber hinaus soll die Benutzbarkeit mithilfe eines Matlab-Plugins und eines Python-Moduls erleichtert werden. Wichtig dabei ist, dass die Implementierung keinen Verlust hinsichtlich der Genauigkeit der berechneten Integrale erleidet.

1.2 Aufbau der Arbeit

In Kapitel 2 dieser Arbeit wird zunächst das in [2] entwickelte Verfahren vorgestellt und einige wesentliche Teilaspekte erläutert. Danach werden in Kapitel 3 die Entwurfsbedingungen erläutert und auf die konkreten Anforderungen sowie die geplante Architektur der Umsetzung eingegangen. In 4 wird die Planung hinsichtlich der Optimierung erläutert, welche in Kapitel 6 ausgewertet und präsentiert werden. Im Kapitel 5 werden einige Details der Implementierung hervorgehoben, sowie die verwendeten Technologien vorgestellt.

2

Das Verfahren

In der Arbeit „An analysis of the steepest descent method to efficiently compute the 3D acoustic single-layer operator in the high-frequency regime“ wird ein Verfahren zum Lösen von Integralen der Form

$$\Delta I_{r,\Delta}(k) = \int_{\Delta} \frac{e^{ik(\|r-r'\|+\theta\cdot r')}}{\|r-r'\|}\, dr' \tag{2.1}$$

vorgestellt. Das Integral $\mathrm{i}_{r,\Delta}(k)$ ist das Integral eines Dreicks δ mit Beobachtungspunkt $r \in \mathbb{R}^3$ für die Wellenzahl k. $\theta \in \mathbb{R}^3$ ist ein Richtungsvektor. Dieses Integral wird auch als *accoustic single-layer Integral-operator* bezeichnet. Mit größer werdender Wellenzahl k wird dieses Problem als im *high frequency regime* liegend beschrieben und die Berechnung dieser ist mit einer hohen Laufzeit verbunden. Die Autoren führen aus wie das Integral sich mithilfe des Cauchysche Integralsatzes gezielt an nicht mit k oszillierenden Pfaden berechnen lässt. Dies ermöglicht es diese Pfade mit einem *Steepest-descent*-Verfahren effizient zu berechnen.

Die Autoren von [2] führen weiter aus, wie Pfade h_a und h_b bestimmt werden können und wie damit das Integral 2 umgestellt werden kann [1].

$$\begin{aligned} I(k,y,a,b) &= \int_{h_a} f(z)e^{ikg(z)}\, dz \int_{h_b} f(z)e^{ikg(z)}\, dz \\ &= \int_0^{\infty} h_a'(t) f(h_a(t)) e^{ikg(h_a(t))}\, dt - \int_0^{\infty} h_b'(t) f(h_b(t)) e^{ikg(h_b(t))}\, dt \end{aligned} \tag{2.2}$$

2.1 Definitionen

Um das Verfahren nachvollziehbar darzustellen sind einige Definitionen notwendig. Zunächst wird in Kapitel 2 von [2] das Integral aus Gleichung 2 hinsichtlich eines Dreiecks $\Delta \in \mathbb{R}^3$ und eines Richtungsvektors $\theta \in \mathbb{R}^3$ betrachtet. Das Dreieck wird als parametrisiertes Einheitsdreieck Λ_1 mit folgender Abbildung definiert: $x \to Ax + b$, mit $x = (x,y)^T$, $A := (A_l)_{1\le l\le 2} \in \mathbb{R}^{3\times 2}$, $A_l \in \mathbb{R}^3$ und $b \in \mathbb{R}^3$

Mit diesen lässt sich das Integral als

[1] [2, Kaptiel 1, Gleichung 4]

$$I_{\Delta}(k) = |A_1 \times A_2| \int_0^1 \int_0^{1-y} \frac{e^{ik(||Ax+b-r||+\theta\cdot(Ax+b))}}{||Ax+b-r||}\, dx\, dy =: |A_1 \times A_2| I^1_{\Delta_1}(k) \quad (2.3)$$

umformulieren. Die Autoren stellen definieren nun $P(x,y) := ||Ax+b-r||^2$ und $P(x)$, welches $P(x,y)$ für ein fixes y berechnet. Dieses lässt sich mithilfe der komplexen Wurzeln c und $\underline{c}$ von P und $c_0 \in [0,\infty[$ unter der Bedingung, dass r nicht auf der Dreiecksfläche liegt, als

$$P(x) = c_0 x^2 + c_0|c|^2 - 2c_0 Re(c) \quad (2.4)$$

umformulieren (vgl. Formel 6, Kapitel 2 [2]).

Bei festem y lässt sich die Funktion g als $g(x) = \sqrt{(}P(x)) + qx + s$ bestimmen. Dabei lassen sich q und s aus $qx + s := \theta \cdot (Ax+b)$ berechnen.

Auf dieser Grundlage lässt sich das Integral als

$$I(k,y,a,b) := \int_a^b \frac{e^{ikg(x)}}{\sqrt{(}P(x))}\, dx \quad (2.5)$$

formulieren.

Die Autoren teilen dieses Problem in zwei Fälle auf, einen eindimensinalen Fall und einen zweidimensionalen Fall der die Berechnung eines Integrals über ein Dreiecks abdeckt.

In der Arbeit [2] wird zunächst beschrieben wie sich das Problem auf Pfaden in der komplexen Ebene berechnen lässt auf denen $e^{ikg(x)}$ nicht oszilliert. Es wird gezeigt wie sogenannte *Splitting-points* berechnet werden, welche als Endpunkte für diese Pfade dienen. Liegt einer dieser Punkte auf einem komplexen Pfad, so kann das *steepest-descent*-Verfahren nicht angewandt werden und es muss auf ein klassisches Integrationsverfahren zurückgegriffen werden, welches diese Singulariäten berechnen kann.

2.2 Cauchyscher Integralsatz

Der Cauchyscher Integralsatz ist ein Satz der Funktionentheorie. Er besagt, dass ein Kurvenintegral $\int_\gamma f(z)\, dz$ enlang eines geschlossenen Weges γ Null ist wenn f auf der ganzen von γ eingeschlossenen Fläche holomorph ist (vgl. Kapitel 4, [3]).

Ein komplexwertige Funktion $f : D \to \mathbb{C}$ ist *holomorph* wenn f in jedem Punkt von D komplex differenzierbar ist. [3]

Für eine holomorphe Funktion $f : D \to \mathbb{C}$ mit offenem Definitionsbereich und homotopen Wegen $\gamma_0, \gamma_1 : [a,b] \to D$ gilt

$$\int_{\gamma_0} f(z)\, dz = \int_{\gamma_1} f(z)\, dz \quad (2.6)$$

2.3 Berechnung der *Splitting-points*

Damit das *Steepest-descent*-Verfahren angewendet werden kann muss sicher gestellt sein, dass die gefundenen Pfade so verbunden werden, dass keine Singularitäten auf ihnen liegen. Die Intervalle mit Singulariäten müssen mit einem klassischen Integrationsverfahren berechnet werden. In Kapitel 3 von [2] wird beschrieben wie diese Punkte berechnet werde können. Es existieren zwei Arten von Singularitäten

1. Nullstellen c_s von g,
2. und Anfangspunkt c_r eines Weges h_{c_r} der durch eine Wurzel t_{c_r} der Funktion P führt.

Da der Realteil von g entlang dieses Pfades konstant ist, wegen $g(h_x(t)) = \sqrt{P(h_x(t))} + qh_x(t) + s = \sqrt{P(x)} + qx + s + it$, können die *Splitting-points* mithilfe von q berechnet werden.

Es wird gezeigt, dass sich mithilfe einer Fallunterscheidung 6 verschiedene Konfigurationen für die Berechnungen von c_s bzw. c_r unterscheiden lassen.

Fall		c_s	c_r
$\|q\| = \infty$		$c, \overline{c}$	$Re(c)$
$\|q\| = 0$		$Re(c)$	$c, \overline{c}$
$\|q\| = \sqrt{c_0}$		$\|c_s\| = \infty$	$\|c_r\| = \infty$
$\|q\| < \sqrt{c_0}$	$q < 0$	$Re(c) + K_{c_s}$	$Re(c) + K_{c_r}$
	$q > 0$	$Re(c) - K_{c_s}$	$Re(c) - K_{c_r}$
$\|q\| > \sqrt{c_0}$	$q < 0$	$Re(c) + K_{c_s}$	$Re(c) + K_{c_r}$
	$q > 0$	$Re(c) - K_{c_s}$	$Re(c) - K_{c_r}$

Tabelle 2.1: Berechnung der *Splitting-points*

Dabei werden K_{c_s} und K_{c_r} wie folgt berechnet:

$$K_{c_s} := \sqrt{Re(c)^2 - \frac{c_0 Re(c)^2 - q^2|c|^2}{c_0 - q^2}} \tag{2.7}$$

$$K_{c_r} := \sqrt{Re(c)^2 + \frac{q^2 Re(c)^2 - c_0^2|c|^2}{c_0 - q^2}} \tag{2.8}$$

Da diese Punkte nicht mit dem *Steepest-descent*-Verfahren berechnet werden können muss der Interval bestimmt werden in welchem sich die *Splitting-points* befinden. Dieses Interval wird im Algorithmus durch die nächsten Nullstellen um diese Singularitäten bestimmt.

2.4 Berechnung von zulässigen Pfade

Die Pfade welche auf denen *Steepest-descent*-Verfahren werden darf, werden in der Arbeit *zulässige*(engl. *admissible*) Pfade genannt. Ein Pfad ist genau dann *zulässig* wenn sein Anfangspunkt zu $\mathbb{R}\backslash c_r, c_s$ gehört (siehe Definition 3, Kapitel 4.1 in [2]).

Für einen Pfad h_x mit Anfangspunkt x wird die Funktion $K_x(t)$ als

$$K_x(t) = \sqrt{P(x)} + qx + it \tag{2.9}$$

definiert und mithilfe dieser die beiden Funktionen $K^1_{h_x}$ und $K^2_{h_x}$ als

$$\begin{aligned} K^1_{h_x} :=& \frac{c_0 Re(c) - qK_x(t)}{c_0 - q^2} \\ K^2_{h_x} :=& \sqrt{\frac{K_x(t)^2 - c_0|c|^2}{c_0 - q^2} - (\frac{qK_x(t) - c_0 Re(c)}{c_0 - q^2})^2} \end{aligned} \tag{2.10}$$

definiert. Mithilfe dieser lassen sich wieder mit einer Fallunterscheidung über q die Pfade berechnen.

Fall		h_x
$\lvert q\rvert = \pm\sqrt{c_0}$		$\frac{K_x(t)^2 - c_o\lvert c\rvert^2}{\pm 2(K_x(t)\sqrt{c_0} \mp c_0 Re(c))}$
$\lvert q\rvert < \sqrt{c_0}$	$x < c_s$	$K^1_{h_x} - K^2_{h_x}$
	$x > c_s$	$K^1_{h_x} + K^2_{h_x}$
$\lvert q\rvert > \sqrt{c_0}$	$q < 0$	$K^1_{h_x} + K^2_{h_x}$
	$q > 0$	$K^1_{h_x} - K^2_{h_x}$

Tabelle 2.2: Berechnung der von h_x

Zur Berechnung des Verfahrens wird neben dem Pfad h_x auch die Ableitung h'_x benötigt. Diese lässt sich, wie in Kapitel 4.2 von [2] gezeigt, mithilfe von h_x berechnen:

$$h'_x(t) = \frac{i\sqrt{P(h_x(t))}}{c_0(h_x(t) - Re(c)) + q\sqrt{P(h_x(t))}} \tag{2.11}$$

2.5 Fall1: Das 1D-Verfahren

Der eindimenisionale Fall wird in Kapitel 5.1 von [2] beschrieben:

$$I(k, y, a, b) = I(k, y, a, a_1) + I(k, y, a_1, b_1) + I(k, y, b_1, b) \tag{2.12}$$

Dabei enthält das Intervall $[a_1, b_1]$ einen sogenannten Splitting-point und das Integral $I(k, y, a_1, b_1)$ muss mit einem klassischen Verfahren berechnet werden.

Es wird gezeigt, dass das Aufteilen der Integration an den *Splitting-points* und das Verbinden der Pfade auf welchen das *Steepest-descent*-Verfahren angewant wird die Voraussetzungen für den Cauchyschen Integralsatz erfüllen. Das erlaubt es das Integral $I(k, y, a, b)$ wie folgt umzustellen:

$$\begin{aligned} I(k, y, a, b) = &\int_0^\infty e^{ik(\sqrt{P(a)}+aq+s)} \frac{e^{-kt}}{\sqrt{P(h_a(t)}} h'_a(t)\, dt - \\ &\int_0^\infty e^{ik(\sqrt{P(b)}+bq+s)} \frac{e^{-kt}}{\sqrt{P(h_b(t)}} h'_b(t)\, dt \end{aligned} \tag{2.13}$$

2.6 Das Verfahren über Dreiecksoberflächen

In Kapitel 5.2 von [2] wird gezeigt wie das Verfahren auf das ursprüngliche Ziel erweitert wird. Dazu wird das zu berechnende Integral für ein Dreieck in n_y parallele Schichten aufgeteilt, welche jeweils mit dem eindimenisionalen Verfahren berechnet werden können. Das zweidimensionale Integral $I^1_{\Delta 1}(k)$ lässt sich ohne *Splitting-point* als

$$I^1_{\Delta 1}(k) = \int_0^1 I(k, y, 0, 1)\, dy \tag{2.14}$$

beschreiben. Dieses lässt sich als

$$\begin{aligned} I^1_{\Delta 1}(k) = \sum_{j=1}^{n_y} \int_{y_{j-1}}^{y_j} \Bigg[&e^{ik(\sqrt{P(0,y)}+q_y(1-y)+q_x y+s)} \int_0^\infty \frac{e^{-kt}}{\sqrt{P(h_0(t), y)}} h'_0(t)\, dt \\ &e^{ik(\sqrt{P(1-y,y)}+q_y(1-y)+q_x y+s)} \int_0^\infty \frac{e^{-kt}}{\sqrt{P(h_{1-y}(t), y)}} h'_{1-y}(t) \Bigg] dy \end{aligned} \tag{2.15}$$

umstellen (siehe Gleichung 19, Kapitel 5.2 [2]). Dabei fällt auf, dass verschachtelte Integrale zu berechnen sind. Die Autoren zeigen, dass diese ausgeklammert werden können. Dabei wird auf die Funktion

$$\Lambda_a : y \to \int_0^\infty \frac{e^{-kz}}{\sqrt{P(h_{a(y)}(t), y)}} h'_{a(y)}(t)\, dt \tag{2.16}$$

zurückgegriffen. Es wird gezeigt, dass sich diese mit einem vorab bestimmbaren vernachlässigbarem Fehlerterm approximieren lässt. Diese Λ_a sind konstant für jede Schicht und lassen sich mithilfe des eindimenisionalen Verfahrens berechnen. Mit dieser Beobachtung wird das Integral letztendlich als

$$\begin{aligned}\hat{I}_{\Delta_1,n_y}(k) = \sum_{j=1}^{n_y} \Bigg[& \Lambda_0(y_{m_j}) \int_{y_{j-1}}^{y_j} e^{ik(\sqrt{(P(0,y))}+q_y y+s)}\, dy \\ & -\Lambda_{1-y_{m_j}}(y_{m_j}) \int_{y_{j-1}}^{y_j} e^{ik(\sqrt{(P(1-y,y))}+q_x(1-y)+q_y y+s)}\, dy \Bigg]\end{aligned} \tag{2.17}$$

umgestellt.

2.7 Gauss-Laguerre Quadratur

Die Gauss-Laguerre Quadratur ist eine Approxmiation zur Auswertung von Integralen der Form

$$\int_0^\infty f(t) e^- t\, dt \tag{2.18}$$

Diese können wie folgt approximatiert werden:

$$\int_0^\infty f(t) e^- t\, dt \approx \sum_{i=1}^{n} w_i f(x_i) \tag{2.19}$$

Dabei werden die Gewichte w_i in dieser Arbeit Gauss-Laguerre-Gewichte und die Punkte x an denen f ausgewertet wird Guass-Laguerre-Knoten genannt.

Im *Steepest-descent*-Verfahren wird diese Approximation genutzt um die Pfade h in den Intervallen, welche keine Singularitäten enthalten zu berechnen. Dies entspricht im eindimenisionalen Fall dem Term $I(k, y, a, a_1)$ und $I(k, y, b_1, b)$.

3

Entwurf

In diesem Kapitel wird auf verschiedene Aspekte der Entwurfsphase eingegangen. Zum einen wird dargelegt mit welcher Herangehensweise der Algorithmus umgesetzt wird und zum anderen werden die Entscheidungen über die verwendeten Technologien begründet.

3.1 Anforderungen

Ziel dieser Arbeit ist es, das in [2] beschriebene Verfahren zu implementieren. Dabei sind drei Anforderungsbereiche auszumachen, welche in diesem Abschnitt behandelt werden.

3.1.1 Korrektheit

Die trivialste Anforderung ist, den beschriebenen Algorithmus korrekt zu implementieren. Diese Anforderung ist erfüllt, wenn die Rechenresultate der Implementierung eine Genauigkeit der gleichen Größenordnung wie die MatLab-Implementierung von [2] erzielen. Dort werden in Kapitel 6.1 (Tabelle 3) für die Parameter

$$A = \begin{pmatrix} 0 & 0 \\ 2 & 0 \\ 0 & 2 \end{pmatrix}, b = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix}, r = \begin{pmatrix} 0.6 \\ 0 \\ 0 \end{pmatrix}, \theta = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{3.1}$$

die relativen Fehler des implementierten Verfahrens mit der Matlab *integral* Funktion für verschiedene Wellenzahlen k verglichen. Das Kriterium der Korrekt-

k	Relativer Fehler
100	1.4410^{-15}
500	1.1510^{-14}
1000	9.7110^{-5}
3000	5.1110^{-7}
5000	1.3110^{-8}

Tabelle 3.1: Genauigkeit der Implementierung von [2]

heit wird als erfüllt angesehen, wenn die Implementierung höchstens Fehler der gleichen Größenordnung liefert.

3.1.2 Bedienbarkeit

Die Anforderung der Bedienbarkeit wird im Rahmen dieser Arbeit wie folgt definiert:

1. Es wird eine C++-Bibliothek bereitgestellt, welche mithilfe des Build-System *CMake* eingebunden werden kann.
2. Es wird eine oder mehrere Matlab-Mex Funktionen ausgeliefert, welche den Algorithmus in Matlab benutzbar machen.
3. Es wird ein Python-Modul bereitgesellt, mit dem die 1D und 2D Algorithmen genutzt werden können.

Dabei ist es nicht das Ziel, die beschriebenen Pakete offiziel auszuliefern. Das eigenhändige Compilieren der Bilbiothek und ggf. der Module ist erforderlich.

3.1.3 Geschwindigkeit

Der implementierte Algorithmus darf nicht langsamer als die bereitgestellte Matlab-Implementierung sein. Zu diesesm Zweck werden einige Auswertungen mit randomisierten Daten mithilfe von MatLab ausgeführt und direkt mit der vorhandenen MatLab-Implementierung verglichen.

3.2 Architektur

Diese Arbeit wird mit der Programmiersprache C++ umgesetzt. Die Anwendung wird in drei Module aufgeteilt. Die Hauptbibliothek `steepest_descent`, in welcher der eigentliche Algorithmus implementiert wird. Diese API-Endpunkte dienen den Python- und Matlab-Wrappern die nötigen Anknüpfpunkte.
Das Python-Modul `stedepy` welches die in der Hauptbibliothek API-Endpunkte bereitstellt, sowie eine Möglichkeit, die benötigten Gauss-Laguerre-Knoten vorab zu berechnen. Das Matlab-Modul, welches sich aus mehreren Mex-Funktionen zusammensetzt, welche die Funkionen der API bereitstellen.

3.2.1 Klassendiagramm

In Abbildung 3.1 wird ein vereinfachter Überblick über die implementierten Klassen und ihr Zusammenwirken gegeben. Die Klassen `integral_1d` und `integral_2d` sind dabei die Schnittstellen für das Python und das Matlab Modul. Die dargestellten Klassen sind verkürzt und die Funktionsparameter wurden ausgelassen, um die Lesbarkeit zu erhalten.

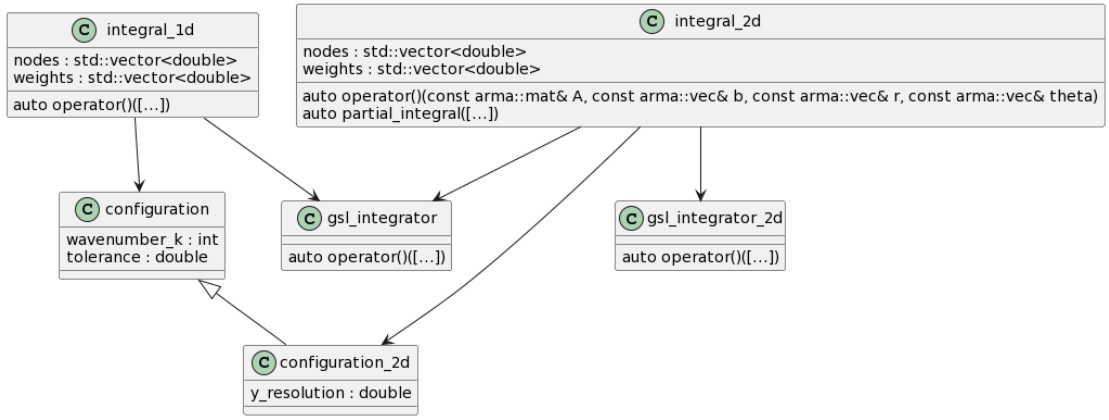

Abbildung 3.1: Vereinfachtes UML-Diagramm der Implementierung

3.2.2 Parameter der Integration

Die einzelnen API-Aufrufe teilen nur wenige gemeinsame Daten:

- Die Wellenzahl k,
- den Beobachtungspunkt r,
- die Anzahl an Knoten für das Gauss-Laguerre-Verfahren,
- sowie die gewünschte Auflösung im zweidimensionalen Fall

Diese werden, abgesehen von dem Beobachtungspunkt r, in einer Konfigurationsklasse (siehe Abbildung 3.2) zusammengefasst.

```
1 struct configuration {
2     double wavenumber_k;
3     double tolerance = 0.1;
4     size_t gauss_laguerre_nodes;
5 };

7 struct configuration_2d : public configuration{
8     double y_resolution = 0.1;
9 };
```

Abbildung 3.2: Die Konfigurationen

Die Parameter einer Integration eines Dreiecks teilen sich wie folgt auf:

- Das Dreieck als parametrisiertes Einheitsdreieck, bestehend aus einer 2×3-Matrix A und einem Verschiebungsvektor b,
- der Beobachtungspunkt r,
- ein Richtungsvektor θ

Zusätzlich wird noch der Beobachtungspunkt als Parameter mitgereicht, dies ermöglicht eine vollständige Kompabilität mit der Matlab-Implementierung.

3.2.3 Functor-Objekte

Die Hauptkomponenten zur Integrationen werden als sogenannte *Functor*-Objekte (Funktor) implementiert. Diese Datenstruktur zeichnet sich dadurch aus, dass sie den C++-Funktionsaufrufsoperator bereitstellen und wie einfache C++-Funktionen aufgerufen werden können. Da Funktoren durch Klassen bzw. *structs* definiert werden, kann in ihnen ein Zustand gespeichert werden. In Abbildung 3.3 ist die Header-Definition des Funktor für das Berechnen des zweidimensionalen Falls gezeigt. In den Konstruktoren werden Parameter übergeben, welche für mehr als eine Integration invariant sind. In der Implementierung des Funktionsaufrufsoperators werden die im vorherigen Abschnitt definierten Parameter erwartet.

```
1  #pragma once
2
3  #include "integration/gsl_integrator.h"
4  #include "integration/gsl_integrator_2d.h"
5  #include "datatypes.h"
6  #include "configuration.h"
7  #include <armadillo>
8  #include <vector>
9  #include <complex>
10
11 namespace integral {
12
13   struct integral_2d {
14     integral_2d(const config::configuration_2d config, integrator::
       gsl_integrator* integrator, integrator::gsl_integrator_2d* integrator_2d)
       ;
15     integral_2d(const config::configuration_2d config, integrator::
       gsl_integrator* integrator, integrator::gsl_integrator_2d* integrator_2d,
        const std::vector<double> nodes, const std::vector<double> weights);
16
17         auto operator()(const arma::mat& A, const arma::vec& b, const arma::
       vec& r, const arma::vec& theta) const->std::complex<double>;
18   private:
19     config::configuration_2d config;
20     integrator::gsl_integrator* integrator;
21     integrator::gsl_integrator_2d* integrator_2d;
22     std::vector<double> nodes, weights;
23
24     auto get_partial_integral(const arma::mat& A, const arma::vec& b, const
       arma::vec& r, const std::complex<double> sPx, const double q,  const std
       ::complex<double> s, const std::complex<double> c, const double c_0,
       const double left_split, const double right_split) const ->std::complex<
       double>;
25   };
26 }
```

Abbildung 3.3: Funktordefinition der zweidimensionalen Integration

4

Vorgehensweise

In diesem Kapitel wird darauf eingegangen, wie die in Kapitel 3.1.3 definierten Ziele erreicht werden.

4.1 Grober Plan

Damit das Ziel der schnellen Performance erreicht wird, wird die folgende Vorgehensweise genutzt.

Zu Beginn wird der Algorithmus ohne Rücksicht auf Perfomanzkriterien implementiert. Dabei wird mithilfe von Unittests sichergestellt, dass sich die Implementierung korrekt verhält. Die korrekten Eregbnisse einzelner Tests können mithilfe der vorliegenden Matlab-Implementierung verifiziert werden.

Nachdem in diesem ersten Schritt ein korrekter Algorithmus vorliegt kann mit der Optimierung begonnen werden. Dazu werden folgende Methoden verwendet, auf welche in den jeweiligen Unterkapiteln eingegangen wird.

- Profiling
- Hotpath-Analyse
- Benchmarking
- Parallelisierung isolierter Bereiche

4.2 Profiling

Mithilfe der Profiling-Werkzeuge der Entwicklungsumgebung Visual Studio lässt sich das Laufzeitverhalten einer Anwendung analysieren. In dieser Arbeit wird dabei das Messen der Geschwindigkeit als primäre Metrik genutzt.

4.3 Hotpath-Analyse

Die Ergebnisse des Profilings ermöglichen es, den sogenannten Hotpath (häufig auch Critical-Path genannt) eines Algorithmus zu finden. Das Ziel dieser Betrachtung ist es, die Funktion(en) zu finden, welche den größten Beitrag zu der gemessenen Laufzeit hat. Die so gefundenen Funktionen sind die besten Kandidaten für

Optimierungsmaßnahmen, da jede andere Funkion weniger zu der Laufzeit beiträgt.

4.4 Benchmarking

Um die Laufzeit der Implementierung beurteilen zu können, wird auf zweierlei Arten von Benchmarks gesetzt.

Zum einen automatisierte Benchmarks, welche vorallem die API-Endpunkte mit verschiedenen Testdaten ausführen und das Laufzeitverhalten messen. Diese Ergebnisse werden genutzt um während der Entwicklung schnelle Rückmeldung über einzelne Maßnahmen zu erhalten.

Die zweite Variante der Benchmarks ist das manuelle Ausführen von API-Aufrufen, welche mithilfe von Timingfunktionen gemessen werden. Diese werden für Tests genutzt, bei denen eine Automatisierung mit erheblichem Mehraufwand verbunden wäre. Diese Vorgehensweise wird beispielsweise beim Vergleich der Matlabimplementierung mit den Matlab-Modulen angewandt.

4.5 Parallelisieren isolierter Bereiche

Einige mögliche Parallesierungen lassen sich direkt aus der Problemstellung ablesen. So ist beispielsweise die Berechnung des zweidimensionalen Falls von zwei Dreiecken Δ_1 und Δ_2 abgesehen von der Wellenzahl k, des Beobachtungspunktes θ und der Anzahl der Gauss-Laguerre-Knoten unabhängig voneinander. Diese Berechnungen lassen sich Dementsprechend relativ unproblematisch Parallelisieren da keine klassischen Threading-Konflikte auftreten können. Da der Zugriff auf diese invarianten Parameter lediglich lesend ist, können die Zugriffe auf diese Ressourcen sogar ohne Synchronisierungen durchgeführt werden.

Ähnliches gilt für das Berechnen der einzelene Schichten im zweidimensionalen Fall. Die Ergebnisse einzelner Schichten beeinflussen sich gegenseitig nicht.

4.6 Sonstige

Desweiteren werden mithilfe von Compiler- und Linker-Flags einige Optimierungen hinsichtlich der Laufzeit ausgewählt.

5

Implementierung

In diesem Kapitel wird dargelegt, wie der Algorithmus implementiert wird. Zu Beginn wird erläutert warum, welche Technologien zum Einsatz kommen und wie diese verwendet werden.

5.1 Verwendete Technologien

In diesem Abschnit werden die verwendeten Technologien vorgestellt.

5.1.1 GNU Scientific Library

Die GNU Scientific Library (GSL) ist eine Sammlung von numerischen Funktionen[1]. Die GSL wird unter der GNU General Public License veröffentlicht und ist in der Programmiersprache C geschrieben.

Die GSL bietet unter anderem Funktionen für *Basic linear Algebrar Subprograms*(BLAS), verschiedene Interpolationsalgorihtmen, Monte-Carlo-Algorithmen, Implementierung für Fast Forier Tranformationen und die für diese Arbeit benötigten Algorithmen für die numerische Integration. GSL bietet eine Reimplementierung des QUADPACK[7], eine Fortran Bibiliothek für numerische Integration. Diese bietet verschiedene Algorithmen an welche ein Integral der Form

$$I = \int_a^b f(x)w(x)\,dx$$

lösen, wobei $w(x)$ eine Gewichtungsfunktion ist. Um die Genauigkeit festzulgen müssen die Grenzen *epsabs* und *epsrel* für die absoluten und relativen Fehler angegeben werden. Die Genauigkeit ist über die Ungleichung

$$|RESULT - I| \leq \max(epsabs, epsrel|I|)$$

definiert, wobei $RESULT$ die Apprixmation des Algorithmus darstellt. Die Algorithmen liefern das erste Ergebnis dessen absoluter Fehler kleiner als *epsabs* bzw. dessen relativer Fehler kleiner als *epsrel* ist.

In dieser Arbeit wird das sogenannte *CQUAD*-Integrationsverfahren verwendet. Im Handbuch der GSL wird dieses Verfahren als ein neues *doubly-adaptive*

general-purpose-Qaudartur-Verfahren [1, Kapitel 17.11] beschrieben.
Dieses Verfahren kann nicht direkt genutzt werden, um die benötigten Integrationen aus [2] zu berechnen, da der *CQUAD*-Algorithmus zum einen nur für eindimensionale Integrale und nur für den reellen Zahlenraum definiert ist.

Um dies zu ermöglichen, sind zwei Anpassungen nötig:

1. Das Verschachteln von zwei *CQUAD* Aufrufen
2. Das Auftrennen in zwei getrennte Aufrufe, jeweils für den Real- und Imaginärteil der Integranden

Die Implementierung für den Realteil ist in 5.1 zu sehen.

```
1  template<class integrand_fun>
2  auto operator()(integrand_fun integrand, const std::complex<double> x_start,
       const std::complex<double> x_end, const double y_start, const double
     y_end) const ->std::complex<double>
3  {
4      double result_real, result_imag;

6      auto f_real = make_gsl_function([&](double x) {
7          double inner_result;
8          auto inner = make_gsl_function([&](double y) {
9              return std::real(integrand(x, y));
10             });
11         gsl_integration_cquad(&inner, y_start, y_end, 1e-16, 1e-6,
     inner_workspace, &inner_result, NULL, NULL);
12         return inner_result;
13         });


16     auto status = gsl_integration_cquad(&f_real, std::real(x_start), std::
     real(x_end), 1e-16, 1e-6, workspace, &result_real, NULL, NULL);
17 [...]
```

Abbildung 5.1: Berechnung von komplexem Integral mit GSL

5.1.2 Armadillo

Armadillo ist eine C++-Bibliothek von Ryan Curtin, welche Datenstrukturen und Algorithmen der linearen Algebra bereitstellt. Die Funktionalität uns Syntax ist an die von Matlab angelehnt mit dem Ziel die Umsetzung von *Researchcode* in Produktivcode möglichst einfach zu gestalten [8].

```
1          arma::mat A = {{0, 0} , {1, 0}, {0, 1}};
2          //use matrix
3          // ...
4
```

```
1          gsl_matrix * m = gsl_matrix_alloc (3, 2);
2          gsl_matrix_set(m, 0, 0) = 0;
```

```
3         gsl_matrix_set(m, 0, 1) = 0;
4         gsl_matrix_set(m, 1, 0) = 1;
5         gsl_matrix_set(m, 1, 1) = 0;
6         gsl_matrix_set(m, 2, 0) = 0;
7         gsl_matrix_set(m, 2, 1) = 1;
8         //use matrix
9         // ...
10        //free
11        gsl_matrix_free (m);
12
```

Abbildung 5.2: Vergleich von Matrizen in Armadillo und GSL

Die GSL bietet zwar auch ein Framework für lineare Algebra, allerdings ist die API von Matlab deutlich moderner und einfacher zu verwenden (siehe Abbildung 5.2).

5.1.3 Intel Threading Building Blocks

Die Intel© oneAPI Threading Building BLocks (oneTBB) ist eine Template-basierte Bibliothek zur effizienten Parallelisierung von Anwendungen. Mithilfe von oneTBB ist es möglich mit wenigen Schritten einen Algorithmus auf mehreren Threads auszuführen.

oneTBB bietet für diverse Parallelisierungsprobleme Algorithmen und Datenstrukturen an, von einfachen Schleifen, bis hin zu Graph-Based-Parallel Computing. In dieser Arbeit wurden lediglich die einfacheren Konzepte von parallelen Schleifen genutzt. Für diese Anwendungsfälle stellt oneTBB, unter anderem, die Funktionen `parallel_for` und `parallel_reduce` zur Verfügung. Diese arbeiten auf sogenannten *Ranges*, welche eine Abstraktion der zu bearbeitenden Daten sind.

Ranges können die zu verarbeitende Datenmenge in kleinere Bereiche aufteilen. Diese Teilbereiche werden von oneTBB dann parallelisiert. Die Daten innerhalb einer Range werden sequenziell verarbeitet. Wie groß diese Teilbereiche werden und welcher Art die Aufteilung ist, kann mithilfe von Parametern gesteuert werden. In diser Arbeit werden dies mit der sogenannten Blocked-Ranges (`blocked_range`) realisiert, welche die Datenmenge in kontinuierliche Blöcke aufteilt. Die größe der jeweiligen Blöcke wird mit der sogenannten *Grainsize* gesteuert. oneTBB bietet verschiedene Arten von Blocked-Ranges für bis zu 3 Dimensionen. Die Grainsize muss dem zu lösenden Problem entsprechend groß gewählt werden.

```
1  return parallel_reduce(
2         // array von floats, mit Blöcken der Größe grainsize
3         blocked_range<float*>( array, array+n, grainsize),
4         // Intialer Wert für Vereinigung
5         0.f,
6         //Transformation
7         [](const blocked_range<float*>& r, float init)->float {
8             // Iteriert über jedes Element des Blocks r
9             for( float* a=r.begin(); a!=r.end(); ++a )
10            // und addiert das transformierte Element zur Gesamtsumme
11                init += do_something(*a);
12            return init;
13        },
```

```
14          // Vereinigung
15          []( float x, float y )->float {
16              return x+y;
17          }
18      );
```

Abbildung 5.3: Minimalbeispiel von `paralell_reduce`

In dieser Arbeit werden ausschließlich Parallelisierungen mithilfe von `parallel_recude` bzw. `parallel_deterministic_reduce` umgesetzt. Dieses bietet die Möglichkeit für jedes Datum eine Transformation durchzuführen und anschließend die tranformierten Daten zu vereinigen. So wird beispielsweise in der Implementierung (siehe Abschnitt 5.2.1) des Gauss-Laguerrre Verfahrens ein zu integrierender Pfad parallel an den einzelnen Gauss-Laguerre-Knotenpunkten ausgewertet und entsprechend gewichtet. Die so berechneten Teilwerte werden nun noch addiert und ergeben so das komplexe (Teil-)Integral des Pfades.

5.1.4 Pybind11

Das Python-Modul wird mithilfe der Bibliothek *pybind11* umgesetzt. Pybind11 ist eine sogenannte Header-only Bibliothek mit dem Ziel eine leichtgewichtige Alternative zu bestehenden Bibliotheken zur Erstellung von Python-bindings für C++-Bibliotheken anzubieten. Pybind11[4] ist ein Opensource-Projekt und wird unter einer BSD-artigen Lizenz veröffentlicht.

Folgendes Beispiel[1] stellt die C++-Funktion *add* als Python-Modul bereit:

```
1  #include <pybind11/pybind11.h>

3  int add(int i, int j) {
4      return i + j;
5  }

7  PYBIND11_MODULE(example, m) {
8      m.doc() = "pybind11 example plugin"; // optional module docstring

10     m.def("add", &add, "A function that adds two numbers");
11 }
```

Dieses kann in einer Pythonumgebung wie folgt verwendet werden:

```
1  import example;
2  example.add(1, 2);
```

[1] Beispielcode stammt von `https://pybind11.readthedocs.io/en/latest/basics.html` aus der pybind-Dokumenation[4].

5.1.5 Matlab Mex-Funktion

Um eine C++-Funktion in Matlab zur Verfügung zu stellen, gibt es mehrere Ansätze. Zum einen kann mithilfe von Matlab C++-Code kompiliert und anschließend genutzt werden. Die für den Endnutzer einfachere Variante ist das Bereitstellen von sogenannten Mex-Funktionen.

Um eine Mex-Funktion zu implementieren müssen die Header-Dateien `mex.hpp` und `mexAdapter.hpp` inkludiert werden und eine Funktion mit dem Namen *MexFunction* implementiert werden.

In Abbildung 5.4 ist eine Beispiel-Implementierung zu sehen.

```
1  #include "mex.hpp"
2  #include "mexAdapter.hpp"

4  class MexFunction : public matlab::mex::Function {
5  public:
6      void operator()(matlab::mex::ArgumentList outputs, matlab::mex::
       ArgumentList inputs) {
7          // Function implementation
8          ...
9      }
10 };
```

Abbildung 5.4: Eine einfache Mex-Funktion ohne Logik

Nachdem diese kompiliert wurde, lässt sie sich aus Matlab heraus aufrufen.

5.1.6 Sonstige

Als Testframework für automatisierte Unit-Tests wird die Bilbiothek *catch2* verwendet. Zum Durchführen der automatisierten Benchmarks wird auf die Bibliothek *Benchmark* von Google zurückgegriffen.

5.2 Ausgewählte Codestellen

In diesem Abschnitt werden Details der Implementierung näher beleuchtet. Es werden ausgewählte Codestellen vorgestellt, die wesentliche Funktionen implementieren.

5.2.1 Gauss laguerre integration und Cauchy Integral Theoerem

Die Berechnung der Pfadintegrale wird mithilfe des Gauss-Laguerre-Verfahrens realisiert. Da die Auswertung der Pfadfunktion an den Gauss-Laguerre-Knoten unabhängig voneinander ist, wird diese Berechnung mithilfe der Funktion `parallel_determinictic_reduce` der oneTBB API parallisiert. Diese Funktion verhält sich ähnlich wie `parallel_reduce`, hat jedoch ein anderes Verhalten hinsichtlich des sogenannten *splittings* der Ranges. Die *Grainsize* ist auf 100 Elemente festgelegt, kleinere Blockgrößen haben in Performanztests keine signifikante Beschleunigung ermöglicht und größere haben die Laufzeit negativ beeinflusst.

```cpp
auto calculate_integral_cauchy(const path_utils::path_function path, std::
    vector<double> nodes, std::vector<double> weights)->std::complex<double>
{
    auto size = (int)nodes.size();
    // Parallelisierung
    auto result = tbb::parallel_deterministic_reduce(tbb::blocked_range(0,
    size, 100), 0. + 0.i, [&](tbb::blocked_range<int> range, std::complex<
    double> integral)
        {
            // Auswertung für den Block
            for (auto i = range.begin(); i < range.end(); ++i)
            {
                // Auswertung von $path$ an Laguerre-Konten i mit
    entsprechendem Gewicht
                integral += (weights[i] * path(nodes[i]));
            }
            return integral;
        },
        // Vereinigung
        std::plus<std::complex<double>>());
    return result;
}
```

Abbildung 5.5: Implementierung der Gauss-Laguerre Quadratur

5.2.2 Gewichtung der Pfade

In der Datei *path_utils.cpp* werden Funktionen zur Berechnung der gewichteten Pfade bereitgestellt. Diese werden dann wie im Abschnitt 5.2.1 beschrieben verwendet um die *zulässigen* Pfade zu berechnen. Es werden drei verschiedene Gewichtungen vorgenommen:

- Für den eindimenisionalen Fall
- Für den Λ-Term im zweidimensionalen
- Für die Pfade im zweidimensionalen

Die Implementierung ist in Abbilung 5.6 zu sehen.

```cpp
//Für den 1D-Fall
auto get_weighted_path(const std::complex<double> split_point, const std::
    complex<double> y, const datatypes::matrix& A, const  arma::vec3& b,
    const  arma::vec3& r, const double q, const double k, const std::complex<
    double> s, const datatypes::complex_root complex_root, const std::complex
    <double> sing_point)->path_function {
    auto Px = math_utils::calculate_P_x(split_point, y, A, b, r);
    auto path = get_complex_path(split_point, Px, q, complex_root,
    sing_point);
    auto derivative = get_path_derivative(path, y, A, b, r, q, complex_root)
    ;

    return [=](const double t) -> auto {
        return derivative(t / k) * std::exp(1i * k * (std::sqrt(math_utils::
    calculate_P_x(split_point, y, A, b, r)) + q * split_point + s)) * (1. / k
    ) * (1. / std::sqrt(math_utils::calculate_P_x(path(t / k), y, A, b, r)));
    };
}

// Für den 2D-Fall
```

```
13 auto get_weighted_path_2d(const std::complex<double> split_point, const std
       ::complex<double> y, const datatypes::matrix& A, const  arma::vec3& b,
       const  arma::vec3& r, const double q, const double k, const std::complex<
       double> s, const datatypes::complex_root complex_root, const std::complex
       <double> sing_point)->path_function {
14     auto Px = math_utils::calculate_P_x(split_point, y, A, b, r);
15     auto path = get_complex_path(split_point, Px, q, complex_root,
       sing_point);
16     auto derivative = get_path_derivative(path, y, A, b, r, q, complex_root)
       ;

18     return [=](const double t) -> auto {
19         return derivative(t / k) * (1. / k) * (1. / std::sqrt(math_utils::
       calculate_P_x(path(t / k), y, A, b, r)));
20     }; /
21 }

23 // Für die Berechnung von Λ_a
24 auto get_weighted_path_y(const std::complex<double> split_point, const std::
       complex<double> y, const datatypes::matrix& A, const  arma::vec3& b,
       const  arma::vec3& r, const double q, const double k, const std::complex<
       double> s, const datatypes::complex_root complex_root, const std::complex
       <double> sing_point)->path_function {
25     auto Px = math_utils::calculate_P_x(split_point, y, A, b, r);
26     auto path = get_complex_path(split_point, Px, q, complex_root,
       sing_point);
27     auto derivative = get_path_derivative(path, y, A, b, r, q, complex_root)
       ;

29     return [=](const double t) -> auto {
30         return derivative(t / k) * (1. / k) * std::exp(1.i * k * (std::sqrt(
       Px) + q * split_point + s));
31     };
32 }
```

Abbildung 5.6: Funktionen zur Berechnung gewichteter Pfade

5.2.3 1D Integration

Die 1D-Integration ist als sogenanntes *Functor-Objekt* implementiert.

```
1 struct integral_1d {
2     integral_1d(const config::configuration config, integrator::
      gsl_integrator* integrator);
3     integral_1d(const config::configuration config, integrator::
      gsl_integrator* integrator, const std::vector<double> nodes, const std::
      vector<double> weights);
4     integral_1d_test(const config::configuration& config, integrator::
      gsl_integrator* integrator, const std::vector<double> nodes, const std::
      vector<double> weights, const path_utils::path_function_generator
      path_function_generator, math_utils::green_fun_generator
      green_fun_generator);


7     auto operator()(const arma::mat& A, const arma::vec3& b, const arma::
      vec3& r, const arma::vec3& theta, const double y, const double left_split
      , const double right_split) const->std::complex<double>;
8     auto operator()(const arma::mat& A, const arma::vec3& b, const arma::
      vec3& r, const double q, const double s, const double y, const double
      left_split, const double right_split) const -> std::complex<double>;
9
10   private:
11     config::configuration config;
```

```
12      integrator::gsl_integrator* integrator;
13      std::vector<double> nodes, weights;
14          path_utils::path_function_generator path_function_generator;
15      math_utils::green_fun_generator green_fun_generator;
16    };
```

Abbildung 5.7

Berechung der Singularität

In Kapitel 5.1 [2, S. 12] (Formel 16) wird die Formel für den eindimenisionalen Fall, im Fall einer Singularität im Interval $[a_1, b_1]$ wie folgt beschrieben:

$$I(k, y, a, b,) = I(k, y, a, a_1) + I(k, y, a_1, b_1) + I(k, y, b_1, b) \tag{5.1}$$

Die partiellen Integrale in den Intervallen $[a, a_1]$ und $[b_1, b_1]$ können mithilfe des *steepest-descent*-Verfahrens berechnet werden. Lediglich das verbleibende Interval muss mit einem klassischen Integrationsverfahren berechnet werden. Dieses Verfahren muss also fähig sein Singularitäten in einem Integral zu berechnen. Dies ist nicht bei allen Integrationsverfahren gegeben (z.B. dem sogenannten QAG-Verfahren aus der GSL).

und wird in dieser Arbeit wie folgt berechnet:

```
1  auto& [sp1, sp2] = split_points;

3  //Berechnung von I(k,y,a,a1) und I(k,y,b1,b)
4  auto I1 = steepest_desc(left_split, sp1);
5  auto I2 =  steepest_desc(sp2, right_split);


8  // Berechnung der um die Singularität I(k,y,a1,b1)
9  auto& local_k = this->config.wavenumber_k;
10 auto green_fun = [k=local_k, y=y, &A, &b, &r, q, s](const double x) -> auto
       {
11     auto Px = math_utils::calculate_P_x(x, y, A, b, r);
12     auto sqrtPx = std::sqrt(Px);
13     auto res =  std::exp(1.i * k * (sqrtPx + q * x + s)) * (1. / sqrtPx);
14     return res;
15 };
16 auto x = integrator->operator()(green_fun, sp1, sp2);
17 // I(k,y,a,a1) + I(k,y,a1,b1) + I(k,y,b1,b)
18 return I1 + x + I2;
```

Abbildung 5.8: Implementierung des 1D-Falls (Singularität)

Dabei werden die Intervalle ohne Singularität mit dem Gauss-Laguerre-Verfahren (siehe Abbildung 5.5) berechnet und die verbleibende partielle Integration, welches die Form

$$I(k, y, a, b) := \int_a^b \frac{e^i k g(x)}{\sqrt{P(x)}}\, dx$$

hat, mithilfe eines Aufruf des GSL-Integrators berechnet (Analog zum 2D Fall, siehe Abbildung 5.1)

5.2.4 2D Integration

Ebenso wie die 1D-Integration wird die 2D-Integration als sogenanntes *Functor-Objekt* implementiert (siehe Abbildung 3.3). Im Konstruktor werden die invarianten Parameter übergeben und in dem -*Operator* werden die Dreiecksparameter, der Blickpunkt r und der Richtungsvektor θ übergeben.

In diesem Abschnitt wird primär auf die Berechnung eines einzelnen Schicht n eingangen. Die Anzahl der zu berechnenden Schichten wird aus der Konfiguration(siehe Abbildung 3.2) abgeleitet. Die Menge der Schichten wird mithilfe der oneTBB Funktion `parallel_reduce` parallelisiert.

```
integration_result = tbb::parallel_reduce(tbb::blocked_range(0,
    number_of_steps, 1), 0. + 0.i, [&](tbb::blocked_range<int> range, std::
    complex<double> integral)
{
    // [...] Berechnung des 2D-Integrals pro Layer
    return integral;
}, std::plus<std::complex<double>>());
```

Abbildung 5.9: Parallelisierung des 2D-Integrals

Dabei hat sich gezeigt, dass eine Blockgröße von 1 zu den besten Ergebnissen führt. Die zu Berechnende Schicht kann nun in zwei Fälle unterteilt werden: Mit Singularität und ohne.

Berechnung von Λ_a

Die Λ_a-Terme werden mithilfe des eindimenisionalen Verfahrens berechnet. Dabei wird dieser mit angepassten Funktionen für die Pfadgewichtung sowie angepasstem Integranden für die klassische Integration.

```
integrator::gsl_integrator integrator_1d;
config::configuration config1d;
config1d.wavenumber_k = config.wavenumber_k;
config1d.tolerance = config.tolerance;

// zu integrierende Funktion für GSL
math_utils::green_fun_generator fun_gen = [](const double& k, const std::
    complex<double>& y, const arma::mat& A, const arma::vec3& b, const arma::
    vec3& r, const double& q, const std::complex<double>& s) -> math_utils::
    green_fun
{
    return [&](const double x) -> auto {
        auto Px = math_utils::calculate_P_x(x, y, A, b, r);
        auto sqrtPx = std::sqrt(Px);
        auto res = std::exp(1.i * k * (sqrtPx + q * x + s));
        return res;
    };
};
integral_1d_test partial_integral(config1d, &integrator_1d, nodes, weights,
    path_utils::get_weighted_path_y, fun_gen);
```

Im Quellcode werden Aufrufe dieser Instanz mit *partial_integral* ausgeführt.

Fall 1: Keine Singularität in Schich n

Dabei ist der Fall ohne Singularität der trivialere Fall:

```
1  // Berechnung der Schrittparameter
2  auto y = config.y_resolution * (double)i;
3  auto u = y + config.y_resolution * 0.5;

5  //Berechnung der potentielllen Singularitäten
6  auto [c, c_0] = math_utils::get_complex_roots(u, A, b, r);
7  auto sing_point = math_utils::get_singularity_for_ODE(q, { c, c_0 });
8  auto spec_point = math_utils::get_spec_point(q, { c, c_0 });


11 // Prüfen ob Singulariäten vorliegen
12 auto is_spec = math_utils::is_singularity_in_layer(config.tolerance,
       spec_point, 0, 1 - u);
13 auto is_sing = math_utils::is_singularity_in_layer(config.tolerance,
       sing_point, 0, 1 - u);

15 // Berechnung der Terme Λ0(y_mj) und Λ1−y_mj(y_mj)
16 auto integration_y =  get_partial_integral(A1, b, r, 0., q1, sx1, c1, c1_0,
       y, y + config.y_resolution);
17 auto integration_1_minus_y =  get_partial_integral(A2, b, r, 1., q2, sx2, c2
       , c2_0, y, y + config.y_resolution);

19 // Berechnung der Pfade mithilfe der Gauss-Laguerre-Quadratur
20 auto s = arma::dot(A.col(0), theta) * u + prod;
21 auto path = path_utils::get_weighted_path_2d(0, u, A, b, r, q, config.
       wavenumber_k, s, { c, c_0 }, sing_point);
22 auto Iin = gauss_laguerre::calculate_integral_cauchy_tbb(path, nodes,
       weights);

24 auto path2 = path_utils::get_weighted_path_2d(1 - u, u, A, b, r, q, config.
       wavenumber_k, s, { c, c_0 }, sing_point);
25 auto Ifin = gauss_laguerre::calculate_integral_cauchy_tbb(path2, nodes,
       weights);

27 //Ergebniss für den Layer
28 integral += Iin * integration_y - Ifin * integration_1_minus_y;
```

Abbildung 5.10: Schicht n ohne Singularität

Fall 2: Singularität in Schicht n

Diesem Fall gehen die Berechnungen des ersten Falles vorraus (angedeutet in Abbildung 5.11) In Schicht n liegt eine Singularität vor. Für diese werden die *Splitting-points* berechnet. Mithilfe dieser wird der Teilpfad für das klassische Integrationsverfahren bestimmt. (Berechnung des Integrals in Zeile 32 Abbildung 5.11). Die Implementierung dieser Funktion ergibt sich nicht komplett aus der Arbeit [2], denn in der zugrundeliegenden Matlab-Implementierung wird ersichtlich, dass für den Fall der Singularität eine weitere Sache berücksichtigt werden muss: Eine von den Autoren der Implementierung genannte *Integrationsbox*, welche eine Ungenauigkeit bei der Berücksichtigung der Singularität darstellt. Diese lässt sich wie folgt erklären: Durch das Aufteilen der zu integrierenden Dreiecksoberfäche in Schichten ist wird die Position der Singularität, bedingt durch die Höheder Schicht, ggf. zu

stark gewichtet. Dies wird mit einer partiellen Integration an dieser Stelle über die vertikale Dimension ausgeglichen (Zeilen 26-29).

```
1  // [...]
2  if (!is_spec && !is_sing) {
3      // Keine Singularität, Fall 1
4      continue;
5  }
6  else if (is_spec) {
7      split_points = math_utils::get_split_points_spec(q, config.wavenumber_k,
        s, { c, c_0 }, 0, 1 - u);
8  }
9  else if (is_sing) {
10     split_points = math_utils::get_split_points_sing(q, config.wavenumber_k,
        s, { c, c_0 }, 0, 1 - u);
11 }
12 auto& [split_point1, split_point2] = split_points;

14 // Singularität entlang der Schicht

16 auto path3 = path_utils::get_weighted_path_2d(split_point1, u, A, b, r, q,
       config.wavenumber_k, s, { c, c_0 }, sing_point);
17 auto Ifin1 = gauss_laguerre::calculate_integral_cauchy_tbb(path3, nodes,
       weights);

19 auto path4 = path_utils::get_weighted_path_2d(split_point2, u, A, b, r, q,
       config.wavenumber_k, s, { c, c_0 }, sing_point);
20 auto Iin2 = gauss_laguerre::calculate_integral_cauchy_tbb(path4, nodes,
       weights);


23 auto sx_intern_1 = arma::dot(A1.col(1), theta) * split_point1 + prod;
24 auto [cIntern1, c_0Intern1] = math_utils::get_complex_roots(split_point1, A1
       , b, r);
25 auto sx_intern_2 = arma::dot(A1.col(1), theta) * split_point2 + prod;
26 auto [cIntern2, c_0Intern2] = math_utils::get_complex_roots(split_point2, A1
       , b, r);

28 auto intYintern1 = get_partial_integral(A1, b, r, split_point1, q1,
       sx_intern_1, cIntern1, c_0Intern1, y, y + config.y_resolution);
29 auto intYintern2 = get_partial_integral(A1, b, r, split_point2, q1,
       sx_intern_2, cIntern2, c_0Intern2, y, y + config.y_resolution);


32 auto integral2_res = integrator.operator()(green_fun_2d, split_point1,
       split_point2, y, y + config.y_resolution);

34 auto integration_result = integral2_res + Iin2 * intYintern2  - Ifin1 *
       intYintern1;
35 integral += integration_result;
```

Abbildung 5.11: Schicht n mit Singularität

6

Analyse und Auswertung

6.1 Testsystem

Die Performanztests wurden auf einem Vierkernprozessor Intel i5-4590 mit 8GB Arbeitsspeicher durchgeführt.

6.2 Analyse mit Valgrind

In diesem Abschnit wird auf die Analyse des Projektes mit Valgrind eingegangen. Zunächst werden Valgrind und die daraus benutzten Tools vorgestellt und dann werden einige Ergebnisse präsentiert.

6.2.1 Valgrind

Valgrind ist einerseits ein Framework zum Erzeugen von dynamsichen Analyse Werkzeugen und andereseits eine Sammlung ebensolcher Werkzeuge (siehe [5]). Valgrind ist offene Software und wird unter der GNU GPL-2 Lizenz veröffentlicht[1]. Die Sammlung bietet acht Werkzeuge[2] an von denen eines noch den Status eines experimentellen Werkzeugs hat:

- Memcheck, ein Werkzeug zum Analysieren von Speicherlecks.
- Cachegrind, ein Cache und Branch-prediction Profiler.
- Callgrind, ein Profiler der einen Aufrufgraphen erzeugt.
- Helgrind und DRD, Werkzeuge um Threading-Fehler zu erkennen.
- Massif und DHAT, Heapprofiler für Analysen hinsichtlich speichereffizienter Programme
- BBV, ein Werkzeug Forschung im Bereich der Rechnerarchitektur

In dieser Arbeit wurden die Werkzeuge Memcheck, Cachegrind und Callgrind verwendet. Die Ergebnisse der Werkzeuge DRD und Helgrind waren unbrauchbar, da diese nicht mit oneTBB kompatibel waren.

[1] `https://valgrind.org/`
[2] vgl. `https://valgrind.org/info/tools.html`

cachegrind und callgrind

Cachegrind und Callgrind, entwickelt von Weidendorfer, Kowarschik und Trinitis, sind sogenannte Cacheprofiler. Diese Werkzeuge werden verwendet um die Programmstellen mit der größten Laufzeit festzustellen und verschiedene Umstellungen im Programmcode vergleichen zu können.

Cachegrind simuliert wie das zu testende Programm mit der Cache-Hierarchie und dem Branch-Predictor einer virtuellen Maschine interagiert. Die dabei simulierte Maschine ist orientiert and der Architektur moderner Maschinen. Dabei werden verschiedene Caches simuliert und die Zugriffe darauf hinsichtlich von Misses ausgewertet. In Abbildung 6.1 ist ein Beispielaufruf zu sehen.

```
==748==
==748== Events    : Ir Dr Dw I1mr D1mr D1mw ILmr DLmr DLmw
==748== Collected : 37949033802 8296932838 3723612328 1702844 225266632 11447
==748==
==748== I   refs:      37,949,033,802
==748== I1  misses:         1,702,844
==748== LLi misses:            18,012
==748== I1  miss rate:           0.00%
==748== LLi miss rate:           0.00%
==748==
==748== D   refs:      12,020,545,166  (8,296,932,838 rd + 3,723,612,328 wr)
==748== D1  misses:       236,713,654  (  225,266,632 rd +    11,447,022 wr)
==748== LLd misses:           481,230  (       48,311 rd +       432,919 wr)
==748== D1  miss rate:            2.0% (          2.7%   +           0.3%  )
==748== LLd miss rate:            0.0% (          0.0%   +           0.0%  )
==748==
==748== LL refs:          238,416,498  (  226,969,476 rd +    11,447,022 wr)
==748== LL misses:            499,242  (       66,323 rd +       432,919 wr)
==748== LL miss rate:             0.0% (          0.0%   +           0.0%  )
```

Abbildung 6.1: Beispielaufruf von Callgrind

Auswerten von callgrind-Ergebnissen

Mithilfe der Anwendung QCachegrind können die Ergebnisse von Valgrinds cachegrind/callgrind grafisch ausgewertet werden. QCachegrind ist ein Windowsbuild der Opensource Anwendung KCacheGrind. KCachegrind ist Teil der Werkzeuge aus der Arbeit [9].

In Abbildung 6.2 ist eine Übersicht einer Auswertung mit QCachegrind zu sehen. In diesem Beispiel wurde mithilfe des Werkzeugs `callgrind` eine Aufzeichnung des zweidimensionalen Falls mit verschiedenen Wellenzahlen und jeweils 40 Stichproben von Beobachtungspunkt und Richtungsektoren für ein festes Dreieck berechnet. Abbildung 6.3 zeigt einen Ausschnitt des Aufrufgraphen vergrößert dar. In diesem lassen sich an Pfeilen ablesen wie oft welcher Programmteil aufgerufen

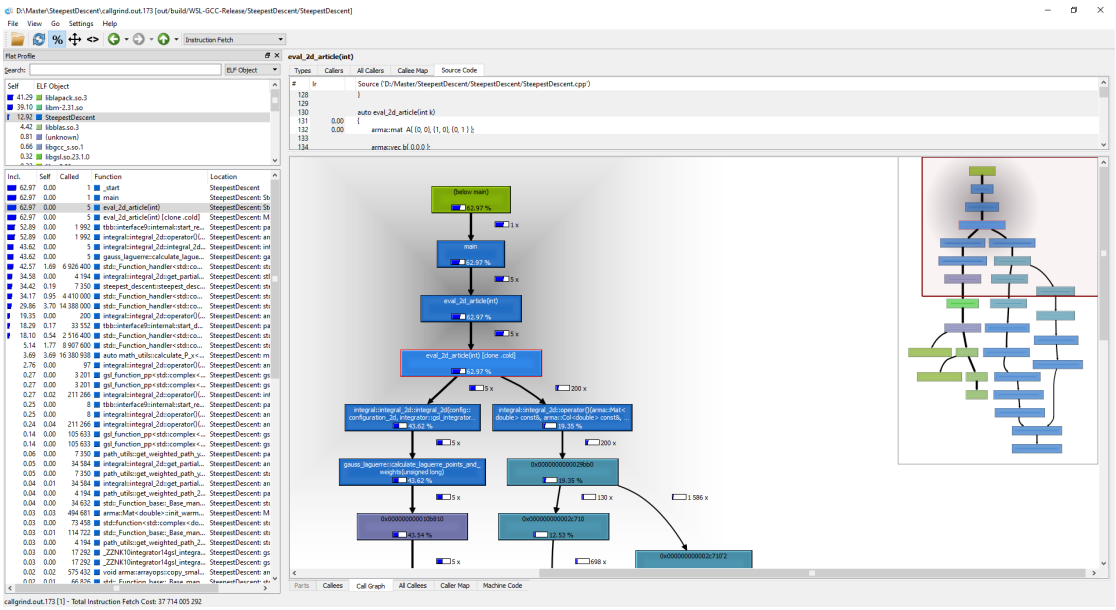

Abbildung 6.2: Übersicht von QCachegrind

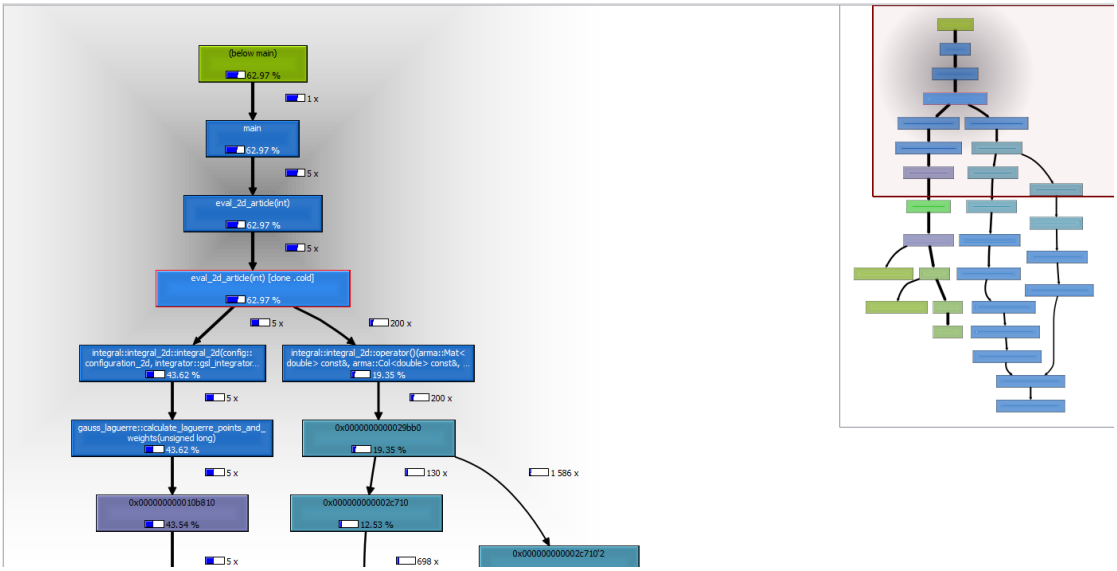

Abbildung 6.3: Aufrufgraph aus QCachegrind

wird und mithilfe einer prozentualen Angabe einsehen wie viel der Laufzeit in diesem Teil verbraucht wird.

Aus diesen und weiteren Auswertungen werden die meistaufgerufene Funktion und der Hotpath ersichtlich.

6.3 Ergebnisse der Optimierungsmaßnahmen

In diesem Abschnitt wird auf einige Ergebnisse der manuellen Performance-Messungen eingegangen.

6.3.1 Hotpath

Der häufigste Fall der Anwendung ist das Berechnen des Integrals in Situationen in denen keine Singularität auftritt. Dementsprechend werden diese in den implementierten Funktionen als der regelfall betrachtet. Durch das vorziehen der Codestellen die diese Szenarien abhandeln werden beispielsweise die benötigten Sprunganweisungen geringer gehalten.

6.3.2 Meistaufgerufene Funktion

Die Funktion mit der größten Laufzeit ist das *Steepest-descent*-Verfahren. Diese Funktion ist über mehrere Iterationen optimiert worden, bis sie die im Kapitel **??** dargestellte Form erreichte. In früheren Versionen war dieses Verfahren in einer eigenen C++-Klasse implementiert, allerdings haben die Auswertungen gezeigt, dass ein direkterer Aufruf der Gauss-Laguerre-Quadratur einen Laufzeitgewinn von fast 50 Prozent erzielen lies.

6.3.3 Vermeiden von Funktionsaufrufen

Bei der Implementierung des zweidimensionalen Falls 5.2.4 hat sich gezeigt, dass das Nutzen der eindimenisionalen Implementierung zu erhöhten Laufzeiten geführt hat.

In diesem Test wird die Laufzeit hinsichtlich veränderter Auflösung $res \in \{0.1, 0.01, 0.001, 0.0001\}$ gemessen. Für jedes k werden 50 zufällige Richtungsvektoren r berechnet mit 600 Gauss-Laguerre-Knoten berechnet.

$$A = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \tag{6.1}$$

Veränderung: läuft über resoltuion nicht K!s
Diese Messungen werden in der Abbildung **??** dargestellt.

Grafik folgt noch

6.4 Auswertung der Laufzeitmessugen

Die folgenen Laufzeitvergleiche wurden aus einer Matlab Umgebung heraus ausgeführt, d.h. es werden die ursprüngliche Implementierung sowie das Matlab-Modul der C++ Implementierung verglichen.

6.4.1 Iteration über Wellenzahl

In diesem Test wird die Laufzeit hinsichtlich veränderter Wellenzahl $k \in \{100, 500, 1000, 3000, 5000\}$ gemessen. Für jedes k werden 50 zufällige Richtungsvektoren r berechnet mit 600 Gauss-Laguerre-Knoten berechnet.

$$A = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \tag{6.2}$$

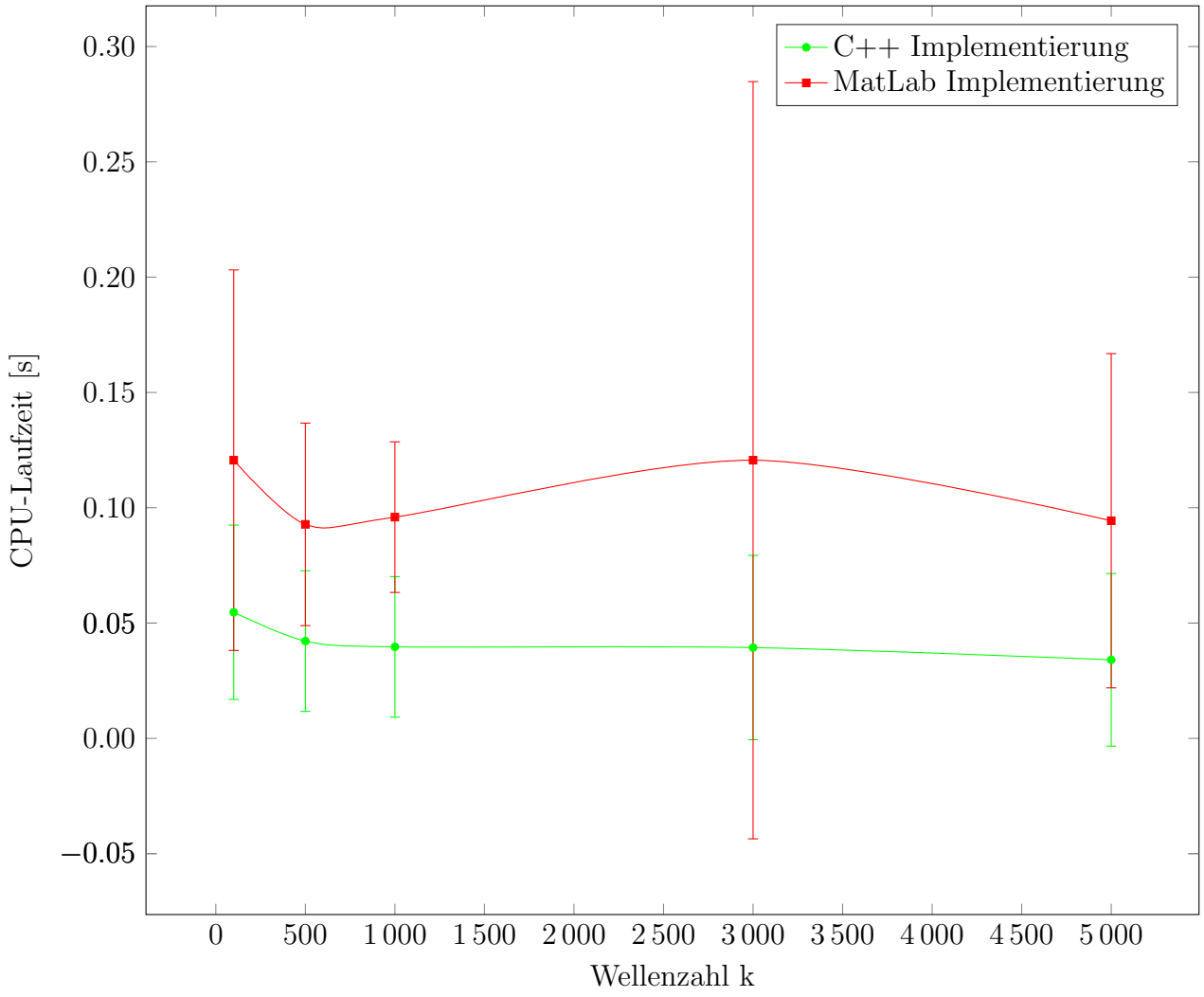

Abbildung 6.4: Laufzeitvergleich über verschiedene Wellenzahlen k

6.4.2 Laufzeitvergleich bei steigender Auflösung

In diesem Test wird die Laufzeit hinsichtlich veränderter Auflösung $res \in \{0.1, 0.01, 0.001, 0.0001\}$ gemessen. Für jedes k werden 50 zufällige Richtungsvektoren r berechnet mit 600 Gauss-Laguerre-Knoten berechnet.

$$A = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \tag{6.3}$$

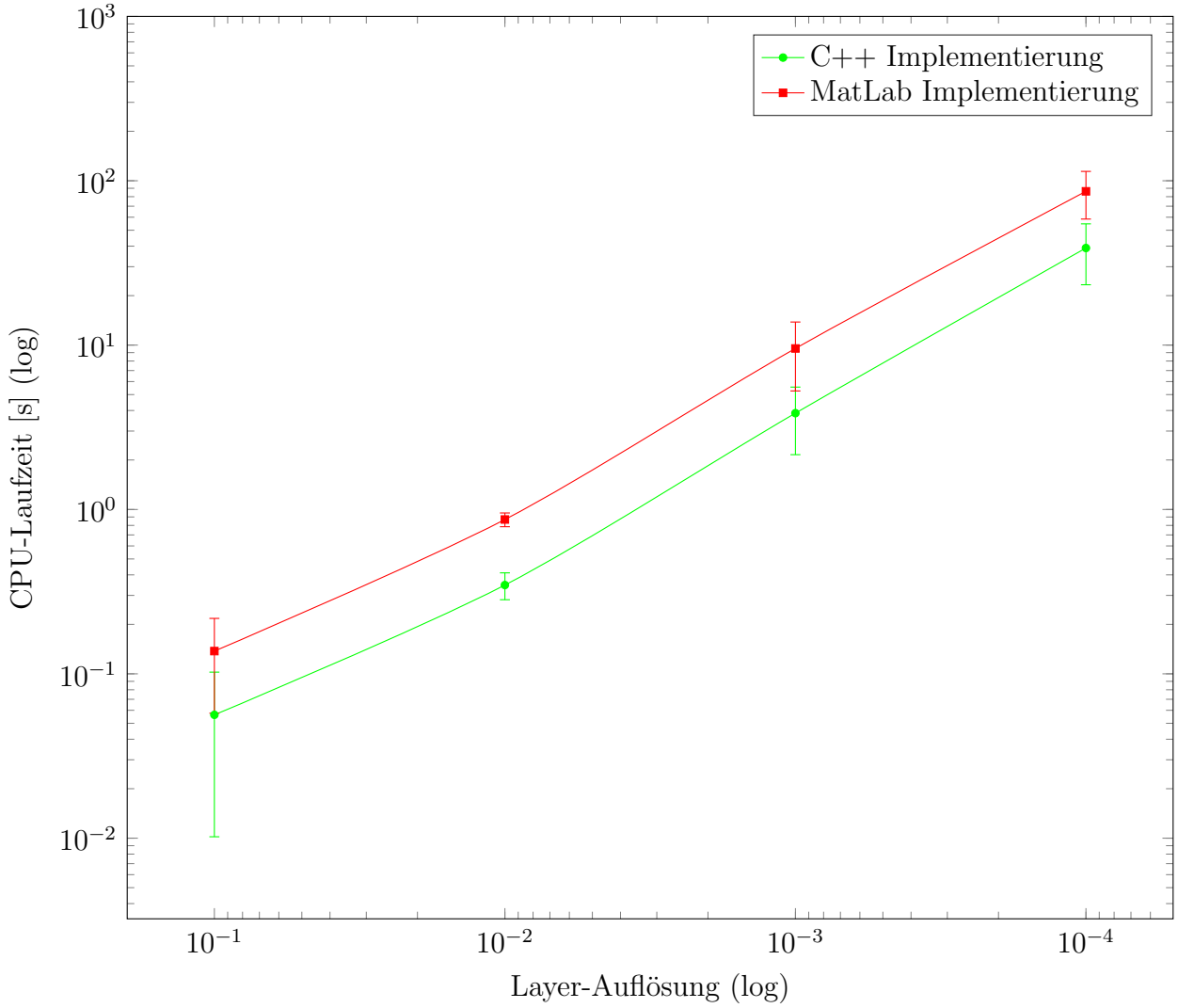

Abbildung 6.5: Laufzeitvergleich über verschiedene Auflösungen, logarithmische Skalen

6.5 Vergleiche der Genauigkeit

Auf das Fehlerszenarion eingehen. Graph mit dem Fehlervergleich
Daten des relativen Fehlers messen =¿ matlab um die 7-e18 =¿ gsl um die 1-e18
Das einfach mal auf einem Lauf rennen lassen

7

Zusammenfassung und Ausblick

Insgesamt hat diese Arbeit die geforderten Anforderungen erfüllt. Die Implementierung ist hinsichtlich der Performanz besser als die zu Grunde liegende Matlab-Implementierung. Durch das Python-Modul und die Matlab-Funktionen ist es relativ einfach möglich die Bibliothek zu verwenden. Die Implementierung hat sich als komplizierter als erwartet gezeigt, denn einige Details werden in der Arbeit [**n**]icht näher beschrieben, die in der Implementierung nicht trivial sind.

So ist beispielsweise die Berechnung der im Programmcode *Splitting-point* genannten Punkte nicht weiter beschrieben und war nur mithilfe des Matlab-Codes nachvollziehbar.

Hinsichtlich der Perfomanz bestehen noch einige Möglichkeiten die in dieser Arbeit nicht ausgeschöpft wurden. Insbesondere könnte eine Beschleunigung der Verfahren mithilfe der GPU enorme Laufzeit-Verbesserungen mit sich bringen.

Literatur

[1] Mark Galassi u. a. *GNU Scientific Library - Reference Manual, Third Edition, for GSL Version 1.12 (3. ed.)*. Jan. 2009. ISBN: 978-0-9546120-7-8. URL: http://www.gnu.org/software/gsl/.

[2] Gasperini, David and Beise, Hans-Peter and Schroeder, Udo and Antoine, Xavier and Geuzaine, Christophe. „An analysis of the steepest descent method to efficiently compute the 3D acoustic single-layer operator in the high-frequency regime“. In: *IMA Journal of Numerical Analysis* (2022). URL: https://hal.archives-ouvertes.fr/hal-03209144.

[3] Andreas Gathmann. *Einführung in die Funktionentheorie*. 2022. URL: https://www.mathematik.uni-kl.de/~gathmann/de/futheo.php.

[4] Wenzel Jakob. *Pybind11*. https://github.com/pybind/pybind11. Zugriffsdatum 7.10.2022.

[5] Nicholas Nethercote und Julian Seward. „Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation“. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, S. 89–100. ISBN: 9781595936332. DOI: 10.1145/1250734.1250746. URL: https://doi.org/10.1145/1250734.1250746.

[6] Nicholas Nethercote und Julian Seward. „Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation“. In: *SIGPLAN Not.* 42.6 (Juni 2007), S. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: https://doi.org/10.1145/1273442.1250746.

[7] Robert Piessens u. a. „Quadpack: A Subroutine Package for Automatic Integration“. In: 2011.

[8] Conrad Sanderson und Ryan Curtin. URL: https://arma.sourceforge.net/docs.html.

[9] Josef Weidendorfer, Markus Kowarschik und Carsten Trinitis. „A Tool Suite for Simulation Based Analysis of Memory Access Behavior“. In: *International Conference on Computational Science*. 2004.