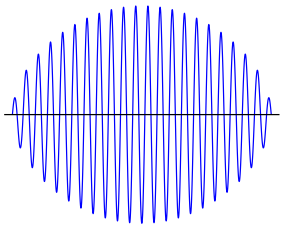
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz



Schnelle Algorithmen zur Berechnung sphärischer harmonischer Koeffizienten

Semesterarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2016/2017

Student:	Hansruedi Patzen
Betreuer:	Prof. Dr. Andreas Müller
Zeitraum:	19.09.2016 - 23.12.2016

Abstract

Mit dieser Arbeit wird neben der sphärischen harmonischen Transformation auch die von Tabea Méndez entwickelte Méndez-Transformation genauer untersucht. Es wird gezeigt, wie die für die Transformation benötigten sphärischen harmonischen Koeffizienten berechnet werden können und wo die Möglichkeit besteht, über Parallelisierung eine kürzere Laufzeit zu erreichen. Dabei finden sich drei Komponenten, die jeweils einzeln betrachtet und optimiert werden können. Es sind dies die Berechnung der Legendre-Polynome, die schnelle Fourier-Transformation und schlussendlich die Berechnung eines diskreten Integrals.

Es ist eine C++ Library entstanden, mit der sich eine einfache Möglichkeit ergibt, die gewünschten Koeffizienten zu berechnen. Dank einem C Interface wird sichergestellt, dass auch andere Programmiersprachen die Möglichkeit haben, die Library zu nutzen. Die vom Algorithmus gegebenen Parallelisierungsmöglichkeiten der einzelnen Komponenten werden neben der CPU-Parallelisierung auch mit einem Prototyp für die schnelle Fourier-Transformation in OpenCL umgesetzt. Eine Laufzeitanalyse zeigt deutlich, dass die Méndez-Transformation besser abschneidet als die normale sphärische harmonische Transformation.

Management Summary

Ausgangslage

Die Berechnung sphärischer harmonischer Koeffizienten ist anspruchsvoll und nicht ohne Weiteres in Programmcode umsetzbar. Es wird nach einer Möglichkeit gesucht, die Koeffizienten schnell und einfach berechnen zu können. Für die Méndez-Transformation soll zudem gezeigt werden, dass damit ein Laufzeitvorteil gegenüber der normalen sphärischen harmonischen Transformation gewonnen werden kann.

Vorgehen / Technologien

Die sphärische harmonische Transformation ist in ihre einzelnen Teile zerlegt worden. Dadurch haben sich drei isolierte Komponenten gezeigt, die jeweils einzeln optimiert werden können, um das Ziel einer schnelleren Berechnung zu erreichen.

Ergebnisse

Das Resultat dieser Arbeit ist die SHCL, kurz für “Spherical Harmonic Coefficient Library”. Mit dieser lassen sich die sphärischen harmonischen Koeffizienten auf einfache Art und Weise berechnen. Sie stellt ein einfaches Interface zur Verfügung, um auch einen Einsatz in anderen Anwendungen zu ermöglichen. Zudem hat sich gezeigt, dass die Méndez-Transformation einen deutlichen Laufzeitvorteil gegenüber der normalen sphärischen harmonischen Transformation besitzt.

Ausblick

Mit der SHCL Library ist eine Grundlage für allfällige Weiterentwicklungen geschaffen worden. Die einzelnen Komponenten können

erweitert und weiter optimiert werden. So fehlen noch die Implementationen der inversen Transformation und die OpenCL Varianten der Legendre-Polynom Berechnung und Integration. Auch die derzeitige OpenCL Lösung der schnellen Fourier-Transformation (FFT) kann noch weiter optimiert und verbessert werden.

Inhaltsverzeichnis

1 Formelsatz

Sphärische harmonische Funktionen, die auch Kugelflächenfunktionen oder “spherical harmonics” im Englischen genannt werden, finden in unterschiedlichsten Gebieten Verwendung. Trotzdem fristen sie ausserhalb von Hochschulen und Universitäten eher ein Schattendasein. Mit diesem Kapitel soll nun ein kurzer Überblick der wichtigsten mathematischen Formeln und Definitionen dieser Funktionen gegeben werden. Es wird zudem auch die, der von Tabea Méndez (siehe [Mé16]) entwickelte, Méndez-Transformation untersucht. Somit ist dieses Kapitel die Grundlage für die weiteren Überlegungen zur Optimierung und auch Parallelisierung des Algorithmus. Für Leser, die sich mit der Materie bereits auskennen, können direkt zu Kapitel 2 springen, wo die Problematik bei der Berechnung diskreter Integrale aufgezeigt wird.

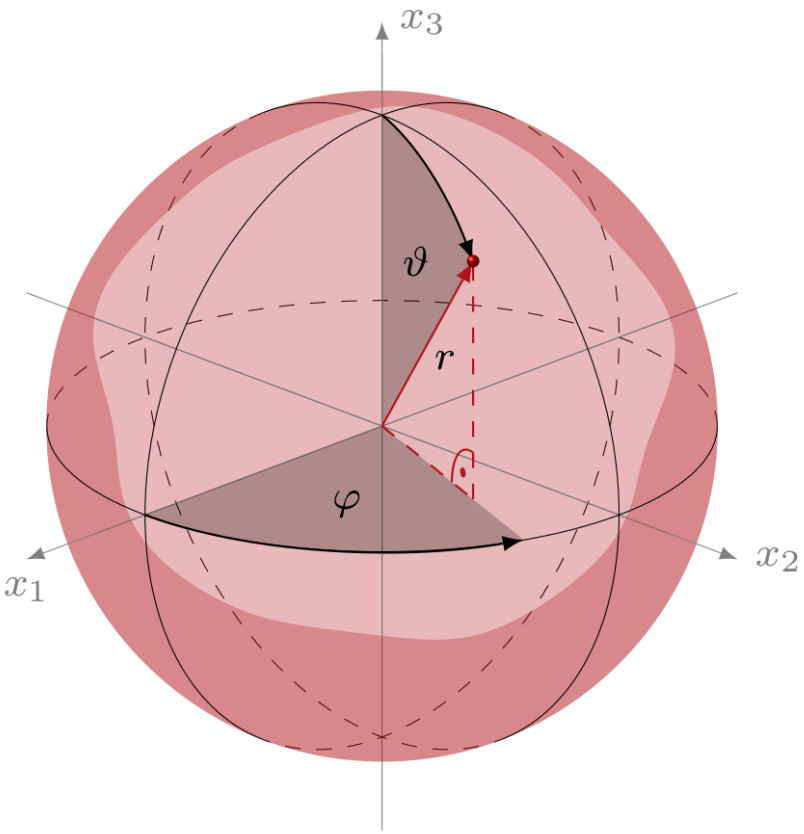


Abbildung 1.1: Darstellung einer Kugel mit ersichtlichem ϑ und φ. Ersteres ist dabei der Winkel zwischen den beiden Polen der x_3-Achse, die oft auch als z-Achse bezeichnet wird. φ hingegen beschreibt das Azimut. Da wir stets mit einer normierten Kugel arbeiten, gilt $r = 1$. (Quelle [Mé16])

1.1 Sphärische harmonische Transformation

Mit Hilfe der sphärischen harmonischen Funktionen $Y_l^m(\varphi, \vartheta)$ können Funkionen $f(\varphi, \vartheta)$ auf einer Kugel mittels Summen

$$f(\varphi, \vartheta) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} c_l^m Y_l^m(\varphi, \vartheta) \tag{1.1}$$

ausgedrückt werden. Wenn wir Gleichung (1.1) genauer anschauen, fällt gleich eine gewisse Ähnlichkeit zu den aus der Fourier-Analysis (siehe [Wikb]) bekannten Fourier-Reihen

$$f(t) = \sum_{k=-\infty}^{\infty} c_k e^{ikt} \tag{1.2}$$

auf, mit der eine Funktion $f(t)$ mittels einer Summe wiedergegeben werden kann.

Die komplexen Fourierkoeffizienten c_k aus Gleichung (1.2) und die sphärischen harmonischen Koeffizienten c_l^m aus Gleichung (1.1) sind als Skalarprodukte definiert. Für c_k gilt

$$c_k = \langle f, \overline{e^{ik}} \rangle = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(t) e^{-ikt}\, dt. \tag{1.3}$$

Bei den c_l^m sieht es auf den ersten Blick schon nicht mehr so einfach

$$c_l^m = \langle f, \overline{Y_l^m} \rangle = \int_0^{\pi} \int_0^{2\pi} f(\varphi, \vartheta) \overline{Y_l^m}(\varphi, \vartheta) \sin(\vartheta) \, d\varphi \, d\vartheta \tag{1.4}$$

aus. Daher ist eine genauere Analyse der c_l^m unabdinglich, um einen schnellen Algorithmus für deren Berechnung zu finden.

Um Gleichung (1.4) zu verstehen, müssen wir erst die $\overline{Y_l^m}$ mit ihrer Definition ersetzen, worauf wir

$$c_l^m = \int_0^{\pi} \int_0^{2\pi} f(\varphi, \vartheta) N_l^m P_l^m(\cos(\vartheta)) e^{-im\varphi} \sin(\vartheta) \, d\varphi \, d\vartheta. \tag{1.5}$$

erhalten. Dabei stellt N_l^m (siehe Abschnitt 1.7) den Normierungsfaktor und P_l^m die assoziierten Legendre-Polynome (siehe Abschnitt 1.6) mit Grad l und Ordnung m dar. Wenn wir Gleichung (1.5) nun noch ein wenig umstellen und den Faktor $\frac{1}{\sqrt{2\pi}}$ aus N_l^m ausklammern, erhalten wir

$$c_l^m = \int_0^{\pi} \underbrace{\frac{1}{\sqrt{2\pi}} \int_0^{2\pi} f(\varphi, \vartheta) e^{-im\varphi} \, d\varphi}_{F_\vartheta(m)} N_l^m P_l^m(\cos(\vartheta)) \sin(\vartheta) \, d\vartheta.$$

Wie man sieht, hat sich dort ein alter Bekannter in Form der Fourier-Transformation $F_\vartheta(m)$ versteckt. Diese wird in Abschnitt 1.4 weiter beschrieben.

Nach ein paar weiteren Umformungen ergibt sich schlussendlich

$$c_l^m = \begin{cases} \displaystyle\int_0^{\pi} F_\vartheta(|m|) N_l^{|m|} P_l^{|m|}(\cos(\vartheta)) \sin(\vartheta) \, d\vartheta & \text{für } m \geq 0, \\ \displaystyle(-1)^{-|m|} \int_0^{\pi} F_\vartheta(-|m|) N_l^{|m|} P_l^{|m|}(\cos(\vartheta)) \sin(\vartheta) \, d\vartheta & \text{für } m < 0. \end{cases}$$

Damit haben wir eine Grundlage für die Umsetzung des Algorithmus für die Berechnung der normalen sphärischen harmonischen Transformation in Programmcode geschaffen.

1.2 Méndez-Transformation

Bei der Méndez-Transformation handelt es sich um einen Spezialfall der normalen sphärischen harmonischen Transformation. Der Unterschied liegt darin, dass sie gemäss Definition, in Gleichung (1.6), nur die Terme für $m = 0$ benötigt.

$$c_l^m = \begin{cases} 0 & \text{für } m \neq 0, \\ \displaystyle\int_0^{\pi} F_\vartheta(0) \sqrt{\frac{2l+1}{2}} P_l(\cos(\vartheta)) \sin(\vartheta) \, d\vartheta & \text{für } m = 0. \end{cases} \tag{1.6}$$

Wir erkennen gleich, dass damit einige Vereinfachungen ermöglicht werden. So fallen beim Normierungsfaktor die Fakultäten weg und wir benötigen auch keine assoziierten sondern nur noch die normalen Legendre-Polynome $P_l(\cos(\vartheta))$. Dazu kommt, dass von der Fourier-Transformation nur noch der "DC"-Term $F_\vartheta(0)$ gebraucht wird, was gleich einer mit einem Skalierfaktor multiplizierten Summen aller Werte um einen Breitenkreis mit Winkel ϑ ist.

1.3 Integration

Die numerische Integration einer Funktion wird oft mittels Riemann-Summen gemacht. Deren Definition lautet

$$\int_a^b f(x)\,dx = \lim_{\max \Delta x_k \to 0} \sum_{k=0}^{n} f(x_k)\Delta x_k. \tag{1.7}$$

Wie Kapitel 2 zeigt, kann diese Definition nicht einfach so ohne weitere Überlegungen auf diskrete Werte angewandt werden.

1.4 Fourier-Transformation

Die Fourier-Transformation ist definiert (siehe [DB06]) als

$$\mathcal{F}\{f(x)\} = F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} \mathrm{d}x$$

Für diskrete Werte findet sich auch eine diskrete Version. Damit wird ein diskretes endliches Signal in ein periodisches Signal umgewandelt, welches wiederum mittels Sinus und Cosinus Funktionen wiedergegeben werden kann. Die diskrete Fourier-Transformation kann unterschiedlich definiert sein. In der Signalverarbeitung ist

$$F_k = \sum_{n=0}^{N-1} f_n e^{-i2\pi k \frac{n}{N}} \tag{1.8}$$

für die Vorwärts- und

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} F_k e^{i2\pi k \frac{n}{N}}$$

für die Inverse-Transformation verbreitet. Alternativ kann auch in beide Richtungen jeweils mit $\frac{1}{\sqrt{N}}$ skaliert werden, um eine symmetrische Gleichung zu erhalten.

1.5 Schnelle Fourier-Transformation (FFT)

Bei der schnellen Fourier-Transformation wird das Gleiche erreicht, wie bei der diskreten Fourier-Transformation. Mit dem kleinen aber feinen Unterschied, dass die Laufzeit nun nicht mehr $O(n^2)$, sondern nur noch $O(n \log n)$ beträgt. Erst die FFT erlaubt uns überhaupt einen sinnvollen Einsatz dieser Transformation. Möglich wird diese Verbesserung indem wir die Summe der Gleichung (1.8) in zwei Teilsummen mit jeweils den geraden und ungeraden Indizes unterteilen.

$$F_k = \underbrace{\sum_{n=0}^{\frac{N}{2}-1} f_{2n} e^{-i2\pi k \frac{2n}{N}}}_{\text{Gerade Indize}} + \underbrace{\sum_{n=0}^{\frac{N}{2}-1} f_{2n+1} e^{-i2\pi k \frac{2n+1}{N}}}_{\text{Ungerade Indize}}$$

Diese können wiederum in zwei Teilsummen unterteilt werden, bis nur noch solche mit zwei Werten vorhanden sind. Falls, wie bei OpenCL, keine Möglichkeit für Rekursion besteht, stellen wir fest, dass wenn man die Binärdarstellung der Indizes umkehrt, gerade die zwei, für eine Zwei-Punkte FFT benötigten, aufeinanderfolgenden Indizes gefunden werden. In Tabelle 1.1 wird eine Bitumkehrung für eine FFT mit 16 diskreten Werten durchgeführt. Danach kann mit der sogenannten Butterfly-Operation die FFT durchgeführt werden, wie in Abb. 1.2 gezeigt wird. Wichtig sind dabei auch die sogenannten Twiddle-Faktoren W^k, für die $W^k = e^{i2\pi \frac{k}{N}}$ gilt. Genauere Erklärungen für das Vorgehen und die Butterfly-Operation finden sich beispielsweise auf [But] oder in [Bur08].

1.6 Legendre-Polynome

Die Legendre-Polynome sind durch die Differentialgleichung [Wikc]

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P_l(x)\right] + l(l+1)P_l(x) = 0.$$

gegeben. Mittels der Rodrigues-Formel folgt daraus

$$P_l(x) = \frac{1}{2^l l!}\frac{d^l}{dx^l}(x^2-1)^l. \tag{1.9}$$

Die assoziierten Legendre-Polynome

$$P_l^m(x) = (-1)^m \sqrt{(1-x^2)^m}\frac{d^m}{dx^m}P_l(x) \tag{1.10}$$

sind eine Erweiterung der normalen Legendre-Polynome. Für den Fall der Berechnung von sphärischen harmonischen Koeffizienten, bei denen $x = \cos(\vartheta)$ gilt, vereinfacht sich $\sqrt{(1-x^2)^m}$ zu $\sin^m(\vartheta)$.

Tabelle 1.1: Bitumkehrung für einen FFT mit 16 diskreten Werten. Zwei aufeinander folgende Startindexe können danach jeweils über einen einfachen Zwei-Punkte FFT verbunden werden. Diese Operation ist auch als sogenannter Butterfly (englisch für Schmettering) bekannt. Als weiterführende Literatur sei hier auf [But] oder [Bur08] verwiesen.

Index	Bitumkehrung			Startindex
0	0000	→	0000	0
1	0001	→	1000	8
2	0010	→	0100	4
3	0011	→	1100	12
4	0100	→	0010	2
5	0101	→	1010	10
6	0110	→	0110	6
7	0111	→	1110	14
8	1000	→	0001	1
9	1001	→	1001	9
10	1010	→	0101	5
11	1011	→	1101	13
12	1100	→	0011	3
13	1101	→	1011	11
14	1110	→	0111	7
15	1111	→	1111	15

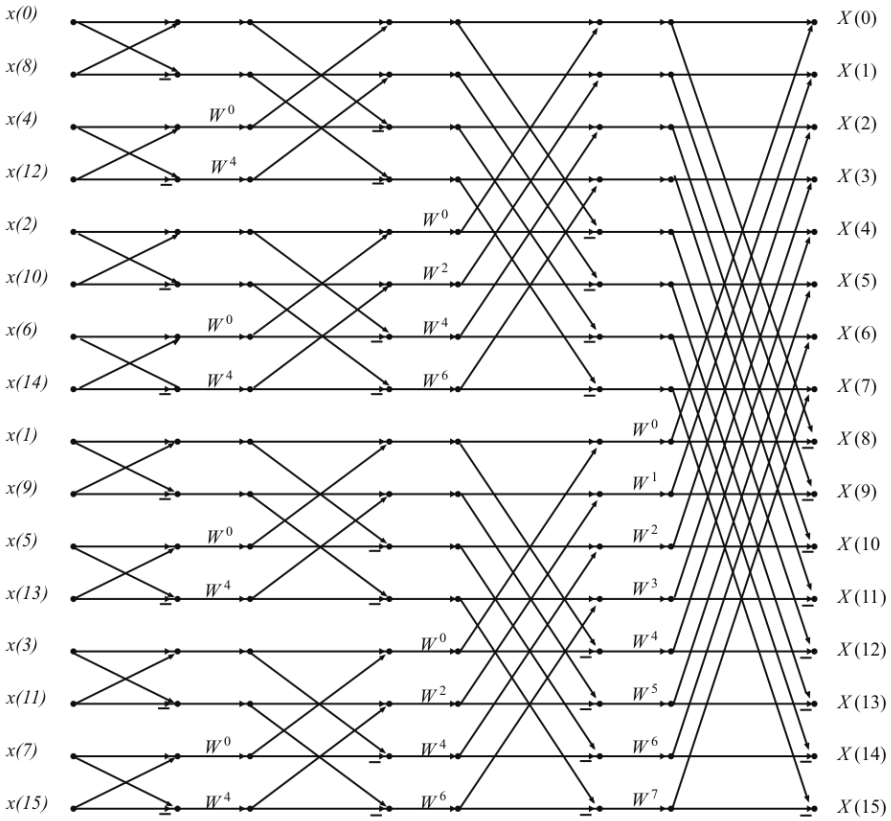


Abbildung 1.2: 16 Punkte Radix-2 FFT. Wie man sieht, entsprechen die Startindexe den in Tabelle 1.1 gewonnen Bitumkehrungen. Bei den verschiedenen W^k handelt es sich um die sogenannten Twiddle-Faktoren. Für diese gilt $W^k = e^{i2\pi \frac{k}{N}}$. (Quelle [Bur08])

Wir sehen sofort, dass die Gleichung (1.10) nur für $m \geq 0$ definiert ist. Dank Gleichung (1.9) kann aber auch der Fall $m < 0$ abgedeckt werden, indem wir die beiden Ableitungen zusammennehmen und

$$\begin{aligned} P_l^m(x) &= (-1)^m \sqrt{(1-x^2)^m} \frac{d^m}{dx^m} \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2-1)^l \\ &= (-1)^m \sqrt{(1-x^2)^m} \frac{1}{2^l l!} \frac{d^{m+l}}{dx^{m+l}} (x^2-1)^l \end{aligned} \tag{1.11}$$

erhalten. Diese Gleichung ist proportional und es gilt somit

$$P_l^{-m}(x) = (-1)^m \frac{(l-m)!}{(l+m)!} P_l^m(x).$$

Zusätzlich haben die assoziierten Legendre-Polynome entweder eine gerade oder ungerade Parität, die sich durch

$$P_l^m(-x) = (-1)^{m+l} P_l^m(x)$$

ausdrückt.

1.7 Normierungsfaktor

Für die Normalisierung der sphärischen harmonischen Transformation gilt

$$N_l^m = (-1)^m \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}}.$$

Je nach Anwendungsgebiet finden sich noch weitere Definitionen. So wird die Condon–Shortley Phase $(-1)^m$ teilweise weggelassen (siehe [Wikd]).

Da wir aber bereits den Faktor $\frac{1}{\sqrt{2\pi}}$ für die Fourier-Transformation benötigen, wird in dieser Arbeit stets

$$N_l^m = (-1)^m \sqrt{\frac{2l+1}{2} \frac{(l-m)!}{(l+m)!}}$$

für die Normierung gewählt.

1.8 Schlussfolgerungen

Mit den in diesem Kapitel gewonnenen Formelsatz haben wir nun ein Werkzeug, mit dem nicht nur die sphärische harmonische Transformation, sondern auch die Méndez-Transformation, in Programmcode umgesetzt werden können. Bevor wir aber in Kapitel 4 beginnen uns Gedanken zu den verschiedenen Parallelisierungs- und Optimierungsmöglichkeiten zu machen, müssen erst ein paar Probleme,

die während der Umsetzung auftreten, aus dem Weg geräumt werden.

2 Probleme diskreter Integrale

Um Integrale einer unbekannten Funktion, von der man nur diskrete Abtastwerte kennt, mittels einem Computer zu berechnen, gibt es unterschiedliche numerische Verfahren. Für alle gilt, dass man nur über eine endliche Anzahl diskreter Funktionswerte integrieren kann und das Resultat somit nur einer Approximation entspricht. Je nach Verfahren, Funktion und Anzahl Funktionswerte erhält man also mehr oder weniger genaue Resultate. Für eine Herleitung der ersten drei Verfahren wird auf [Lov11] verwiesen.

In den nachfolgenden Summen gilt, für den Fall, dass die obere Summationsgrenze kleiner als die untere ist, die leere Summe. Diese ist gleich 0 zu setzen.

2.1 Riemann-Integral und Riemann-Summe

Unter dem Begriff des Integrals wird normalerweise das Riemann-Integral verstanden. Wie wir in Gleichung (1.7) sehen können, wird das Integral genauer, je kleiner die Δx_k werden. Für diskrete Werte bedeutet dies, dass mittels einer schnelleren Abtastrate, oder höherer Auflösung bei Bildern, auch das Integral genauer berechnet werden kann.

Ein grosser Vorteil dieser direkten Methode ist, dass sie einfach und schnell berechnet werden kann. Δx_k ist bei diskreten Werten aufgrund des Integrationsbereichs und der Anzahl, sofern nichts anderes bekannt ist, bereits fix gegeben. Danach ist nur ein Zusammenzählen aller Werte und Teilen durch Δx nötig, um das Resultat zu erhalten. Es wird dabei davon ausgegangen, dass

$$\Delta x_1 = \Delta x_2 = \cdots = \Delta x_k = \Delta x,$$

gilt. Für eine N-Mal abgetastete Funktion, mit

$$\Delta x = h = \frac{v - u}{N - 1},$$

erhält man somit

$$\int_u^v f(x)\,dx \approx h \sum_{k=0}^{N-1} f_k,$$

wobei u die untere und v die obere Grenze des Integrals darstellen.

Das so erhaltene Resultat hat je nach Originalfunktion und Abtastrate grössere oder kleinere Fehler. Ausser für einzelne, spezielle Funktionen wie beispielsweise $f(x) = c$, wo das Resultat exakt ist, lässt sich das Resultat meist nur mit vielen Abtastwerten annähern. Als Beispiel sei eine Sinusschwingung in Abb. 2.1 mit neun diskreten Werten gegeben. Es zeigt, dass wir mit dieser Approximation entweder viel zu tief oder zu hoch liegen, je nach Wert von $f(x)$ und $f'(x)$. Die Grössenordnung des Fehlers, dieser auch Links-Punkt genannten Riemann-Summe, liegt somit in etwa bei

$$O(h^2).$$

Es gibt auch noch die Mittelpunkt Riemann-Summe bei der jeweils die Punkte zwischen den beiden x_k und $x_k + h$ berechnet und mit h multipliziert werden. Dort haben wir einen Fehler von

$$O(h^3),$$

welcher etwa im gleichen Bereich, wie derjenige der Trapezregel, liegt. Für die Rechts-Punkt Riemann-Summe gilt dasselbe wie bei der Links-Punkt Riemann-Summe.

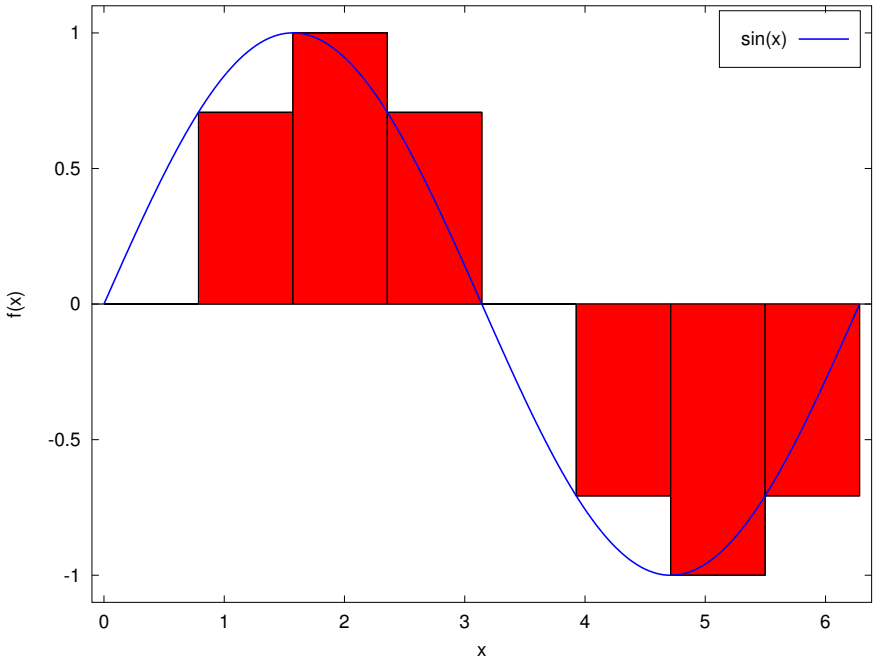


Abbildung 2.1: Berechnung $\int_0^{2\pi} \sin(x)dx$ mittels einer Riemann-Summe. Wie man sehen kann, ist die Flächenapproximation entweder zu gross oder zu klein, was das Resultat je nach gewähltem Intervall deutlich verfälschen kann.

2.2 Trapezregel

Die Trapezregel ist eine Erweiterung der Berechnung mittels Riemann-Summen. Statt aber jeweils den Anfangswert mit dem Δx zu multiplizieren, wird ein Trapez konstruiert, indem jeweils der Mittelwert von f_k und f_{k+1} gebildet und mit Δx multipliziert wird. Damit wird ermöglicht, dass nicht nur $f(x) = c$ sondern auch $f(x) = bx + c$ exakt wiedergegeben werden können. Für die Berechnung gilt

$$\int_u^v f(x)dx \approx h \sum_{k=0}^{N-1} \frac{f_k + f_{k+1}}{2}$$

$$\Leftrightarrow \qquad \int_u^v f(x)\,dx \approx h\left(\frac{f_0}{2} + \sum_{k=1}^{N-2} f_k + \frac{f_{N-1}}{2}\right).$$

Das so erhaltene Resultat sehen wir in Abb. 2.2. Man sieht gleich, dass die ursprüngliche Funktion somit besser angenähert werden kann und das Resultat genauer ist.

Die Trapezregel ist fast identisch mit dem Riemann-Summe, wenn wir den Rechenaufwand betrachten. Funktionen können damit allerdings deutlich besser angenähert werden, da der Fehlerordnung nur rund

$$O(h^3)$$

beträgt. Sie ist der Links- und Rechts-Punkt Riemann-Summe daher in jedem Fall vorzuziehen. Die Mittelpunkt Riemann-Summe mit gleicher Fehlerordnung kommt für uns nicht in Frage, da wir nur dis-

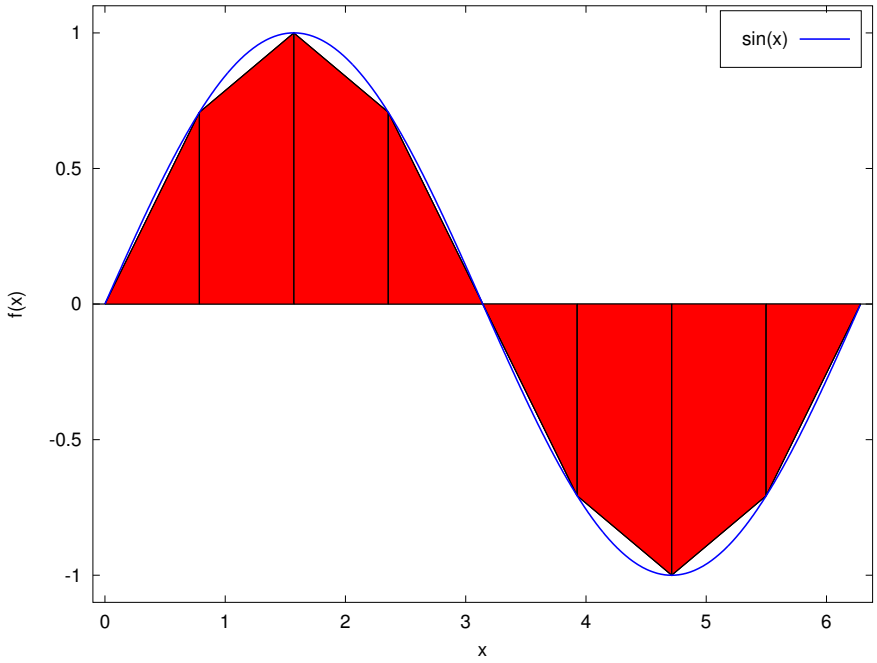


Abbildung 2.2: Berechnung $\int_0^{2\pi} \sin(x)dx$ mittels der Trapezregel. Man erkennt bereits einen deutlichen Vorteil zur einfachen Links-Punkt Riemann-Summe, da die Fläche unter der Kurve nur noch leicht von der realen Fläche abweicht und das trotz gleich vieler Messwerte.

krete Werte besitzen und die Funktion nicht kennen und somit nicht einfach den Wert für $f(x + h/2)$ berechnen können.

2.3 Simpson-Regel

Die Simpson-Regel ist ein weit verbreitetes Verfahren um Integrale zu berechnen. Im Unterschied zur Riemann-Summe oder Trapezregel werden nicht mehr einfach Geraden durch zwei Punkte gelegt, sondern eine Parabel durch drei Punkte. Dadurch können auch Funktionen der Form $f(x) = ax^2 + bx + c$ exakt wiedergeben werden. In Abb. 2.3 sieht man, dass die Funktion bereits mit wenigen diskreten Punkten fast identisch ist. Es gilt

$$\int_u^v f(x)\,dx \approx \frac{h}{3}\left(f_0 + 4\sum_{k=0}^{\frac{N-1}{2}} f_{2k+1} + 2\sum_{k=1}^{\frac{N-1}{2}} f_{2k} + f_{N-1}\right)$$

Ein grosser Nachteil der Simpson-Regel ist, dass sie nur mit einer geraden Anzahl Intervallen beziehungsweise einer ungeraden Anzahl diskreter Punkte funktioniert. Dafür ist sie, wenn man dieses Kriterium erfüllt, eine gute Annäherung an die ursprüngliche Funktion. Die Simpson-Regel hat eine Fehlerordnung von nur ungefähr

$$O(h^5),$$

was nochmals eine grosse Steigerung gegenüber der Trapezregel zeigt. Damit ist es möglich, bereits mit viel weniger Messwerten annähernd genaue Integrale zu berechnen.

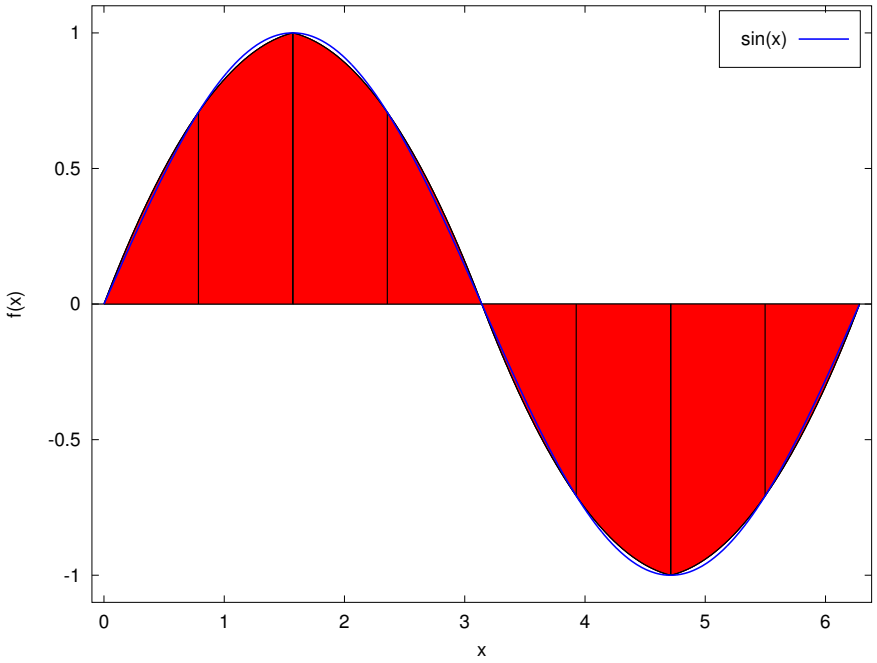


ABBILDUNG 2.3: Berechnung $\int_0^{2\pi} \sin(x)dx$ mittels der Simpson-Regel. Hier wird es bereits schwierig, den Unterschied zur ursprünglichen Funktion zu sehen. Die Steigerung gegenüber der Trapezregel ist mit einem Fehler von $O(h^5)$ gegenüber $O(h^3)$ gut sichtbar.

2.4 Hollingsworth-Hunter-Regel

Die Hollingsworth-Hunter-Regel [HH59] ist eine weniger bekannte Erweiterung der Simpson-Regel. Mit ihr ist es möglich, auch kubische Funktionen der Form $f(x) = wx^3 + ax^2 + bx + c$ exakt wiederzugeben. Zudem ist sie sowohl für eine gerade als auch ungerade Anzahl diskreter Punkte definiert. Leider ist die Definition nicht mehr so einfach, da man mindestens sechs Werte benötigt und sonst eine Fallunterscheidung vorgenommen werden muss. Es gilt

$$\int_u^v f(x)\,dx \approx \begin{cases} 0 & \text{für } N = 1, \\ \frac{h}{2}(f_0 + f_1) & \text{für } N = 2, \\ \frac{h}{3}(f_0 + 4f_1 + f_2) & \text{für } N = 3, \\ \frac{h}{8}(3f_0 + 9f_1 + 9f_2 + 3f_3) & \text{für } N = 4, \\ \frac{h}{24}(9f_0 + 28f_1 + 22f_2 + 28f_3 + 9f_4) & \text{für } N = 5, \\ \frac{h}{24}\left(9f_0 + 28f_1 + 23f_2 + 24\sum_{k=3}^{N-4} f_k + 23f_{N-3} + 28f_{N-2} + 9f_{N-1}\right) & \text{sonst.} \end{cases} \tag{2.1}$$

Der Fehler kann mit dieser Funktion allerdings nicht mehr verbessert werden und somit hat sie ebenfalls eine Fehlerordnung von

$$O(h^5).$$

Da wir allerdings damit unabhängig gegenüber der Anzahl gemessener diskreter Werte sind und der Fehler in etwa vergleichbar ist, haben wir damit eine gute Alternative zur Simpson-Regel gefunden.

2.5 Schlussfolgerungen

Für die Umsetzung der Programmbibliothek wurden die vier genannten Integrationsverfahren implementiert. Der Benutzer kann je nach Bedarf das jeweilige Verfahren aussuchen, denn jedes hat seine eigenen Vor- und auch Nachteile. Die Riemann-Summe sollte aber nur dann verwendet werden, wenn die anderen keine Option darstellen und genügend Messwerte vorhanden sind. Insbesondere die Trapezregel ist der Riemann-Summe aufgrund des praktisch identischen Rechenaufwands und der gewonnen Genauigkeit vorzuziehen.

Mit der Simpson- und der Hollingsworth-Hunter-Regel stehen zwei Verfahren mit etwa gleich grossem Fehler zur Verfügung. Da die Simpson-Regel aber eine ungerade Anzahl Messwerte erfordert und diese nicht immer gewährleistet werden kann, sollte die Hollingsworth-Hunter-Regel bevorzugt werden.

3 Numerische Instabilität der Legendre-Funktionen

Die Berechnung der Legendre-Polynome stellt eine grössere Hürde dar als vermutet. Der Versuch, die Formel direkt in Code abzubilden, führte nur für kleine l zum Ziel, bei grösseren l wird der numerische Fehler immer deutlicher sichtbar (siehe Abb. 3.4). Wie sich herausstellt, liegt dies nicht nur an der Umsetzung, sondern stellt ein numerisches Problem dar, dass nicht einfach zu lösen ist.

3.1 Fehlerquelle

Den Ursprung der fehlerhaften Werte finden wir bereits in der rekursiven Definition der Legendre-Polynome (siehe [Sel+13])

$$P_l^{m+2}(z) - 2(m+1)\frac{z}{\sqrt{1-z^2}}P_l^{m+1}(z) + (l-m)(l+m+1)P_l^m(z) = 0.$$

Auf den ersten Blick sieht der Term noch mehr oder weniger harmlos aus. Allerdings hat man ein Problem, sobald z sich den Polstellen des Faktors

$$\frac{z}{\sqrt{1-z^2}}$$

bei 1 und -1 nähert, da dieser dann immer grösser wird bis er schlussendlich gar nicht mehr definiert ist. Das alleine wäre aber noch nicht ganz so schlimm.

Wenn wir den Term ein wenig umstellen

$$P_l^{m+2}(z) = 2(m+1)\frac{z}{\sqrt{1-z^2}}P_l^{m+1}(z) - (l-m)(l+m+1)P_l^m(z),$$

sehen wir, dass in Nähe der Polstellen eine riesige Zahl mit einer im Verhältnis winzigen Zahl addiert wird, um das P_l^{m+2} zu erhalten. Das ist mit endlicher Gleitkommaarithmetik nicht handhabbar und führt zu immer grösser werdendem Fehler.

3.2 Auswirkungen

Es zeigt sich, dass die Auswirkungen des Fehlers sich bei grösser werdendem l immer mehr auf das Resultat auswirkt. Dies zeigt sich beispielsweise bei einer ersten direkten Implementation der Berechnung in Abb. 3.1 und Abb. 3.2. Wie man leicht erkennen kann, wird der Fehler grösser, je näher man zu den Polstellen kommt.

Ein Versuch, die Polynome in den Griff zu bekommen indem weitere Terme in die jeweiligen Einzelschritte mit eingerechnet werden, hat leider auch nicht zum Erfolg geführt. Die Hoffnung, dass damit das Resultat daran gehindert werden kann, aus dem Wertebereich von $[-1; 1]$ auszubrechen und sich so das numerische Problem löst, hat sich nicht bestätigt. Wenn wir die Resultate in Abb. 3.3 und Abb. 3.4 anschauen, sieht man zwar eine minimale optische Verbesserung, insbesondere für $P_{56}^0(\cos(\vartheta))\sin(\vartheta)$, trotzdem ist das Resultat so nicht brauchbar.

Ein Blick auf Wolfram Alpha zeigt allerdings auch, dass selbst bekannte Projekte damit Probleme haben. Abb. 3.5 zeigt, dass das Resultat ähnlich demjenigen der direkten Implementation dieser Arbeit, und somit unbrauchbar, ist.

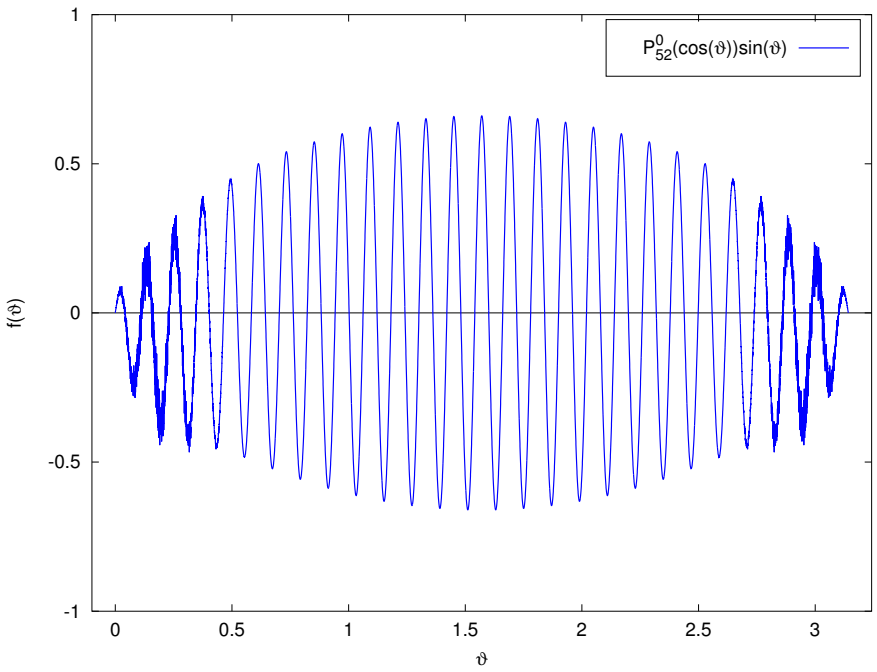


Abbildung 3.1:
$P^0_{52}(\cos(\vartheta))\sin(\vartheta)$ zeigt bereits erste Zeichen, dass etwas nicht stimmen kann. In der Mitte um $\frac{\pi}{2}$ ist zwar alles noch in Ordnung, doch je näher man 0 beziehungsweise π kommt, umso sichtbarerer wird der Fehler.

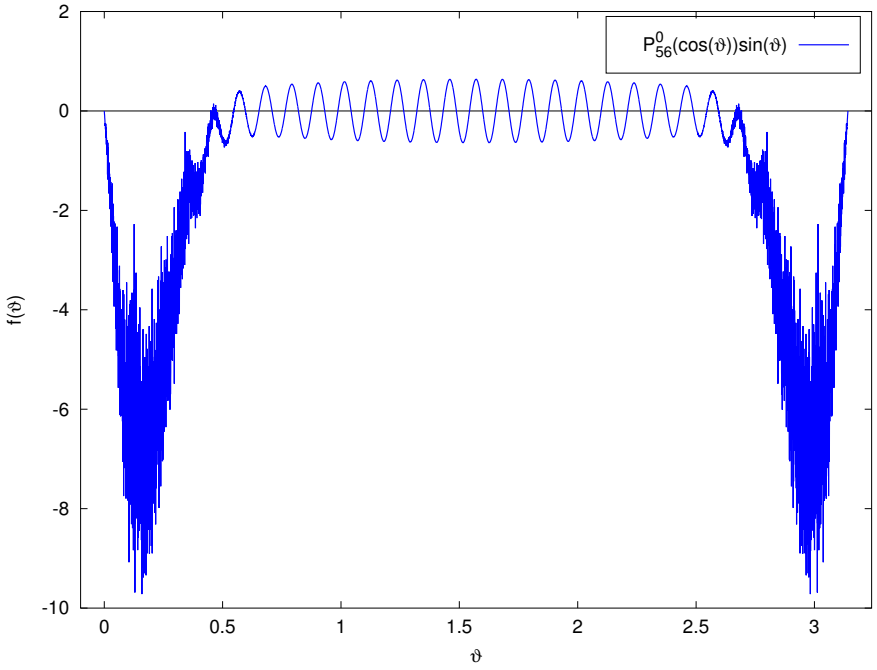


Abbildung 3.2:
$P^0_{56}(\cos(\vartheta))\sin(\vartheta)$ zeigt deutlich, dass etwas nicht stimmen kann. In der Nähe der Polstellen ist das Resultat nun weit ausserhalb des erlaubten Bereichs von Wertebereichs $[-1;1]$.

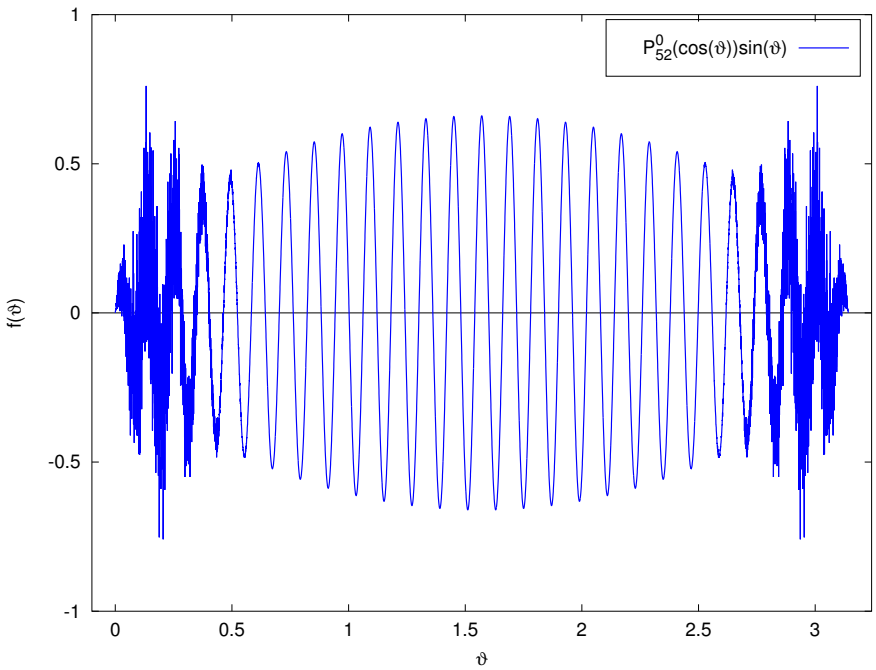


ABBILDUNG 3.3: $P^0_{52}(\cos(\vartheta))\sin(\vartheta)$ mit einer alternative Implementation umgesetzt. Leider führt dies in diesem Fall zu noch schlechteren Resultaten als bei der direkten Implementation in Abb. 3.1.

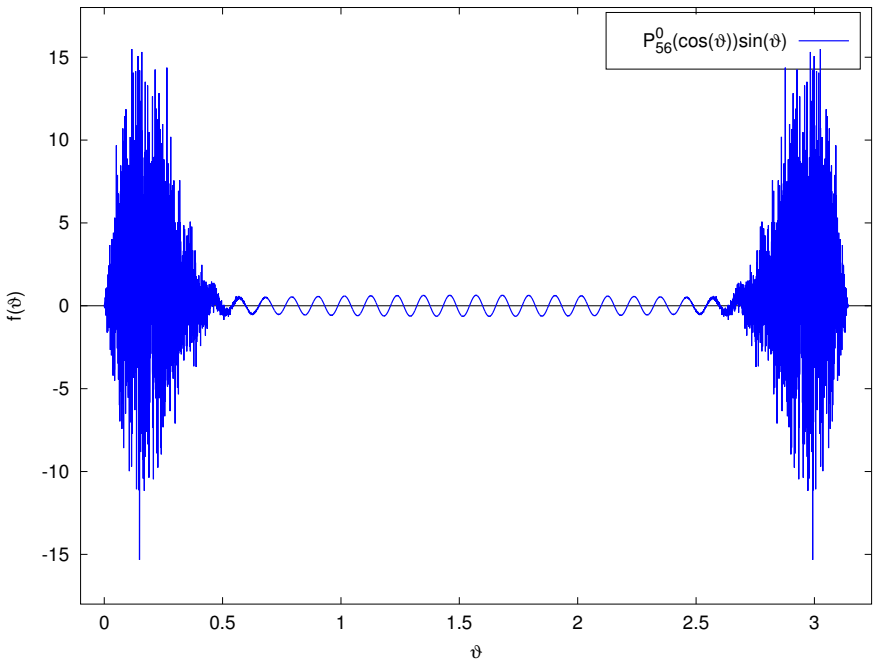


ABBILDUNG 3.4: Für $P^0_{56}(\cos(\vartheta))\sin(\vartheta)$ ist eine optische Verbesserung gegenüber Abb. 3.2. Das ändert aber nichts daran, dass das Resultat immer noch falsch und somit unbrauchbar ist.

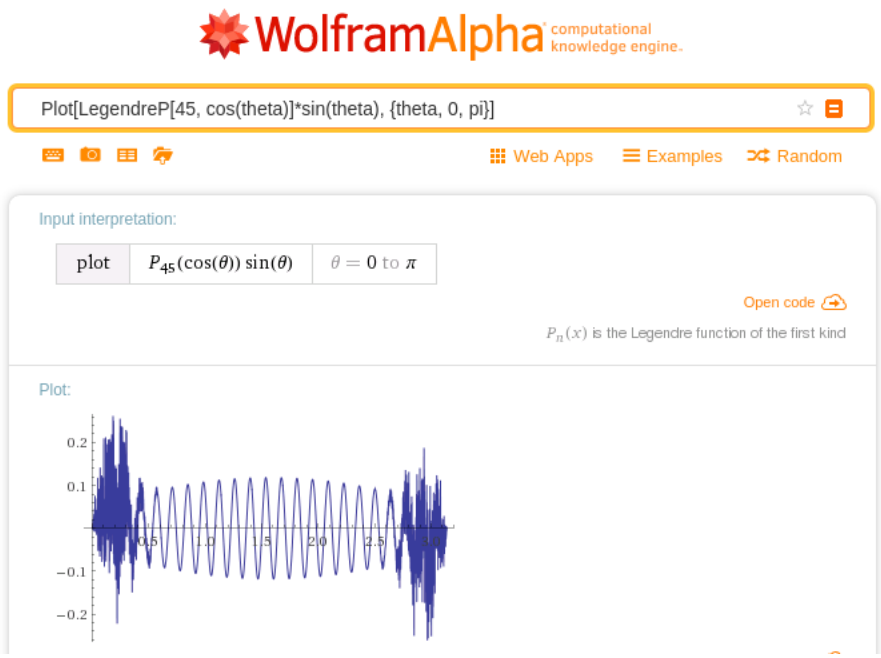


Abbildung 3.5: Wolfram Alpha zeigt ähnliches Verhalten, wie es bei der direkten Implementation zu sehen ist, und die Resultate brechen in der Nähe der Polstellen aus dem Wertebereich aus.

3.3 Problemlösung

Für 64-Bit Gleitkommazahlen, die meist mit dem `double` Datentyp gegeben sind, können die Legendre-Polynome mit Hilfe des von Holmes und Featherstone [FH02] gefundenen Algorithmus bis zu einer Grössenordnung von $l \approx 2700$ effektiv berechnet werden. Es ist auch bereits gezeigt worden, dass mittels einer Erweiterung auf 128-Bit Gleitkommazahlen, die Legendre-Polynome bis zu $l \approx 10800$ berechnet werden können. Mehr dazu findet man im Paper [KL07].

Glücklicherweise gibt es mit der GSL Library bereits eine Implementation, die diesen Algorithmus mit 64-Bit Gleitkommazahlen unterstützt. Berechnungen mit dieser Library zeigen auch gleich erste Erfolge, wie wir in Abb. 3.6 und Abb. 3.7 sehen können. Es gibt aber noch ein paar Bedenken und zwar muss bei grösser werdendem l auch die Anzahl ϑ-Werte erhöht werden, da die Anzahl Schwingungen der Legendre-Polynome proportional zu l zunimmt. In Abb. 3.9 sehen wir bereits, dass 1024 ϑ-Werte für $l = 1000$ nicht mehr das gewünschte Resultat liefern. Erst mit weitaus mehr Werten (siehe Abb. 3.10) stellt sich dieses ein.

Diesen Effekt nennt man Aliasing (siehe [Wika]). Er tritt auf, da Aufgrund des Informationsverlusts bei der Diskretisierung von kontinuierlichen Signalen mehrere Signale die gleichen Messwerte liefern, wie in Abb. 3.8 zu sehen ist. Ein bekanntes Beispiel, wo man Aliasing selbst beobachten kann, ist ein beschleunigendes Auto. Bei einer gewissen Geschwindigkeit kann das menschliche Gehirn nicht

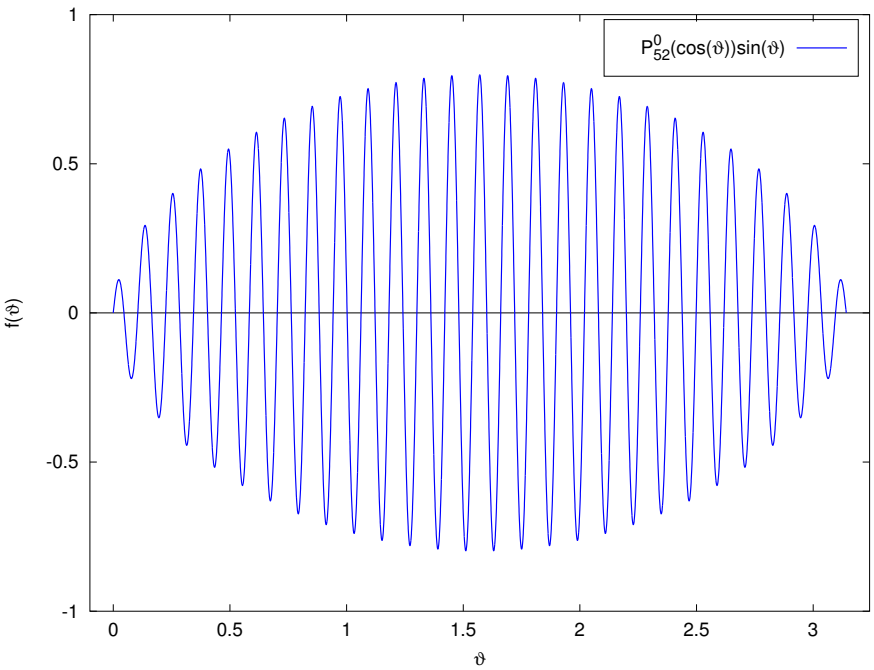


ABBILDUNG 3.6: $P^0_{52}(\cos(\vartheta))\sin(\vartheta)$ mittels der GSL-Implementierung berechnet. Der Fehler tritt nun nicht mehr auf und die Legendre-Polynome können korrekt gezeichnet und somit genutzt werden.

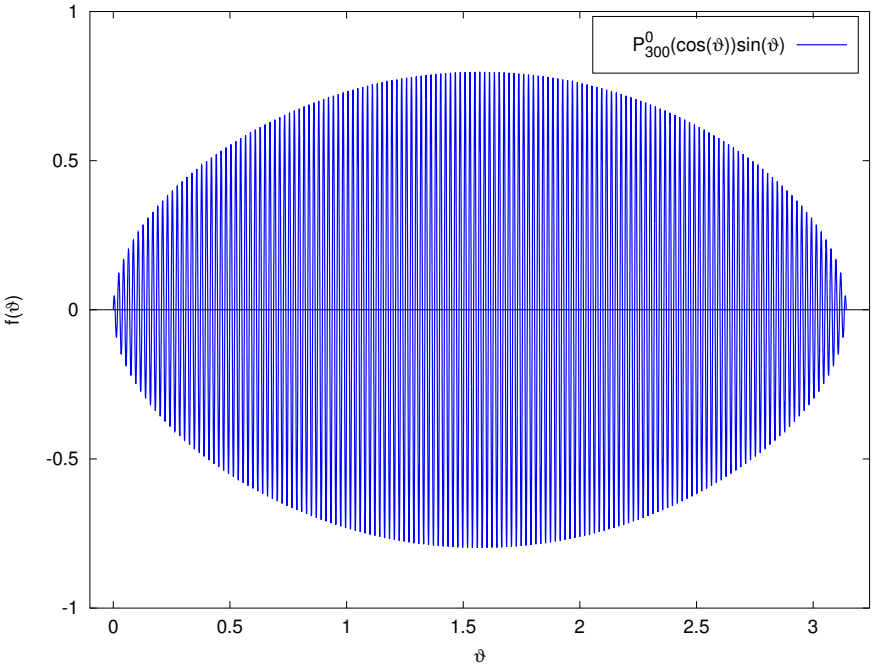


ABBILDUNG 3.7: $P^0_{300}(\cos(\vartheta))\sin(\vartheta)$ mittels der GSL-Implementierung berechnet. Auch grosse l wie hier mit $l = 300$ stellen kein Hinderniss mehr dar und können Problemlos berechnet und ausgegeben werden.

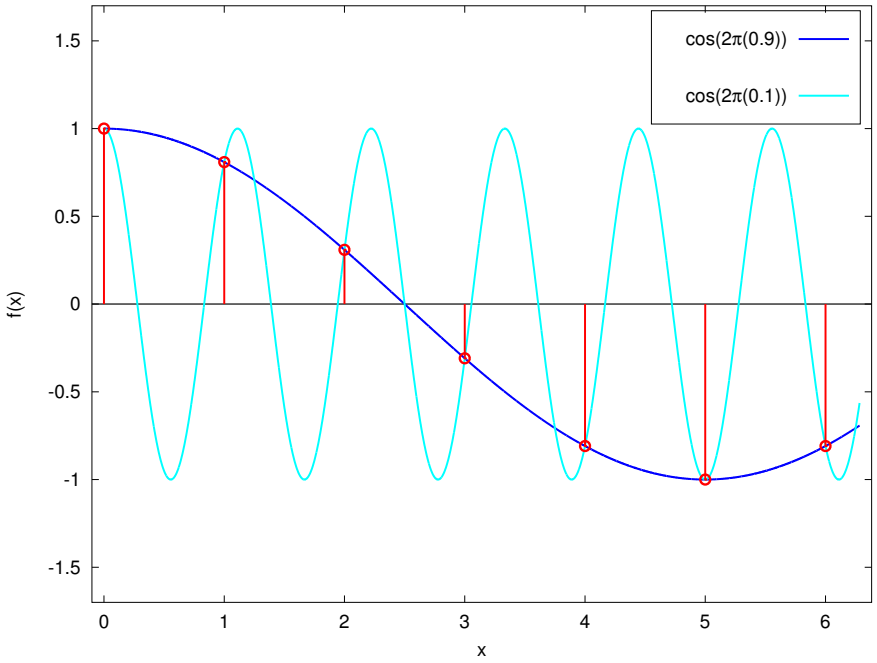


Abbildung 3.8: Aliasing Effekt bei dem mittels diskreten Werte nicht mehr zwischen zwei Signalen unterschieden werden kann. Allein mit den sieben Messwerten kann nicht zwischen den beiden Signalen unterschieden werden. Meist wird in einem solchen Fall das Signal mit der tiefsten Frequenz angenommen.

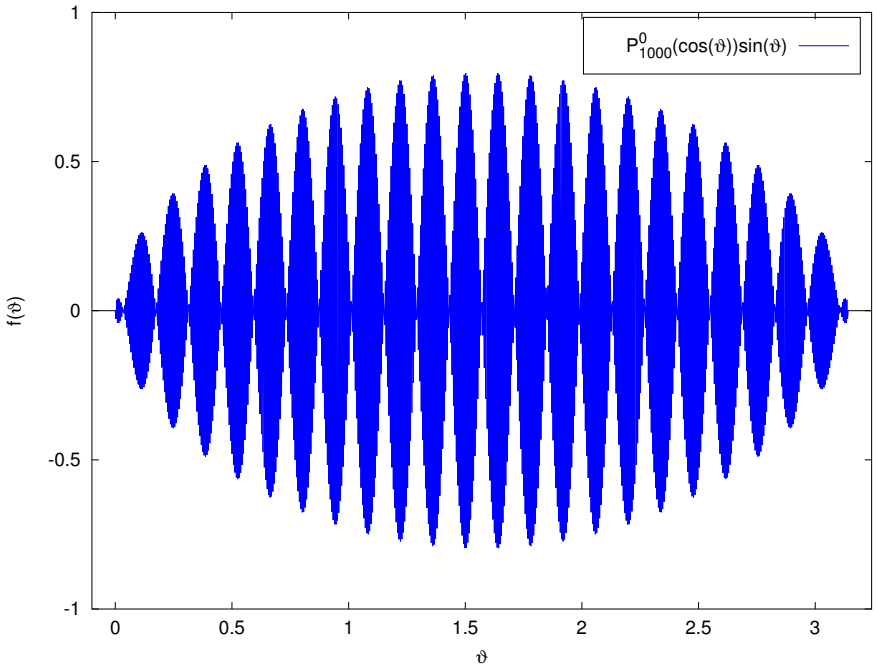


Abbildung 3.9: $P^0_{1000}(\cos(\vartheta))\sin(\vartheta)$ mit 1024 diskreten ϑ-Werten. Es ist klar, dass man zu wenig Werte hat, um das Legendre-Polynom wiedergeben zu können. Der auftretende Effekt nennt man Aliasing und ist ein bekanntes Problem der Fourier-Theorie.

mehr unterscheiden, ob es sich nun um eine positive oder negative Frequenz handelt und darum sieht es dann so aus, als ob das Rad die Drehrichtung ändern würde.

3.4 Schlussfolgerungen

Wie sich gezeigt hat, ist die Berechnung der Legendre-Polynome nicht trivial, daher wurde vorerst auf die Implementation der GSL

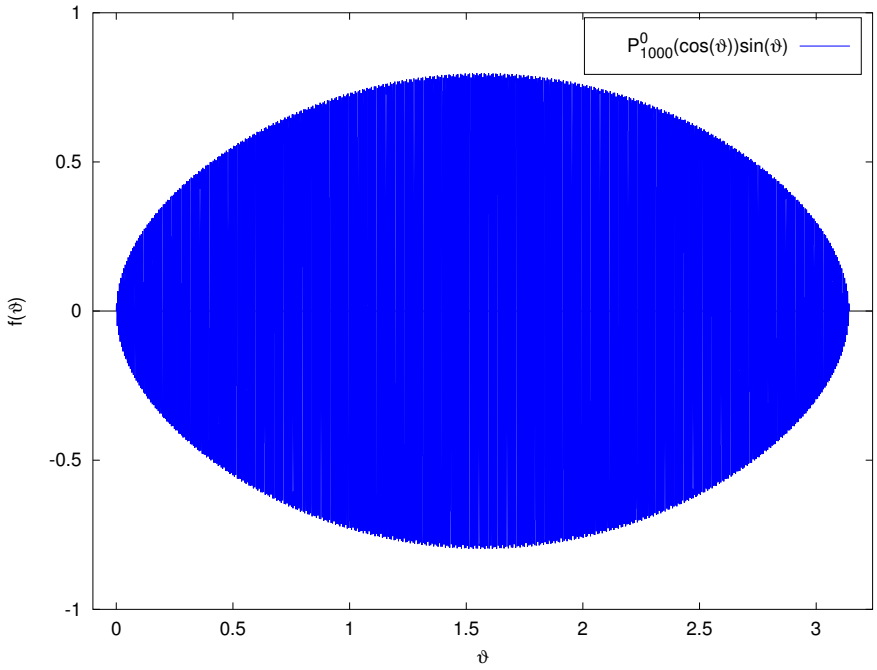


Abbildung 3.10: $P^0_{1000}(\cos(\vartheta))\sin(\vartheta)$ mit 8192 diskreten ϑ-Werten. Mit so vielen Messwerten kann das Legendre-Polynom annähernd wiedergegeben werden. Es ist daher darauf zu achten, die Anzahl Messwerte dem gewünschten maximalen l anzupassen.

Library zurückgegriffen. Ein möglicher Weg, die Legendre-Polynome zu berechnen, könnte darin liegen, die verschiedenen $P^l_l(\cos(\vartheta))$ mittels 64-Bit Gleitkommazahlen vorauszuberechnen, dann die von Holmes und Featherstone [FH02] gefundenen Rekursionsmethode auszurollen und mittels OpenCL zu implementieren.

4 Parallelisierung

In Kapitel 1 haben wir drei Teile gefunden, die in sich selbst unabhängig sind. Es ist dies die Berechnung des diskreten Integrals über ϑ, die Legendre-Polynome $P_l^m(\cos(\vartheta))$ und die Fourier-Transformation $F_\vartheta(m)$. Sie sind aber nicht alle zeitlich unabhängig, so müssen die Legendre-Polynome und die Fourier-Transformation vor der Berechnung des Integrals geschehen, damit ein sinnvolles Resultat entsteht.

Für die Parallelisierung bedeutet dies, dass wir jeden Teil in sich parallelisieren können, ohne das Gesamtresultat zu verfälschen, solange sichergestellt ist, dass sie Berechnung fertig ist bevor das Resultat von einem anderen Teil des Algorithmus verwendet wird.

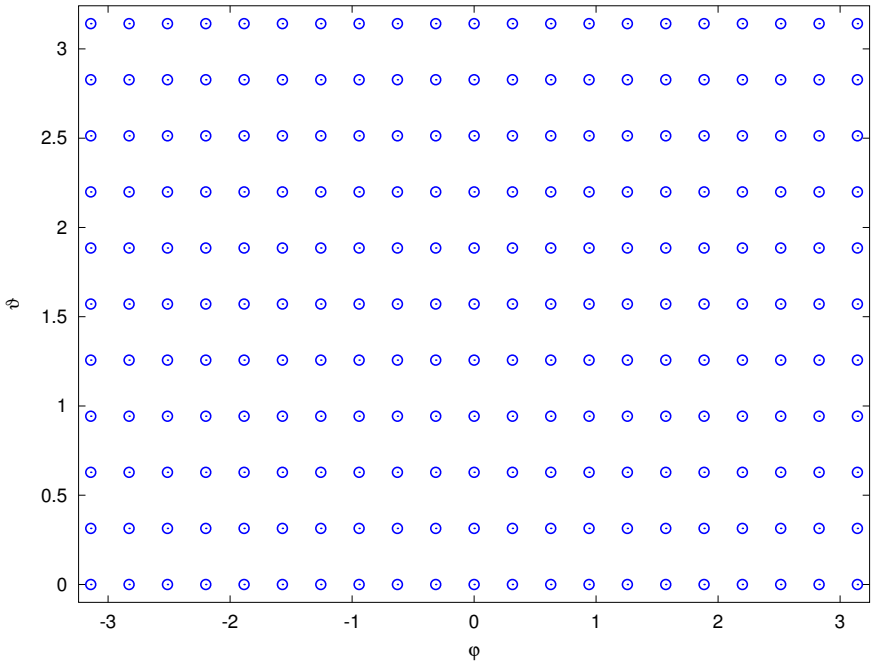


Abbildung 4.1: Ausgangslage für die Transformation. Die blauen Punkte stellen jeweils die gemessenen diskreten Werte $f(\varphi, \vartheta)$ dar.

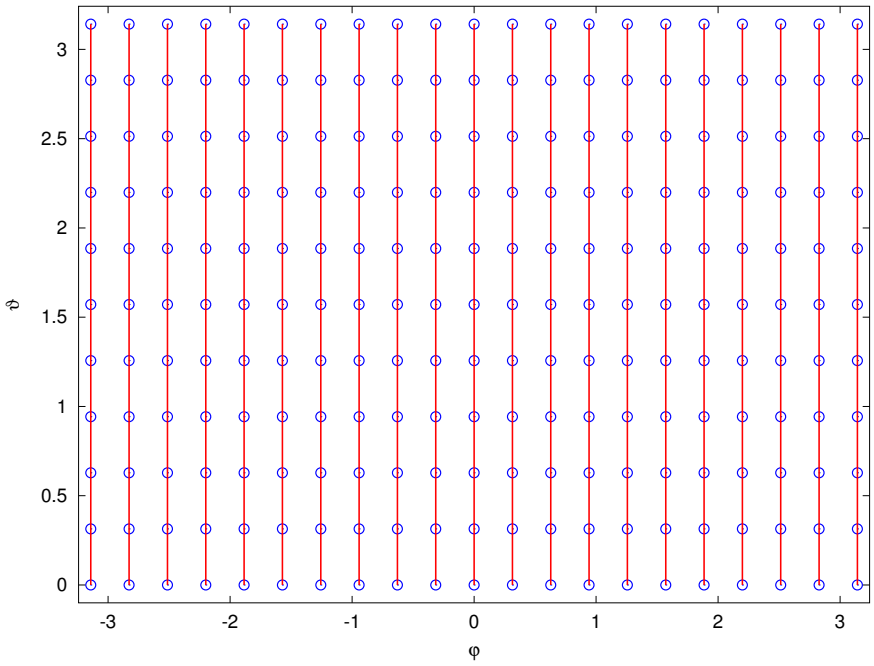


Abbildung 4.2: Parallelisierung über die Breite. Hierbei werden die ϑ-Werte für jedes φ gebündelt. Die verbundenen Punkte stellen jeweils die Daten für einen Thread dar. Die einzelnen Threads sind voneinander unabhängig und können daher ohne Probleme parallel abgearbeitet werden.

4.1 ... über die Breite

Die Grundlage für die Parallelisierung über die Breite, wie auch bei der über die Tiefe, stellt Abb. 4.1 dar. Bei der Parallelisierung über die Breite geht es darum, dass die Berechnung mit den jeweiligen ϑ-Werte eines φ-Wertes zusammen gehören und man sie somit ohne Synchronisationsoverhead parallelisieren kann. Abb. 4.2 zeigt eine Visualisierung des Vorgangs.

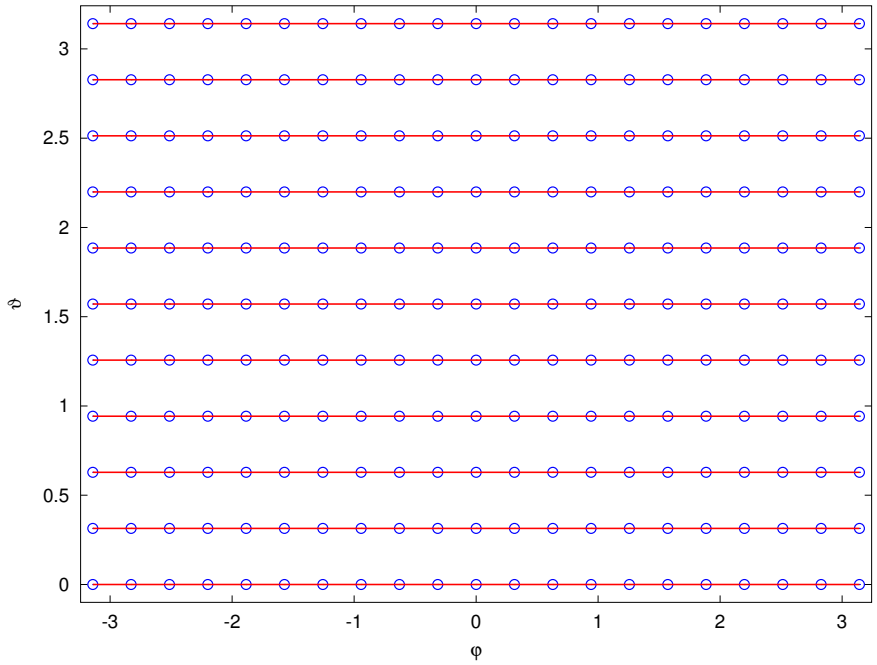


ABBILDUNG 4.3: Parallelisierung über die Tiefe. Anders als bei Abb. 4.2 sind es hier die φ-Werte die zusammen gehören und somit parallel bearbeitet werden können.

4.2 ... über die Tiefe

Die Parallelisierung über die Tiefe ist ähnlich derjenigen über die Breite. Der Unterschied liegt aber darin, dass statt den ϑ-Werten die φ-Werte unabhängig der ϑ-Werte gebündelt werden. Abb. 4.3 zeigt eine graphische Darstellung des Vorgangs.

4.3 Fourier-Transformation

Die Fourier-Transformation lässt sich auf verschiedenste Arten parallelisieren. Eine Möglichkeit gibt sich direkt aus dem Algorithmus, indem eine Parallelisierung über die Breite gemacht wird. Dabei werden die φ-Werte aufgeteilt. Wir müssen dabei aber zwingend nach jeder Runde zwischen den einzelnen Threads synchronisieren. Für OpenCL bedeutet dies, dass wir die Messwerte für die jeweiligen Work-Groups aufteilen müssen. Diese wiederum teilen die Punkte weiter auf, sodass schlussendlich jedes Work-Item innerhalb der Work-Group eine bestimmte Anzahl Messwerte seriell abarbeitet und sich danach mit der Work-Group synchronisiert bis diese fertig ist. Sollten mehr Werte vorhanden sein, als in einer Work-Group behandelt werden können, muss man dann nach jeder Runde global synchronisieren, was mit OpenCL 1.2 nur über den Start neuer Kernel möglich ist, was einen gewissen zusätzlichen Overhead mit sich bringt. Alternativ dazu können wir auch den Algorithmus über die Tiefe parallelisieren. Dazu würde jeweils ein ganzer FFT pro

Thread behandelt, dafür aber für jedes ϑ gleichzeitig. Bei der FFTW Implementation kann dies nicht direkt gesteuert werden, sondern es wird einfach die gewünschte Anzahl Threads bei der Erstellung des FFTW-Planes angegeben.

4.4 Legendre-Polynome

Bei den Legendre-Polynomen haben wir eigentlich keine Wahl und es gibt nur die Möglichkeit, über die Tiefe zu parallelisieren. Einerseits ist dies durch die GSL Library bereits vorgegeben, da jeweils direkt eine ganze Reihe Werte für einen ϑ-Wert berechnet wird. Andererseits macht eine Parallelisierung über die Breite, indem für die verschiedenen m-Werte parallel gerechnet wird, gar keinen Sinn, da diese direkt voneinander abhängig sind.

4.5 Integration

Für die Integration der Funktion hat man mehrere Möglichkeiten. Einerseits kann man ähnlich der FFT Berechnung über die Breite gehen und die Berechnung der verschiedenen m für ein spezifisches l parallel laufen lassen, oder man parallelisiert über die Tiefe, indem man die unterschiedlichen l parallel rechnet. Da die m auch negative Werte annehmen können, ist die Parallelisierung über die l bequemer. Es könnte sich aber trotzdem ein Vorteil daraus ergeben, da damit nicht jeder Thread den gleichen Arbeitsaufwand hat. Schliesslich hängen die Grenzen für das m direkt von l ab.

Bei der Méndez-Transformation ist insbesondere die Parallelisierung über die m gar nicht möglich, da diese stets 0 ist für alle $m \neq 0$.

5 SHCL

Mit der SHCL, kurz für “Spheric Harmonic Coefficient Library”, wurde eine Library geschaffen, mit der die sphärischen harmonischen Koeffizienten auf unterschiedliche Weise berechnet werden können. Die Library wurde in C++ geschrieben, wobei für einzelne Rechenschritte aber auf die jeweiligen Spezialbibliotheken zurückgegriffen wird. So wird für die schnelle Fourier-Transformation FFTW, kurz für “Fastest Fourier Transform in the West”, verwendet. Als Alternative kann auch eine experimentelle OpenCL Implementation des FFT-Algorithmus genutzt werden. Diese ist aber noch nicht ausgereift und nur für Experimente geeignet. Die Legendre-Polynome können dank der GSL, kurz für “GNU Scientific Library”, bis zu einem Grad l von 2700 berechnet werden. Um auch anderen Programmierumgebungen und -sprachen, wie Matlab, C, Python etc., die Möglichkeit zu geben, die Library zu nutzen, wird das API mittels C, genauer gesagt das C89 Subset mit zusätzlichem `const`, umgesetzt.

5.1 Memory Management

Benutzer der Library müssen aufpassen, dass sie bei Funktionen die ein `shcl_error *` erwarten, diesen unbedingt prüfen und danach mittels der entsprechenden `shcl_destroy_error(shcl_error err)` wieder deallozieren müssen. Am Ende des Funktionsrahmens müssen alle mittels `shcl_create_xxx()` oder `shcl_calloc_xxx()` erstellten Objekte wieder mittels entsprechender `shcl_destroy_xxx()` beziehungsweise `shcl_free()` zerstört werden. Dazu gehört auch das `shcl_error *` Objekt.

5.2 Info und Plan

Die zwei essentiellen Bauteile der SHCL Library sind das Info- und das Plan-Objekt. Ersteres enthält alle wichtigen Informationen, welche für die Transformation von Bedeutung sind. Damit ist es möglich, dass man beim allozieren der c_l^m beispielsweise automatisch die richtige Speichergrösse zurück erhält, je nachdem ob es sich um eine Méndez-Transformation oder um eine normale sphärische harmonische Transformation handelt. Zudem kann so das gleiche Info-Objekt für unterschiedliche Eingangsdaten verwendet werden.

Das Info Objekt enthält aber auch die Legendre-Polynome. Diese werden beim Erstellen berechnet, was zu Verzögerungen bei diesem Schritt führen kann. Das Ganze hat den Vorteil, dass man sowohl für die Vorwärts- als auch für die Inverse-Transformation jeweils das gleiche Info-Objekt benutzen kann, ohne dass die Polynome neu berechnet werden müssen. Dafür darf das Info-Objekt aber zwingend erst nach dem Plan zerstört werden.

Der Plan verhält sich ähnlich demjenigen der FFTW Library. Beim Erstellen wird ein Info-Objekt und die Richtung, mittels den Parametern `SHCL_FORWARD` oder `SHCL_INVERSE`, des Plans festgelegt. Nun kann man den Plan ausführen, indem `shcl_execute_plan(plan)` aufgerufen wird. Dies führt dann die Berechnung entweder der c_l^m oder eine Approximation der ursprünglichen Funktion durch und speichert das Resultat in das Ausgangsdatenarray.

5.3 Modularer Aufbau

Die SHCL ist so aufgebaut, dass einzelne Teile des Algorithmus einfach ausgetauscht werden können. Dies hat den Vorteil, dass man

jeweils ein Teilproblem optimieren kann, was gleich zu einer Optimierung der gesamten Laufzeit führt.

Legendre

Im Legendre-Modul wird jegliche Berechnung der Legendre-Polynome vorgenommen. Diese findet momentan einzig mittels GSL Library statt, kann aber jederzeit erweitert werden, sollte man es schaffen eine Implementation in OpenCL zu erstellen. Für die Méndez-Transformation und die normale sphärische harmonische Transformation stehen jeweils eigene Klassen zur Verfügung. Insbesondere die Méndez-Transformation wird dadurch optimiert, dass nur die $P_l(\cos(\vartheta))$ berechnet und gespeichert werden müssen.

Fourier

Beim Fourier-Modul gibt es bereits jetzt zwei mögliche Varianten, die mit `SCHL_CPU` und `SCHL_OCL` ausgewählt werden können. So wird einerseits die Möglichkeit gegeben, mittels der FFTW Library die Berechnung durchzuführen, andererseits gibt es bereits einen ersten FFT Prototypen in OpenCL. Leider funktioniert die Speichersynchronisation noch nicht ganz korrekt, was dazu führt, dass der FFT nur solange die richtigen Resultate liefert, wie er in den lokalen Speicher einer Work-Group passt. Dazu kommen noch weitere Einschränkungen für die Anzahl φ-Messwerte, welche zwingend einer Zweierpotenz entsprechen müssen, damit die OpenCL Variante funktioniert. Zudem wird momentan nur die Berechnung von `float`, also 32-Bit Gleitkommazahlen, unterstützt, was eine Einschränkung aufgrund der Möglichkeiten der meisten Grafikkarten darstellt. Für genaue Resultate und die Umgehung der Zweierpotenz-Einschränkung, wird momentan der FFTW-Algorithmus mittels `SHCL_CPU` bevorzugt.

Wenn es sich um eine Méndez-Transformation handelt, wird bei der CPU Variante zudem noch auf die Berechnung der FFT verzichtet und stattdessen nur eine Summe über die jeweiligen φ-Werte für jedes ϑ berechnet und normiert. Diese zusätzliche Optimierung ist derzeit noch nicht in OpenCL umgesetzt.

Integration

Die Integration wird, wie bereits gezeigt, mittels vier Algorithmen umgesetzt, die jeweils mit `SHCL_HOLLINGSWORTH`, `SHCL_SIMPSON`,

SHCL_TRAPEZOIDAL, und SHCL_RIEMANN ausgewählt werden können. Insbesondere bei der normalen sphärischen harmonischen Transformation macht der Integrationsvorgang für grössere l einen beachtlichen Teil der Berechnungsdauer aus. Hier würde sich allenfalls eine weitere OpenCL Implementation der verschiedenen Varianten lohnen, um eine zusätzliche Laufzeitverbesserung zu erhalten.

5.4 Anwendungsbeispiel

Mit einer kleinen C++ Client Implementation soll gezeigt werden, wie man die Library benutzen kann, um sphärisch harmonische Koeffizienten zu berechnen. Das Beispiel zeigt auf, wie zuerst ein shc_info Objekt erstellt wird. Danach wird Speicher für die Ein- und Ausgabedaten bereitgestellt und dem shcl_plan mitgegeben. Mittels shcl_plan_execute(plan) wird dieser dann umgesetzt und die Berechnung durchgeführt. Bei jedem Schritt wird jeweils, wie von der Library verlangt, geprüft, ob ein Fehler aufgetreten ist, und danach das Programm wieder abgeräumt. Bei einer Weiterentwicklung könnte diese beispielsweise in einem C++ Destruktor passieren, was den Code sicherlich noch übersichtlicher erscheinen liesse.

```
1  /*
2   * Sample implementation for a program using
       the SHCL Library
3   */
4
5  #include <shcl.h>
6
7  #include <complex>
8  #include <iostream>
9
10 int main() {
11   // Specify image dimensions.
12   int columns { 2048 };
13   int rows { 1024 };
14
15   // Specify the number of images to
       transform. (Must be >= 1)
16   int numberOfImages { 1 };
17
18   // Specify the maximum l you want to
       calculate.
```

```
19  // This must be smaller than the number of
       rows!
20  int maxL { 10 };
21
22  // Create an error object to allow passing
       erros from the library to the application.
23  shcl_error error { nullptr };
24
25  // Create an info object with all
       information needed for the transform.
26  shcl_info_const info {
27    shcl_create_info(
28      SHCL_MENDEZ,
29      SHCL_ROW_MAJOR,
30      columns,
31      rows,
32      numberOfImages,
33      maxL,
34      SHCL_CPU,
35      SHCL_HOLLINGSWORTH,
36      SHCL_SCALE_SYMMETRIC,
37      &error
38    )
39  };
40
41  // Check if an error happend and if so
       print it to std::cout, destroy the
       current state including the error!
42  if (error) {
43    // Print the error message.
44    std::cout << "Error while creating info:
         " << shcl_error_message(error);
45    shcl_destroy_error(error);
46    shcl_destroy_info(info);
47    return 1;
48  }
49
50  // Allocate space for the input data.
51  // It will be enough for the specified
       image dimensions and numbers.
52  shcl_complex * inData {
```

```
    shcl_calloc_image(info, &error) };

// As alwys check for an error and do the cleanup if true.
if (error) {
  std::cout << "Error while allocation input data memory: " << shcl_error_message(error);
  shcl_destroy_error(error);
  shcl_free(inData);
  shcl_destroy_info(info);
  return 1;
}

// Allocate space for the output data.
// It will be enough for all clm depending on the transform type.
shcl_complex * outData {
    shcl_calloc_clm(info, &error) };

// Error check and cleanup if true.
if (error) {
  std::cout << "Error while allocation output data memory: " << shcl_error_message(error);
  shcl_destroy_error(error);
  shcl_free(inData);
  shcl_free(outData);
  shcl_destroy_info(info);
  return 1;
}

// Fill input data.
for (int t { 0 }; t < numberOfImages; ++t) {
  for (int r { 0 }; r < rows; ++r) {
    for (int c { 0 }; c < columns; ++c) {

      // Get the image array index.
      int index { shcl_image_index(info, c, r, t, &error) };

```

```
85          // Error check and cleanup if true.
86          if (error) {
87            std::cout << "Error while
                 calculating image index: " <<
                 shcl_error_message(error);
88            shcl_destroy_error(error);
89            shcl_free(inData);
90            shcl_free(outData);
91            shcl_destroy_info(info);
92            return 1;
93          }
94
95          // Fill image data point.
96          inData[index][0] =
               static_cast<double>(c);
97          inData[index][1] = 0;
98        }
99      }
100   }
101
102   // Create a plan with the given info, data
       and direction.
103   shcl_plan plan { shcl_create_plan(info,
       inData, outData, SHCL_FORWARD, &error) };
104
105   // Error check and cleanup if true.
106   if (error) {
107     std::cout << "Error while creating the
          shcl plan: " <<
          shcl_error_message(error);
108     shcl_destroy_error(error);
109     shcl_free(inData);
110     shcl_free(outData);
111     shcl_destroy_plan(plan);
112     shcl_destroy_info(info);
113     return 1;
114   }
115
116   // Execute the transformation by executing
       the plan.
117   shcl_execute_plan(plan, &error);
```

```
118
119   // Error check and cleanup if true.
120   if (error) {
121     std::cout << "Error while executing the
          shcl plan: " <<
          shcl_error_message(error);
122     shcl_destroy_error(error);
123     shcl_free(inData);
124     shcl_free(outData);
125     shcl_destroy_plan(plan);
126     shcl_destroy_info(info);
127     return 1;
128   }
129
130   // Print the transformed coefficients.
131   for (int i { 0 }; i < numberOfImages; ++i) {
132     std::cout << "Image: " << (i + 1) << "\n";
133     for (int l { 0 }; l <= maxL; ++l) {
134
135       // Get the image index. If since this
            is the Méndez-Transform the m is
            always equal 0.
136       int index {
            shcl_coefficient_index(info, l, 0, i,
            &error) };
137
138       // Error check and cleanup if true.
139       if (error) {
140         std::cout << "Error while calculating
              the clm index: " <<
              shcl_error_message(error);
141         shcl_destroy_error(error);
142         shcl_free(inData);
143         shcl_free(outData);
144         shcl_destroy_plan(plan);
145         shcl_destroy_info(info);
146         return 1;
147       }
148       std::cout << std::complex<double> {
            outData[index][0], outData[index][1]
            } << "\n";
```

```
149     }
150   }
151
152   // Do a final cleanup after finishing all
          the work.
153   // This includes destroying the error!
154   shcl_destroy_error(error);
155   shcl_free(inData);
156   shcl_free(outData);
157   shcl_destroy_plan(plan);
158   shcl_destroy_info(info);
159   return 0;
160 }
```

5.5 Ausbaumöglichkeiten

Die Library ist noch ausbaufähig. So werden momentan nur komplexwertige Ein- und Ausgangswerte unterstützt. Dies macht bei der Bildbearbeitung wenig Sinn, da man dort meist mit reellwertigen Bildern arbeitet. Bei der Méndez-Transformation kommt hinzu, dass bei reellem Eingangsbild die c_l^m auch stets reell sind, womit eine zusätzliche Speichereinsparungen möglich ist.

Es gibt auch noch einiges an Optimierungs- und Verbesserungspotential. So könnte man beispielsweise noch OpenCL Module für die Integration oder die Berechnung der Legendre-Polynome erstellen, wobei letzteres ein nicht ganz so triviales Unterfangen ist. Auch die FFT Implementation kann noch deutlich ausgebaut und verbessert werden, um eine für den Algorithmus optimale Berechnung zu ermöglichen. So wäre eine Parallelisierung der FFT über die Tiefe sinnvoller als die Parallelisierung über die Breite, wie es derzeit implementiert ist. Zudem fehlt momentan noch die Implementation der inversen Transformation

6 Performance

In dieser Arbeit steht die schnelle Berechnung der c_l^m im Vordergrund. Um die Resultate zu sehen, müssen Messungen durchgeführt werden. Alle Messungen sind auf einem HP EliteBook Folio 1040 Notebook mit einem Intel®Core™i7-4600U @ 2.10GHz Prozessor und 8 GB DDR3-1600 RAM auf einem Linux Betriebssystem ausgeführt worden. Für jeden Messpunkt ist die Messung jeweils 50 Mal wiederholt und dann gemittelt worden, um externe Einflüsse, wie andere gleichzeitig Laufende Tasks, zu vermindern.

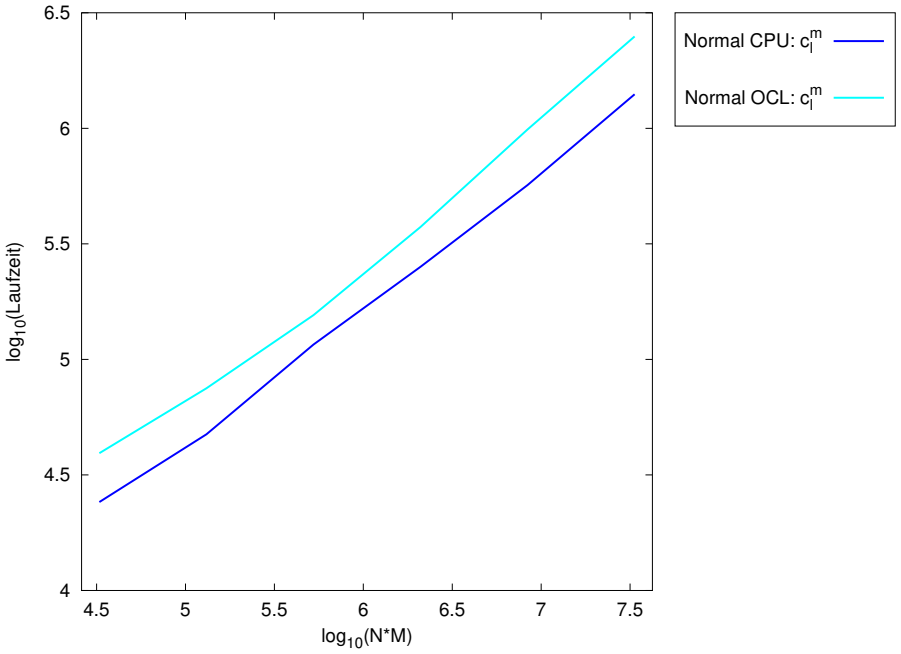


Abbildung 6.1: Vergleich der OpenCL- und CPU-Implementation für die normale sphärische harmonische Transformation. Die beiden Kurven sind bis auf einen konstanten Faktor praktisch identisch. ($l = 100$, $N * M =$ 256x128 bis 8192x4096 Punkte)

6.1 Normale Transformation

Bei der normalen sphärischen Transformation zeigt sich, dass sie nicht nur rechen- sondern auch speicherintensiv ist. So konnten die Messungen, bei der jeweils das maximal mögliche l gewählt wurde, nur bis 2048x1024 Punkte durchgeführt werden. Für die nächst grössere Messung bei 4096x2048 Punkten hat das System, selbst mit einer 9 GB Swap-Partition, nicht genügend Speicher zur Verfügung stellen können.

In Abb. 6.1 sieht man einen Vergleich der OpenCL und der FFTW Variante für die Transformation, bei der l eine Konstante ist. Wie man sieht, ist die Laufzeit bis auf einen konstanten Faktor beinahe identisch. Es liegt nahe, dass neben dem Overhead für die Erstellung der Kernel in OpenCL auch die bessere Optimierung der FFTW Variante die in dem konstanten Faktor enthalten sind.

Abb. 6.2 zeigt den Fall, dass l nicht mehr konstant ist, sondern proportional zunimmt. Die ersten Messwerte zeigen deutlich, dass bei OpenCL ein gewisser Overhead für die Erstellung der Kernel vorhanden ist. Sobald die Eingangsdaten grösser werden, gleicht sich dies aber wieder aus und die Werte sind beinahe identisch.

6.2 Méndez-Transformation

Für den Vergleich der Méndez-Transformation gelten die gleichen Voraussetzung wie bei der normalen sphärischen harmonischen

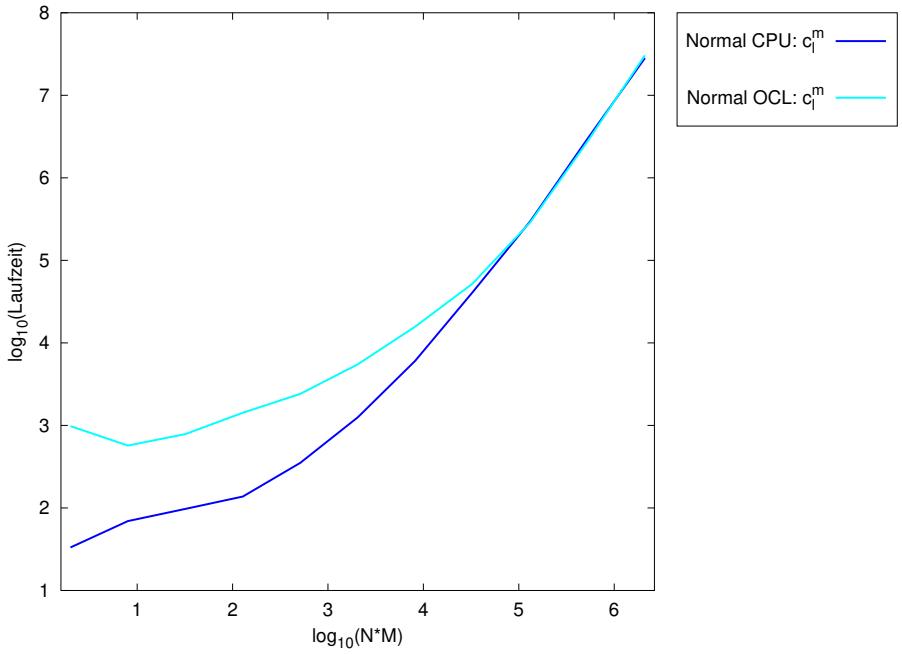


Abbildung 6.2: Vergleich der OpenCL- und CPU-Implementation für die normale sphärische harmonische Transformation. Für den Fall eines proportional zur Bildauflösung wachsenden l wird der Unterschied bei wachsender Punktezahl immer kleiner. (l proportional, $N * M$ = 2x1 bis 2048x1024 Punkte)

Transformation. Zugegebenermassen sind die Vergleiche zwischen der OpenCL und der CPU Variante bei dieser Transformation nicht unbedingt fair, da bei letzterer gar kein FFT mehr gemacht wird, sondern nur die Summe aller Zahlen berechnet und skaliert wird. Wir stellen sofort fest, dass die Méndez-Transformation eine erhebliche Performance-Steigerung gegenüber der normalen sphärischen harmonischen Transformation aufzeigt.

Abb. 6.3 mit konstantem l zeigt, dass bei der OpenCL Lösung die Laufzeit jeweils um einen konstanten Faktor grösser ist.

In Abb. 6.4 scheinen sich die Kurven anzunähern, was aber nur langsam passiert und so muss sich auch hier die OpenCL Variante geschlagen geben.

6.3 Schlussfolgerungen

Für die normale sphärische harmonische Transformation sieht man vor allem bei der Datenintegration und der Berechnung für die assoziierten Legendre-Polynome ein grosses Potential zur Performance-Steigerung. Bei der Méndez-Transformation hat sich gezeigt, dass eine deutlich schnellere Berechnung der gesuchten Koeffizienten im Gegensatz zur normalen sphärischen harmonischen Transformation möglich ist.

Ein direkter Vergleich der beiden Transformationsarten in Abb. 6.5 macht deutlich, das die Méndez-Transformation um einiges schneller

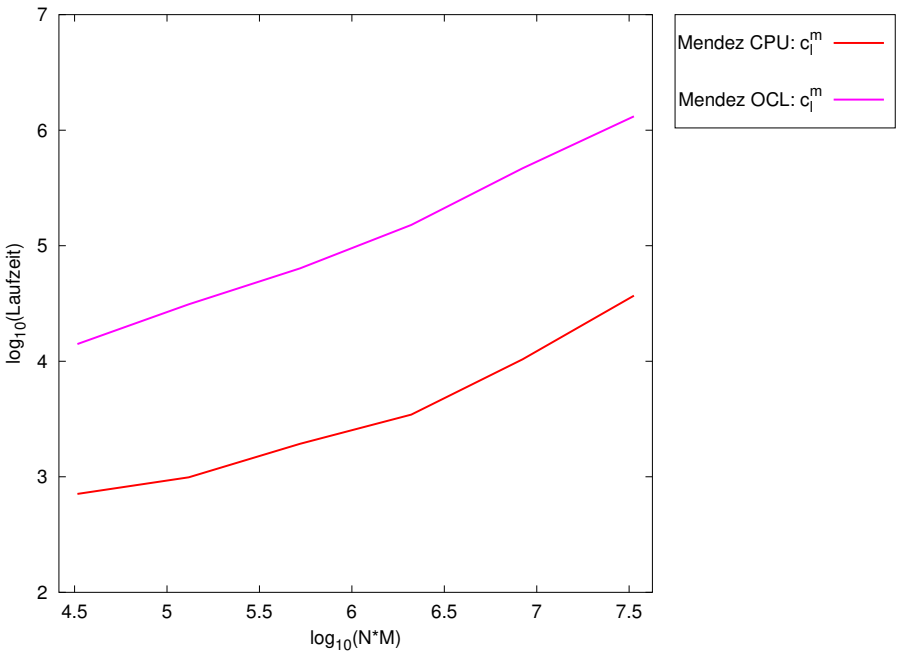


Abbildung 6.3: Vergleich der OpenCL- und CPU-Implementation für die Méndez-Transformation. Hier sind die beiden Kurven bis auf einen konstanten Faktor praktisch identisch. ($l = 100$ konstant, $N * M$ = 256x128 bis 8192x4096 Punkte)

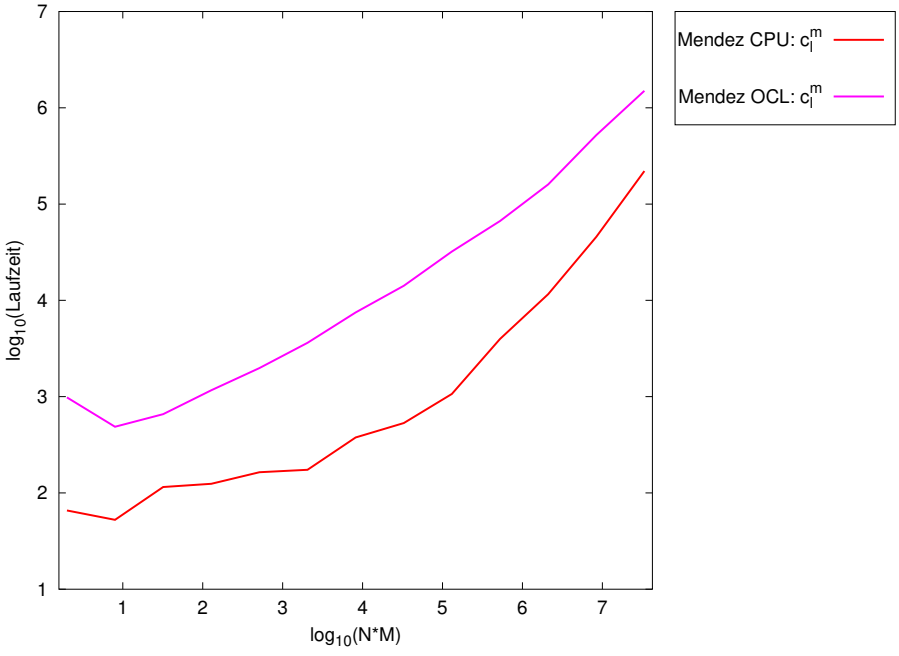


Abbildung 6.4: Vergleich der OpenCL- und CPU-Implementation für die Méndez-Transformation. Wie man erkennen kann, ist der Aufwand bis auf einen konstanten Faktor in etwa gleich. (l proportional, $N * M$ = 2x1 bis 8192x4096 Punkte)

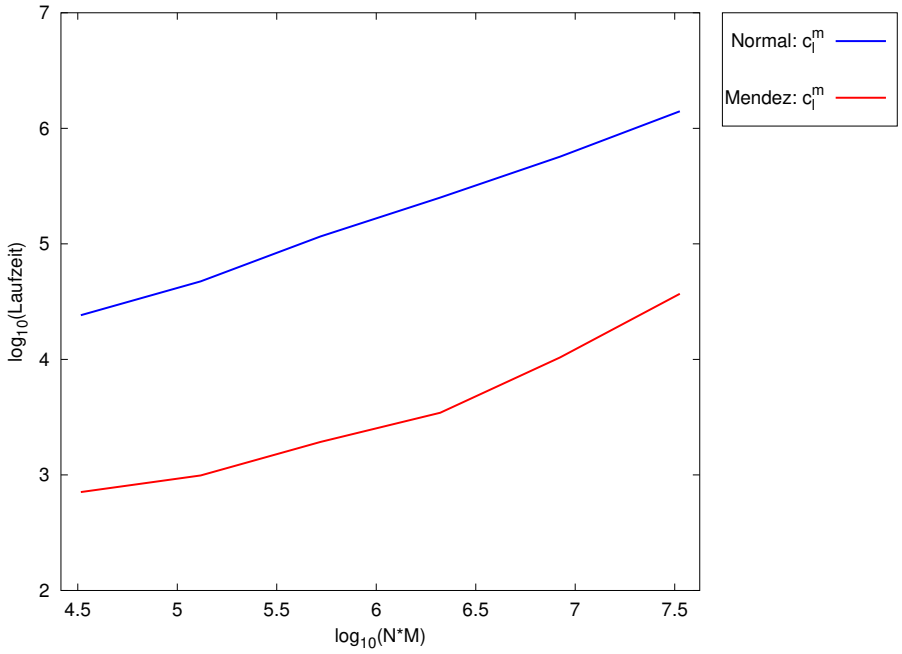


Abbildung 6.5: Vergleich der normalen sphärischen harmonischen Transformation und der Méndez-Transformation. Beide weisen bei gleich bleibendem l eine ähnliche Steigung Laufzeit auf. Der Konstante Faktor macht aber einen grossen Unterschied aus. ($l = 100$, $N * M$ = 256x128 bis 8192x4096 Punkte)

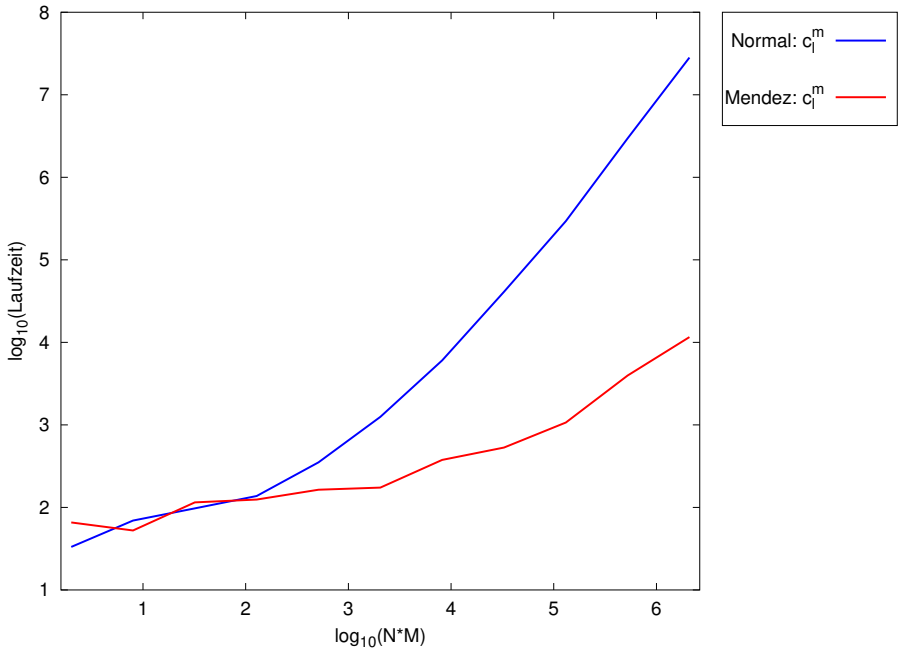


Abbildung 6.6: Vergleich CPU Varianten der normalen sphärischen harmonischen Transformation und der Méndez-Transformation. Obwohl die beiden für kleine Auflösungen noch gleich sind, zeigt sich schnell, dass die Méndez-Transformation bei grösserer Auflösung bedeutend schneller ist. (l proportional, $N * M$ = 2x1 bis 2048x1024 Punkte)

ist. Wenn man noch, wie in Abb. 6.6, das l proportional anwachsen lässt, macht sich der Vorteil noch klarer bemerkbar.

Um die Ergebnisse etwas fassbarer zu machen zeigt Abb. 6.7 das Resultat noch in absoluten Zahlen.

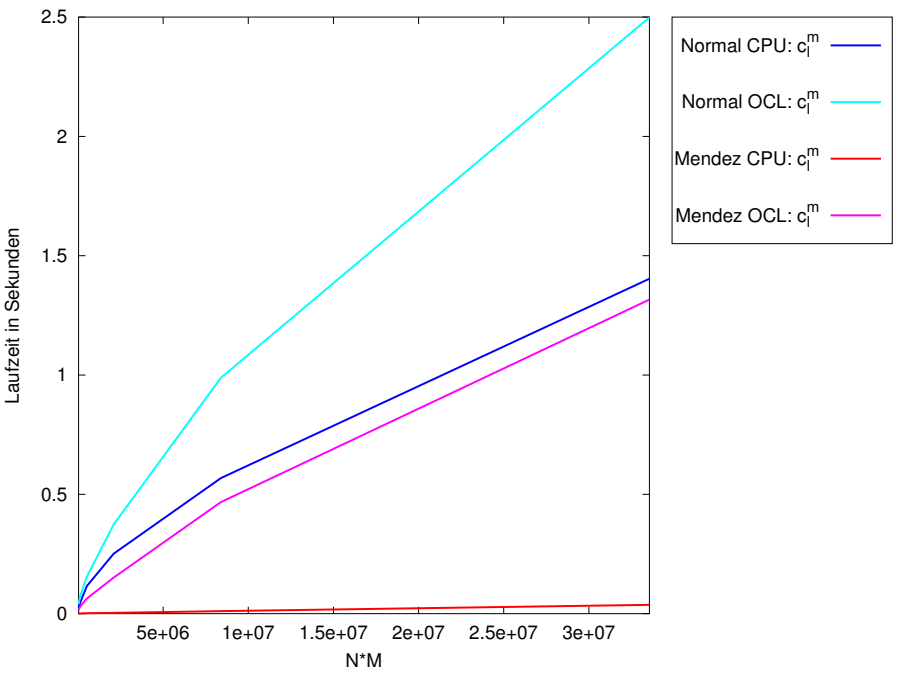


Abbildung 6.7: Vergleich der OpenCL- und CPU-Implementation für die beiden Transformationen in absoluten Zahlen. Man erkennt, dass die Méndez-Transformation deutlich schneller ist als die normale sphärische harmonische Transformation. Die OpenCL Varianten sind momentan noch nicht so ausgereift, was sich wiederum in der Laufzeit widerspiegelt. ($l = 100$, $N * M =$ 256x128 bis 8192x4096 Punkte)

Literatur

[Ben] E. A. Bender. (). Deriving the Trapezoidal Rule Error, Adresse: http://math.ucsd.edu/~ebender/20B/77_Trap.pdf (besucht am 12/2016).

[Bur08] C. S. Burrus, *Fast Fourier Transforms*. Okt. 2008. Adresse: http://cnx.org/content/col10550.

[But] (). FFT: An 8 Input Butterfly, Adresse: http://www.alwayslearn.com/dft%20and%20fft%20tutorial/DFTandFFT_FFT_Butterfly_8_Input.html (besucht am 12/2016).

[DB06] L. Debnath und D. Bhatta, *Integral Transform and Their Applications*, 2. Aufl. Chapman and Hall/CRC, Okt. 2006, ISBN: 978-1-584-88575-7.

[FH02] W. E. Featherstone und S. A. Holmes, „A Unified Approach to the Clenshaw Summation and the Recursive Computation of Very High Degree and Order Normalised Associated Legendre Functions“, *Journal of Geodesy*, Bd. 76, Nr. 5, S. 279–299, Mai 2002. DOI: 10.1007/s00190-002-0216-2.

[HH59] J. W. Hollingsworth und H. F. Hunter, „Simpson's rule for an odd number of intervals“, *ACM '59 Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*, S. 1–2, 1959. DOI: 10.1145/612201.612205.

[KL07] J.-H. Kwon und J.-K. Lee, „Numerical Computation of Ultra-High-Degree Legendre Function“, *Journal of the Korean Society of Surveying Geodesy Photogrammetry and Cartography*, Bd. 27, S. 63–68, 1 Feb. 2007.

[Lov11] J. Loviscach. (März 2011). 23.04 Numerische Integration, Trapezregel, Simpson-Regel, Adresse: https://www.youtube.com/watch?v=cvHUpXNZmlM.

[Mé16] T. Méndez, *Nichtkommunative Bildbearbeitung*. Aug. 2016, ISBN: 978-3-033-05830-9.

[Sel+13] I. A. Selezneva, Y. L. Ratis, E. Hernández, J. Pérez-Quiles und P. Fernández de Córdoba, „A Code to Calculate High Order Legendre Polynomials and Functions“, *Revista de la Academia Colombiana de Ciencias Exactas, Físicas y Naturales*, Bd. 37, S. 541 –544, Dez. 2013, ISSN: 0370-3908.

[Wika] (). Aliasing, Adresse: https://en.wikipedia.org/wiki/Aliasing (besucht am 12/2016).

[Wikb] (). Fourier-Analysis, Adresse: https://de.wikipedia.org/wiki/Fourier-Analysis (besucht am 12/2016).

[Wikc] (). Legendre Polynomials, Adresse: https://en.wikipedia.org/wiki/Legendre_polynomials (besucht am 12/2016).

[Wikd] (). Spherical harmonics, Adresse: https://en.wikipedia.org/wiki/Spherical_harmonics (besucht am 12/2016).

A API

Die API der SHCL Library ist in C geschrieben. Für externe Programme genügt das inkludieren der `shcl.h` Datei. Für die Dokumentation wird `doxygen` verwendet.

A.1 shcl_types.h File Reference

This file contains the type definitions for the SHCL Library.

Typedefs

- typedef double shcl_complex[2]

  Specifies the complex data type used in the SHCL.
- typedef struct lib_error * shcl_error

  Specifies a pointer to a SHCL error object.
- typedef struct lib_info const * shcl_info_const

  Specifies a pointer to the constant SHCL info object.
- typedef struct lib_plan * shcl_plan

  Specifies a pointer to a SHCL plan object.

Enumerations

- enum shcl_transform {
  SHCL_NORMAL,
  SHCL_MENDEZ
  }

  Specifies the available transform types.
- enum shcl_direction {
  SHCL_FORWARD,
  SHCL_INVERSE
  }

  Specify in which direciton you want to transform.
- enum shcl_platform {
  SHCL_CPU,
  SHCL_OCL
  }

  Specify which platform you want to use.
- enum shcl_array_major {
  SHCL_ROW_MAJOR,
  SHCL_COLUMN_MAJOR
  }

  Specify which array major the data is in.
- enum shcl_scaling {
  SHCL_SCALE_NONE,
  SHCL_SCALE_SYMMETRIC,
  SHCL_SCALE_FORWARD,
  SHCL_SCALE_INVERSE
  }

  Specify what type of scaling you want to use.

- enum shcl_integration {
  SHCL_RIEMANN,
  SHCL_TRAPEZOIDAL,
  SHCL_SIMPSON,
  SHCL_HOLLINGSWORTH
  }

  Specify which integral calculation method you want to use to integrate over the theta values..

Detailed Description

This file contains the type definitions for the SHCL Library.

Typedef Documentation

shcl_complex

`shcl_complex`

Specifies the complex data type used in the SHCL.

Author
: Hansruedi Patzen

Since
: 0.1.0

shcl_error

`shcl_error`

Specifies a pointer to a SHCL error object.

Author
: Hansruedi Patzen

Since
: 0.1.0

shcl_info_const

`shcl_info_const`

Specifies a pointer to the constant SHCL info object.

Author
: Hansruedi Patzen

Since
: 0.1.0

shcl_plan

`shcl_plan`

Specifies a pointer to a SHCL plan object.

Author

Hansruedi Patzen

Since

0.1.0

Enumeration Type Documentation

shcl_array_major

`enum shcl_array_major`

Specify which array major the data is in.

Author

Hansruedi Patzen

Since

0.1.0

Parameters

SHCL_ROW_MAJOR	Use this if your data is in a row major format.
SHCL_COLUMN_MAJOR	Use this if your data is in a column major format.

shcl_direction

`enum shcl_direction`

Specify in which direction you want to transform.

Author

Hansruedi Patzen

Since

0.1.0

Parameters

SHCL_FORWARD	Use this if you want to run a forward transformation.
SHCL_INVERSE	Use this if you want to run an inverse transformation.

shcl_integration

`enum shcl_integration`

Specify which integral calculation method you want to use to integrate over the theta values..

Author
: Hansruedi Patzen

Since
: 0.1.0

Parameters

SHCL_RIEMANN	Use normal left-point Riemann sums.
SHCL_TRAPEZOIDAL	Use the trapezoidal rule.
SHCL_SIMPSON	Use the Simpson rule.
SHCL_HOLLINGSWORTH	Use the Hollingsworth-Hunter rule.

shcl_platform

`enum shcl_platform`

Specify which platform you want to use.

Author
: Hansruedi Patzen

Since
: 0.1.0

Parameters

SHCL_CPU	Use normal CPU instructions and libraries.
SHCL_OCL	Use OpenCL Code whenever possible.

shcl_scaling

`enum shcl_scaling`

Specify what type of scaling you want to use.

Author
: Hansruedi Patzen

Since
0.1.0

Parameters

SHCL_NONE	Do not scale in any direction.
SHCL_SYMMETRIC	Scale in both directions using 1/sqrt(N).
SHCL_FORWARD	Scale only when doing a forward transform using 1/N.
SHCL_INVERSE	Scale only when doing an inverse transform using 1/N.

shcl_transform

`enum shcl_transform`
Specifies the available transform types.

Author
Hansruedi Patzen

Since
0.1.0

Parameters

SHCL_NORMAL	Use this if you want to run a normal spherical transformation.
SHCL_MENDEZ	Use this if you want to run a Méndez transformation.

A.2 shcl_memory.h File Reference

This file contains the interface to handle SHCL memory acquisition and release.

```
#include <shcl/internal/visibility.h>
#include <shcl/internal/shcl_types.h>
```

Functions

- SHCL_EXPORT shcl_complex * shcl_calloc_image (shcl_info_const const info, shcl_error *out_error)

  Allocate and initialize enough memory to hold the number of image data.

- SHCL_EXPORT shcl_complex * shcl_calloc_clm (shcl_info_const const info, shcl_error *out_error)

  Allocate and initialize enough memory to hold the number of clm specified by the shcl_transform.

- SHCL_EXPORT void shcl_free (shcl_complex *data)

  Free the memory for the given data array pointer.

Detailed Description

This file contains the interface to handle SHCL memory acquisition and release.

Function Documentation

shcl_calloc_clm()

```
SHCL_EXPORT shcl_complex* shcl_calloc_clm (
                shcl_info_const const info,
                shcl_error * out_error )
```

Allocate and initialize enough memory to hold the number of clm specified by the shcl_transform.

Author
: Hansruedi Patzen

Since
: 0.1.0

Warning
: The out_error must be checked after calling this function.

Parameters

in	*info*	The info object containing the information needed to calculate the space to allocate.
out	*out_error*	A pointer to an error object. Might be NULL.

Returns
A pointer to the first index of the data array.

shcl_calloc_image()

```
SHCL_EXPORT shcl_complex* shcl_calloc_image (
                shcl_info_const const info,
                shcl_error * out_error )
```

Allocate and initialize enough memory to hold the number of image data.

Author
Hansruedi Patzen

Since
0.1.0

Warning
The out_error must be checked after calling this function.

Parameters

in	*info*	The info object containing the information needed to calculate the space to allocate.
out	*out_error*	A pointer to an error object. Might be NULL.

Returns
A pointer to the first index of the data array.

shcl_free()

```
SHCL_EXPORT void shcl_free (
                shcl_complex * data )
```

Free the memory for the given data array pointer.

Author
Hansruedi Patzen

Since
0.1.0

Parameters

in	*data*	A pointer to the shcl_complex data which needs to be freed.

A.3 shcl_info.h File Reference

This file contains the interface to handle and interact with SHCL info objects.

```
#include <shcl/internal/visibility.h>
#include <shcl/internal/shcl_types.h>
#include <stdint.h>
```

Functions

- SHCL_EXPORT shcl_info_const shcl_create_info (shcl_transform transform_type, shcl_array_major array_major, int32_t columns, int32_t rows, int32_t number_of_images, int32_t max_l, shcl_platform platform, shcl_integration integration_method, shcl_scaling scaling, shcl_error *out_error)

  Creates a constant info object.

- SHCL_EXPORT void shcl_destroy_info (shcl_info_const info)

  Cleanup and destroy the info object and all its data.

- SHCL_EXPORT int32_t shcl_coefficient_index (shcl_info_const const info, int32_t const l, int32_t const m, int32_t const image_number, shcl_error *out_error)

  Calculates the coefficient index based on the given information.

- SHCL_EXPORT int32_t shcl_image_index (shcl_info_const const info, int32_t const column_id, int32_t const row_id, int32_t const image_number, shcl_error *out_error)

  Calculates the image index based on the given information.

Detailed Description

This file contains the interface to handle and interact with SHCL info objects.

Function Documentation

shcl_coefficient_index()

```
SHCL_EXPORT int32_t shcl_coefficient_index (
                  shcl_info_const const info,
                  int32_t const l,
                  int32_t const m,
                  int32_t const image_number,
                  shcl_error * out_error )
```

Calculates the coefficient index based on the given information.

Author

Hansruedi Patzen

Since
0.1.0

Warning
The out_error must be checked after calling this function.

Parameters

in	*info*	The info object use for boundary checks.
in	*l*	Specifies the l which one wants to access.
in	*m*	Specifies the m which one wants to access. Must be *smaller or equal* than l.
in	*image_number*	Specifies for which image the requested transformed point belongs to. Must be *smaller* than the specified number_of_images when the info was created.
out	*out_error*	A pointer to an error object. Might be NULL.

Returns
Returns the coefficients index.

shcl_create_info()

```
SHCL_EXPORT shcl_info_const shcl_create_info (
                  shcl_transform transform_type,
                  shcl_array_major array_major,
                  int32_t columns,
                  int32_t rows,
                  int32_t number_of_images,
                  int32_t max_l,
                  shcl_platform platform,
                  shcl_integration integration_method,
                  shcl_scaling scaling,
                  shcl_error * out_error )
```

Creates a constant info object.

Author
Hansruedi Patzen

Since
0.1.0

Allows a user to create a constant info object which can be used to execute a shcl_plan. This object can be used to run different plans given the same data dimensions.

Warning

Upon creation the info object will calculate all needed Legendre-Polynomials. Which could take some time.
The out_error must be checked after calling this function.

Parameters

in	*transform_type*	Specifies the kind of transformation the user wants to run. Valid values are: • SHCL_MENDEZ • SHCL_NORMAL
in	*array_major*	Specifies in which array major the given data is present. Valid values are: • SHCL_ROW_MAJOR • SHCL_Column_MAJOR
in	*columns*	Specifies the number of columns the images has. This is equal to the number of discrete phi values.
in	*rows*	Specifies the number of rows the image has. This is equal to the number of discrete theta values.
in	*number_of_images*	Specifies the number of images one wants to transform with a single plan. This can be used when multiple images are present in a sequence.
in	*max_l*	Specifies the maximum l one is interested in for this specific transform. In conjunction with the inverse transform this will determine the number of clm present in the input. This *must* be smaller than the number of rows.
in	*platform*	Specifies the platform on which the transform will be run. Valid values are: • SHCL_CPU • SHCL_OCL

Parameters

in	*integration_method*	Specifies the preferred integration method. Valid values are: • SHCL_HOLLINGSWORTH • SHCL_SIMPSON • SHCL_TRAPEZOIDAL • SHCL_RIEMANN
in	*scaling*	Specifies whether the transform should be scaled and if so in which direction. Valid values are: • SHCL_SCALE_SYMMETRIC • SHCL_SCALE_FORWARD • SHCL_SCALE_INVERSE • SCHL_SCALE_NONE
out	*out_error*	A pointer to an error object. Might be NULL.

Returns

Returns a pointer to a constant info object.

shcl_destroy_info()

```
SHCL_EXPORT void shcl_destroy_info (
                shcl_info_const info )
```

Cleanup and destroy the info object and all its data.

Author

Hansruedi Patzen

Since

0.1.0

Warning

Calling destroy multiple times on the same error object will result in undefined behavior.
The info object must be destroyed *AFTER* every depending plan has been destroyed!

Parameters

in	*info*	The info object to be destroyed.

shcl_image_index()

```
SHCL_EXPORT int32_t shcl_image_index (
                  shcl_info_const const info,
                  int32_t const column_id,
                  int32_t const row_id,
                  int32_t const image_number,
                  shcl_error * out_error )
```

Calculates the image index based on the given information.

Author

Hansruedi Patzen

Since

0.1.0

Warning

The out_error must be checked after calling this function.

Parameters

in	*info*	The info object use for boundary checks.
in	*column_id*	Specifies the column in which the point is found.
in	*row_id*	Specifies the row in which the point is found.
in	*image_number*	Specifies for which image the requested point belongs to. Must be smaller than the specified number_of_images when the info was created.
out	*out_error*	A pointer to an error object. Might be NULL.

Returns

Returns the image index.

A.4 shcl_plan.h File Reference

This file contains the interface to handle and interact with SHCL plan objects.

```
#include <shcl/internal/visibility.h>
#include <shcl/internal/shcl_types.h>
#include <stdint.h>
```

Functions

- SHCL_EXPORT shcl_plan shcl_create_plan (shcl_info_const const info, shcl_complex ∗in_data, shcl_complex ∗out_data, shcl_direction direction, shcl_error ∗out_error)
- SHCL_EXPORT void shcl_destroy_plan (shcl_plan plan)

  Cleanup and destroy the plan object and all its data.
- SHCL_EXPORT void shcl_execute_plan (shcl_plan const plan, shcl_error ∗out_error)

  Execute the plan which runs the specified transformation and puts the result in the output data array.

Detailed Description

This file contains the interface to handle and interact with SHCL plan objects.

Function Documentation

shcl_create_plan()

```
SHCL_EXPORT shcl_plan shcl_create_plan (
                  shcl_info_const const info,
                  shcl_complex * in_data,
                  shcl_complex * out_data,
                  shcl_direction direction,
                  shcl_error * out_error )
```

Author
: Hansruedi Patzen

Since
: 0.1.0

Warning
: The out_error must be checked after calling this function.

Parameters

in	*info*	An info object containing all the information needed to run the plan including the Legendre-Polynomials.
in	*in_data*	The input data, depending on the direction this is either the image or the clm data.
in	*out_data*	The output data, depending on the direction this is either the clm or the image data.
in	*direction*	Specifies the transformation direction. Valid values are: • SHCL_FORWARD • SHCL_INVERSE
out	*out_error*	The error object whose message should be retrieved. Might be NULL.

Returns

Returns a pointer to a plan object.

shcl_destroy_plan()

```
SHCL_EXPORT void shcl_destroy_plan (
                shcl_plan plan )
```

Cleanup and destroy the plan object and all its data.

Author

Hansruedi Patzen

Since

0.1.0

Warning

Calling destroy multiple times on the same error object will result in undefined behavior.
The plan object must be destroyed *BEFORE* its info has been destroyed!

Parameters

in	*plan*	The plan object to be destroyed.

shcl_execute_plan()

```
SHCL_EXPORT void shcl_execute_plan (
                  shcl_plan const plan,
                  shcl_error * out_error )
```

Execute the plan which runs the specified transformation and puts the result in the output data array.

Author
: Hansruedi Patzen

Since
: 0.1.0

Warning
: The out_error must be checked after calling this function.

Parameters

in	*plan*	The plan you want to run.
out	*out_error*	The error object whose message should be retrieved. Might be NULL.

A.5 shcl_error.h File Reference

This file contains the interface to handle and interact with SHCL error object.

```
#include <shcl/internal/visibility.h>
#include <shcl/internal/shcl_types.h>
```

Functions

- SHCL_EXPORT char const * shcl_error_message (shcl_error const error)

  Retrieve the contained error message from the error object.
- SHCL_EXPORT void shcl_destroy_error (shcl_error error)

  Cleanup and destroy the error object.

Detailed Description

This file contains the interface to handle and interact with SHCL error object.

Function Documentation

shcl_destroy_error()

```
SHCL_EXPORT void shcl_destroy_error (
                shcl_error error )
```

Cleanup and destroy the error object.

Author
: Hansruedi Patzen

Since
: 0.1.0

Warning
: Calling destroy multiple times on the same error object will result in undefined behavior.
  This function must be called after every *positive* error check to prevent leaking memory!

Parameters

in	*error*	The error object to be destroyed.

shcl_error_message()

```
SHCL_EXPORT char const* shcl_error_message (
```

```
shcl_error const error )
```

Retrieve the contained error message from the error object.

Author
: Hansruedi Patzen

Since
: 0.1.0

Warning
: If the error is not empty it must be destroyed by the user!

Parameters

in	*error*	The error object whose message should be retrieved. Might be NULL.

Returns
: A pointer to the message contained in the error or NULL if there was no error.