LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

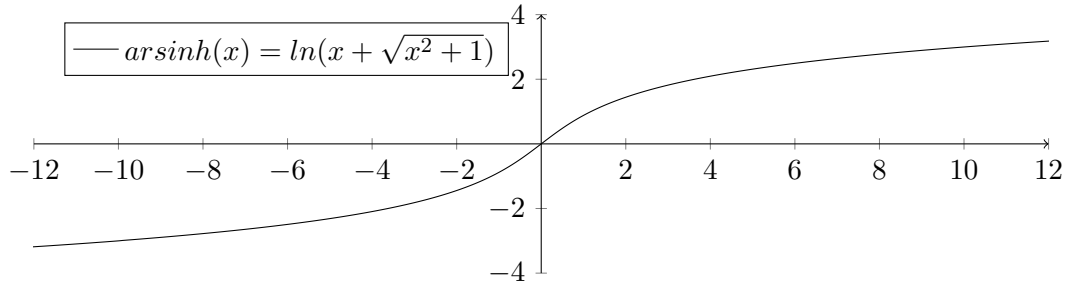
Grundlagenpraktikum: Rechnerarchitektur

Gruppe 233 – Abgabe zu Aufgabe A316
Sommersemester 2023

Ludwig Gröber Julian Pins Daniel Safyan

1 Einleitung

In den Bereichen der speziellen Relativitätstheorie [1], Kosmologie [4, 2] und allgemein in der Berechnung von Wellen [5] gibt es zahlreiche Anwendungen, in denen die Berechnung des *Areasinus Hyperbolicus* ($arsinh(x)$) von großer Bedeutung ist. Er wird verwendet, um Küsten vor zerstörerischen Wellen zu warnen als auch um die Ausdehnung des Universums zu berechnen. In dieser Arbeit werden Methoden zur genauen und effizienten Berechnung dieser vielseitigen Funktion vorgestellt und analysiert.



1.1 Analyse und Spezifikation der Aufgabenstellung

Die Aufgabe A316 ist eben diesen $arsinh(x)$ im C17 Standard von C zu approximieren. Spezifischer verlangt die Aufgabenstellung explizit zwei Implementierungen, die nur auf einfachen arithmetischen Ausdrücken basieren sowie eine Vergleichsimplementierung, die komplexe Instruktionen benutzen darf. Eine dieser Implementierungen soll dabei eine reine Reihe sein, während die andere einen Tabellen-Lookup benutzen soll.

Für unsere Implementierung wurden folgende Annahmen getroffen: Eine **"reinen" Reihenentwicklung** soll hierbei eine Berechnung der Form $\sum_{k=0}^{\pm\infty} a_k x^k$ oder $\sum_{k=0}^{\pm\infty} a_k x^{-k}$ sein und bis auf einzelne Inputs keine weiteren Fallunterscheidungen machen. Hierbei sollte also eine Reihe gewählt werden, deren Konvergenzbereich am ehesten für ein mögliches Anwendungsgebiet geeignet ist. Eine **Reihenentwicklung mit mehreren Reihen** soll hier eine einzige Fallunterscheidung für $x \leq 1$ machen, um die Anzahl an Fallunterscheidungen gering zu halten. Für die **Lookuptabelle** war der Speicherplatz von zentraler Bedeutung, da für die meisten Anwendungen Speichereffizienz wichtig ist. Wir haben uns zudem für eine lineare Interpolation zwischen den gespeicherten Werten entschieden. Als **Vergleichsimplementierung** verwenden wir Funktionen der C math Bibiliothek. Diese wird in erster Linie für die Bewertung der Genauigkeit herangezogen.

Diese vier Implementierungen werden in der Folge erklärt und auf Performanz und Genauigkeit untersucht, verglichen und anhand der Ergebnisse bewertet.

2 Lösungsansatz

Sowohl für die Implementierung mit einer Reihendarstellung, als auch für die Lookuptabelle haben wir uns die Punktsymmetrie der $arsinh$-Funktion zunutze gemacht: $arsinh(-x) = -arsinh(x)$. Die Funktionswerte für negative Eingabewerte werden per Fallunterscheidung mit $-arsinh(|x|)$ berechnet.

2.1 Reihendarstellung

2.1.1 Problematik und Ansätze

Bei der Interpolation durch eine reine Reihe der Form $\sum_{k=0}^{\pm\infty} a_k x^k$ werden Overflows zum Problem. Unabhängig welche Reihe genutzt wird, können die Zwischenergebnisse $x^{\pm k}$ entweder für sehr große Werte oder sehr kleine Werte von x nicht als $double$ dargestellt werden, was zu einem falschen Endergebnis führt. Daher haben Reihendarstellungen stets begrenzte Gültigkeitsbereiche. Es wird versucht, eine Reihenentwicklung zu finden, die für einen sinnvollen Konvergenzbereich möglichst genaue Ergebnisse liefert. Hierzu haben wir unter anderem die Methode der kleinsten Quadrate verwendet, um geeignet Koeffizienten für Reihenglieder mit positiven oder negativen Potenzen zu finden. Die auf diese Weise hergeleitete Reihenentwicklung liefert allerdings für Werte zwischen den verwendeten Datenpunkten zu ungenaue Ergebnisse, da die $arsinh$-Funktion nicht um ein lineares. Wir haben daher stattdessen Reihenentwicklungen für verschiedene Intervalle hergeleitet, die im Folgenden näher erläutert werden sollen. Für bessere Verständlichkeit werden die drei Reihen mit *TaylorArsinh*, *TaylorLn* und *Restreihe* bezeichnet. Die Berechnungen werden im Folgenden näher erläutert:

2.1.2 Reihendarstellung für $|x| \leq 1$

Für $|x| \leq 1$ wird die Taylor-Reihe des $arsinh(x)$ um den Entwicklungspunkt 0 verwendet:

$$\textbf{\textit{TaylorArsinh}} := arsinh(x) = \sum_{k=0}^{\infty} \frac{(2k-1)!!(-x^2)^k}{(2k)!!(2k+1)} = \sum_{k=0}^{\infty} \frac{(-1)^k (2k)! x^{2k+1}}{(2k+1)(2^k * k!)^2}$$

Durch das Auflösen der Doppelfakultäten entsteht eine leichter implementierbare Formel.

2.1.3 Reihendarstellung für $|x| \geq 1$

Die Berechnung für $|x| > 1$ folgt aus folgender Näherung:

$$arsinh(x) = \ln(x + \sqrt{x^2 + 1}) = \ln(2x) + error(x)$$

Der Fehler dieser Näherung kann durch eine dritte Reihe berechnet werden, die vor allem für den Funktionswert von Eingabewerten $|x| < 2^8$ notwendig ist:

$$\textbf{\textit{Restreihe}} := error(x) = \sum_{k=1}^{\infty} \frac{(-1)^{k-1}(2k)!}{2k(2^k \cdot k!)^2} \cdot \frac{1}{x^{2k}}$$

Für die Berechnung von $\ln(2x)$ wird die Taylor-Reihe des natürlichen Logarithmus um den Entwicklungspunkt 1 verwendet. Es ist zu beachten, dass diese Reihe nur im Wertebereich $[0, 2]$ konvergiert.

$$\textbf{\textit{TaylorLn}} := \ln(x) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k}(x-1)^k$$

Um diese beiden Reihen zu einer reinen Reihe zu vereinigen, verwenden wir die Folgende Umformung:

$$arsinh(x) \approx ln(2x) + error(x) = ln(2) - ln(\frac{1}{x}) + error(x)$$

Da $|x| > 1$ gilt, steht im Argument des natürlichen Logarithmus nun ein Wert $\in]0, 1]$, für den die **TaylorLn**-Reihe konvergiert. Um die reine Reihe zu erhalten, können durch Ausmultiplizieren der ersten n Reihenglieder die ersten n Koeffizienten a_k der neuen Reihe bestimmt werden, wodurch sich eine reine Reihendarstellung der Form $\sum_{k=0}^{n} a_k x^{-k}$ ergibt.

In der Vergleichsimplementierung ohne die Beschränkung auf eine reine Reihendarstellung, kann das *double* Datenformat genutzt werden, um die Berechnung signifikant auf Genauigkeit zu optimieren. Die Implementierung mit verschiedenen Reihen verwendet Exponent und Mantisse von x um folgende Umformung zu treffen:

$$x = M \cdot 2^E \;\; mit \;\; M = implizierteMantisse \;\; E = implizierterExponent$$
$$\ln(2x) = \ln(2) + \ln(x) = \ln(2) + \ln(M \cdot 2^E) = \ln(2) + E \cdot \ln(2) + \ln(M)$$

Da $M \in [1, 2[$ liegt, kann nun *TaylorLn* für die Berechnung von $\ln(M)$ verwendet werden.

Die Koeffizienten der einzelnen Reihenglieder zu berechnen ist laufzeitintensiv, da insbesondere die Berechnung der Fakultät viel Zeit kostet. Daher lässt sich die Berechnung optimieren, indem die Koeffizienten aller drei Reihen bereits vor Runtime berechnet werden. Durch Anwendung des Horner-Schemas lässt sich nun die Reihe besonders effizient berechnen.

2.1.4 Umsetzung

In der Implementierung mit einer reinen Reihenentwicklung wird die *TaylorArsinh* verwendet. Diese Entscheidung entsteht primär aus einer Abwägung aus Genauigkeit und Konvergenzbereich. Da im Datentyp double für $|x| < 1$ gleich viele Werte existieren

wie für $|x| \geq 1$, beinhaltet der Konvergenzbereich der Reihen für die beiden Intervalle gleich viele mögliche double-Eingabewerte. Durch Genauigkeitsmessungen hat sich jedoch ergeben, dass die TaylorArsinh(x) für den Großteil der Eingabewerte unter 1 mit wenigen Reihengliedern bereits ein exaktes Ergebnis liefern kann, während das Ergebnis der Näherung $arsinh(x) \approx ln(2x) + error(x) = ln(2) - ln(\frac{1}{x}) + error(x)$ ungenauer wird, je größer $|x|$ ist.

Für die gemischte Reihenentwicklung können wir zudem eine Fallunterscheidung nach den Eingabewerten treffen, um für alle Eingabewerte eine möglichst genaue Näherung zu treffen:

$$\operatorname{arsinh}(x) = \begin{cases} \textit{TaylorArsinh} & \text{falls } |x| < 1 \\ \ln(2x) + error(x) & \text{falls } |x| > 1 \\ x & \text{falls } x \in \{\pm\inf, \pm Nan\} \end{cases}$$

2.1.5 Abwägung Genauigkeit und Performanz

Die Anzahl an benutzten Reihenglieder, entsteht aus einer Abwägung zwischen Performanz und Genauigkeit. Wie viele Reihenglieder für ein möglichst genaues Ergebnis berechnet werden müssen, hängt von dem Eingabewert ab. Da die Reihenglieder aller drei Reihen für Eingabewerte innerhalb des jeweiligen Konvergenzbereichs immer kleiner werden, gilt: Je kleiner das letzte berechnete Reihenglied, desto genauer ist auch das Ergebnis.

Wir betrachten zunächst *TaylorArsinh*: Das k-te Reihenglied hat den Betrag

$$\frac{(2k)!}{(2k+1)(2^k * k!)^2} \cdot x^{2k+1}$$

Aus dieser Formel lässt sich bereits ablesen, dass ein Reihenglied kleiner ist, je kleiner x ist. Die Reihe liefert demnach bereits mit weniger Reihengliedern genaue Ergebnisse, je näher x an 0 liegt. Mithilfe dieser Formel lässt sich feststellen, dass für x = 0.25 alle Reihenglieder ab dem etwa 13. Glied keinen Einfluss mehr auf das Endergebnis haben, da sie aufgrund der Beschränkung der Mantisse auf 52 Bit diese bei der Addition absorbiert werden. Für x = 0.5 müssten 26 Reihenglieder berechnet werden. Für x = 0.99 sind bereits etwa 1500 Reihenglieder nötig. Die Anzahl der zu berechnenden Reihenglieder wird auf 13 festgelegt und damit ein exponentiell ansteigender relativer Fehler ab $|x| > 0.25$ in Kauf genommen.

Für die gemischte Reihe lassen sich ähnliche Berechnungen mit *TaylorLn* und *Restreihe* aufstellen. Um ein sinnvolles Verhältnis von Genauigkeit und Performanz zu erhalten, wird Anzahl der zu berechnenden Reihenglieder für *TaylorArsinh* und *Restreihe* auf 13 festgelegt. Bei *TaylorLn* berechnen wir 27 Reihenglieder, da damit auf dem gesamten Wertebereich ein exaktes Ergebnis erreicht wird.

Es sei an dieser Stelle bereits erwähnt, dass ausgehend von diesen Untersuchungen eine weitere Optimierung umsetzbar wäre, die (beispielsweise mithilfe des Exponenten des Eingabewertes) die Anzahl der nötigen Reihenglieder bestimmt und nur diese berechnet. Durch diese Optimierung wird Overhead durch die Berechnung von für das

Ergebnis irrelevanten Reihengliedern vermieden. Da Fallunterscheidungen grundsätzlich vermieden werden sollten sind, wurde auf diese Optimierung verzichtet.

2.2 Tabellen-Lookup

Die Lookuptabelle speichert einige vorberechnete Werte der $arsinh$-Funktion und interpoliert für die verbleibenden Werte linear zwischen den beiden nächstgelegenen Tabellenwerten. Bei der Implementierung galt es zunächst zwischen Speicherplatz der Tabelle und Genauigkeit des Ergebnisses abzuwägen. Im Rahmen der Ausarbeitung wurde ein Limit von 500KB für die Lookuptabelle festgelegt. Diese Größe ermöglicht uns das Speichern von rund 33000 Werten. Ebenso wurde eine logarithmische Verteilung der Werte festgelegt, da auch die Verteilung aller möglichen Werte im Datentyp $double$ logarithmisch ist. Für alle positiven Eingabewerte wurde für jeden möglichen Exponenten $[-1023, +1023]$ eine feste Anzahl an Werten vorberechnet. Das gesetzte Speicherlimit beschränkt die Genauigkeit auf 16 Werte pro Exponent. Folglich werden 32753 Werte gespeichert. Die Folgen dieser Designentscheidung werden im Kapitel Genauigkeit noch näher betrachtet.

Diese Verteilung ermöglicht zudem ein besonders effizientes Mapping der Eingabewerte auf den zugehörigen Index in der Lookuptabelle durch eine Hashfunktion. Durch eine Bitmaske lassen sich Exponent und die ersten 4 Bits der Mantisse ermitteln. Diese 15 Bits bilden den Index i des nächstkleineren Wertes in der Lookuptabelle. Mit diesem und dem nächsten Tabellenwert gilt $table[i] \leq arsinh(x) < table[i+1]$. Daraufhin wird gemäß

$$arsinh(x) \approx \frac{x - x_i}{x_{i+1} - x_i} \cdot (y_{i+1} - y_i) + y_i$$

linear interpoliert.

3 Genauigkeit

Als Maßstab für die Genauigkeit unserer Ergebnisse wird im Folgenden der relative Fehler der Implementierung bezüglich des tatsächlichen mathematischen Funktionswert verwendet. Die Diagramme wurden erstellt, indem für rund 40000 logarithmisch verteilte Eingabewerte der relative Fehler für alle vier Implementierungen errechnet und mittels matplotlib aus Python dargestellt wurde.

3.1 Reine Reihenentwicklung

Wie deutlich in Abb.2.1 und Abb.2.2 erkennbar ist, liefert die Reihenentwicklung für $|x| < 0.25$ nahezu immer den exakten Funktionswert. Wie in Abb.2.2 erkennbar, steigt der relative Fehler ab etwa $|x| = 0.25$ exponentiell zum Eingabewert an. Dies lässt sich anhand der Konvergenz der Reihenglieder für verschiedene Eingabewerte erklären. Wie in Kapitel 2.1.1 erläutert, ist das Ergebnis genauer, je kleiner das letzte berechnete Reihenglied ist. Wir zeigen zunächst, dass bei 13 Reihengliedern die Grenze für die

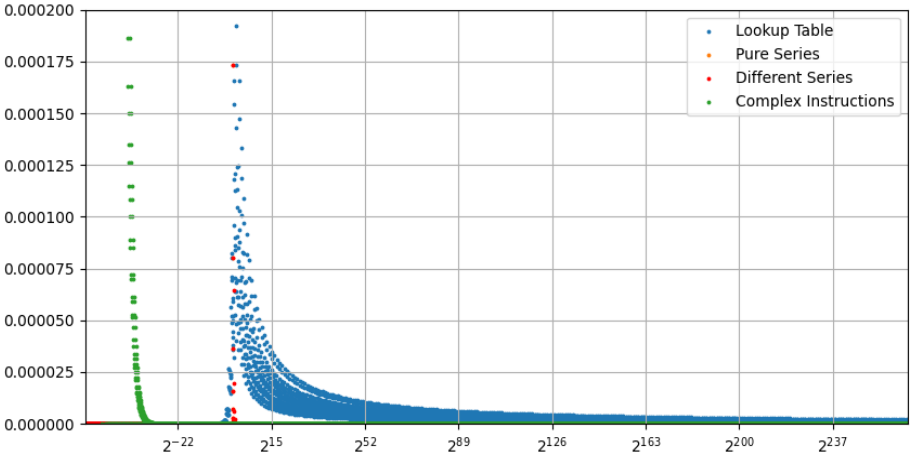


Abbildung 1

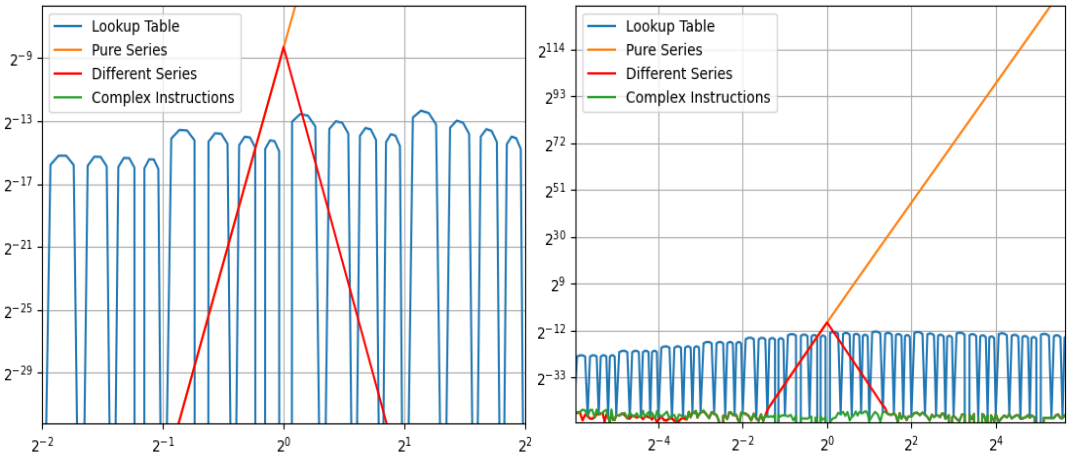


Abbildung 2

Entstehung von Ungenauigkeiten bei etwa 0.25 liegt. Hierzu berechnen wir jeweils das letzte Reihenglied für x=0.25:

$$\textit{TaylorArsinh}: \frac{(2\cdot 13)!0.25^{2\cdot 13+1}}{(2\cdot 13+1)(2^1 3\cdot 13!)^2} \approx 2^{-59}$$

Da des letzte Reihenglied noch unter $2^{-52}\cdot asinh(0.25)$ liegt und die die Mantisse von doubles nur 52 bit beinhaltet, hat das Ergebnis noch maximale double Genauigkeit. Für größer werdende x-Werte, wird jedoch das letzte Reihenglied immer größer, wodurch der relative Fehler exponentiell ansteigt. Daher hat die reine Reihenentwicklung innerhalb ihres Konvergenzbereichs $|x| \leq 1$ ihren maximalen relativen Fehler von circa $2^{-8} \approx 0.39\%$ am Eingabewert 1. Berechnen wir nun das letzte (13.) berechnete Reihenglied für den Eingabewert 1 so erhalten wir:

$$\textit{TaylorArsinh}: \frac{(2\cdot 13)!1^{2\cdot 13+1}}{(2\cdot 13+1)(2^1 3\cdot 13!)^2} \approx 0.00574 \approx 2^{-8}$$

Da das letzte Reihenglied die Größenordnung 2^{-8} hat, liegt auch der relative Fehler etwa in dieser Größenordnung. Innerhalb ihres Konvergenzbereiches hat die reine

Reihenentwicklung also einen maximalen Fehler von 2^{-8} Außerhalb des Konvergenzbereichs $|x| \leq 1$ konvergieren die einzelnen Reihenglieder nicht gegen 0, wodurch die Reihe nicht mehr anwendbar ist. Da das letzte Reihenglied weiter exponentiell ansteigt, steigt auch der relative Fehler exponentiell an, wie auch in Abbildung 2.1 deutlich erkennbar.

3.2 Reihenentwicklung mit mehreren Reihen

In Abb.1 und Abb.2.2 lässt sich ähnlich zur reinen Reihenentwicklung erkennen, dass die Reihenentwicklung außerhalb der Intervalle $0.25 < |x| < 4$ nahezu immer den exakten Funktionswert liefert. Es gibt einige Ausnahmen mit einem verschwindend geringen relativen Fehler $\leq 2^{-50}$, welche sich durch floating-point-Rundungsfehler bei der Umrechnung der Polynomkoeffizienten erklären lassen. Da der Fall $1 < x < 0.25$ exakt äquivalent zu der reinen Reihe ist, betrachten wir hier die Ungenauigkeiten im Intervall $1 < x < 4$. Dazu berechnen wir das letzte Reihenglied der *Restreihe* für $|x| = 4$.

$$\textit{Restreihe} : \frac{(2 \cdot 13)!}{2 \cdot 13(2^{13} \cdot 13!)^2} \cdot \frac{1}{4^{2 \cdot 13}} \approx 2^{-59}$$

Da des letzte Reihenglied unter 2^{-52} liegt, hat das Ergebnis noch maximale double Genauigkeit. Ähnlich wie bei der reinen Reihe, wächst das letzte Reihenglied für x-Werte, die näher an 1 liegen exponentiell, was zu einem exponentiellen Anstieg des relativen Fehler führt. Für das letzte Reihenglied für den Eingabewert 1 erhalten wir:

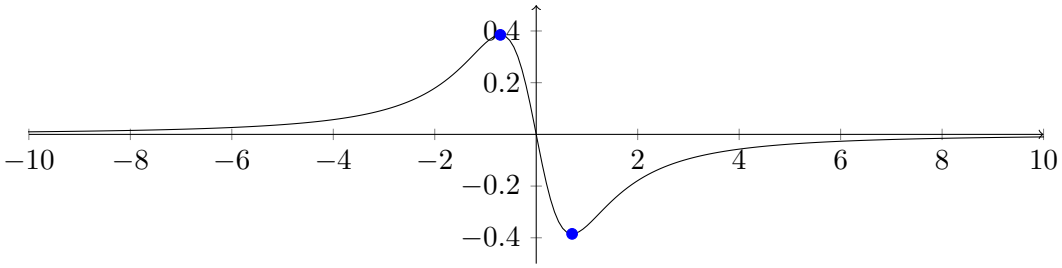
$$\textit{Restreihe} : \frac{(2 \cdot 13)!}{2 \cdot 13(2^{13} \cdot 13!)^2} \cdot \frac{1}{1^{2 \cdot 13}} \approx 0.00596 \approx 2^{-8}$$

Da das letzte Reihenglied die Größenordnung 2^{-8} hat, stimmt der relative Fehler hier mit der reinen Reihe auch von der anderen Seite aus überein.

3.3 Lookuptabelle

Die Genauigkeit des Ergebnisses bei der Verwendung der Lookuptabelle hängt stark von den Eingabewerten ab. Ist der Eingabewert exakt einer der vorgespeicherten Werte in der Tabelle, so liefert die Funktion logischerweise das exakte Ergebnis. Am ungenauesten ist die Funktion für Eingabewerte, die genau zwischen zwei gespeicherten Tabellenwerten liegen. Das lässt sich deutlich an der großen vertikalen Streuung der Fehlerwerte erkennen.

Im ersten Diagramm ist zudem deutlich erkennbar, dass die Funktion wie auch die Reihenentwicklung deutlich ungenauere Ergebnisse für Eingabewerte nahe an 1 liefert. Dies lässt sich mithilfe des Krümmungsverhaltens der Funktion begründen. Der folgende Graph zeigt die zweite Ableitung des $arsinh''(x) = \frac{-x}{x^2 \cdot \sqrt{x^2+1} + \sqrt{x^2+1}}$



Wie man sieht, hat der $arsinh(x)$ eine maximale Krümmung bei circa ± 0.707. Dementsprechend wird auch eine lineare Interpolation im Bereich dieser Stellen eine maximale Ungenauigkeit aufweisen.

3.4 Implementierung mit komplexen Instruktionen

Wie in Abb.1 zu sehen ist, liefert die Implementierung mit komplexen mathematischen Instruktionen der C-Math Bibliothek für den Großteil des Wertebereichs den genauest möglichen Funktionswert für den Datentyp double. Die einzige Ausnahme bilden sehr kleine x-Werte. Dies lässt sich mit der Absorption beziehungsweise Auslöschung von Bits begründen, welche bei der Addition von doubles unterschiedlicher Größenordnung unvermeidbar auftritt. Betrachten wir nun zunächst den Term $\sqrt{x^2+1}$ der allgemeinen Gleichung für den $arsinh(x)$. Ab $|x| < 2^{-26}$ gilt stets $x^2 < 2^{-52}$. x^2 wird damit bei der Addition mit 1 vollständig absorbiert und $\sqrt{x^2+1}$ evaluiert zu 1. Hierdurch entsteht ein für kleinere x immer größerer relativer Fehler. Ab $|x| < 2^{-52}$ wird zudem x bei Addition auf 1 immer absorbiert, wodurch $\ln(x+\sqrt{x^2+1})$ automatisch zu 0 evaluiert. Für $|x| < 2^{-52}$ ist die Implementierung mit komplexen mathematischen Instruktionen demnach kaum noch anwendbar, da das Ergebnis (mit Ausnahme von 0) immer einen relativen Fehler von 100% haben wird.

4 Performanzanalyse

Die Performanz der Implementierungen wird anhand der Laufzeit gemessen. Diese wird im Folgenden mit der Libaray $time.h$ gemessen.

4.1 Methodik und Annahmen 0,5Seite

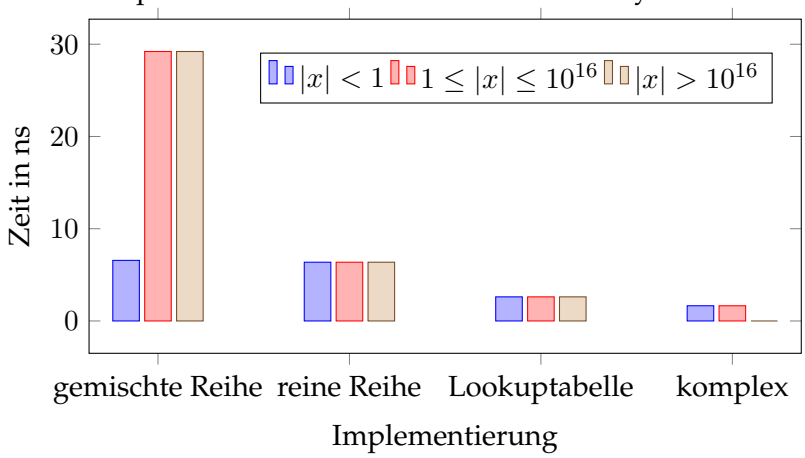
Gemessen wurde auf einem System mit Intel i5-10210U Prozessor, 1.60 GHz, 8 GiB Arbeitsspeicher, Ubuntu 22.04.2 LTS, 64 Bit, 5.19.0-46-generic Linux-Kernel. Kompiliert wurde mit GCC 11.3.0 mit der Option -O3. Um eine optimale Vergleichbarkeit der Ergebnisse zu garantieren, wurden alle nicht für das Betriebssystem nötigen Prozesse beendet.

Da die Reihenentwicklung zwei sehr verschiedene Berechnungen für $|x| > 1$ und $|x| \geq 1$ durchführt, wurden bei der Laufzeitmessung diese beiden Fälle unterschieden. Ansonsten hängt die Laufzeit aller drei Implementierungen nur bedingt von der Größenordnung der Eingabewerte ab, daher wurde die Laufzeitmessung für die beiden

Intervalle jeweils mit repräsentativen Werten verschiedener Größenordnungen durchgeführt. Die Methoden wurden mit 10 verschiedenen Werten jeweils 100.000.000 mal aufgerufen, um dann das arithmetische Mittel der Zeit für einen Funktionsaufruf zu ermitteln.

4.1.1 Zeitmessung der Implementierungen

Der folgende Graph zeigt die durchschnittliche Laufzeit eines Funktionsaufrufes der drei Implementierungen mit Reihenentwicklung, Tabellen-Lookup und der Implementierung mithilfe komplexer Instruktionen der C-math-library in Nanosekunden:



4.2 Bewertung, Einordnung und Erklärung der Ergebnisse

Wie in den Abbildungen zu sehen ist, gibt es starke Performanzunterschiede zwischen den verschiedenen Implementierungen. Wir betrachten zunächst die Implementierung mit komplexen Instruktionen. Diese ist hierbei im Vergleich zu den anderen Implementierungen deutlich schneller. Es offenbart sich auch innerhalb der Funktion eine signifikante Diskrepanz zwischen den Eingabewerten unterhalb von 10^{16} und jenen oberhalb von 10^{16}. Ab dem gewählten Grenzwert von 10^{16} wird die Näherung $arsinh(x) = ln(2x)$ für die Berechnung verwendet. Diese Berechnung wird bei Optimierungsstufe O3 so stark optimiert, dass sogar bei 10^{12} Iterationen eine kaum noch messbare Laufzeit aufweist. Für den Großteil der Werte deren Betrag kleiner ist als 10^{16} sind allerdings neben der Logarithmusberechnung zudem Multiplikation, Addition und Wurzelberechnung für die exakte Berechnung erforderlich, wodurch die Funktion in diesem Intervall eine durchschnittliche Laufzeit von etwa 1.648 ns hat.

Die Lookuptabelle hat für alle reellen Eingabewerte etwa die gleiche Laufzeit von 2.613 ns, da die Berechnung unabhängig von der Höhe des Eingabewerts ist. Durch die effiziente Hashfunktion mittels Bitmaske, die für das Mapping der Eingabewerte auf den jeweiligen Tabellenwert verwendet wird, ist diese Implementierung nur etwa 1.5 mal so langsam wie die worst-case Laufzeit der Implementierung mit komplexen Instruktionen. Neben der Hashfunktion werden nur wenige einfache Instruktionen für die lineare

Interpolation benötigt, was ebenfalls zu der hohen Performanz dieser Implementierung beiträgt.

Im Vergleich zu der Lookuptabelle, braucht die reine Reihenentwicklung für die von uns beschlossene Anzahl berechneter Reihenglieder mit 6.369 ns etwa 2.5 mal so lange wie die Lookuptabelle. Dies ist damit zu begründen, dass die Reihenentwicklung deutlich mehr Multiplikationen und Summen über mehrere Reihenglieder verwendet, wodurch die Berechnung länger dauert.

Die Reihendarstellung mit verschiedenen Reihen benötigt für $|x| < 1$ etwa gleich viel Zeit wie die reine Reihenentwicklung, da derselbe Algorithmus wiederverwendet wird. Für die Approximation der Eingabewerte $|x| \geq 1$ werden allerdings zwei verschiedene Reihen (TaylorLn und Restreihe) sowie einige komplexere Bitoperationen verwendet. Hierdurch benötigt diese Implementierung für $|x| \geq 1$ rund 29.212 ns und damit etwa 12 mal so lange wie die Lookuptabelle. Um eine vergleichbare Genauigkeit mit den Eingabewerten $|x| < 1$ zu erhalten, sind diese Operationen allerdings notwendig.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Der Trade-off zwischen Performance, Genauigkeit und Speicherverbrauch ist für die beiden Implementierungen mit einem Tabellenlookup beziehungsweise einer Reihenentwicklung sehr unterschiedlich. Der Tabellenlookup ist schnell, benötigt allerdings mehr Speicherplatz, um genauere Werte zu erzielen. In der Tabellenlookup Implementierung mit insgesamt etwa 30000 zu speichernden Werten in einer Lookup-Tabelle, hat die Berechnung einen maximalen relativen Fehler von 0.02%.

Die reine Reihenentwicklung auf der anderen Seite liefert mit 13 gespeicherten Werten für etwa 49.4% aller doubles im Intervall $[-1, 1]$ den exakten Funktionswert, wird aber für Eingabewerte nahe an Eins ungenau. Da jedoch der interessanteste Bereich über 1 hinausgeht, lässt sich schließen, dass die Nutzung einer reinen Reihe, egal ob $|x| \geq 1$ oder $|x| \leq 1$ verwendet, für die gegebene Problemstellung nicht geeignet ist. Die gemischte Reihe liefert für 99,8% aller doubles den exakten Funktionswert. Jedoch benötigt die Reihenentwicklung verglichen mit dem Lookup für $|x| > 1$ die bis zu 11-fache Zeit für die Berechnung. Andererseits verbraucht diese über 600-mal weniger Speicherplatz für eine genaue Berechnung.

5.2 Anwendung

Welche Implementierung verwendet werden sollte, hängt also von den genauen Rahmenbedingungen ab. Der Areasinus Hyperbolicus findet in erster Linie Anwendung in verschiedensten Bereichen der Physik und Ingenierswissenschaften [**?**]. Für die Verwendung in Echtzeitsystemen eignet sich somit die Lookuptabelle besser, da Messungen in der stets kleine Fehler aufweisen und die Lookuptabelle für viele Werte aufgrund der höheren Performanz schneller Ergebnisse liefert. Steht jedoch Genauigkeit im Vordergrund,

so eignet sich die Reihenentwicklung mit einer geeigneten Menge an Reihengliedern deutlich besser.

5.3 Ausblick

In unserer Implementierung des $arsinh(x)$ haben wir uns auf eine reine Reihenentwicklung und eine einfache Lookup-Tabelle mit linearer Näherung beschränkt. Unter weniger strengen Rahmenbedingungen ließe sich die Implementierung allerdings sowohl in Bezug auf Laufzeit, als auch Genauigkeit noch signifikant optimieren.

Ein möglicher weiterführender Ansatz ist die Verwendung von Splines in der Lookup-Tabelle: Statt der bisher linearen Näherung zwischen zwei Messpunkten wäre es möglich den $arsinh(x)$ in diesem Intervall durch ein Polynom anzunähern. Hierzu werden in der Regel Polynome vom Grad drei verwendet. [**?**] Dadurch verbessert sich die Genauigkeit, insbesondere für x-Werte die genau zwischen zwei Werten in der Lookup-Tabelle liegen deutlich, während sich die Laufzeit nur minimal erhöht. Dafür steigt allerdings der benötigte Speicherplatz, da nun für jedes Intervall ein Polynom gespeichert werden muss.

Eine mögliche weitere Verbesserung des Lookup-Tables ist eine effizientere Verteilung der Tabellenwerte. Durch lineare Interpolation entsteht eine höhere Ungenauigkeit, je stärker die Krümmung der arsinh Funktion an dieser Stelle ist. Würde also an stark gekrümmten Stellen eine höhere Dichte an Messpunkten verwendet und eine geeignete Mappingfunktion erstellt, würde die Lookuptabelle bei der gleichen Anzahl an vorgespeicherten Werten deutlich genauere Ergebnisse liefern.

Ein weiterer Ansatz wäre, eine Kombination aus Reihenentwicklung und Lookup-Tabelle zu verwenden: Wie bereits beobachtet, liefert die Berechnung mit einer Reihenentwicklung für x-Werte, die nicht nahe an Eins liegen, besonders genaue Ergebnisse. Für $|x|<0.125$ brauchen werden beispielsweise 13 Reihenglieder benötigt, für $|x|>8$ werden 13 Reihenglieder der Restreihe, sowie bis zu 33 Reihenglieder der Taylorreihe für ein exaktes Ergebnis benötigt. Außerhalb dieser Intervallen bietet es sich an, eine reine Reihenentwicklung zu verwenden. In dem Intervall $0.125<=|x|<=8$ werden allerdings mehr Reihenglieder benötigt, wodurch es zu starken Performanzeinbußen kommt. Es wäre demnach sinnvoll in diesem Intervall eine Lookup-Tabelle zu verwenden. Dieser Ansatz würde ein besonders genaues Ergebnisses erzielen.

Die Wahl des Datentyps bietet eine weitere einfache Möglichkeitzur Verkleinerung des Speicherverbrauchs. Gemäß IEEE 754 [3] könnte auch die einfache Genauigkeit der Fließkommaarithmetik gewählt werden, wodurch sich der Speicherverbrauch halbieren würde.

Literatur

[1]

[2] A. G. Walker. On Milne's theory of world-structure. Proc. Lond. Math. Soc., Band 42, 1936. S. 90–127.

[3] American National Standards Institute. IEEE Standard for Binary Floating-Point Arithmetic. The Institute of Electrical and Electronics Engineers, Inc, 1985. S. 4.

[4] H. P. Robertson. Kinematics and world structure. Astrophysical Journal, Band 82, 1935. S. 284–301.

[5] Patrick Holmes. Coastal Processes: Waves. Organisation of American States, 2001. http://www.oas.org/cdcm$_{t}rain/courses/course21/chap_{0}5.pdf, visited2023 - 07 - 12.$