Untersuchung und Bewertung von Posit-Arithmetik für Mixed-Precision-Runge-Kutta-Verfahren

David Schuberth

Bayreuth Reports on Parallel and Distributed Systems

No. 18, 2025

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

UNIVERSITÄT BAYREUTH

Untersuchung und Bewertung von Posit-Arithmetik für Mixed-Precision-Runge-Kutta-Verfahren

Analysis and evaluation of Posit arithmetic for mixed-precision Runge-Kutta methods

Bachelorarbeit

Universität Bayreuth
Lehrstuhl für Angewandte Informatik 2

Prüfer: **Prof. Dr. Matthias Korch**
Prof. Dr. Thomas Rauber

David Schuberth

Bayreuth den 11. Juli 2025

Inhaltsverzeichnis

Zusammenfassung

In dieser Arbeit soll die Nutzbarkeit des alternativen Posit-Zahlenformates im Vergleich zu den herkömmlichen IEEE-574 Gleitkommazahlen bei der Verwendung in Runge-Kutta-Verfahren mit gemischter Genauigkeit untersucht werden. Diese sogenannten "Mixed-Precision" Verfahren verwenden für den Löser zwei verschiedene Zahlenformate, eine genaueres und eines mit reduzierter Genauigkeit, wobei bei herkömmlichen Verfahren dieser Art von IEEE Float Formaten unterschiedlicher Bitlängen Gebrauch gemacht wird. Ziel dieser Arbeit ist es nun, durch Untersuchung experimenteller Ergebnisse auszuwerten, ob sich diese Datentypen durch Posits austauschen lassen, und diese Verfahren dann erstens dasselbe Konvergenzverhalten erreichen können und zweitens mit dem Fehlerverhalten der IEEE Floats mithalten können oder dieses gar übertreffen können. Zu diesem Zweck wurde ein Framework entwickelt, dass per Softwaresimulation Posit-Arithmetik emuliert, und damit Lösungsverfahren gemischter Genauigkeit nach den Schemata aus Publikationen von [1] und [2] konstruiert. In der Evaluation konnten für diese mit Posits dieselben aus der Theorie zu erwartenden Fehlerverhalten reproduziert werden. Im direkten Vergleich zu ihren gleich großen Float-Äquivalenten konnten die Posits stets die gleiche Größenordnung von Fehler erreichen, und für einige Szenarien sogar diesen Fehler verringern.

Abstract

This thesis seeks to evaluate the usability of the alternative Posit number format as a replacement of IEEE-574 floating point numbers in mixed precision Runge-Kutta methods. Whereas the conventional mixed precision solvers use differently sized IEEE Floats as the formats for the reduced precision and work precision parts of their calculations, this thesis aims to replace these datatypes with equivalent Posit formats in order to evaluate the impact of this alternative arithmetic on the results of the ODE solvers. To prove their usabiliy, these solvers are compared against their Float-using counterparts in regards to their convergence and error behaivour. In order to support these evaluations, a framework that provides software simulation of Posit arithmetic was implemented, which was then used to implement the two types of mixed precision Runge-Kutta solvers as proposed by [1] and [2]. Evaluating their results showed that the methods using posits could replicate the same convergency as floats and fullfill expectations from theory. Furthermore, comparing them to Floats of the same size, Posists could always reach the same order of magnitude of error, and in some cases even manage to outperform the Float results.

1 Einleitung

Im Jahre des Verfassens dieser Arbeit feiert der IEEE-754 Floating-Point-Standard, erschienen 1985, seinen vierzigsten Geburtstag und bestimmt damit seit vier Jahrzehnten maßgebend die Art und Weise wie Computer mit reellen Zahlen rechnen. Dabei ist er aber abgesehen von kleineren Erweiterungen in seinen Grundprinzipien seitdem unverändert geblieben. Da die IEEE-754 Floats mit ihren 40 Jahren aus einer anderen Zeit stammen und mit den Anforderungen für ältere Hardware entworfen wurden, die heute nicht mehr zeitgemäß sind, erscheinen immer wieder alternative Zahlenformate, die die Computerrepräsentation von Fließkommazahlen zu modernisieren versuchen. Das bekannteste davon sind heute die Posits, welche 2017 von Gustafson [3] vorgeschlagen wurden. Neben einer flexibleren Darstellung, die ermöglichen soll größere Genauigkeiten und Zahlenbereiche als Floats erreichen zu können, adressieren sie zusätzliche Unstimmigkeiten der IEEE-754 Fließkommazahlen wie der doppelten Kodierung der Null.

Ein weiterer Entwicklungstrend, der sich im Feld des numerischen Rechnens beobachten lässt, ist die häufigere Verwendung von verringerten Genauigkeiten, um mit weniger Speichertransfers die Speichergebundenheit von Programmen zu überwinden [4]. Diese Arbeit versucht diese beiden Ansätze zu fusionieren, also Verfahren von gemischter Genauigkeit (Mixed-Precision) mit dem modernen Posit-Zahlenformat anstatt von IEEE Floats durchzuführen. Speziell wurden hier Runge-Kutta-Verfahren zum Lösen gewöhnlicher Differentialgleichungen als das zu untersuchende numerische Verfahren gewählt. Die hier verwendeten Löser entstammen den Arbeiten von Grant [1], der implizite Verfahren betrachtet, sowie Croci [2] der ein Schema für explizite Runge-Kutta-Chebyshev-Verfahren mit gemischter Genauigkeit vorstellt. Für ersteres existieren bereits mehrere empirische Auswertungen zu dessen Stabilitätsbereich in [5] sowie zu dessen Fehlerverhalten und Performance in [6], [7], welches hier versucht werden soll für Posits zu reproduzieren.

Die Zielsetzung ist dabei die Untersuchung, ob Posits die IEEE Fließkommazahlen in den Mixed-Precision-Runge-Kutta-Verfahren austauschen können, ohne das Verfahren in seiner Konvergenz zu stören, und ob sie sich auf die Genauigkeit des Endergebnisses positiv auswirken können. Das Erreichen der gleichen Ergebnisgenauigkeit durch Verwendung von weniger Bits könnte einen großen Vorteil bedeuten, da es speichergebundene Algorithmen beschleunigen könnte. Mangels Hardwareunterstützung für Posits können die Auswirkungen auf die Performance hier nicht akkurat bemessen werden, da die Posits per software simuliert werden und durch diese langsame Simulation die Speichergebundenheit der Verfahren verloren geht, weswegen der Fokus dieser Arbeit deswegen umso mehr auf einer Fehleranalyse im Vergleich mit den Floats liegt.

Zur Untersuchung von Posits in Runge-Kutta-Verfahren gemischter Genauigkeit exisitert dabei heute noch keinen Vergleichsliteratur. Einzig für gewöhnliche Runge-Kutta-Verfahren erschien im späten Verlauf der Ausarbeitung eine Evaluation von Murillo [8], in der das klassische RK4 Verfahren für Posits erfolgreich mit geringerem Fehler als für Float-Typen derselben Bitlänge getestet wird. Aktuelle praktische Untersuchung der Posits fokussieren sich dagegen

eher auf deren Nutzen in neuronalen Netzen [8], [9] oder andere numerische Verfahren [10]. Diese Arbeit selbst entstand als Fortführung vorheriger Abschlussarbeiten am Lehrstuhl, die sich mit Mixed-Precision-Runge-Kutta-Verfahren [11], [12] sowie Posits [13] gesondert befasst haben.

Diese Arbeit ist dabei in drei Teile untergliedert: Im ersten wird die Theorie hinter den beiden betrachteten Zahlenformaten, IEEE Float und Posit, erläutert, sowie die Grundlagen der Runge-Kutta Verfahren und die konkreten Schemata der Mixed-Precision Verfahren aufgezeigt. Der zweite Abschnitt befasst sich mit der Implementierung der Theorie und der umgebenden Systeme, die zur Auswertung der Differentialgleichungslöser benötigt werden. Besagte Auswertung findet im letzten Abschnitt statt, in dem die Löser mit Posts für verschieden Formate, Szenarien und Anfangswertprobleme getestet und mit den Lösern mit Floats verglichen werden.

2 Theorie

2.1 Zahlenformate

Gleitkommazahlen sind die maßgebende Grundlage zur Darstellung von reellen Zahlen in modernen Computersystemen. Zusammengesetzt aus einer Mantisse m und einem Exponenten e, vermögen sie einen größeren Zahlenbereich der Reellen Zahlen abzubilden als Festkommazahlen, da mit der beliebigen Wahl des Exponenten das Komma verschoben werden kann bzw. gleiten kann, was sie zu ihrem Namen bringt. Diese genannten Komponenten stellen reelle Zahlen als $m \cdot \beta^e$ dar [14], wobei β die Basis ist, die im Kontext von Computern aufgrund ihrer Digitallogik als $\beta = 2$ gewählt ist. Zur Kodierung dieser zwei Komponenten gibt es nun verschiedene Varianten auf Computern.

2.1.1 IEEE 754

Zu Beginn der Computerentwicklung waren viele verschiedene Versionen von Gleitkommazahlen in Verwendung, da jeder Hersteller eigene Größen und Kodierungen für Vorzeichen, Mantisse und Exponent verwendete. Mit der Einführung des IEEE-754 Standards im Jahre 1985 wurde ein einheitliches Format zur Speicherung, Kodierung und Rundung von Gleitkommazahlen definiert, das bis heute in fast jedem geläufigen Rechnersystem Verwendung findet. Der Standard gibt dabei genaue Vorschriften zur Kodierung vor: Die Mantisse m wird aufgeteilt in ihren Betrag $|m|$ und ein Vorzeichen, das durch ein zusätzliches Bit S als $(-1)^S$ kodiert wird. Der Betrag der Mantisse $|m|$ muss ferner normiert sein, also immer mit $1, \ldots$ anfangen, jegliche Verschiebung des Kommas wird immer durch den Exponenten dargestellt. Diese führende 1 ist in der Darstellung implizit, es wird nur der darauffolgende Nachkommateil in M binärkodiert. Der Exponent e wird vor dem Speichern auf ein Bias e_B addiert, sodass $E = e + e_B$ in der gespeicherten Darstellung immer eine positive Zahl ist. Dies dient vor allem dazu, Sortierung und Vergleichsoperationen etwas zu vereinfachen.[14], [15]

Diese drei Bestandteile S, E, M werden wie in Abb. 2.1 gezeigt in einer Speicherzelle abgelegt und stellen eine kodierte Gleitkommazahl x dar als:

$$x = (-1)^S \cdot 2^{(E-e_B)} \cdot (1 + M)$$

Die Größe des Exponentenbias oder der Bitlängen der W_E, W_M von Exponent und Mantisse werden festgelegt vom Format der IEEE-754 Gleitkommazahl.

Formate Der Standard definiert 4 Größen an binären Gleitkommazahlen für Speichergrößen W von 16, 32, 64 und 128 Bit. Diese spezifizieren die jeweiligen Exponenten- und Mantissenlängen W_E, W_M sowie das Exponentenbias e_B wie in Tabelle 2.1 gelistet.

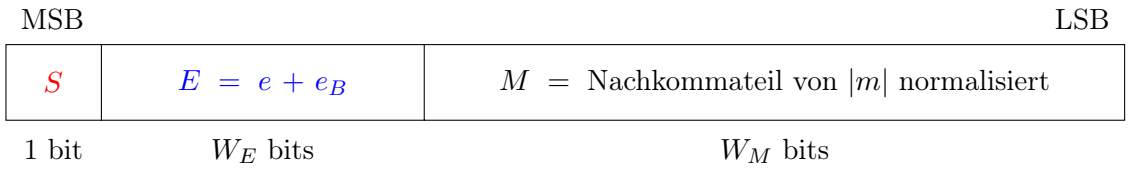


Abbildung 2.1: Allgemeine Kodierung einer IEEE-754 Gleitkommazahl.

Tabelle 2.1: Verschiedene Formate von IEEE 756 Fließkommazahlen. [14, Tab. 3.4]

Gesamtlänge W	16 bit	32 bit	64 bit	128 bit
Eigenname	half	single / float	double	quad
Mantissenlänge W_M	10	23	52	112
Exponentenlänge W_E	5	8	11	15
Exponentenbias e_B	15	127	1023	16383

Tabelle 2.2: Sonderzustände der IEEE Fließkommazahlen [14, Tab. 3.6]. In den Spalte "Exponent" und "Mantisse" stehen hier die Werte die tatsächlich in der Speicherzelle stehen würde, also bereits nach Biasaddition bzw. Normalisierung

Beschreibung	Exponent	Mantisse	Anmerkung
$0\ldots0$	$0\ldots0$	$0\ldots0$	Vorzeichen beliebig
$\pm\infty$	$1\ldots1$	$0\ldots0$	
NaN (Not a Number)	$1\ldots1$	$\neq 0\ldots0$	Vorzeichen beliebig
$\pm$Denormalisierte Zahl	$0\ldots0$	$\neq 0\ldots0$	

Tabelle 2.3: IEEE Float-Formate und deren darstellbarer Zahlenbereich.

Format	Minimum $float_{\min}$	Maximum $float_{\max}$
16bit half	$2^{-14-10} \approx 5,69 \cdot 10^{-8}$	$(2-2^{10}) \cdot 2^{15} = 65504$
32bit float	$2^{-126-23} \approx 1,401 \cdot 10^{-45}$	$(2-2^{-23}) \cdot 2^{127} \approx 3,403 \cdot 10^{38}$
64bit double	$2^{-1022-52} \approx 4,941 \cdot 10^{-324}$	$(2-2^{52}) \cdot 2^{1023} \approx 1,798 \cdot 10^{308}$
128bit quad	$2^{-16382-112} \approx 6,68 \cdot 10^{-4966}$	$(2-2^{-112}) \cdot 2^{16386} \approx 1,19 \cdot 10^{4932}$

Sonderfälle Unter allen Bitmustern die eine IEEE-Fließkommazahl darstellen können, gibt es für jedes Format einige vorreservierte Sonderfälle, die in Tabelle 2.2 gelistet sind.

Da die führende 1 der Mantisse immer implizit gegeben ist, ist zur Darstellung der 0 ein spezielles Bitmuster vorgesehen, in dem Exponent und Mantisse nur aus Nullen bestehen. Das Vorzeichen ist dabei beliebig, was dafür sorgt, dass es zwei Darstellungen von 0 gibt, eine positive (+0) und eine negative (−0).

Bei Überläufen von der Null weg wird das Ergebnis zu $\pm\infty$ während bei illegalen Operationen wie der Division $\frac{0}{0}$ das Ergebnis auf NaN gesetzt wird. Wie in Tab. 2.2 gelistet, gilt jede Bitkombination mit einem Exponent aus nur Einsen und einer Mantisse $M \neq 0$ als NaN, was bedeutet, dass es $2^{W_M+1} - 2$ Bitkombinationen zur Kodierung von NaN gibt. Aus dem Standard [15] geht hervor, dass im Falle von gescheiterten Operationen, die NaN erzeugen, in den restlichen Bits Diagnoseinformationen kodiert werden sollen. Besonders für kleiner Float-Formate ist der Prozentsatz an mit NaN belegten Bitmustern relativ hoch, bei Half-Floats mit 16 Bit Speichergröße liegt dieser bei $\frac{2^{11}-2}{2^{16}} = \frac{2046}{65536} \approx 3.12\%$.

Im Bereich der Unterläufe zur Null hin werden noch denormalisierte Zahlen verwendet, um den Zahlenbereich zu erweitern. Besteht der Exponent nur aus Nullen, nimmt also seinen kleinstmöglichen Wert $e_{\min} = -e_B = -2^{(W_E-1)} + 1$ an, so gelten in diesem Fall die Mantissen als nicht mehr normalisiert. Die implizite führende Eins wird also bei der Interpretation nicht mehr angefügt, die dargestellte Zahl wird also zu:

$$x = (-1)^S \cdot 2^{e_{\min}} \cdot (0 + M)$$

Dieses Vorgehen erlaubt es den Zahlenbereich nach unten noch etwas zu vergrößern, verkompliziert allerdings die Hardware die zur Dekodierung der Zahl benötigt wird massiv, was sie zu einer der wohl kontroversesten Eigenschaften der IEEE-754 Fließkommazahlen macht [14].

Darstellbarer Zahlenbereich: Die betragsmäßig minimale darstellbare IEEE-754 Gleitkommazahl ist eine denormalisierte Zahl mit Exponent $e_{\min} = -2^{(W_E-1)} + 1$ und minimaler Mantisse bei der nur das LSB eine Eins ist, also $m = 2^{-W_M}$, was zu $float_{min} = 2^{e_{\min}} \cdot 2^{-W_M}$ führt. Der betragsmäßig maximal darstellbare Float besteht aus dem maximalen Exponenten, einem Bitstring aus nur Einsen, der auf eine Null endet (da nur Einsen für NaN reserviert sind) $e_{\max} = 2^{W_E-1} - 1$, und der maximalen Mantisse, bestehend nur aus Einsen, $m = \sum_{i=0}^{-W_M} 2^i = 2 - 2^{-W_M}$ und ergibt sich damit zu $float_{\max} = 2^{e_{\max}} \cdot (2 - 2^{-W_M})$. Tabelle 2.3 führt die Maximal- und Minimalwerte der IEEE-754 Float Formate auf.[14], [15]

Weitere IEEE-754 orientierte Formate: Prinzipiell lassen sich nach dem Vorbild des IEEE-754 Standard auch noch weitere Zahlenformate definieren, indem man die Mechanismen der

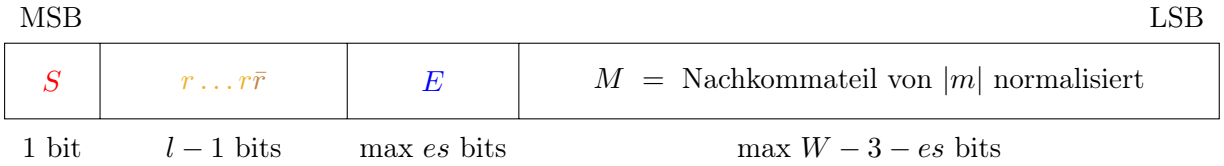


Abbildung 2.2: Allgemeine Kodierung eines Posits

Kodierung übernimmt und schlichtweg andere Gesamtlängen W und Größen für Mantisse und Exponent W_M, W_E definiert.

Besonders mit der wachsender Relevanz von Machine Learning stieg auch das Interesse an kompakterer Zahlendarstellung, was alternative Formate wie Brain Float oder Tensorfloat schuf, aber auch 16bit IEEE Half Floats größere Beliebtheit bescherte.[4]

2.1.2 Posits

Der Posit ist ein alternatives Zahlenformat, das erstmals 2017 von John L. Gustafson vorgestellt wurde [3]. Posits sind die zweite Weiterentwicklung des von Gustafson 2015 vorgestellten *unum* Formates ("universal numbers"), welches ursprünglich eine Erweiterung der IEEE-754 Fließkommazahlen auf variable Mantissen- und Exponenten mit zusätzlicher Intervallarithmetik war [16]. Das Ziel der Weiterentwicklung war dabei ein hardwarefreundlicheres Design der unums, da die Vorgängerversion in der Praxis ausschließlich über Lookup-Tabellen funktionieren konnte.

Im Jahr 2022 wurden Posits schlussendlich in einem Standard [17] fundamentiert, der neben weiterer Literatur von Gustafson [3], [16] als Hauptquelle für den nachfolgenden Abschnitt fungiert.

Grundlegend sind Posits genauso Fließkommazahlen wie auch IEEE-754 Floats, sie verwenden allerdings eine andere Art der Kodierung von Vorzeichen, Exponent und Mantisse. Wie in 2.2 gezeigt, bestehen Posits noch aus einem zusätzlichen Teil Regime $R = r \dots r\bar{r}$. In diesem liegen die Daten unärkodiert, sprich, die Lauflänge l von aufeinanderfolgenden Nullen oder Einsen $r \dots r, r \in \{0, 1\}$ bestimmt den Wert den das Feld darstellt. Der kodierte Wert k ergibt sich wie in Gleichung 2.1 dargestellt.

$$k = \begin{cases} -l & \text{wenn R aus Nullen besteht} \\ l - 1 & \text{wenn R aus Einsen besteht} \end{cases} \tag{2.1}$$

Beendet wird die Lauflängenkodierung mit einem einzigen invertierten Bit $\bar{r}$, das Regime beansprucht also in der Kodierung immer $m + 1$ Bit an Speicher und ist minimal 2 Bits lang. Dabei ist das Regime in der Gleitkommazahldarstellung etwa wie ein weiterer Exponent zu sehen, der den Faktor $useed^k$ zur repräsentierten Zahl beisteuert. $useed = 2^{(2^{es})}$ berechnet sich dabei aus dem Parameter es, der die Bitlänge des Exponentenfeldes beschreibt, und der für ein Positformate konstant gewählt werden muss.

Besagter Exponent E hat für Posits, im Gegensatz zu IEEE-754 Gleitkommazahlen kein Bias, sondern speichert den Exponentenwert e immer als vorzeichenlosen Integer. Negative Gesamtexponenten erreicht man mithilfe negativer Regime.

Das Vorzeichenbit S ist analog zu dem der Floats kodiert, jedoch haben Posits die zusätzliche Eigenschaft, dass negative Posits im Zweierkomplement invertiert werden. Dies hat den großen Vorteil, dass Vergleiche, die bei Floats zusätzlicher Logik bedürfen, zu einfachen Vergleichen von vorzeichenbehafteten Integern verkommen. Das ist jedoch nur garantiert, solange die Posits dieselbe Bitlänge wie der vorzeichenbehaftete Integer, der als Vergleichstyp verwendet wird, haben, sodass das Vorzeichenbit S des Posits im MSB des Integers liegt, wodurch es in der Zweierkomplementdarstellung seine negative Gewichtung bekommt.

Wie auch bei Floats ist die Mantisse M normalisiert und die führende Eins implizit. Anders als bei Floats gibt es allerdings keine subnormalen Zahlen, es gilt also immer $m = 1 + M$

Damit ergibt sich im Allgemeinen eine Interpretationsfunktion von:

$$x = (-1)^S \cdot useed^k \cdot 2^E \cdot (1 + M) = 2^{2^{es} \cdot k + E} \cdot (1 + M) \tag{2.2}$$

Sonderfälle Die Interpretation des dargestellten Posits verkompliziert sich, wenn das Regime betragsmäßig groß wird, in der Kodierung also sehr viele Bits benötigt. Mit wachsendem Regime, kann es zuerst dazu kommen, dass für die Mantisse keine Bits mehr übrig sind. Aufgrund der impliziten führenden Eins der Mantisse und fehlender Bits für die Nachkommastellen, verkommt die Mantisse zu $m = 1, 0$.

Wird das Regime noch länger, so kann es dazu kommen, dass der Exponententeil E teilweise oder ganz aus dem Register "rutscht". Hier gilt, dass alle der es Bits, die nicht dargestellt werden können, implizit mit 0 belegt sind.

Im Gegensatz zu IEEE Floats gibt es nur eine Darstellung der Null, bei der das gesamte Register mit Nullen gefüllt ist. Die Darstellung einer Negativen 0, bei der das Vorzeichen-Bit $S = 1$ gesetzt ist, gibt es hingegen bei Posits nicht, stattdessen ist das Bitmuster $10 \ldots 0$ für den Sonderfall NaR reserviert. "Not A Real" ist sozusagen das Posit-Äquivalent zu NaN, und kodiert ein ungültiges Ergebnis einer Rechnung, wie z.B. die Quadratwurzel einer negativen Zahl.

Darstellbarer Zahlenbereich Der Zahlenbereich den Posits abdecken können hängt von der Wahl von Exponentenlänge es und der Gesamtbitlänge W des Posits ab. Der betragsmäßig maximale Posit besteht nur aus dem größten maximalen Regime, Mantisse ist also implizit $m = 1$ und Exponent $e = 0$. Da das Regime Lauflängenkodiert ist hängt der Wert von der maximalen Lauflänge ab, welche sich als $W - 2$ aufeinanderfolgende gleiche Ziffern ergibt, da von der vollen Bitlänge ein Bit noch vom Vorzeichen S verwendet wird und ein anderes als $\bar{r}$ benötigt wird, um die Lauflängenkodierung zu terminieren. Damit ergeben sich als betragsmäßig maximaler und minimaler Posit:

$$posit_{\max} = useed^{(W-2)-1} = 2^{2^{es} \cdot (W-3)}; \qquad posit_{\min} = useed^{-(W-2)} = 2^{2^{es} \cdot (-W+2)} \tag{2.3}$$

Die Exponentenlänge es bestimmt also einerseits den Zahlenbereich den das Positformat abdecken kann, und andererseits die Genauigkeit, also die Anzahl der gültigen Stellen der Mantisse die gespeichert werden kann. Damit ist die Wahl von es wichtig für die Charakteristik des Zahlenformats das man bekommt. In älteren Publikationen lässt Gustafson diese Wahl dem Nutzer offen und liefert exemplarische Werte für es, sodass der Dynamikbereich eines Positformates an das IEEE Float Format derselben Bitlänge angepasst wird [16]. Der Posit Standard

[17] hingegen legt $es = 2$ fest, während unterschiedliche Implementierungen [18] von Posits wieder andere Werte für es verwenden. Um in dieser Arbeit alle Fälle abzudecken, wurden alle Implementierungen von Posits generisch gehalten.

Weitere Posit-verwandte Formate Gustafson definiert durch das Zusammenfügen zweier Posits und das Hinzunehmen eines *ubit* (uncertainty/Ungewissheits-bit) die sogenannten *Valids* als eine Form der Intervallarithmetik [3]. Ferner beschreibt der Posit Standard den sogenannten *Quire* als Registerformat für Zwischenergebnisse vor dem Runden (2017 noch "Exact accumulator" genannt in [3]). Damit ist ein Register gemeint, das groß genug ist (nach Standard $16 \cdot W$) um das exakte Ergebnis von Summen, Skalarprodukte oder FMA-Operationen vieler Operanden in einem Register zu akkumulieren und nur einmal danach zu runden. Dies eliminiert Rundungsschritte und damit Rundungsfehler von Zwischenergebnissen ähnlich zu den FMA-Operationen von IEEE-754 Floats, allerdings auf mehr Operanden [3], [17]. Diese beiden Formate werden in dieser Arbeit jedoch nicht thematisiert.

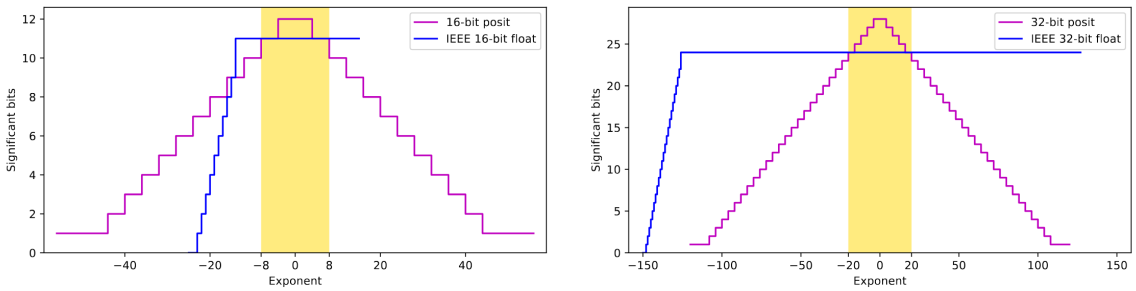
Als Anmerkung für den Rest dieser Arbeit: Auch wenn Posits technisch gesehen genauso Gleitkommazahlen sind, wird die englische Abkürzung *Float* im Folgenden als gleichbedeutend mit den IEEE-754 Gleitkommazahlen verwendet, um eine verständlichere Abgrenzung zu den Posits zu schaffen.

2.1.3 Unterschiede zwischen Floats und POSITs

Wie bereits bei der Beschreibung der Posits erwähnt, unterscheiden sich die Sonderfälle der Kodierung zu Floats maßgeblich: Die Vielzahl von NaN reservierten Bitkombinationen kann nun zusätzlich zur Darstellung von Zahlen verwendet werden, während die negative Null wegfällt und stattdessen zur Kodierung von NaR verwendet wird [16]. Genauso entfallen $\pm\infty$, die aufgrund der Rundungsregeln der Posits nicht nötig sind. Posits runden nach Standard [17] immer zum nächsten gültigen Posit, was Unter- und Überläufe wie sie von Floats bekannt sind ausschließt. Diese Reduktion der Sonderfälle ist vorteilhaft für Posits, da sie dadurch mehr reele Zahlen als Floats für dieselbe Bitlänge darstellen können, und die Ausnahmebehandlung bei Rechenoperationen simpler wird, was ebenfalls eine Vereinfachung von Schaltkreisen bedeutet [8].

Ebenfalls einfacher ist die Abhandlung von Vergleichsoperationen zwischen Posits. Einerseits sind diese ebenfalls durch die Reduktion der Sonderfälle begünstigt (z.B. $+0 = -0$ muss nicht beachtet werden), anderseits profitieren sie noch zusätzlich von der Sortiereigenschaft der kodierten Posits. Da bei negativen Posits der Bitstring (mit ausnahme des Vorzeichenbits S) im Zweierkomplement negiert wird, können die Posits einfach wie vorzeichenbehaftete Integer verglichen werden [16] (unter der oben genannten Voraussetzung).

Wie bereits erwähnt haben Posits ein anderes Rundungsverhalten als Floats: Es gibt keinen Überlauf zu ∞ und keinen Unterlauf zu 0, stattdessen gilt die Regel, dass immer zum nächsten darstellbaren Posit gerundet werden muss. Statt Über- bzw. Unterläufen wären das der betragsmäßig größte bzw. kleinste darstellbare Posit. Liegt ein Ergebnis in der Mitte zwischen zwei darstellbaren Posits, so wird zu dem gerundet, der auf eine 0 endet, man spricht hierbei auch von "*round-to-nearest-even*" [8]. Fällt die Mantisse aufgrund eines langen Regimes weg und es liegt ein Exponentenbit im LSB, so muss nach dem Exponenten gerundet



(a) 16-bit formats. **(b)** 32-bit formats.

Abbildung 2.3: Genauigkeit der Mantisse (Significant bits) von Posits und IEEE Floats für verschiedene Exponenten[8]

werden. Hierbei rundet man zur logarithmisch nächsten Zahl. Dieses Verhalten ist in der verfügbaren Literatur nicht dokumentiert und wird von Gustafson nur in einem Forumsbeitrag als "*Twilight-Zone*" Rundung bezeichnet [19] und muss bei Implementierung noch zusätzlich berücksichtigt werden.

Posits haben keine Subnormalen Zahlen wie Floats und damit auch keinen "*gradual underflow*" (allmählichen Unterlauf) im herkömmlichen Sinne. Stattdessen lässt sich das Verhalten der Subnormalen, dass für einen betragsmäßig größeren Exponenten die Zahl der signifikanten Stellen abnimmt für alle Posits betrachten [16]. Diese Erscheinung nennt man auch "*tapered precision*" (konische Genauigkeit). Die Werte der Posits liegen deswegen in einer Normalverteilung um die Null herum, und nahezu die Hälfte aller Posits auf dem Intervall $[-1, 1]$. Da hier die meisten Berechnungen vieler Anwendungsbereiche wie z.B. Deep Learning oder Numerik stattfinden, können diese von der erhöhten Genauigkeit auf diesem Intervall profitieren [8]. Das Verhältnis vom Exponenten zur verfügbaren Genauigkeit wird in Grafik 2.3 gezeigt.

Ein weiterer praktischer Vorteil der Posits ist eine einfacher Konvertierung zwischen zwei Formaten. So erhält man durch das Anhängen einer 0 hinter dem LSB denselben Zahlenwert im Posit-format, das um ein Bit länger ist [8]. Haben nun zwei Posit-Formate denselben Wert für die Exponentenlänge es (was bei Einhaltung des Standards [17] immer gegeben wäre), so wären Konvertierungen zu einem Format längerer Bitlänge nichts anderes als Bitshifts nach links, und umgekehrt nur Rechtsshifts mit Rundung.

Der Posit Standard versucht auch noch das Problem von verschiedenen Ergebnissen auf verschiedenen Rechnern, unter dem Floats leiden, anzusprechen. Im Gegensatz zum IEEE-754 verlangt er, dass alle arithmetischen Operationen korrekt gerundet sein müssen [16]. Weiter verlangt er, dass alle Fused-Operationen in Quellcode unterscheidbar zu normalen Rechenoperatoren geschrieben werden müssen und verbietet das Umsortieren der Ausführungsreihenfolge von Berechnungen, was die Implementierungen vom Posits so einschränken soll, sodass jede replizierbare Ergebnisse liefert. [17]

Neben all der Verbesserungen, die die Posits mitbringen, liegen sie doch gegen Floats im Nachteil was die Dekodierung angeht. Die variable Größe des Regimes erzwingt eine Sequentialisierung der Dekodierung, da zur Dekodierung der Mantisse und des Exponenten zuerst die Länge des Regimes bekannt sein muss, was das Hardwaredesign verkompliziert [16].

Untersuchungen von Gustafson Dass Posits besser als Floats sind, lässt sich nicht pauschal feststellen. Im Wesentlichen sind Posits genau wie IEEE Floats als Gleitkommazahlen nur eine Approximation der Reellen Zahlen, die in jedem Rechenschritt einen Rundungsfehler erzeugen kann. Das Regime variabler Länge erlaubt Posits einen großen Zahlenbereich abzudecken und gleichermaßen für kleine Werte eine hohe Genauigkeit zu bieten, was ihnen erlaubt mit weniger Bits gleichmäßig gute Ergebnisse zu erzielen als Floats [16]. So haben sie zwar Vorteile was die Ausnutzung der verfügbaren Bits angeht, leiden aber trotzdem auch unter Rundungsfehlern die mit ihrer Funktion der teilweisen Abbildung der reellen Zahlen inhärent einhergehen, mit dem Unterschied, dass diese für unterschiedliche Größenordnungen unterschiedlich groß ausfallen.

Gustafson stellt in seinen Publikationen Demonstrationen zur Schau, in denen Posits durchaus bessere Ergebnisse als Floats liefern. Ob sich dieses Verhalten auch auf Mixed-Precision Ansätze überträgt, und inwiefern die numerischen Lösungsverfahren von den Eigenschaften der Posits zu profitieren vermögen wird im weiteren Verlauf dieser Arbeit betrachtet.

2.2 Gewöhnliche Differentialgleichungen

Eine gewöhnliche Differentialgleichung ist eine Gleichung

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f(t, y(t)), \qquad y : \mathbb{R} \to \mathbb{R}^d, f : (\mathbb{R}, \mathbb{R}^d) \to \mathbb{R}^d \tag{2.4}$$

in der eine Funktion y sowie deren zeitliche Änderung $\frac{\mathrm{d}y}{\mathrm{d}t}$ vorkommen. Dabei sei $d \in \mathbb{N}^+$ die Dimension des Systems.[20] Im englischen Sprachgebrauch spricht man von "Ordinary Differential Equation" mit der Abkürzung ODE, welche in der restlichen Arbeit ebenfalls verwendet wird.

Mit der zusätzlichen Bedingung $y(t_0) = y_0$ beschreibt man ein Anfangswertproblem (AWP) mit Anfangswert $y_0 \in \mathbb{R}^d$ am Anfangszeitpunkt $t_0 \in \mathbb{R}$. [20]

Ist die rechte Seite f nicht von t abhängig, so spricht man von *autonomen* Differentialgleichungen, die Gleichung vereinfacht sich in diesem Falle zu

$$\frac{\mathrm{d}}{\mathrm{d}t}y(t) = f(y(t)), \qquad f : \mathbb{R}^d \to \mathbb{R}^d$$

Differentialgleichungen beschreiben im Wesentlichen einen Zusammenhang der zeitlichen Änderung eines Wertes zum Wert selbst und werden damit häufig verwendet, um in den Naturwissenschaften Systeme zu modellieren. Da es für viele Differentialgleichungen keine analytische Lösung gibt, zieht man stattdessen numerische Verfahren zurate, zu denen die Runge-Kutta-Verfahren zählen.

2.2.1 Runge-Kutta-Verfahren

Runge-Kutta Verfahren sind eine Klasse von Einschrittverfahren zum numerischen Lösen gewöhnlicher Differentialgleichungen. Einschrittverfahren approximieren die Lösung einer ODE

im Allgemeinen durch eine Zeitdiskretisierung $\tilde{y}(t)$ der Lösungsfunktion $y(t)$ auf einem Zeitgitter $\mathcal{T}$ aus N Zeitschritten:

$$\mathcal{T} = \{t_0, \ldots, t_N\}, \quad t_0 < t_1 < \cdots < t_N, \quad N \in \mathbb{N}$$

In all unseren Anwendungen berechnet sich $\forall 1 \leq n \leq N : t_n = t_0 + n \cdot h$ aus einer konstanten Schrittweite $h \in \mathbb{R}$. Auf diesem Gitter soll nun eine *Gitterfunktion* $\tilde{y} : \mathcal{T} \to \mathbb{R}^d$ existieren, die für jeden Zeitpunkt $t_n \in \mathcal{T}$ im Zeitgitter $\tilde{y}(t_n) \approx y(t_n)$ approximiert.[20]

Ein Einschrittverfahren ist nun eine Funktion

$$\Phi : (\mathbb{R}, \mathbb{R}^d, \mathbb{R}) \to \mathbb{R}^d$$

aus der, rekursiv mit gegebenem Anfangswert y_0, eine Gitterfunktion definiert werden kann:

$$\tilde{y}(t_0) = y_0, \qquad \tilde{y}(t_{n+1}) = \Phi(t_n, \tilde{y}(t_n), h) \text{ für } i \in \{0, \ldots, N-1\}$$

Einschrittverfahren beschreiben also eine Berechnungsvorschrift mithilfe derer man aus dem Wert vom Zeitschritt t_n das Ergebnis des nächsten Zeitschrittes $\tilde{y}(t_{n+1})$ berechnen kann.[20]

Da die Funktion $\tilde{y}$ sowieso nur für diskrete $t_n \in \mathcal{T}$ definiert ist, schreiben wir die Schrittwerte stattdessen als Index $u^{(n)} = \tilde{y}(t_n),$ für $n \in \{0, \ldots, N\}$

Runge-Kutta-Verfahren (RKV) sind eine Klasse von Einschrittverfahren. Ein s-stufiges RKV wird beschrieben durch

$$k^{(i)} = f\left(t + c_i h, y + h \sum_{j=1}^{s} a_{ij} k^{(j)}\right) \text{ für } i \in 1, \ldots, s, \ s \in \mathbb{N}^+ \tag{2.5}$$

$$\Phi(t, y, h) = y + h \sum_{i=1}^{s} b_i k^{(i)}, \text{ und damit } u^{(n+1)} = u^{(n)} + h \sum_{i=1}^{s} b_i k^{(i)} \tag{2.6}$$

wobei die Wahl der Stufenzahl s und der Koeffizienten $b, c \in \mathbb{R}^s$ und $A \in \mathbb{R}^{s \times s}$ ein Verfahren konkretisieren. Die Zwischenwerte $k^{(i)}, i \in \{1, \ldots, s\}$ bezeichnet man als i-te Stufe des Verfahrens [20], Sie müssen in jedem Zeitschritt n für $y = u^{(n)}$ neu berechnet werden. In der Literatur werden die Koeffizienten häufig in Form eines Butcher-Tableaus notiert.

Wenn die Matrix A in der unteren Dreiecksform ist, ist jede Stufe $k^{(i)}$ nur von vorhergehenden Stufen $k^{(j)}, j < i$ abhängig, man spricht hier von *expliziten* Runge-Kutta-Verfahren. Ist das nicht der Fall, so erhält man ein Gleichungssystem $k = G(k), k = [k^{(1)}, \ldots, k^{(s)}]^T$, das gelöst werden muss, um alle Stufen zu ermitteln, diese Verfahren nennt man *implizite* RKV.[20]

Von den herkömmlichen RKV wird in der Arbeit nur das klassische Runge-Kutta-Verfahren mit Stufenzahl $s = 4$, auch RK4 genannt, verwendet. Dessen Koeffizienten sehen im Butcher Tableau wie folgt aus [20]:

0				
$\frac{1}{2}$	$\frac{1}{2}$			
$\frac{1}{2}$	0	$\frac{1}{2}$		
1	0	0	1	
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$

Weiter charakterisiert sind Einschrittverfahren durch ihre *Ordnung* q, die angibt wie schnell das Verfahren konvergiert und in welcher Größenordnung sich der globale Fehler $\mathcal{O}(h^q)$ aufhält. Bei den herkömmlichen RKV wird versucht die Koeffizienten so zu wählen, dass die Ordnung q für gegebene Stufenzahl s möglichst groß wird. Für $1 \leq q \leq 4$ ist es möglich ein Verfahren mit $s = q$ zu konstruieren, für größere q benötigt man immer $s > q$ Stufenzahlen. [20]

Die *Stabilität* ist ebenfalls eine Eigenschaft von Einschrittverfahren, und ein Maß dafür, wie *steif* eine Differnetialgleichung sein darf, sodass das Verfahren, in Abhängigkeit der Schrittweite h, noch konvergiert. Die Steifigkeit einer Differentialgleichung zu definieren wäre umständlich und würde den Umfang dieser Arbeit überschreiten, die wichtigste Feststellung ist, dass Steifigkeit eine intrinsische Eigenschaft einer Differentialgleichung ist, die für manche ODEs größer und für andere kleiner ist. Jeder Löser hat nun einen Stabilitätsbereich, in dem die zu lösende ODE, abhängig von der gewählten Schrittweite h, liegen muss, sodass die Konvergenz des Verfahrens garantiert ist. Eine ODE kann außerhalb des Stabilitätsbereichs liegen, wenn die Steifigkeit zu groß ist, und die Schrittweite h nicht hinzureichend klein gewählt ist. Das heißt, dass man theoretisch mit einer genügend kleinen Schrittweite immer die Konvergenz garantieren könnte, was jedoch natürlich mit einem großen Aufwand verbunden wäre. [20]

Der Unterschied ihrer Stabilitätsregionen ist der maßgeblichste Vorteil der impliziten RKV gegenüber der expliziten; Implizite RKV sind A-Stabil, garantieren also immer eine Konvergenz, ungeachtet der Steifigkeit der ODE, während explizite RKV relativ kleine Stabilitätsregionen aufweisen.

2.3 Mixed Precision

Mixed-Precision, im Deutschen "Gemischte Genauigkeit", beschreibt das Verwenden mehrere Datentypen von verschiedener Genauigkeit in Numerischen Berechnungen. Die Motivation dahinter ist eine Einsparung von Speichertransfers und Berechnungzeit durch das geschickte Austauschen von Berechnungen in hoher Genauigkeit durch billigere Berechnungen auf einer reduzierten Genauigkeit.[4]

Da diese Reduzierung der Genauigkeit sich natürlich auch in einer größeren numerischen Abweichung des Ergebnisses bemerkbar macht, werden Verfahren verwendet, die durch geschickte Korrekturen den Fehlerterm trotz der teilweisen Verwendung ungenauerer Datentypen klein halten. Im Allgemeinen versuchen Mixed-Precision Methodiken also gleichermaßen die Geschwindigkeit des reduzierten und die Genauigkeit des genauen Datentyps auszunutzen.

Im Rahmen dieser Arbeit werden zwei verschiedene Ansätze zur Nutzung von Gemischter Genauigkeit für Runge-Kutta Verfahren betrachtet:

2.4 Implizite Mixed-Precision-Runge-Kutta-Verfahren

In seinem Paper "Perturbed Runge Kutta Methods For Mixed Precison Applications"[1] aus dem Jahr 2022 stellt Zachary Grant ein Verfahren zur Anpassung von impliziten RKV vor, um das implizite Gleichungssystem auf reduzierter Genauigkeit zu berechnen. Das Paper beschränkt sich in seinen Definitionen nur auf autonome ODEs.

Am Anfang steht dabei eine Umformung von Gleichung 2.6 durch das Herausziehen der Funktionsauswertung f aus der Berechnung für $k^{(i)}$ mit der Definition von $k^{(i)} = f(\kappa^{(i)})$:

$$\kappa^{(i)} = u^{(n)} + h \sum_{j=1}^{s} a_{ij} f\left(\kappa^{(j)}\right), \quad i \in \{1, \ldots, s\} \tag{2.7}$$

$$u^{(n+1)} = u^{(n)} + h \sum_{j=1}^{s} b_j f\left(\kappa^{(j)}\right) \tag{2.8}$$

Der Mixed-Precision Ansatz sieht vor, dass das Gleichungssystem 2.7 in verringerter Genauigkeit gelöst wird. Sei $\hat{f}(y)$ nun die Funktionsauswertung in verringerter Genauigkeit von $f(y)$ mit einer Perturbation von $f(y) - \hat{f}(y) = \epsilon\tau(y)$ aus dem Rundungsfehler, so ergibt sich:

$$\hat{\kappa}^{(i)} = u^{(n)} + h \sum_{j=1}^{s} \hat{a}_{ij} \hat{f}\left(\hat{\kappa}^{(j)}\right) \tag{2.9}$$

Diese $\hat{\kappa}^{(i)}$, die mit einem Störungsfehler (Perturbation Error) behaftet sind, werden statt $\kappa^{(i)}$ in Gl. 2.8 eingesetzt, die noch in voller Genauigkeit ausgeführt wird. Trotzem wirkt sich die Abweichung auf den globalen Fehler des gesamten RKVs negativ aus. Wie Grant [1] vorschlägt diesen Fehler zu kompensieren wird im Folgenden konkret anhand des Lösers zur Impliziten Mittelpunktsregel (englisch "Implicit Midpoint") betrachtet.

Mit den Butcher-Tableau der Impliziten Mittelpunktsregel

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$$

ergibt sich für den Löser aus den Gl. 2.7 und 2.8 :

$$\hat{\kappa}^{(1)} = u^{(n)} + \frac{h}{2} \hat{f}\left(\hat{\kappa}^{(1)}\right) \tag{2.10}$$

$$u^{(n+1)} = u^{(n)} + h f\left(\hat{\kappa}^{(1)}\right) \tag{2.11}$$

Für dieses Schema ergibt sich ein globaler Fehler der Größenordnung $E = \mathcal{O}(h^2) + \mathcal{O}(\epsilon h)$ [1], [5], d.h für große Schrittweiten h tritt der Perturbationsfehler kaum zum Vorschein, dominiert aber den Fehlerterm für kleine h und stört damit die Konvergenz des Verfahrens, was sich noch in der Auswertung betrachtent lassen wird.

Grant zeigt, dass sich dieser Fehlerbeitrag noch durch $p - 1$ zusätzliche explizite Korrekturschritte, die in hoher Genauigkeit berechnet werden, reduzieren lässt, und erweitert das Schema somit auf:

$$\begin{aligned} \kappa_{[0]}^{(1)} &= u^{(n)} + \frac{h}{2} \hat{f}\left(\kappa_{[0]}^{(1)}\right) \\ \kappa_{[k]}^{(1)} &= u^{(n)} + \frac{h}{2} f\left(\kappa_{[k-1]}^{(1)}\right) \text{ für } k = 1, \ldots, p-1 \\ u^{(n+1)} &= u^{(n)} + h f\left(\kappa_{[p-1]}^{(1)}\right) \end{aligned} \tag{2.12}$$

Damit lässt sich ein globaler Fehler von $E = \mathcal{O}(h^2) + \mathcal{O}(\epsilon h^p)$ erreichen, der Beitrag des Perturbationsfehlers wurde also verringert.

Die Publikation [1] beschreibt darüber hinaus auch noch Schemata für weitere RKV Löser bereit. Deren Konstruktion gestaltet sich komplexer und beinhaltet verschiedene Butcher-Koeffizienten für die Terme verringerter Genauigkeit $\hat{A}, \hat{b}$ und hoher Genauigkeit A, b zusätzlich zu den Korrekturschritten. Diese werden betrachtet und empirisch analysiert in den Publikationen von Burnett [5], [6] oder Dravins [7], liegen aber nicht im Fokus dieser Arbeit.

2.5 Explizite Mixed-Precision-Runge-Kutta-Chebyshev-Verfahren

Einen weiteres Mixed-Precision RKV wurde 2022 von Croci [2] vorgestellt. Dabei handelt es sich um explizite stabilisierte Runge-Kutta-Verfahren.

Unter stabilisierten RKV versteht man Verfahren, die, anstatt möglichst wenige Stufen s für eine Ordnung q zu verwenden, den Fokus auf die Stabilität (siehe 2.2.1) legen und versuchen die Stabilitätsregion zu maximieren [21]. Croci verwendet davon konkret die Runge-Kutta-Chebyshev-Verfahren (RKCV), die allgemein für s Stufen wie folgt formuliert sind:

$$\begin{aligned} &k^{(0)} = 0, \quad k^{(1)} = \mu_1 hf\left(u^{(n)}\right) \\ &k^{(j)} = \nu_j k^{(j-1)} + \psi_j k^{(j-2)} + \mu_j hf\left(u^{(n)} + k^{(j-1)}\right) + \gamma_j hf\left(u^{(n)}\right), \text{ für } j \in \{2, \dots, s\} \\ &u^{(n+1)} = u^{(n)} + k^{(s)} \end{aligned} \tag{2.13}$$

Für die Berechnung der Koeffizienten existieren verschiedene Notationen, zur Konsistenz wurden hier die Notation aus [2] übernommen. $\mu_j, \nu_j, \psi_j, \gamma_j$ sind für $j \in \{2, \dots, s\}$ gegeben als

$$\mu_1 = b_1\omega_1, \quad \mu_j = 2\omega_1\frac{b_j}{b_{j-1}}, \quad \nu_j = 2\omega_0\frac{b_j}{b_{j-1}}, \quad \psi_j = -\frac{b_j}{b_{j-2}}, \quad \gamma_j = -\mu_j a_{j-1} \tag{2.14}$$

Die Koeffizienten $a_j = 1 - b_j T_j(\omega_0)$ für $j \in \{0, \dots, s\}$ ergeben sich aus den Chebyshev-Polynomen $T_j(x)$ erster Ordnung, welche den Runge-Kutta-Chebyshev-Verfahren ihren Namen geben. Die Chebyshev-Polynome lassen sich rekursiv definieren als

$$T_j(x) = \begin{cases} 1 & \text{falls } j = 0 \\ x & \text{falls } j = 1 \\ 2xT_{j-1}(x) - T_{j-2}(x) & \text{sonst} \end{cases} \tag{2.15}$$

Weiter braucht man noch die Koeffizienten ω_0, ω_1 und $b_j, j \in \{0, \dots, s\}$. Diese sind abhängig davon wie man die Ordnung q des Verfahrens wählt; Für $q = 1$ (man schreibt als Abkürzung RKC1) gilt:

$$\omega_0 = 1 + \frac{\varepsilon}{s^2}, \quad \omega_1 = \frac{T_s(\omega_0)}{T_s'(\omega_0)}, \quad b_j = \frac{1}{T_j(\omega_0)}, \; j \in \{0, \dots, s\} \tag{2.16}$$

Dabei ist ε der *Dämpfungsfaktor*, der Einfluss auf die Stabilitätsregion nimmt und diese allgemein etwas "abrundet". Für RKC1 ist $\varepsilon = 0.05$ gängig [2], [22]. $T_j'(x)$ bezeichnet die erste

Ableitung $\frac{\mathrm{d}}{\mathrm{d}x}T_j(x)$ die definiert ist als $T_j'(x) = jU_{j-1}(x)$ [23] wobei $U_j(x)$ das Chebyshev-Polynom zweiter Art ist, das ähnlich zu dem der ersten Ordnung wiederrum rekursiv definiert ist, als:

$$U_j(x) = \begin{cases} 1 & \text{falls } j = 0 \\ 2x & \text{falls } j = 1 \\ 2xU_{j-1}(x) - U_{j-2}(x) & \text{sonst} \end{cases} \tag{2.17}$$

Für RKC2 mit $q = 2$ sind die Koeffizienten

$$\omega_0 = 1 + \frac{\varepsilon}{s^2}, \quad \omega_1 = \frac{T_s'(\omega_0)}{T_s''(\omega_0)}, \quad b_0 = b_1 =_2,\ b_j = \frac{T_j''(\omega_0)}{T_j'(\omega_0)^2},\ j \in \{2, \ldots, s\} \tag{2.18}$$

wobei hier ein geläufiger Wert für den Dämpfungsfaktor $\varepsilon = 2/13$ beträgt [2].
Wie sich aus der Definition berechnen lässt, wird für RKC1

$$\forall j \in 0, \ldots, s : a_j = 1 - b_j T_j(\omega_0) = 1 - \frac{1}{T_j(\omega_0)} T_j(\omega_0) = 0 \implies \gamma_j = 0$$

der $\gamma_j h f(u^{(n)})$ Term fällt also aus Gl. 2.13 heraus.
Croci konstruiert nun einen Schritt eines Mixed-Precision Verfahren mit $\hat{u}^{(n)} \approx u^{(n)}$ als:

$$\begin{aligned} &\hat{k}(0) = 0, \quad \hat{k}^{(1)} = \mu_1 h f\left(\hat{u}^{(n)}\right) \\ &\hat{k}^{(j)} = \nu_j \hat{k}^{(j-1)} + \psi_j \hat{k}^{(j-2)} + \mu_j h\left(f\left(\hat{u}^{(n)}\right) + \widehat{\Delta f}_{j-1}\right) + \gamma_j h f\left(\hat{u}^{(n)}\right), \text{ für } j \in \{2, \ldots, s\} \\ &\hat{u}^{(n+1)} = \hat{u}^{(n)} + \hat{k}^{(s)} \end{aligned} \tag{2.19}$$

Maßgebliche Änderung zu Gl. 2.13 ist dabei das Austauschen der Zwischenschritte $k^{(j)}$ durch ihre Approximationen $\hat{k}^{(j)}$, und der Funktionsauswertung im μ_j-Term durch eine Approximation in niedriger Genauigkeit. Für $j \in \{1, \ldots, s-1\}$ ist $\widehat{\Delta f}_j$ definiert als Abschätzung

$$\widehat{\Delta f}_j \approx \Delta f_j = f(\hat{u}^{(n)} + \hat{k}^{(j)}) - f(\hat{u}^{(n)}) \tag{2.20}$$

ohne die man (also für $\widehat{\Delta f}_j = \Delta f_j$) das ursprüngliche Schema aus Gl. 2.13 erhalten würde.
Damit dieses Schema die Konvergenzordnung $q = 1$ bewahrt, muss für die Abweichung von $\widehat{\Delta f}_j$ gelten:

$$\forall 1 \leq j < s : \widehat{\Delta f}_j = \Delta f_j + \mathcal{O}(\xi h) \tag{2.21}$$

wobei $\xi \in \mathbb{R}^+$ eine kleine Konstante ist [2]. Um RK2 noch zur zweiten Ordnung zu bewahren, benötigt es noch weitere Schritte die in [2] aufgeführt sind, diese Arbeit beschränkt sich allerdings auf RK1.

Im Paper werden abhängig von der Form der rechten Seite f verschiedene Ansätze zur Berechnung von $\widehat{\Delta f}_j$ präsentiert. Um eine Kompatibilität mit allen AWPs zu garantieren, wurde der allgemeinste der Ansätze gewählt;

$$\widehat{\Delta f}_j = \hat{J}_f(\hat{u}^{(n)}) \cdot \hat{k}^{(j)} \tag{2.22}$$

Hier beschreibt $\hat{J}_f$ die Jacobimatrix von $\hat{f}$, die in verringerter Genauigkeit an der Stelle $\hat{u}^{(n)}$ evaluiert wird. Besonders wenn die Jacobimatrix analytisch ermittelbar/bekannt ist, bietet sich dieses Vorgehen an. Andernfalls kann man auf numerische Approximationen zurückgreifen, wofür ebenfalls in [2] verschiedene Optionen gelistet werden, die der Anforderung an die maximal erlaubten Abweichung gerecht werden.

3 Implementierung

3.1 Allgemeine Entwurfsentscheidungen und Aufbau

Um die Auswirkungen von Posits in Mixed-Precision-Runge-Kutta-Verfahren analysieren zu können, benötigt man ein Framework, das schnell mit Posits rechnen kann und generisch implementiert ist, sodass sich Datentypen einfach austauschen lassen um verschiedene Konfigurationen gemischter Genauigkeiten betrachtet zu können.

Aufgrund mangelnder, weitläufiger Hardwareunterstützung des Positformats fiel die Entscheidung auf eine Softwaresimulation des Zahlenformats. Es sei angemerkt, dass es bereits Ausarbeitungen wie [24] darüber gibt, wie die Hardware von Posit-Recheneinheiten aussehen müsste, die sich mit FPGA simulieren ließen, allerdings bietet in diesem Falle hier eine Softwaresimulation mehr Flexibilität um beliebige Posit-Formate zu simulieren.

Unter [25] ist von den Herausgebern des Posit-Standards eine Liste von existierenden Posit-Implementierungen geführt, allerdings erwies sich die einzige aktiv weiterentwickelte Bibliothek, die ebenfalls beliebige Positformate unterstützt, "universal" [26] aufgrund ihres großen Umfanges als unhandlich und zu langsam. Da die das Format bestimmenden Größen der Bitlänge und *es* zur Kompilierzeit feststehen müssen und das Programm sehr oft für verschieden Größen ausgeführt und damit neukompiliert werden musste, war der langsame Build-Prozess der Bibliothek so nicht hinnehmbar.

Somit wurde die Softwaresimulation eigens implementiert. Da besonders zur effizienteren Dekodierung der Posits die Nutzung von eingebetteten Assemblerbefehlen notwendig war, schränkte sich die Wahl auf eine hardwarenahe Programmiersprache ein und fiel schließlich auf C.

Auf dieser Softwaresimulation liegt ein "Mixed-Precison" System, das die Generizität des Programms garantiert und per Makros die Funktionen die hinter Berechungen stehen austauschen kann, und somit Posits z.B. durch Float-Formate zu substituieren. Dieses benutzen die Solver dann um eine Menge definierter Anfangswertprobleme zu lösen. Zusätzlich sind für die Solver einige numerische Algorithmen, z.B. zum Lösen linearer und nichtlinearer Gleichungssysteme, notwendig, die ebenfalls generisch für alle Zahlentypen implementiert wurden. Um dieses Framework herum wurde zur Automatisierung der Kompilierung und Auswertung ein Hilfsprogramm aus Python-Skripten entworfen, dass die Benutzung für den Endnutzer mit einer gesammelten Kommandozeilenschnittstelle vereinfachen soll.

3.2 Posit Softwaresimulation

Bei der Implementierung der softwaresimulierten Posits lag der Fokus auf der Geschwindigkeit der Berechnungen, da besonders für implizite Runge-Kutta-Verfahren relativ viel Rechenaufwand anfallen kann. So wurde versucht alle Berechnungen zur Kompilezeit zu tätigen, die möglich sind.

```
typedef struct {
    uint8_t sign;
    int32_t exponent;
    uint64_t mantissa;
} unpacked_t;
```

Listing 3.1: Defintion von `unpacked_t`.

Deswegen ist ein Posit im C-Code nichts weiteres als ein Alias für den nächstgrößeren Integetypen in den die Bitlänge des Posits passt, dieser Typ wird im weiteren Verlauf auch Container-Typ genannt, da er den Posit in sich enthält. Bitlänge und Exponentenlänge es die zur Kodierung notwendig sind, müssen zur Kompilezeit festgelegt sein, wo sie der Präprozessor in den Code einfügt und damit bereits Konstanten vorberechnen kann. Dieses Vorgehen hat auch den Vorteil im Vergleich zu einer dynamischen Implementierung, in der ein Posit durch ein Struct dargestellt würde, das zusätzlich noch Bitlänge und Exponentenlänge speichert, dass nicht unnötig viele zusätzliche Speicherzugriffe für das Laden eines Posits notwendig sind.

Um das Potenzial moderner Hardware auszunutzen und Prozesse wie das Dekodieren des Regimes zu beschleunigen, wurde Gebrauch von Befehlssatzerweiterungen des X86 Maschinencodes gemacht. Man beachte, dass das Framework damit nicht plattformunabhängig nutzbar ist, sondern nur auf 64 Bit X86 Prozessoren läuft, die die Befehlssatzerweiterungen `bmi1`, `bmi2` und `lzcnt` unterstützen, was aber für die meisten Prozessoren die nicht älter als 15 Jahre sind gegeben sein sollte.

Die integralen Bestandteile für jede Rechenoperation sind dabei das Ent- und Verpacken des Posits. So werden die Funktionen genannt, die die Konvertierung zwischen dem als Bitstring kodierten Posit und einem im Programm `unpacked_t` genannten Zwischenformat, definiert in Listing 3.1 vornehmen.

In diesem ist das Vorzeichenbit in sein eigenes Byte entpackt, das Regime k und der Exponent e sind zusammen zu einem Gesamtexponenten $k2^{es} + e$ kombiniert gespeichert und die Mantisse als Festkommazahl so abgespeichert, dass die sonst implizite führende Eins explizit im MSB des 64 Bit Integers liegt, d.h das Komma steht immer an Stelle 63.

Die Entpackungsfunktion `unpack`, gezeigt in Listing 3.2, kümmert sich darum, einen als Bitstring in einem Integer verpackten Posit in das Zwischenformat zu dekodieren. Dabei werden Schritt für Schritt die Komponenten aus dem Posit gemäß den Definitionen von Kapitel 2.1.2 dekodiert: Zuerst wird das Vorzeichenbit extrahiert, bei einer negativen Zahl wird noch der gesamte Bitstring negiert, danach die Lauflänge des Regimes mithilfe der Instruktion `lzcnt` ermittelt, die die Zahl führender Nullen eines Registers zählt. Abhängig von der Größe des Regimes werden die restlichen Bits, insofern es welche gibt, weiter dekodiert. Zuerst werden so viele Exponentenbits wie möglich entnommen und mit dem Wert des Regimes zum Gesamtexponenten kombiniert, implizite Bits werden auf Null gesetzt. Sind nun noch Bits für die Mantisse übrig, wird diese auf die Kommastelle am 63ten Bit ausgerichtet und die implizite führende Eins vor dem Komma angefügt.

Für jede Rechenoperation werden zu Beginn, nach dem Abfangen etwaiger Sonderfälle (z.B. Division durch Null), die Operanden zu `unpacked_t` entpackt. Nun können Rechnungen analog wie für allgemeine Gleitkommazahlen ausgeführt werden. Um bei Berechnungen die maximal

```
#define MANTISSA_DP 63 /*Decimal Point position in the mantissa*/
#define MANTISSA_ONE (1 << MANTISSA_DP)
#define SIGN_BIT_POS (BITLEN - 1) /*Bit Position of the sign in the posit*/
/*Empty bits when posit is smaller than its container*/
#define EMPTY_LEADING_BITS (CONTAINERLEN - BITLEN)
static inline posit_t complementIfNegative(posit_t p, uint8_t sign) {
    if(sign) {
        p = _bzhi_u64(p, SIGN_BIT_POS); // clear sign and above
        p = -p; // 2s complement
        p = _bzhi_u64(p, SIGN_BIT_POS); // clear again
        return (((posit_t)sign) << SIGN_BIT_POS) | p; // add sign
    }
    else return _bzhi_u64(p, SIGN_BIT_POS);
}
static inline unpacked_t unpack(posit_t p) {
    unpacked_t u = {0, 0, 0};
    p = _bzhi_u64(p, BITLEN);// clear all above the posits size
    if(p == 0) return u;
    u.sign = _bextr_u64(p, SIGN_BIT_POS, 1);
    if(u.sign) p = -p; // invert if negative
    p = _bzhi_u64(p, SIGN_BIT_POS); // clear after potential inverting
    uint8_t firstRegimeBit = _bextr_u64(p, SIGN_BIT_POS - 1, 1);
    // left-align & conditionally invert the regime to use lzcnt for regime length counting
    uint64_t regimeLZS = (uint64_t) p << (1 + EMPTY_LEADING_BITS);
    if (firstRegimeBit) regimeLZS = ~regimeLZS;
    // set a terminating 1 to the end of the posit to stop lzcnt
    // This is needed when BITLEN < Containerlen
    regimeLZS |= ((uint64_t) 1 << EMPTY_LEADING_BITS);
    uint8_t regimeLength = __lzcnt64(regimeLZS);// count the leading zeroes
    int16_t regime;
    if (!firstRegimeBit) // regiment started with a 0 -> k is negative
        regime = -regimeLength;
    else regime = regimeLength - 1;
    regimeLength++; // add the terminating 0/1 to regimentLength;
    // get as many exponent bits as possible
    int8_t remainingBits = BITLEN - 1 - regimeLength;
    if(remainingBits <= ES) {
        // shift the bits es bits that are implicit zeros outside the register in
        uint8_t exp = _bzhi_u64(p << (ES - remainingBits), ES);
        u.exponent = (regime << ES) | exp;
        u.mantissa = MANTISSA_ONE;
        return u;
    } // else: a mantissa exists!
    // mask out the mantissa:
    int8_t encMantissaBitlen = BITLEN - 1 - regimeLength - ES;
    u.mantissa = _bzhi_u64(p, encMantissaBitlen);
    // align as fixed point number
    u.mantissa <<= MANTISSA_DP - encMantissaBitlen;
    u.mantissa |= MANTISSA_ONE; // prepend implicit leading 1 at bit 64
    uint16_t exp = _bextr_u64(p, encMantissaBitlen, ES);
    u.exponent = (regime << ES) | exp;
    return u;
}
```

Listing 3.2: Symbolische Posit-Entpackungsfunktion für einen generischen Posit mit Bitlänge `BITLEN` und Exponentenlänge `ES`

mögliche Genauigkeit zu erhalten, muss man die 64 Bit der Mantisse voll ausnutzen. Dafür ist es wichtig hier jede Art von Überlauf zu berücksichtigen, um keine Bits vor der Rundung zu verlieren. Jegliche überlaufende Information die nicht in den 64 Bit Mantissen Platz findet, wird in einem Zusatzwert `remainder` aufgefangen, der `pack(unpacked, remainder)` am Ende als Zusatzinformation zur Rundung mitgegeben wir. So wird beim Angleichen der Exponenten bei Addition oder Subtraktion der Teil, der beim Rechtsverschieben der Mantisse verloren gehen würde, in diesem Wert gespeichert, wo er später genutzt werden kann um z.B bei Unterläufen durch Subtraktion von rechts wieder neue gültige Bits einzuschieben oder bei der Rundung weitere Details zu liefern. Multiplikationen nutzen eingebettetem Assemblercode um die zwei 64 Bit Mantissen zu einem 128 Bit Ergebnis zu multiplizieren, von dem dann die obere Hälfte das Ergebnis enthält während die untere als `remainder` der Rundung mehr Details liefert. Analog dazu verwendet die Division ein 128 Bit Register für den Dividenden, in dessen obere Hälfte die Mantisse des Dividenden geladen wird. Nach Division durch die 64 Bit Mantisse des Divisors erhält man als Ergebnis wieder einen 64 Bit langen Quotienten. Hier dient der Rest der Integerdivision als Zusatzinformation `remainder` für die Rundung.

Dies ist nur ein überschaubarer Überblick über die Funktionsweise der Softwaresimulation, neben dem genannten Vorgehen sind noch einige Sonderfälle zu berücksichtigen, Überläufe abzufangen und zusätzliche Bitshifts nötig um am Ende wieder eine Mantisse mit Fixkommastelle an Position 63 zu erhalten. Für weitere Details wie diese sei auf die Kommentare im Quellcode der Datei `longposit.c` verwiesen. Insgesamt wurden die Grundrechenarten (Addition, Subtraktion, Multiplikation und Division) sowie einige zusätzliche Funktionen (Quadratwurzel, Potenzierung und Negierung) implementiert, die vollständige Liste findet sich im Header `posit.h`. Da sich die Quadratwurzel und Potenzfunktion in keinem der evaluierten Löser wiederfindet, sind diese nicht ausgiebig getested oder optimiert worden und entsprechen wegen wiederholter interner Rundungsschritte wahrscheinlich auch nicht den Anforderungen des Standards.

Nach jeder Rechnung kümmert sich dann die Verpackungsfunktion `pack` darum, das Zwischenformat richtig in einen Posit zu kodieren. Diese ist in Listing 3.3 aufgeführt. Dabei ist das Verpacken umständlicher als das Entpacken, da hier noch eine korrekte Rundung garantiert werden muss. Zu Beginn muss die Lauflänge des Regimes ermittelt werden, und alle damit einhergehenden Sonderfälle berücksichtigt werden. So könnte das Regime zu lange für die Bitlänge sein, der Posit läuft damit über bzw. unter zu $posit_{\max}$ bzw. $posit_{\min}$. Weiter bestimmt die Regimelänge ob anhand der Mantissen- oder Exponentenbits gerundet werden muss.

Die Rundung selbst wurde mittels dreier Flags *Guard* G, *Round* R und *Sticky* S implementiert, die alle drei Bits um die Rundungsstelle an der der Bitstring abgeschnitten werden soll, beschreiben. Guard notiert dabei das letzte Bit links vor der Rundungsstelle, Round als Rundungsbedingung das erste Bit direkt nach der Rundungsstelle, und Sticky alle weiteren Bits rechts von Round. Dabei wird Sticky jedoch nur einmal auf > 0 evaluiert, es ist also nicht wichtig die genauen Bits nach Round darin abzuspeichern, sondern nur, ob welche davon auf Eins gesetzt sind. Das Äquivalent dieser Rundung im Dezimalsystem wäre, dass Round aussagt ob der Nachkommateil $\geq 0,5$ ist, Sticky verschärft mit Round zusammen die Bedinung auf $> 0,5$ wobei Guard nur die letzte Stelle vor dem Komma speichert. Guard wird benötigt um das Runden zur nächsten geraden Zahl zu implementieren, wie es der Posit-Standard verlangt. Ein Aufrunden findet immer dann statt, wenn die Bedingung $R \wedge (G \vee S)$ erfüllt ist, also bei einem Wert in der exakten Mitte zwischen zwei Zahlen ($R = 1, S = 0$) wird nur dann

```c
#define POSIT_MIN 1 /*Smallest posit with minimal regime 0...01 */
#define POSIT_MAX (1 << (BITLEN - 1)) - 1 /*Biggest posit with max regime 01...1*/
static inline posit_t pack(unpacked_t u, uint64_t remainder) {
    posit_t p = 0x00;
    if(u.mantissa == 0) return p;
    // Determine the regime length (without terminating bit)
    int32_t regime = u.exponent >> ES;
    uint16_t reglen = regime < 0 ? -regime : regime + 1;
    if(reglen >= BITLEN - 1) { // Check for over and underflow
        if(regime < 0) return complementIfNegative(POSIT_MIN, u.sign);
        else return complementIfNegative(POSIT_MAX, u.sign);
    }
    // Keep track of rounding in 3 bits: Guard | Round Sticky...
    uint8_t Guard = 0, Round = 0;
    uint64_t Sticky = remainder; // For rounding, only sticky > 0 matters
    posit_t exponent = _bzhi_u64(u.exponent, ES);
    uint64_t mantissa = _bzhi_u64(u.mantissa, MANTISSA_DP); // remove implicit leading 1
    int16_t mantissalen = BITLEN - 1 - ES - (reglen + 1); // reglen + terminator bit
    // shift: highest mantissa bit from MANTISSA_DP - 1 to mantissalen
    uint16_t mantissa_rem_len = (MANTISSA_DP - mantissalen);
    // Rounding: Round is the first bit of the remainder, round the rest.
    Sticky |= _bzhi_u64(mantissa, mantissa_rem_len - 1);
    Round = _bextr_u64(mantissa, mantissa_rem_len - 1, 1);
    mantissa >>= mantissa_rem_len;
    if(mantissalen >= 0)
        exponent <<= mantissalen;
    else {
        // there are no mantissa bits => get Round and Sticky from the exponent
        int8_t explen = BITLEN - 1 - (reglen + 1); // <= ES
        Sticky |= _bzhi_u64(exponent, ES - explen - 1);
        exponent >>= (ES - explen - 1);
        Round = exponent & 0b1;
        exponent >>= 1;
        Sticky |= mantissa; // combine conditions exp_remainder > 0 | mantissa > 0
        mantissa = 0;
    }
    // Run Length encode the regime
    posit_t regimeRLE = 0;
    if (regime < 0) {
        // place a terminating 1 at the end of leading zeroes
        regimeRLE = ((posit_t) 1) << (BITLEN - 2 + regime);
    } else {
        // create a series of leading ones of the right length and shift it into position
        posit_t rleOnes = ((((posit_t) 1) << (regime + 1)) - 1);
        regimeRLE = rleOnes << (BITLEN - 2 - regime);
    }
    p = regimeRLE | exponent | mantissa;
    if(Round) { // Round up conditionally
        Sticky = Sticky != 0 ? 1 : 0; // Evaluate sticky into a boolean
        Guard = p & 0b1; // Guard is the current LSB of the representation
        p += Guard | Sticky;
    }
    return complementIfNegative(p, u.sign);
}
```

Listing 3.3: Symbolische Posit-Verpackungsfunktion für einen generischen Posit mit Bitlänge `BITLEN` und Exponentenlänge `ES`.

aufgerundet wenn das Rundungsergebnis eine gerade Zahl wird, die Zahl vor der Rundung also ungerade ($G = 1$) ist.

In der Verpackungsfunktion werden diese drei Flags je nach Regimelänge von verschiedenen Stellen bezogen. Ist das Regime zu lange, sodass die Mantisse keinen Platz mehr findet, liegt die Rundungsstelle im Exponenten, weswegen Guard und ggf. auch Round hier aus dem Exponenten genommen werden müssen, und die Mantisse nur in Sticky berücksichtigt wird. Da jedoch in den allermeisten Fällen in der Mantisse gerundet wird, werden die Rundungsbedingungen am häufigsten aus den Mantissenbits genommen, dieser Fall tritt so oft auf dass er in der Implementierung in Listing 3.3 aus der Fallunterscheidung herausgenommen wurde. In jedem Fall werden die Zusatzinformationen der Rechenoperationen aus `remainder` nur zu den Sticky-Bits hinzugenommen, da selbst für die längstmögliche Mantisse (61 Bits für 64 Bit lange Posits) Guard und Round noch immer direkt in der 64 Bit Mantisse liegen würden. Nachdem das Regime Lauflängenkodiert und mit Exponent und Mantisse zusammengefügt wurde, wird schließlich die Rundungsbedingung auf dieses Endergebnisses angewendet.

Dieser Rundungsalgorithmus wurde ausgehend vom Standard [17] entwickelt und mit der Implementierung aus "universal" [26] als Orientierung für allgemeine Bitlängen und Exponentenlängen eines generischen Positformates erweitert.

Die Ent- und Verpackungsfunktion sind in C als `inline`-Funktionen definiert. Untersuchungen des ausgegebenen Assemblycodes zeigt auf, dass es der Compiler schafft das Zwischenformat `unpacked_t` in den Registern zu halten und auf weitere Speicherzugriffe zu verzichten. Das ist dahingehend hilfreich, dass es Vergleiche zwischen Posits und Floats vereinfacht, weil für Posits-Operationen genau wie für Floats nur die Operanden aus dem Speicher geladen werden müssen.

Die Generizität der Posit-Implementierung ist gegeben durch Präprozessorersetzungen. Die beiden Werte `BITLEN` (Bitlänge) und `ES` müssen dem Präprozessor übergeben werden, der anhand dieser einerseits die Logik anpasst, aber auch die Namen von Konstanten und Funktionen setzt. Alle Funktions- und Konstantennamen sind mittels eines Makros gesetzt, das ihnen allen ein Präfix der Form `p_BITLEN_ES_` verleiht, in dem `BITLEN` und `ES` durch die jeweiligen Werte ausgetauscht ist. Funktionen haben dieses Präfix in Kleinbuchstaben, Konstanten in Großbuchstaben. Diese makrogenerierten Namen werden ebenfalls im zugehörigen Header `posit.h` verwendet, was es ermöglicht, ihn mehrmals für verschieden definierte Werte von `BITLEN` und `ES` zu inkludieren, was die Basis der Mixed-Precision Implementierung bildet.

Um die getrennte Notation von Bitlänge und Exponentenlänge abzukürzen haben sich während der Entwicklung des Frameworks die Abkürzungen `posit<BITLEN,ES>` oder `positBITLEN,ES` etabliert, die von nun an verwendet werden und auch an vielen Stellen in den Quellcodekommentaren wiederzufinden sind.

Aufgrund ihrer Optimierung mit X86 Befehlssatzerweiterungen finden sich die Implementierungen der Posits im Verzeichnis `x86posit`, dies wird oft auch als Name der Positbibliothek verwendet. Neben der hier beschriebenen Implementierung findet sich in der Datei `posit_impl.c` noch eine veraltete Version, die nur bis zu Bitlängen 32-bit funktioniert und von deren Verwendung generell abgeraten wird.

Für die Softwaresimulation einer anderen Arithmetik sind zwei Eigenschaften unabdingbar: Eine schnelle Ausführung und fehlerfreies Rechnen. Mit der Garantie beider dieser Anforderung befassen sich die folgenden Abschnitte.

Tabelle 3.1: Millionen Operationen pro Sekunde (MPOPs) einger Positformate im Vergleich zu den MFLOPs von double. Die rechteste Spalte zeigt als einheitslose Größe das Verhälthis der double MFLOPs zu den Durschschnittlichen Posit MPOPs

	`posit16,2`	`posit32,3`	`posit64,4`	`double`	Posit Durchsch.	Verhält. Double/Posit
$+$	71,8	82,6	76,6	2072,0	77,0	26,9
$-$	66,5	76,2	75,9	2234,4	72,9	30,7
$\times$	70,3	70,0	75,5	2062,8	71,9	28,7
$\div$	57,7	59,6	63,7	751,9	60,3	12,5

3.2.1 Performance

Mithilfe einer Schleife die immer wieder dieselbe Operation auf zwei Operanden ausführt, wurde im Programm `positbench.c` für alle Operatoren die Zahl der Operationen pro Sekunde für ein generisches Posit Format ermittelt und gegen die FLOP/s Rate derselben Berechnung im Datentyp `double` verglichen. Auf einem Ryzen 5 2600 Prozessor der auf seine Basisfrequenz von $3,4$GHz fixiert wurde, ergaben sich bei 10^9 Tests für die exemplarischen Positformate `posit16,2`, `posit32,3`, `posit64,4` und Vergleichsformat `double` die in Tabelle 3.1 genannten Millionen Operationen pro Sekunde (MFLOPs/MPOPs).

Verglichen mit den Frameworks aus [25] kann diese Implementierung also gut mithalten, und bietet den zusätzlichen Vorteil schneller Kompilierung und eines generischen Formates. Die Tabelle zeigt dabei einige interessante Umstände auf: So sind Subtraktionen auf Posits immer leicht langsamer als die Additionen, da `sub` nur den zweiten Operanden negiert und danach `add` aufruft. Über die verschiedenen Bitlängen der Posits hin lässt sich kaum eine signifikante Änderung der Geschwindigkeit feststellen, da im Hintergrund immer mit den Mantissen in vollen 64 Bit Größe gerechnet wird. Dies könnte auch erklären, warum die kleineren Formate etwas langsamer sind, da hier der Compiler gegebenfalls noch Operationen einbaut um die Konvertierungen von und zu den kleinen Typen vorzunehmen. Bemerkenswert ist auch der Unterschied der OPs Rate bei der Division, die für beidermaßen Posits und Floats langsamer als andere Operationen sind, was der größeren Komplexität von Integer- und Float-Divisionen auf der CPU geschuldet ist. Jedoch ist der Verlust an Performance für Posits hier nicht so stark ausgeprägt wie für double, was die Posit Division relativ gesehen schneller als andere Operationen macht.

3.2.2 Korrektheit

Um die Korrektheit der Simulation der Zahlen zu prüfen, wurde in `posittest.cpp` ein Programm entwickelt, das für einen gegebenen Operator potenziell alle möglichen Berechnungen eines Formates simulieren kann und deren Ergebnisse mit einem Vergleichsformat abgleicht. Dazu wurde erstens ein Algorithmus entwickelt, der mit einem Vergleichsergebnis aus einem Float-Format (float, double oder quad) prüft, ob das ermittelte Posit-Ergebnis am nächsten am Float-Ergebnis liegt, d.h. es wird getestet ob der nächstgrößere oder -kleinere Posit näher am Float-Ergebnis liegt als es das tatsächliche Posit-Ergebnis tut. Diese ganze Prüfung ist etwas verkompliziert durch die verschiedenen Rundungsverhalten der beiden Zahlenformate, da einerseits Über- und Unterläufe abgefangen werden müssen, aber auch die angepasste Rundung in der "Twilight Zone" (vgl. Abschnitt 2.1.2) berücksichtigt werden muss.

Eine bessere Prüfung der Korrektheit bietet sich ein direkter Abgleich mit einer etablierten Posit-Softwaresimulation an. Dazu wurde die "softposit"-Bibliothek [18] verwendet. Diese bietet zwar nur eine beschränkte Menge an Positformaten an, namentlich `posit8,0`, `posit16,1` und `posit32,2`, dafür sind diese aber ausführlich getestet worden. Leider wird sie aber nicht mehr aktiv weiterentwickelt, weswegen einige Korrekturen am Code notwendig sind bevor er kompilieren kann. Installation, Anpassung und Kompilierung werden alle von den Hilfsskripten erledigt, die später noch genauer in Kapitel 3.7.1 gezeigt werden. Mithilfe dieser Bibliothek können die Ergebnisse der eigenen Posit-Implementierung noch zusätzlich zum Vergleich mit Floats abgeglichen werden. Um die gesteigerte Komplexität der vielen Vergleichstypen, Fällen und Operatoren etwas eleganter zu verwalten, wurde hier ausnahmsweise C++ verwendet, um dessen generische Templates ausnutzen zu können.

Die einzige Hürde daran, nun jedes Posit-Format umfassend zu Testen, ist die Zeitkomplexität des Programms. Da es für einen Posit der Bitlänge W insgesamt 2^W mögliche Kombinationen gibt, gibt es bei zwei Operanden also 4^W mögliche Berechnungen die getestet werden müssten. Aufgrund dieser exponentiellen Zeitkomplexität ist es praktisch unmöglich größere Positformate zu 100% zu testen. Auch wenn das Programm parallelisiert wurde und so mehrere Tests gleichzeitig pro Sekunde ausführen kann, so benötigt es auf dem im vorigen Abschnitt verwendeten Prozessor z.B. zum vollständigen Testen von `posit32,2` eine geschätzte Zeit von 3000 Jahren, für `posit64,5` sogar mehr als 500 Milliarden Jahre. Mithilfe einer anpassbaren Schrittweite lassen sich hier stichprobenartige Tests ausführen.

Mithilfe dieses Programmes wurden die Positformate `posit8,0` und `posit16,1` vollständig, und `posit32,2` mit Schrittweite 10^4 fehlerfrei getestet. Für größere Positformate blieb nur der Vergleich mit 128 Bit Floats, hier wurde `posit64,5` mit Schrittweite 10^{15}, was etwa $340 \cdot 10^6$ Stichproben entspricht, fehlerfrei getestet. Die getesteten Operatoren waren dabei neben den von Posit implementierten Rechenoperationen auch die Konvertierungsoperationen zu Float, Double und Quad. Diese beginnen für große Positformate wie zu erwarten Tests nicht zu bestehen, da durch die Konvertierung zu float oder double mit kürzeren Mantissen ein Informationsverlust entsteht.

3.3 Mixed-Precision

Um mehrere beliebige Posit-Formate in Lösern gemischter Genauigkeit zu verwenden, benötigt man ein System mit dem die Zahlentypen schnell ausgetauscht werden können. Aufgrund mangelnder generischer Typen und Templates in C, wird die Generizität mithilfe des Präprozessors verwirklicht.

Dazu werden neue Präfixes definiert: `wp` für "Work Precision", womit der Typ höherer Genauigkeit gemeint ist, und `rp` für "Reduced Precision" was den Typen verringerter Genauigkeit beschreibt. Nun gibt es im Verzeichnis `mixed-precision/number` eine Reihe von Headerdateien `*_num.h`, die nur aus Makros bestehen. Deren Aufgabe besteht daraus, Typnamen, Konstanten und Funktionsnamen wie `wp_add()` oder `rp_mul()` auf die Namen vom richtigen zugehörigen Datentypen abzubilden. Gesteuert wird jeder dieser Header mit drei Definitionen, z.B. `num_wp.h` erhält in `WP_BITLEN` und `WP_ES` die Bitlänge und Exponentenlänge des Formates sowie mit `WP_LIB` die ID des verwendeten Formates (1 für Posit und 2 für Float). Zum Beispiel wird `wp_add(a,b)` für die Längen `WP_BITLEN = 32` und `WP_ES = 2` dann abhängig des Formates bei `WP_LIB = 1` auf `p_32_2_add(a, b)`, und bei `WP_LIB = 2` auf `(a + b)` übersetzt.

Tabelle 3.2: Verfügbare Zahlenformate im Mixed-Precision-System.

	`_LIB`	`_BITLEN`	`_ES`
Posit	1	1..64	0..BITLEN-2
Half	2	16	5
Float	2	32	8
Double	2	64	11
Quad	2	128	15

In seiner Gänze unterstützt das Mixed-Precision System noch weitere Datentypen: Um IEEE-754 Floats der Größe 128 Bit zu unterstützen wird der vom GNU C Compiler unterstützte Datentyp `__float128` verwendet, für 16 Bit Half-Floats wurde in `mixed-precision/float16` aufgrund mangelnder passender verfügbarer C-Bibliotheken eine Softwaresimulation eigens implementiert. Diese funktioniert dabei in ihren Grundprinzipien so wie die Positsimulation, nur dass die Ent- und Verpackungsfunktionen anders sind, und die anderen Sonderfälle und Vergleichsoperationen sowie das andere Rundungsverhalten berücksichtigt werden. Genau wie die Posit-Softwaresimulation wurden auch die Half-Floats mit dem Testprogramm aus Abschnitt 3.2.2 erfolgreich auf ihre Korrektheit geprüft, und mit dem Programm aus Abschnitt 3.2.1 auf demselben Prozessor in ihrer Performance getestet, wobei sie über alle Operatoren ziemlich gleichmäßig 415 MFLOPs erreichen konnten. Als Überblick wird eine Liste aller unterstützter Formate zusammen mit den Parametern, die dem Präprozessor übergeben werden müssen, in Tabelle 3.2 geführt.

Zwischen all diesen Datentypen stellt das Framework Konvertierungsfunktionen bereit. Dem Nutzer direkt zugänglich sind dabei Konvertierungen zwischen Work- und Reduced-Precision (`wp_to_rp(a)`, `rp_to_wp(a)`), sowie zwischen jedem der IEEE Float-Typen und WP bzw. RP (`wp_from_float(a)`, `rp_to_double()`, etc. mit Abkürzungen `wp_ff(a)`, `rp_td`). Alle Typkonvertierung, sofern nicht nativ von der Hardware unterstützt, erfolgt über das gemeinsame Zwischenformat `unpacked_t`. Das Verzeichnis `mixed-precision/packing` stellt die Ver- und Entpackungsfunktionen aller Float-Datentypen bereit, während die Posit-Implementierung ihre `inline` Packfunktionen mittels eines Wrappers ebenfalls im globalen Namensraum verfügbar macht. Wie alle anderen Operatoren wird die Konvertierung ebenfalls per Makroersetzung bereitgestellt, indem der Präprozessor die Ent- und Verpackungsfunktion der richtigen zugehörigen Typen verkettet.

Da die Header sowieso mit ihren Makros schon für jede Rechenoperation eine Ersetzung vornehmen, wurde das Zählen der Rechenoperationen hier gleich mit eingebaut. Mithilfe des Komma-Operators von C, der besagt, dass bei einer Folge kommaseparierter Operanden alle evaluiert werden aber nur das Ergebnis des letzten zurückgegeben wird, wird also bei obigem Beispiel für `wp_add` stattdessen `(p_32_2_count_operation(), p_32_2_add(a,b))` zurückgegeben, was denselben Effekt hat, aber währenddessen noch den Zähler eines globalen Structs inkrementiert. Dies geschieht hier über Funktion um den Anteil von WP und RP Operationen getrennt zählen zu können. Neben Operationen werden noch andere Metriken während der Programmausführung gezählt, für die ebenfalls Makroersetzungen ausgenutzt werden.

Wie bereits aus dieser Beschreibung hervorgeht, unterscheiden sich die Header die WP und RP bereitstellen nicht stark in ihrer Funktion, tatsächlich sogar nur in den Präfixen die sie den Funktionsnamen voranstellen. Da generische Makros die Kapazität des C-Präprozessors überschreiten, diese Redundanzen allerdings doch ziemlich umständlich sind, wenn man kleine

Änderungen vornehmen möchte, werden die Header `*_num.h` mithilfe des GNU M4 Präprozessors aus der Datei `num.m4` erzeugt. Das Shellskript im selben Verzeichnis kümmert sich dabei um die richtigen Aufrufe des von M4.

Für die Anwendungsprogramme, die das Mixed-Precision Framework nutzen, bedeutet das, dass nur der Header `mp.h` inkludiert werden muss, um Berechnungen auf den Typen `wp_num` und `rp_num` auszuführen. Diese werden dann zur Kompilierzeit vom Präprozessor durch die richtigen Funktionsaufrufe ausgetauscht, je nachdem welche Datentypen für die WP und RP gewünscht sind. Darum, dass zusätzlich zu den richtigen Funktionsnamen in den Headern auch mit den richtigen Objektdateien gelinkt wird, kümmern sich die Hilfsskripte die in Abschnitt 3.7.1 beschrieben werden.

3.4 Algebra- und BLAS-Funktionen

Um die im Theoriekapitel beschriebenen Löser zum Implementieren, benötigt man noch einige numerische Algorithmen. Um allgemein das Arbeiten mit mehrdimensionalen Größen in Form von Vektoren zu vereinfachen, wurden die BLAS-Routinen `axpy`, `scal`, `copy`, `asum` und `gemv` implementiert. Der Code entstand in Anlehnung an die Lapack Bibliothek [27], deren Algorithmen von Fortran zu C übertragen wurden und an das Mixed-Precision System angepasst wurden. Dabei wurde versucht die Funktionssignaturen zu bewahren, mit Außnahme des Präfixes `s`,`d` das den Datentyp notiert, da stattdessen eine eigene Notation zur internen Identifizierung des Datentypen, ähnlich der Funktionsnamen der Posits, verwendet wird.

Neben diesen simplen Vektor- und Matrixoperationen benötigen die Impliziten Runge-Kutta-Verfahren noch einen Löser für das implizite Gleichungssystem 2.7. Eine gängige Option dafür ist das Newton-Raphson-Verfahren [28], welches ein iteratives Verfahren ist, das mithilfe der Ableitung die Nullstelle einer Funktion approximieren kann. Für eine vektorwertige Funktion $g : \mathbb{R}^n \to \mathbb{R}^n$ kann es die Lösung von $g(x) = 0$ annähern, indem iterativ die folgende Berechnungsvorschrift für N Schritte wiederholt wird:

$$x^{(i+1)} = x^{(i)} - J_g^{-1}\left(x^{(i)}\right) \cdot g\left(x^{(i)}\right), \quad \forall 0 \leq j \leq N : x^{(j)} \in \mathbb{R}^n \tag{3.1}$$

Dabei beschreibt $J_g^{-1}(x^{(i)})$ die inverse Jakobimatrix von g, ausgewertet am Ergebnis des vorherigen Schrittes $x^{(i)}$. Das Newton-Rasphson-verfahren konvergiert quadratisch mit jeder Iteration näher an das Ergebnis, Gl. 3.1 wird also so oft wiederholt bis die Differenz $||x^{(i+1)} - x^{(i)}|| < \xi$ eine gewisse Schwelle ξ unterschreitet, oder das Verfahren eine maximale Anzahl an Wiederholungen N überschreitet. Wichtig ist auch, dass initial ein Startwert $x^{(0)}$ gewählt werden muss, von dem das Verfahren aus nach der Nullstelle sucht.

Da das Berechnen einer Matrixinverse in Gl. 3.1 sehr umständlich ist, bietet es sich eher an die Gleichung umzuformulieren und stattdessen das sich so ergebende lineare Gleichungssystem zu lösen:

$$J_g\left(x^{(i)}\right) \cdot \left(x^{(i)} - x^{(i+1)}\right) = g\left(x^{(i)}\right) \tag{3.2}$$

Gleichungen der Form $A \cdot x = b$, wobei $A \in \mathbb{R}^{n \times n}$ eine Matrix und $b, x \in \mathbb{R}^n$, lässt sich relativ einfach algorithmisch nach x auflösen. Das bekannteste Verfahren dafür ist das Gaußverfahren [28], das auch in dieser Arbeit implmentiert wurde.

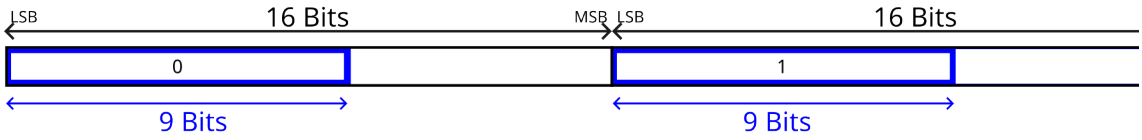


Abbildung 3.1: Speicherbelegung eines unkomprimierten Arrays von 9 Bit Posits in ihren 16 Bit Containern.

Bevor das lineare Gleichumssystem aus 3.2 gelöst werden kann, benötigt man die Jacobimatrix J_g der Funktion $g(x)$. Falls die analytische Ableitung der Funktion g nicht vorliegt, so kann die Jacobimatrix auch approximiert werden. Dieses Framework unterstützt dabei die Methode der Zentralen Differenzen [28]. Die Ableitung einer eindimensionalen Funktion $f : \mathbb{R} \to \mathbb{R}$ wird approximiert als

$$\frac{\partial f}{\partial x} \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{2\Delta x} \tag{3.3}$$

wobei $\Delta x > 0$ eine kleine Konstante ist, die die Schrittweite beschreibt auf der f approximiert wird.[28] Dies lässt sich auf eine vektorwertige Funktion $g : \mathbb{R}^n \to \mathbb{R}^n$ erweitern, indem man die Formel auf jedes Element der Jacobimatrix anwendet

$$\begin{gathered} J_g(x) = \left[\frac{\partial g(x)}{\partial x_1} \cdots \frac{\partial g(x)}{\partial x_n}\right]; \quad \frac{\partial g(x)}{\partial x_i} = \begin{bmatrix} \frac{\partial g_1(x)}{\partial x_i} \\ \vdots \\ \frac{\partial g_n(x)}{\partial x_i} \end{bmatrix}; \\ \forall i, j \in \{1, \ldots, n\} : \frac{\partial g_j}{\partial x_i} \approx \frac{g_j(x_1, \ldots, x_i + \Delta x, \ldots, x_n) - g_j(x_1, \ldots, x_i - \Delta x, \ldots, x_n)}{2\Delta x} \end{gathered} \tag{3.4}$$

Die Implementierungen der Algorithmen `newton_raphson`, `gauss` und `approx_jacobian` liegen zusammen mit den BLAS-Funktionen in `algebra.c` im Verzeichnis `mixed-precision/algebra`. Mittels eines ähnlichen Systems aus Headern und Makros werden sie für beide Genauigkeiten WP und RP kompiliert und mit `mp.h` eingebunden.

3.5 Komprimierte Vektoren

Da Posits beliebiger Größe untersucht werden sollen, und für Posits in der Softwaresimulation immer der Integerdatentyp mit der Bitlänge der nächstgrößeren Zweierpotenz als Container dient, fallen potenziell viele ungenutzte Bits an Speicher an. Bits die nicht nur zusätzlichen Hauptspeicher benötigen, sondern auch die Speicherbandbreite unnütz mitverbrauchen und wertvollen Platz in den Caches verschwenden. Im schlimmsten Falle, für Posits der Bitlänge $W = 33$, die als Container einen 64bit Integer verwenden, beträgt der Anteil verschwendeter Bits sogar $31/64 \approx 48,44\%$. Graphik 3.1 zeigt exemplarisch diesen verschwendeten Speicher in einem Array aus `posit9,x`

Um dieser Verschwendung entgegenzuwirken, wurde ein System *vector* entwickelt, welches Vektoren in den numerischen Algorithmen ersetzt und eine Kompaktierung wie in Grafik 3.2 gezeigt unterstützt. Mithilfe von Bitshifts wird der Leerraum der Container somit für den

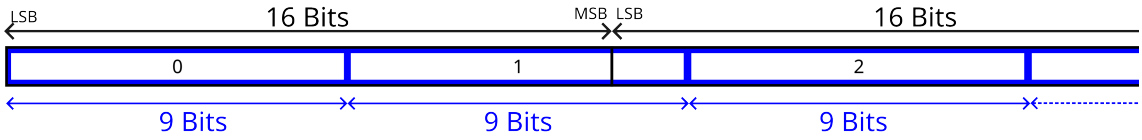


Abbildung 3.2: Speicherbelegung eines komprimierten Arrays von 9 Bit Posits in ihren 16 Bit Containern.

jeweils nächsten Posit des Vektors verwendet, die Posits rücken sozusagen auf und überlappen dadurch möglicherweise zwei Container. Durch dieses Vorgehen wird der verschwendete Speicher von ungenutzten Bits in jedem Container reduziert auf lediglich das Padding nach dem letzten Element des Vektors, sofern die Gesamtbitzahl des Arrays kein Vielfaches der Containerlänge ist.

Da aufgrund der möglichen Teilerfreiheit von Bitlänge und Containerlänge lange Ketten von Posits die stets zwei Speicherzellen überlappen entstehen können, Computer ihren Speicher aber zumeist in Blöcken verwalten, wurde dem Komprimierungsverfahren eine Blockgröße l_B hinzugefügt. Diese garantiert, dass nach l_B Containern das nächste Arrayelement wieder am Anfang eines Containers ausgerichtet ist. Dies bedeutet zwar etwas zusätzlichen Speicherverlust, begünstigt allerdings eine Verwendung dieser komprimierten Vektoren im Shared Memory. So könnte man z.B. Cache-Invalidierungen infolge von False Sharing vorbeugen, indem man die Blockgröße gleich der Länge einer Cache-Line setzt. In allen Betrachtungen dieser Arbeit wurde stets eine Blockgröße von 64 Byte gewählt. Diese Designentscheidung war jedoch eher zukunfsgerichtet gedacht, da dieses Verhalten in den Untersuchungen aufgrund mangelnder Parallelität nicht zum Vorschein tritt.

Im Quellcode ist diese gesamte Logik in der Datei `compressed_vector.c` im `algebra` Ordner. Die Quellcodeabschnitte 3.4 und 3.5 zeigen dabei auf, wie die Kompaktierung der Lese- und Schreibezugriffe auf den Vektor implementiert wurde. Die Berechnung der Indizes wurde dabei aufgrund der Gleichheit in ein Makro ausgelagert. Um die Position eines Elements i der Bitlänge `_BITLEN` in einer Reihe von Containern der Größe `_CONTAINERLEN` zu lokalisieren, muss zuerst der Index des Blocks berechnet werden, danach der Index des Elements im Block. Hieraus kann man den Index des Containers in Block berechnen, in dem der abgelegte Posit anfängt, welcher zusammen mit dem Blockindex als `array_index` die Position dieses Containers im Gesamtarray angibt. Zuletzt wird die Bitposition in diesem Container ermittelt, an der der abgespeicherte Posit beginnt. Hier ist noch darauf zu achten, dass die Elemente sich von kleinem Bitindex zum größeren erstrecken, in den Abbildungen 3.1 und 3.2 ist dies bereits bedacht, indem die einzelnen Container entgegen der Konvention umgedreht dargestellt sind, das MSB ist also stattdessen an der Position des LSB.

Die Lese- und Schreibvorgänge selbst erfolgen abhängig davon, ob das Element in einem Container liegt oder zwei überlappt. Bei Schreiben muss noch zusätzlich auf eine richtige Maskierung geachtet werden, sodass keine falschen Bereiche überschrieben werden, und keine vorherigen Werte in dem zu schreibenden Element unbeabsichtigt bestehen bleiben. Einen weitere Abschnitt den sich beide Funktionen teilen ist die Präprozessorfallunterscheidung, die für den Fall, dass die Elementlänge gleich der Containerlänge ist, den gesamten Komprimierungsteil überspringt. Eigentlich sollte dieser, um seinen Zusatzaufwand bemessen zu können, in jedem

```c
// Macro to determine the main indices needed for accessing (reading and writing)
#define GET_ARRAY_INDICES(i) \
    uint32_t block_index = (i) / ELEMENTS_PER_BLOCK;\
    uint32_t element_index = (i) % ELEMENTS_PER_BLOCK;\
    /*Array index of the container in the target block*/\
    uint32_t container_index = (element_index * _BITLEN) / _CONTAINERLEN;\
    /*Bit position inside the container*/\
    uint32_t internal_index = (element_index * _BITLEN) % _CONTAINERLEN;\
    uint32_t array_index = block_index * CONTAINERS_PER_BLOCK + container_index;
inline _num FN_PREFIX(v_get)(FN_PREFIX(v_t)* vec, size_t i) {
    MEMORY_COUNT(_BITLEN, 1, statistics.read_bits); // counts statistics
    #if _BITLEN == _CONTAINERLEN // Necessary because float/double cannot be bitshifted
    return vec->baseptr[i];
    #else
    GET_ARRAY_INDICES(i)
    _num out = vec->baseptr[array_index] >> internal_index;
    if(internal_index > _CONTAINERLEN - _BITLEN) {
        // bits of this element have spilled from this container into the next one
        out |= vec->baseptr[array_index+1] << (_CONTAINERLEN - internal_index);
    }
    return _bzhi_u64(out, _BITLEN);
    #endif
}
```

Listing 3.4: Symbolische Posit-Entpackungsfunktion für einen generischen Posit mit Bitlänge `BITLEN` und Exponentenlänge `ES`.

Fall mit eingeschlossen sein, jedoch lässt sich der Code mit Bitshifts auf den Datentypen float und double, die ebenfalls die Vektoren benutzen, nicht kompilieren, weswegen dieser Abschnitt unter dieser Bedingung ausgeschlossen wird.

Um einerseits einen Vergleich zu naiven Arrayzugriffen zu haben und andererseits den Zusatzaufwand durch den Funktionsaufruf bzw. des Inlinings bemessen zu können, stehen zwei weitere Varianten mit diesen Implementierungen als Alternative zu den komprimierten Arrayzugriffen zur Verfügung. Im Allgemeinen werden alle Zugriffe auf Elemente von Vektoren, die zuvor noch durch Arrayzugriffe implementiert waren, nun nur noch durch Makros getätigt. Diese werden vom Präprozessor im Rahmen des Mixed-Precision Systems durch den entsprechenden Funktionsaufrufe oder direkte Arrayzugriffe der Form `array[idx]` substituiert.

Dafür gibt es wieder eine Reihe von mit M4 generierten Headerdateien `*_vec.h`, die die Zugriffsmethoden für beidermaßen WP und RP bereitstellen und nach Einstellung auf die richtigen Funktion im Hintergrund aliasen. Sie stellen die Vektortypen `wp_num_v_t` und `rp_num_t`, sowie die Lese- und Schreibfunktionen `*_v_get()` und `*_v_set()` zur Verfügung, und analog dazu noch einen Typ für Matrizen `*_num_m_t`, der dieselben Zugriffsfunktionen nur mit zwei Indizes anbietet und dieselben komprimierten und nichtkomprimierten Varianten anbietet. Neben den Zugriffsmethoden werden ebenfalls Makros zur Initialisierung bereitgestellt, die Allokation auf Stack und Heap unterstützen. Da die komprimierten Vektoren auf einem Pointer basieren wurde hier mithilfe zusätzlicher Hintergrundvariablen eine Stackallokation nachempfunden, um sie nicht mit dem zusätzlichen Nachteil eines `malloc` Systemaufrufs zu belasten. Gesteuert wird die Variante der verwendeten Implementierung über den Parameter `vector_variant` in den Build-Skripten, die sich um das Linken mit der richtigen Objektdatei

```
inline void FN_PREFIX(v_set)(FN_PREFIX(v_t)* vec, size_t i, _num val) {
    MEMORY_COUNT(_BITLEN, 1, statistics.write_bits); // counts statistics
    #if _BITLEN == _CONTAINERLEN
    vec->baseptr[i] = val;
    #else
    val = _bzhi_u64(val, _BITLEN);
    GET_ARRAY_INDICES(i)
    if(internal_index <= _CONTAINERLEN - _BITLEN) {
        // the element ends before the end of the container => mask in only the leading
        ↪ `internal_index` bits and the ones following after the end of the element
        _num mask = ~(((1ul << BITLEN) - 1) << internal_index);
        vec->baseptr[array_index] = (vec->baseptr[array_index] & mask) | (val <<
        ↪ internal_index);
    }
    else {
        // mask out the upper bits of the lower register
        vec->baseptr[array_index] = _bzhi_u64(vec->baseptr[array_index], internal_index) |
        ↪ (val << internal_index);
        // insert the bits that overflowed the register to the left into the next register
        // mask out the lower _BITLEN - (_CONTAINERLEN - internal_index) bits
        uint32_t lower_len = _CONTAINERLEN - internal_index;
        uint32_t upper_len = _BITLEN - lower_len;
        vec->baseptr[array_index + 1] = ((vec->baseptr[array_index + 1] >> upper_len) <<
        ↪ upper_len) | (val >>  lower_len);
    }
    #endif
}
```

Listing 3.5: Schreibefunktion für komprimierte Vektoren.

kümmern. Da die Vektorvariante nicht in den Funktionsnamen der generischen Funktionen enthalten sind, können diese für verschiedene Varianten vom Linker nicht unterschieden werden, es kann also immer nur eine Variante für das gesamte Programm gewählt werden, und nicht für RP oder WP separat.

3.6 Löser

Um die Eignung von Posits als Datentypen für MP Solver zu prüfen wurden mehrere Löser implementiert. Neben den in Abschnitten 2.4 und 2.5 vorgestellten Verfahren wurden noch ein explizites Eulerverfahren und das klassische RK4-Verfahren programmiert, die hauptsächlich als Vergleichsmaß und für Referenzergebnisse herangezogen werden. Alle Löser des Frameworks befinden sich im Verzeichnis `solver`. Verwaltet werden alle Solver in einer Liste von Structs, von denen jedes Element Informationen über den Solver und einen Funktionszeiger zur Lösungsfunktion speichert, was ein dynamisches Auswählen des zu verwendenden Solvers zur Laufzeit ermöglicht. Im Allgemeinen sind die Implementierungen der Solver nur direkte Übertragungen der Berechnungsvorschriften zu Code unter der Verwendung der Mixed-Precision Datentypen und Algebrafunktionen, es folgen also nur spezifische Einzelheiten zu den zwei wichtigen Solvern:

3.6.1 Implizite Mittelpunktsregel

Der Löser zur impliziten Mittelpunktsregel arbeitet das Schema aus Gleichung 2.12 ab. Zum Lösen des nichtlinearen Gleichungssystems $\kappa^{(1)} = u^{(n)} + \frac{h}{2} \cdot f(\kappa^{(1)})$ auf verringerter Genauigkeit wird das Newton-Raphson-Verfahren aus Abschnitt 3.4 verwendet, um die Nullstelle der Funktion $g(\kappa^{(1)}) = u^{(n)} + \frac{h}{2} \cdot f(\kappa^{(1)}) - \kappa^{(1)} = 0$ zu finden. Hierfür benötigt man die Jacobimatrix J_g von g, welche sich, wenn die Jacobimatrix J_f von f bekannt ist, analytisch berechnen lässt.

$$
\begin{aligned}
J_g\left(\kappa^{(1)}\right) &= \left[\frac{\partial g_j}{\partial \kappa_i^{(1)}}\right]_{i,j=0}^{d} \\
\frac{\partial g_j}{\partial \kappa_i^{(1)}} &= \frac{\partial}{\partial \kappa_i^{(1)}}\left(u_j^{(n)} + \frac{h}{2} f_j\left(\kappa^{(1)}\right) - \kappa_j^{(1)}\right) = \frac{h}{2}\frac{\partial f_j}{\partial \kappa_i^{(1)}}\left(\kappa^{(1)}\right) - \begin{cases} 1 & \text{falls } i = j \\ 0 & \text{falls } i \neq j \end{cases} \\
\implies J_g\left(\kappa^{(1)}\right) &= \frac{h}{2} J_f\left(\kappa^{(1)}\right) - I_d
\end{aligned}
\tag{3.5}
$$

I_d beschreibt die Identitätsmatrix derselben Dimension $d \in \mathbb{N}^+$ der ODE.

Nachdem in jedem Zeitschritt das implizite System gelöst ist, werden die von Grant [1] vorgeschlagenen Korrekturschritte durchgeführt. Der Parameter p der deren Anzahl bestimmt, kann vom Nutzer dem Solver übergeben werden. Mit der Wahl $p = 0$ und `WP=RP` wird der Solver zur gewöhnlichen Mittelpunktsregel ohne gemischter Genauigkeit.

3.6.2 RKC1

Als explizites Verfahren benötigen die Runge-Kutta-Chebyshev-Verfahren kein Newton-Raphson-Verfahren, sondern können ihre Koeffizienten explizit berechnen. Trotzdem sind noch einige zusätzliche Funktionen notwendig.

So werden die Werte des Chebyshev-Polynoms (Gl. 2.15 und 2.17) mithilfe von Memoization berechnet, anstatt das rekursive Verfahren direkt auszuführen, was aufgrund seiner exponentiellen Zeitkomplexität sehr langsam wäre.

Eine direkte Abbildung der Rechenvorschrift aus 2.19 war in diesem Falle nicht eindeutig möglich, da aus der Publikation nicht deutlich hervorgeht welche der Funktionsauswertungen in verringerter, und welche in höherer Genauigkeit ausgeführt werden sollten. Da es keinen veröffentlichten Programmcode gibt und auch keine detaillierteren zitierenden Publikationen existieren, wurde die Berechnungsvorschrift so gut es ging an die existierenden Beschreibungen des Schemas angepasst.

Das Paper zu den RKC-Methoden sieht vor, dass die Stufenzahl s des Lösers genauso gewählt werden soll, dass der Stabilitätsbereich groß genug wird um die Steifigkeit der betrachteten ODE zu enthalten, was mit in Abschätzungen möglich ist [2]. Da diese jedoch mit einem enormen Zusatzaufwand an Numerischen Verfahren (z.B. zur Ermittlung des Spektralradius' der Matrix der ODE) verbunden wären, wurden in allen Auswertungen die Stufenzahlen aus den Beispielen der Publikation [2] als Konstante übernommen.

Die Implementierung unterstützt drei verschiedene Varianten wie der Löser die gemischte Genauigkeit verwendet. Mit der ersten lässt sich die gemischte Genauigkeit ausschalten, sodass

der Löser nur mit dem WP-Datentypen rechnet, also das Schema aus 2.13 abarbeitet. Als zweite wird ein "naiver" Ansatz verwendet, der die Funktionsauswertungen von f in reduzierter Genauigkeit durchführt, für den die Konvergenzbedingung nicht mehr garantiert wird. Mit der dritten Option werden die Zwischenwerte wie in Gl. 2.19 gezeigt in gemischter Genauigkeit so bestimmt, dass die Konvergenzordnung bewahrt wird [2]. Intern wird die Wahl der Variante über den Wert in p, der eigentlich nur für die Anzahl der Korrekturschritte aus ImpMid verwendet wird, gesteuert.

3.7 Programme und Hilfsskripte

Im Verlauf der vorigen Kapitel wurden bereits einige der Programme die das Framework nutzen vorgestellt. Neben diesen Testprogrammen für die Posit-Zahlenformate gibt es noch weitere ausführbare Einstiegspunkte in das Framework. Um das Kompilieren, Ausführen und Auswerten dieser Programme zu beschleunigen, wurde eine Reihe von Python-Skripten entwickelt, die sich im Verzeichnis `scripts` im Repository finden.

3.7.1 Buildsystem

Da das Framework stark vom Präprozessor abhängig ist, der viele Argumente zur Kompilezeit benötigt, was manuelle Kompilation und Aufrufe kompliziert macht, ist in den Skripten ebenfalls der Build-Vorgang mit enthalten. Auch wenn die Wahl für Gnumake hier naheliegend scheint, fiel die Entscheidung auf ein vollständiges Buildsystem mittels Kompileraufrufen aus Python, um genauere Kontrolle über den Vorgang zu bekommen. Ein großes Ziel bei der Entwicklung war vor allem ein schneller Kompileprozess, weswegen versucht wurde so gut wie möglich unnötige Neukompilierungen von bestehenden Bestandteilen des Programms zu vermeiden. Im Grunde gehen die Skripte wie folgt vor:

Innerhalb des Buildsystems gibt es für jede Teilbibliothek des Frameworks (Zahlen, Algebra, Vektoren und Hilfsbibliotheken) eine Subklasse, deren Funktionen beschreiben wie diese Bibliothek kompiliert wird, wie sich deren Zielpfad ergibt, und welche Flags sie für den Linkingprozess voraussetzt. Dabei können diese Klassen entweder allgemeine Bibliotheken beschreiben, oder welche die in Abhängigkeit eines Zahlenformates verschiedene Ergebnisse liefern, wie z.B. die Positimplementierung, die abhängig von Bit- und Exponentenlänge speziellen Code erzeugt. Für diese komplexeren Bibliotheken wird das Zahlenformat, mit dem sie kompiliert wurden im Dateinamen mitgespeichert, um die erzeugte Objektdatei wiederverwenden zu können, sollte später noch einmal dieselbe Bibliothek vom selben Zahlenformat benötigt werden. Da in Bibliotheken aber noch mehr Parameter als das Zahlenformat einfließen, die sie potentiell inkompatibel mit der Variante machen, die tatsächlich benötigt wird, werden diese ebenfalls in den erzeugten Objektdateien kodiert. Mithilfe von `objdump` wird den Dateien eine Sektion für Metadaten angehägt, in denen die Textform der Schlüssel-Wert-Paare der Zusatargumente gespeichert wird. Wird nun eine Bibliothek erneut benötigt, so wird aus der vorher erzeugten Datei erst die Sektion extrahiert und mit den Anforderungen verglichen, stimmen diese nicht überein, so wird neu kompiliert.

Diese Bibliotheken werden im nächsten Schritt zu Programmen gelinkt. Dabei hat jedes Programm (Posittest, Positbenchmark, Positode, etc.) ebenfalls eine Subklasse die den Kompilierungs- und Verlinkungsprozess spezifiziert. Die kompilierten Bibliotheken werden aus der `LibraryRegistry`

Tabelle 3.3: Alle möglichen Eingabeargumente des Positode-Programms

Option	Datentyp	Beschreibung
`--solver`	Enum	Wahl des Solvers, Optionen: (explicit-euler, explicit-rk4, implicit-midpoint, 1op-rkc1)
`--problem`	Enum	Wahl des AWPs, Optionen: (linear-lambda, arenstorf, vander-pol, heat2d, brusselator)
`--h/-h`	Double	Schrittweite die das Verfahren verwendet
`--t-range/-t`	Double	Differenz des Zielzeitpunkts zum Anfangszeitpunt $t_N - t_0$
`--corrections/-p`	Integer	Anzahl der Korrekturschritte wenn Solver implizites MP-RKV, MP-Variante wenn Solver MP-RKCV ist
`--repetitions/-r`	Integer	Optionale Anzahl wie oft der Solver ausgeführt werden soll.
`--solver-args`	String	Zeichenfolge aus `schlüssel=wert` Paaren mit Zusatzargumenten für den spezifischen Solver
`--problem-args`	String	Analog zu `solver-args` kodiert, aber mit Argumenten für das AWP
`--output/-o`	Pfad	Optionaler Dateispeicherort für die Ausgabe des Ergebnisses des Solvers, Binär serialisiert statt in Texform wie die Standardausgabe
`--reference-solution`	Pfad	Pfad zu einer Vergleichslösung zu der der Fehler, bzw. die Abweichung berechnet werden soll
`--error-method`	Enum	Auswahl der Fehlerberechnungsmethode, Optionen: (mse, euclid, mean, max, max_all, mean_all)

bezogen, die mithilfe von asynchroner Python-Funktionen die Teilbibliotheken parallel kompiliert. Zur Ausführung definieren die Klassen noch eine zusätzliche Funktion in der Eingabeargumente richtig übergeben werden und Ergebnisse entsprechend dekodiert werden. Genau wie bei den Bibliotheken existiert für die Programme auch der Wiederverwendungsmechanismus, generische Argumente werden also im Namen oder in den Metadaten abgespeichert.

Das wichtigste Programm das von Buildsystem kompiliert wird, ist wohl:

3.7.2 Positode-Programm

Das Programm `positode` ist der Haupteinstiegspunkt des Frameworks in die Solver. Die Aufgabe des C-Programms ist der Aufruf eines ausgewählten Solvers für eines der unterstützen Anfangswertprobleme, für eine bestimmte Schrittweite und Zielzeitpunkt. Die Übergabe der Parameter für die verwendeten Zahlenformate WP und RP und das Linken gegen die zugehörigen Bibliotheken geschieht, wie für die anderen Programme auch, durch das Buildsystem.

Zu Programmstart parst `positode` die Eingabeargumente mithilfe von GNU `getopt`. Tabelle 3.3 gibt eine Übersicht über die verfügbaren Argumente. Neben den erwarteten Einstellungen für Solver, AWP, Schrittweite, Zielzeitpunkt und Korrekturschritte bietet das Programm noch weitere Optionen an. So lässt sich optional eine Anzahl an Wiederholungen, wie oft der Solver ausgeführt werden soll, setzen, um bei einer sehr kurzen Ausführungszeit einen besser gemittelten Durchschnittswert zu bekommen. Auch optional sind die Zusatzargumente für die Solver und AWPs, mithilfe derer Solver- bzw. Problemspezifische Argumente an das Programm

übergeben werden können, die in der allgemeinen Programmschnittstelle nicht enthalten sind. Diese erwarten einen speziell formatierten String, in dem eine Liste von Schlüssel-Wert Paaren in der Form `schlüssel1=wert1&schlüssel2=wert2` usw. abgelegt ist und übersetzen diesen in ein Dictionary, das mithilfe eines sortierten Binärbaums implementiert wurde. Aus diesem können nun Solver und Probleme bei ihrer Initialisierung die Argumente entnehmen, die sie verwenden können.

Für die Ausführung des Lösers gibt es zwei Modi: Einen der nur das Endergebnis nach dem letzten Zeitschritt zurückgibt, und einer der die Ergebnisse aller Zeitschritte speichert und ausgibt. Zweiteres muss zur Kompilezeit mit der Flagge `SOLVER_STEP_VALUES` an den Präprozessor aktiviert werden.

Ebenfalls übernimmt das Programm die Fehlerberechnung, d.h. es berechnet die Abweichung von einem Referenzwert. Für die spätere Evaluation wird als Quelle für diesen Referenzwert immer das Ergebnis eines genaueren Solvers mit einer kleineren Schrittweite dienen, prinzipiell wären aber auch die Übergabe von analytischen Ergebnissen möglich, sofern diese existieren. Da die Referenzwerte entweder Vektoren sind, wenn man die Abweichung nach dem letzten Schritt betrachtet, oder Listen von Vektoren, wenn man den globalen Fehler berechnen möchte, benötigen diese vor allem für große Differentialgleichungssysteme schnell viel Speicher, weswegen sie binärkodiert in Dateien geschrieben werden und nur ein Dateipfad zur Referenz übergeben wird.

Um mit einem Aufruf von `positode` eine solche Datei zu erzeugen, um das Ergebnis des Lösers später als Referenz zu verwenden, muss beim Aufruf im Argument `--output/-o` ein Dateipfad angegeben werden, an dem die Lösung geschrieben werden soll. Nun kann derselbe Pfad bei einem anderen Aufruf von Positode im Argument `--reference-solution` mitgegeben werden. Zusammen mit der Angabe einer Fehlerberechnungsmethode durch `--error-method` bestimmt das Programm nach dem Fertigwerden des Solvers die Abweichung seines Ergebnisses. Die Tabelle 3.4 gibt einen Überblick über die verfügbaren Methoden. Allerdings nimmt diese vereinfachenderweise an, dass sich die Lösung und Referenzlösung dieselbe Schrittzahlen $N = \tilde{N}$ teilen, was natürlich in der Realität meist nicht der Fall sein sollte. Das bedeutet aber auch nur, dass das Programm für die Berechnung der globalen Fehler für den Zeitschritt j den richtigen korrespondierenden Zeitschritt $\tilde{j}$ der Referenzlösung finden muss, sodass $hj = \tilde{h}\tilde{j}$, wenn $\tilde{h}$ die Schrittweite der Referenzlösung ist, sodass $t_j = \tilde{t}_{\tilde{j}}$. Für die anderen Fehlertypen wird in jedem Fall einfach das Ergebnis des letzten Zeitschrittes verglichen. Dieses Vergleichsverfahren hat logischerweise die Voraussetzung, dass sich Lösung und Referenz im betrachteten Zeitbereich $t_N = t_{\tilde{N}}$ gleichen. Hier sei noch betont, dass für die globale Fehlerberechnung beide Solver mit Speicherung der Schrittwerte (Präprozessor `SOLVER_STEP_VALUES`) ausgeführt werden müssen.

Zuletzt kümmert sich das Programm noch um die Ausgabe. Dabei werden neben des Ergebnisses des Solvers und der Fehlerberechnung noch Statistiken wie die Ausführungszeit, Anzahl der durchgeführten Rechenoperationen, Arrayzugriffe und Allokationen ausgegeben. Da diese potenziellen zusätzlichen Overhead bedeuten, kann sie der Präprozessor mit der Flagge `SUPPRESS_OPERATION_COUNTING` aus dem Programm auszuschließen, um die Ausführung zu beschleunigen wenn keine Statistiken benötigt werden.

Tabelle 3.4: Methoden zur Berechnung der Abweichung zwischen den Schritten der Lösung $u^{(j)},\ j \in 0, \ldots, N$ und den Schritten der Referenzlösung $\tilde{u}^{(j)},\ j \in 0, \ldots, N$ unter der Annahme dass beide dieselbe Schrittzahl N haben.

Name	Beschreibung	Berechnung
`mse`	Mean Squared Error	$\frac{1}{d}\sum_{i=1}^{d}\left(u_i^{(N)} - \tilde{u}_i^{(N)}\right)^2$
`euclid`	Euklidische Norm der Abweichung	$\sqrt{\sum_{i=1}^{d}\left(u_i^{(N)} - \tilde{u}_i^{(N)}\right)^2}$
`mean`	Durchschnitt der Abweichung	$\frac{1}{d}\sum_{i=1}^{d}\left(u_i^{(N)} - \tilde{u}_i^{(N)}\right)$
`mean_all`	Durchschnitt der globalen Abweichung	$\frac{1}{dN}\sum_{j=0}^{N}\sum_{i=1}^{d}\left(u_i^{(j)} - \tilde{u}_i^{(j)}\right)$
`max`	Maximale Abweichung	$\max_{1\le i\le d}\left(u_i^{(N)} - \tilde{u}_i^{(N)}\right)$
`max_all`	Maximale globale Abweichung	$\max_{0\le j\le N}\left(\max_{1\le i\le d}\left(u_i^{(j)} - \tilde{u}_i^{(j)}\right)\right)$

3.7.3 Benchmarks und Plots

Wie der vorherige Abschnitt zeigt, wird ein einzelner Aufruf eines Lösers für gegebene Datentypen relativ komplex, da beidermaßen Argumente zur Kompilezeit und zur Laufzeit zusammenpassen müssen. Besonders wenn man viele Ergebnisse verschiedener Solver für verschiedene Parametergrößen für ein Diagramm benötigt, führt das zu viel händischer Arbeit, selbst wenn man das Buildsystem aus Abschnitt 3.7.1 verwendet.

Darum ist eine große Aufgabe der Skripte das automatische Erstellen von Diagrammen. Dazu nutzen sie eine Abstraktion mithilfe von Klassen: Ein *Sample* beschreibt eine Auswertung eines AWPs duch einen Solver für gegebene Datentypen und Aufrufparameter. Es speichert alle Argumente für eine eindeutige Zuordnung, definiert eine Methode die sich um die Kompilierung und das Aufrufen des `positode` Programms kümmert, und verarbeitet die Ergebnisse dieses Programmaufrufs. Für spezifischere Programmaufrufe existieren Subklassen wie *ReferenceSample*, das sich zusätzlich um die Erzeugung der temporären Datei kümmert, in der die Vergleichsergebnisse gespeichert werden, die von Samples der Subklasse *ErrorSample* dann bei Programmaufruf zusammen mit einer Fehlerberechnungsmethode übergeben wird.

Da die Solver an sich keine Parallelisierung unterstützen, bietet es sich an, die Samples parallel zu berechnen, also mehrere Instanzen von `positode` gleichzeitig laufen zu lassen. In den Skripten wird dies mithilfe eines Prozess-Pools verwirklicht, der eine Menge von Samples erhält und diese parallel auswertet. Diese Parallelisierung muss für jedes Sample erst erlaubt werden, da sie die Zeitmessung durch die Zusatzlast von anderen Samples auf dem Prozessor verfälscht. Da das Auswerten der Samplepunkte doch ein langwieriger Prozess sein kann, werden die Ergebnisse in einer SQLite-Datenbank abgespeichert, um sie wiederverwenden zu können, sollte an späterer Stelle nochmal dasselbe Sample benötigt werden.

Mithilfe dieser Samples lassen sich nun einfach Messreihen zusammenstellen. Genutzt wird dies von einer Menge an Plot-Typen die im Rahmen des Skriptsystems implementiert wur-

den. Der simpelste davon dient, um den zeitlichen Wertverlauf zweidimensionaler Probleme zu zeigen (`showplot`), ein anderer dient dem Vergleich komprimierter Vektoren mit nichtkomprimierten (`compressed-vector-plot`). Der wichtigste Diagrammtyp, der exemplarisch weiter betrachtet wird, ist der Fehlerplot `h-err` der das Verhältnis von Schrittweite **h** zum Fehler **err** darstellt. Sehr ähnlich zu diesem werden noch `work-precision` Diagramme unterstützt, die statt der Schrittweite h eine Metrik für den Arbeitsaufwand wie die Laufzeit oder ausgeführte Operationen auf die x-Achse abbilden.

Die Skripte erlauben es, einen Plot mithilfe einer Konfigurationsdatei zu beschreiben. Konfigurationen erfolgen durch Beschreibung von Objekten und Listen im YAML-Format, wie im Listing 3.6 exemplarisch gezeigt, die den in Grafik 3.3 gezeigten Plot erzeugt. Anhand dieses Beispiels soll nun die Funktionsweise des Systems betrachtet werden. In der Datei sind zwei `benchmarks` definiert, diese entsprechen im Diagramm den zwei farblichen Gruppen an Graphen. Wie viele Graphen pro Benchmark erzeugt werden, steuert die Liste `groups`. Die Anzahl der Samplepunkte aus denen jeder Graph zusammengesetzt wird, bestimmt die Liste `samplepoints`. Auf der obersten Ebene der Konfiguration werden allegemeine Einstellungen wie der Titel und Plot-Typ spezifiziert, sowie eine `reference` und die Fehlerberechnungsmethode `error_type`.

Die Einstellungen für die Samples sind hierarchisch, d.h. Argumente auf einer höheren Ebene werden von daruntergelegenen überschrieben bzw. hinzugefügt, wenn sie noch nicht existieren. So erhält jeder Samplepunkt von jeder Gruppe jedes Benchmarks durch ein schrittweises Zusammenführen der Schlüssel-Wert-Paare dieser verschiedenen Stufen seine Argumente.

Wie im Beispiel bereits vorgeführt, wird hier eine spezielle Schreibweise zur Beschreibung der Zahlenformate verwendet. Um ein Ausschreiben von Bit- und Exponentenlänge sowie Name zu vermeiden, existieren die Abkürzungen wie `p16,2@1`, die eine Erweiterung der bereits in Kapitel 3.2 vorgestellten Notation sind. Neben Posits erlauben sie auch Floats wie z.B. `f32` zu beschreiben, wobei keine Exponentenlänge angegeben werden darf. Die Notation beginnt also immer mit dem Formatnamen `posit`, `float` (oder dessen Anfangsbuchstabe) und nennt danach die für das Format wichtigen Bitlängen. Darauffolgend lässt sich für Posits noch die ID der Implementierung hinter einem `@` anhängen. Diese dient zur Unterscheidung der Implementierungen und war ursprünglich gedacht, um mehrere verschieden optimierte Implementierungen zu handhaben, bietet aber jetzt nur die Optionen 0 und 1, wobei 1 für die neuere `longposit.c` Implementierung steht die auch im Kapitel 3.2 thematisiert wird, und 0 für die veraltete `posit_impl.c` die nur Posits bis zur Bitlänge 32 unterstützt. Das Versionskürzel `@1` wird demnach eigentlich überall verwendet und der Übersichtlichkeit halber in vielen Grafiken gleich weggelassen.

Mit den Konfigurationsdateien als Eingabe kümmern sich die Plot-Skripte um die Ausführung des `positode` Programms, verarbeiten die Ergebnisse und erstellen mittels der Pythonbibliothek Matplotlib Diagramme, die im `.svg` Format exportiert werden. Da dieses auf textbasierten XML-Beschreibungen basiert, kann in die Dateien einfach ein Kommentar eingefügt werden, in den die Konfigurationsdatei sowie die Ausgaben der Programmaufrufe kopiert werden, was eine spätere Zuordnung eines Diagramms deutlich begünstigt.

Wie in Abbildung 3.3 auch deutlich wird, sind die Namen der Graphen noch nicht sehr aussagekräftig, was daran liegt, dass sie automatisch generiert werden. Die im Evaluierungsteil eingefügten Diagramme werden alle nachbearbeitet sein um die Beschriftungen zu verbessern.

```yaml
type: h-err
name: Van der Pol, RKC1 [h/Error (euclid)]
problem: van-der-pol
solver: implicit-midpoint
t_range: 25.0
args:
  solve_parallel: True
problem_args:
  mu: 15.0
solver_args:

plot_args:
  error_type: "euclid"
groups:
  - p: 0
  - p: 2
samplepoints:
  - h: 0.1
  - h: 0.01
  - h: 0.001
  - h: 0.0001
  - h: 0.00001
  - h: 0.000001
benchmarks:
  - wp: f32
    rp: p16,2@1
  - wp: f64
    rp: p32,3@1
reference:
  wp: f128
  rp: f128
  solver: explicit-rk4
  h: 0.0000001
```

Listing 3.6: Beispiel einer Konfigurationsdatei für einen Fehlerplot.

3.7.4 Kommandozeilenschnittstelle

Um alle der im Implementierungskapitel beschriebenen Programme zu starten, stellen die Skripte eine gesammelte Kommandozeilenschnittstelle (CLI) bereit. Diese findet sich in der Datei `main.py` und nutzt die `argparse` Bibliothek um verschiedene, in Tabelle 3.5 gelistete, Modi mit verschiedenen Parametern bereitzustellen. Im Allgemeinen sollte jeder Programmaufruf über diese Schnittstelle getätigt werden, da sich die Skripte hier um beidermaßen die Kompilierung und Ausführung kümmern und die angegebenen Argumente korrekt auf diese beiden aufteilen.

Die meisten der in Tabelle 3.5 genannten Programme wurden bereits in vorherigen Abschnitten vorgestellt, mit der Ausnahme zweier. Der Aufruf `build clean` sorgt für ein Zurücksetzen aller zwischengespeicherter Dateien, also vorkompilierter Objektdateien und der Sample-Datenbank. Zudem kümmert er sich um die Erzeugung einiger benötigter Verzeichnisse, sowie die Generierung der Header, die mit dem M4-Präprozessor erstellt wurde. Das zweite Programm ist `positunittest.c`, das Mithilfe der `utest.h` Bibliothek [29] einige Testfälle durchführt, mit denen die Funktionen der Algebra- und Vektor-Bibliotheken auf ihre Korrektheit

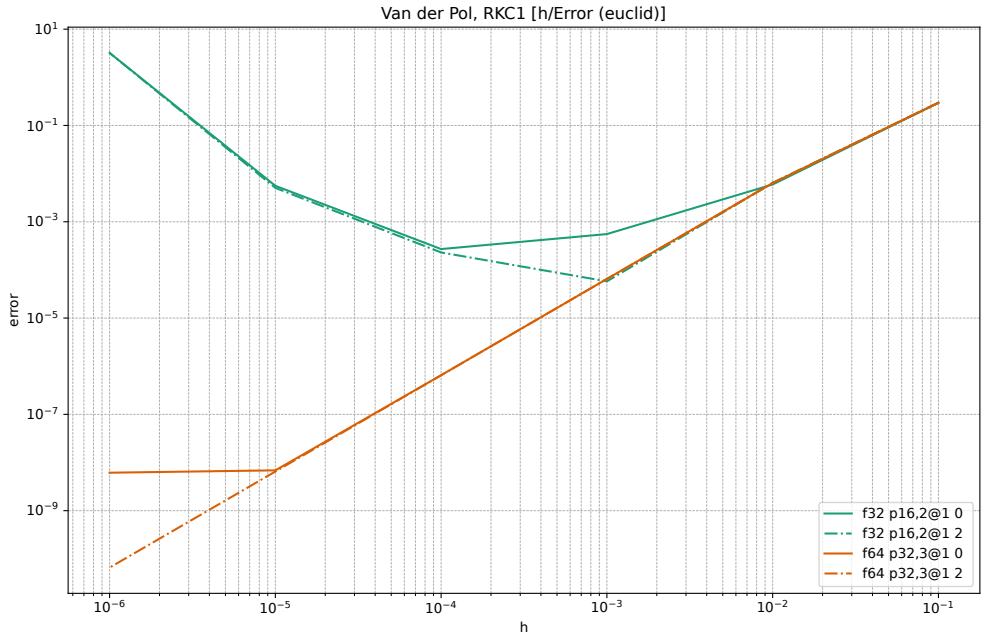


Abbildung 3.3: Erzeugnis der Konfigurationsdatei aus Listing 3.6.

Tabelle 3.5: Liste der Modi die die CLI zur Verfügung stellt.

Modus	Überblick möglicher Argumente	Beschreibung
`build`	`clean`	Löscht alle Caches, vorkompilierte Objekte und generiert Header mit dem M4 Präprozessor neu
`numbertest`	`--format`, `--operator`, `--stride`, `--comp-type`	Führt das in Absch. 3.2.2 erklärte Programm für gegebenes Format, Operator, Schrittweite und Vergleichstyp aus.
`benchmark`	`--format`, `--operator`, `--samples`	Führt das in Abschnitt 3.2.1 vorgestellte Programm für gegebenes Format, Operator und Anzahl der Tests aus
`run`	Argumente von `positode` ohne Ausgabe/Fehlerberechung, `--wp`, `--rp`, `--vector-variant`	Führt das Positode-Programm (aus 3.7.2) aus (mit Ausnahme der Fehlerberechung und Dateiausgabe)
`plot`	`<plot-typ>` `<konfigurationsdatei>`	Erzeugt wie in Absch. 3.7.3 beschrieben Diagramme aus gegebenen Diagrammtypen und Konfigurationsdateien
`unittest`	`--format`, `--vector-variant`	Führt eine Reihe von Unit-Tests für Algebra-Routinen und Zugriffe auf komprimierte Vektoren durch.

geprüft werden. Wird das Argument `--format` ausgelassen, wird eine große Vorauswahl an Formaten nacheinander getestet. Besonders beim Testen des Gauss-Verfahrens macht sich der Unterschied zwischen den Zahlenformaten bemerkbar, da dieses öfters für die schmaleren Datentypen Abweichungen erzeugt, die über dem Schwellwert liegen, während die größeren Formate diese Prüfung problemlos bestehen. Um hier eine Chancengleichheit zu garantieren, verwendet das Skript für alle Aufrufe des Unittest-Programms denselben Seed für den Pseudozufallszahlengenerator, wodurch jeder Datentyp mit denselben Zahlen rechnet.

Neben den in Tabelle 3.5 gezeigten Argumenten, die für jeden Modus spezifisch sind, gibt es eine kleine Auswahl globaler Argumente, die für jeden Programmaufruf (mit Ausnahme von `build clean`) vor dem Modusnamen angehängt werden können. Mit `--log` kann das Logging-Level gesetzt werden, Optionen sind hier `DEBUG`, `INFO`, `WARNING`, `ERROR`, `CRITICAL`, wobei der Standardwert `INFO` ist. Dieses Logging-Level steuert nur die Details der Ausgabe der Skripte, nicht die Ausgabe der von den Skripten aufgerufenen Programme. Mit `--debug/-g` erzeugt das Buildsystem ein Programm mit Debug-Symbolen, während `--profile` die Kompilation unter Einbezug des GNU Profiling-Tools `gprof` durchführt. Nach Ausführung des Programms werden dann die erzeugten Profiling-Ergebnisse von gprof ausgewertet und mit `gprof2dot | dot` in ein Bild von Aufrufgraph umgewandelt, das im Verzeichnis `results/profiling` abgespeichert wird. Hier sei angemerkt, dass das Profiling bei paralleler Ausführung vieler Positode-Programme zu keinen Ergebnissen führt, da gprof die Ausgabe immer in dieselbe Datei versucht zu schreiben, was bei mehreren gleichzeitigen Programmen zu Konflikten führt.

Weiter Informationen über Abhängigkeiten, Installation und genaue Struktur der Programmaufrufe kann der `README.md` und den `--help`-Befehlen der CLI entnommen werden.

4 Evaluation

4.1 Betrachtete Probleme

Um die Solver zu testen wurde eine Reihe von Anfangswertproblemen,die praxisnahe Probleme abbilden, implementiert. Um vergleichbare Ergebnisse zu bekommen, wurden so gut wie möglich die Anfangswertprobleme aus den betrachteten Papern [1], [2] übernommen. Neben den drei im Folgenden vorgestellten AWP gibt es im Quellcode noch zwei weitere, Arenstorf und Linear-Lambda, die für internes Testen verwendet wurden und aufgrund ihrer Auslassung in der Evaluation hier nicht mit aufgeführt werden.

4.1.1 Van der Pol

Der Van-der-Pol-Oszillator wird beschrieben durch das Gleichungssystem [30]

$$\begin{aligned} \frac{\mathrm{d}y_1}{\mathrm{d}t} &= y_2 \\ \frac{\mathrm{d}y_2}{\mathrm{d}t} &= \mu(1-y_1^2)y_2 - y_1 \end{aligned} \tag{4.1}$$

Seien die gewählten Anfangswerte $y_1(0) = 2, y_2(0) = 0$, wobei μ ein Parameter ist, mit dem sich die Steifigkeit der Differentialgleichung einstellen lässt, der im Normalfall $\mu = 1$ ist.

Die zugehörige Jacobimatrix lässt sich einfach berechnen als:

$$J_f = \begin{bmatrix} \frac{\mathrm{d}}{\mathrm{d}y_1}y_2 & \frac{\mathrm{d}}{\mathrm{d}y_2}y_2 \\ \frac{\mathrm{d}}{\mathrm{d}y_1}\mu(1-y_1^2)y_2 - y_1 & \frac{\mathrm{d}}{\mathrm{d}y_2}\mu(1-y_1^2)y_2 - y_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2\mu y_1 y_2 - 1 & \mu(1-y_1^2) \end{bmatrix} \tag{4.2}$$

4.1.2 Brusselator

Brusselator ist der Name einer partiellen Differentialgleichung die das Verhalten der Konzentrationen zweier Substanzen $u(t,x), v(t,x)$ zueinander in einem System chemischer Reaktionen beschreibt.

$$\begin{aligned} \frac{\mathrm{d}u(t,x)}{\mathrm{d}t} &= \alpha \cdot \Delta u + u^2 v - (b+1)u + a \\ \frac{\mathrm{d}v(t,x)}{\mathrm{d}t} &= \alpha \cdot \Delta v - u^2 v + bu \end{aligned} \tag{4.3}$$

Dabei betrachten wir den eindimensionalen Fall $x \in \mathbb{R}$, mit Laplace-Operator $\Delta = \frac{\partial^2}{\partial x^2}$ und Dirichlet-Randbedingungen

$$u(t,0) = u(t,1) = a, \quad v(t,0) = v(t,1) = b, \quad u(0,x) = \sin(2\pi x), \quad v(0,x) = b$$

(aus [2]) wobei $a, b \in \mathbb{R}$ eindimensionale Konstanten sind.

Da diese Differentialgleichung so noch partiell ist und demnach nicht mit den vorgestellten Runge-Kutta-Verfahren gelöst werden kann, wird sie auf einem eindimensionalen Raumgittern $\{x_1, \ldots, x_m\},\ x_j - x_{j-1} = \Delta s, m \in \mathbb{N}^+$ mittels des zentralen Differenzenquotienten

$$\frac{\partial^2 y}{\partial x^2} \approx \frac{y(x + \Delta s) - 2y(x) + y(x - \Delta s)}{\Delta s^2} \tag{4.4}$$

für $1 \leq j \leq m$ diskretisiert:

$$\begin{aligned} \frac{\mathrm{d}u_j(t)}{\mathrm{d}t} &= \alpha \frac{u_{j+1}(t) - 2u_j(t) + u_{j-1}(t)}{\Delta s^2} + u_j(t)^2 v_j(t) - (b+1)u_j(t) + a \\ \frac{\mathrm{d}v_j(t)}{\mathrm{d}t} &= \alpha \frac{v_{j+1}(t) - 2v_j(t) + v_{j-1}(t)}{\Delta s^2} - u_j(t)^2 v_j(t) + bu_j(t) \end{aligned} \tag{4.5}$$

Dies ergibt insgesamt ein Differentialgleichungssystem der Dimension $d = 2m$ mit $y(t) = [u_1(t), \cdots, u_m(t), v_1(t), \cdots, v_m(t)]^T$ und $f = [f_u, f_v]^T$. Die Jacobimatrix $J_f(t)$ ist aus [30] übernommen:

$$\begin{aligned} J_f &= \begin{bmatrix} \frac{\mathrm{d}f_u}{\mathrm{d}u} & \frac{\mathrm{d}f_u}{\mathrm{d}v} \\ \frac{\mathrm{d}f_v}{\mathrm{d}u} & \frac{\mathrm{d}f_v}{\mathrm{d}v} \end{bmatrix} \\ J_f &= \begin{bmatrix} \mathrm{diag}(2u_i v_i - (B+1)) & \mathrm{diag}(u_i^2) \\ \mathrm{diag}(B - 2u_i v_i) & \mathrm{diag}(-u_i^2) \end{bmatrix} + \frac{\alpha}{\Delta s^2} \begin{bmatrix} K & 0 \\ 0 & K \end{bmatrix} \text{ wobei } K \in \mathbb{R}^{m \times m}: \\ K &= \begin{bmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix} \end{aligned} \tag{4.6}$$

Da im Programm für die konstanten Randbedingungen die jeweils ersten und letzten Elemente $u_1 = u_m = a$ sowie $v_1 = v_m = b$ verwendet werden, werden die Zeilen $1, m, m+1$ und $2m$ der Jacobimatrix, die deren Ableitung enthalten, zu 0 gesetzt. In den folgenden Betrachtungen wird für die Koeffizienten $a = 1$, $b = 3$ und $\alpha = 1/50$ verwendet und die Gitterzellengröße $\Delta s = 1/(m-1)$ gesetzt.

4.1.3 Heat2d

Das letzte Anfangswertproblem ist die Wärmeleitungsgleichung. Diese ist wie der Brusellator eine partielle Differentialgleichung

$$\frac{\mathrm{d}u}{\mathrm{d}t} = \alpha \Delta u = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{4.7}$$

von der der zweidimensionale Fall $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ betrachtet wird. Auf einem zweidimensionalen Raumgitter $x_1, \ldots, x_m;\ y_1, \ldots, y_m$ der Dimension $m \times m$ wird diese zu einem gewöhnlichen Differentialgleichungssystem diskretisiert. Auf dem Raumgitter sei dann $u(x_i, y_j, t) = u_{i,j}(t)$,

womit sich durch Approximation mithilfe des zentralen Differenzenquotienten aus Gl. 4.4 ergibt:

$$\begin{aligned} \frac{\mathrm{d}u_{i,j}(t)}{\mathrm{d}t} &= \frac{\alpha}{\Delta s}\left(u_{i+1,j}(t) + u_{i-1,j}(t) + u_{i,j+1}(t) + u_{i,j-1}(t) - 4u_{i,j}(t)\right), \text{ für } 1 < i, j < m \\ \frac{\mathrm{d}u_{i,j}(t)}{\mathrm{d}t} &= T_b \text{ für Ränder } i = 0, i = m \text{ oder } j = 0, j = d \end{aligned} \tag{4.8}$$

Dabei ist T_b die konstante Temperatur an den Rändern.

Für die Wärmeleitungsgleichung wurde die Jacobimatrix nicht analytisch berechnet und implementiert, sondern mithilfe der Zentralen Differnzenquotienten approximiert.

4.2 Testumgebung

Wie schon die in Kapitel 3.2.1 angestellten Tests wurde die gesamte Auswertung auf einem AMD Ryzen 5 2600 Prozessor mit 6 Kernen, der auf 3,7GHz Frequenz gesperrt wurde, ausführt. Dies erfolgte auf einem Linux-System mit der Kernelversion 6.15.4, als Compiler wurde die aktuelle GCC-Version 15.1.1 verwendet, die Python Version für die Skripte war 3.13.5. Tests ohne Zeitmessungen wurde zuweilen auch mit identischer Software auf einem AMD Ryzen 5 4650U durchgeführt.

Da es eine sehr große Anzahl an Parametern gibt, die für jede Lösung eines Anfangswertproblems gesetzt werden können, kann hier der Übersicht wegen nur eine exemplarische Auswahl an Fällen betrachtet werden. Die langsame Softwaresimulation begrenzt ebenfalls den Umfang der Löser auf nicht all zu große Schrittzahlen, vor allem bei großen ODE-Systemen.

4.3 Mixed-Precision Implizite Mittelpunktsregel

4.3.1 Vergleich mit Literatur

Um zuerst die Richtigkeit des Lösungsverfahrens zur Mittelpunktsregel gemischter Genauigkeit zu prüfen, wurde dieses vorerst mit Float-Zahlen für den gleichen Testfall, der auch im Paper [6] untersucht wird, betrachtet. Es wurde hierbei das Van-der-Pol System in der nichtsteifen Variante $\mu = 1$ auf einem Zeitintervall $T = 1$ mit dem Löser der Impliziten Mittelpunktsregel für eine Reihe an Kombinationen von Genauigkeiten ausgeführt, und untersucht wie sich die Schrittweite h bei verschiedenen Datentypen auf den Fehler, hier die euklidische Norm der Abweichung, auswirkt. Die Abbildung 4.1, generiert mithilfe des Skriptsystems aus 3.7.3 zeigt das Ergebnis dieser Auswertung. Dabei werden Verfahren einfacher und gemischter Genauigkeiten betrachtet. Für zweitere werden jeweils die zwei Fälle, ohne Korrekturschritten $p = 0$ und mit Korrekturschritten $p = 2$ betrachtet. Im Diagramm teilen sich diese für zusammengehörende Zahlenformate eine Farbe und sind nur unterschiedlich gestrichelt bzw. durchgezogen.

An dieser Grafik, die den Ergebnissen von Burnett ([6] Abbildungen 1. und 2.) gleicht, lässt sich schön das Fehlerverhalten der Mixed-Precision Verfahren erkennen. Ohne Korrekturschritte wird der Fehlerterm $\mathcal{O}(h^2) + \mathcal{O}(\epsilon h)$ für kleiner werdende Schrittweiten h irgendwann vom konstanten Rundungsfehler ϵ dominiert, was im Graph dazu führt, dass die Linien von Verfahren ohne Korrektuschritte ab einem gewissen Knickpunkt nach oben links weiterverlaufen,

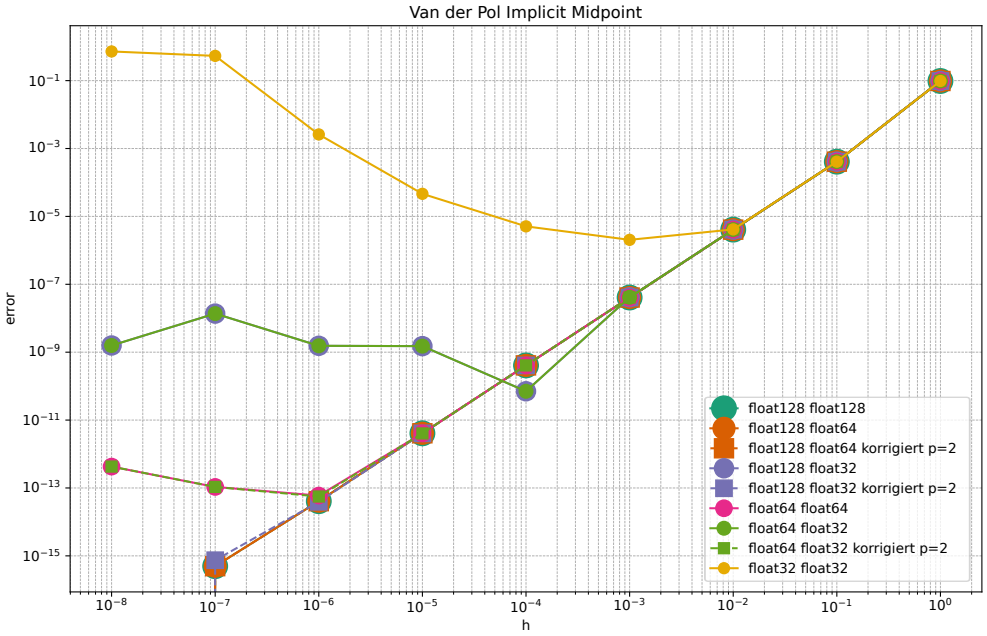


Abbildung 4.1: Fehler des Mixed-Precision ImpMid Solvers für $p = 0$ oder $p = 2$ Korrekturschritte

der Fehler also trotz einer kleineren Schrittweite stagniert bzw. durch die Summation vieler interner Rundungsfehler sogar wächst. Die Verfahren mit Korrekturschritten, in der Abbildung mit gestrichelten Linien gekennzeichnet, laufen aufgrund ihres angepassten Fehlerterms $\mathcal{O}(h^2)+\mathcal{O}(\epsilon h^2)$, bei dem ϵ einen kleineren Beitrag ausmacht, länger in diagonaler Richtung mit nach links unten. Sie liegen also näher an den $\mathcal{O}(h^2)$ die ein normales Runge-Kutta-Verfahren ohne Mixed-Precision, wie in der Abbildung `float128 float128`, also die Implizite Mittelpunktsregel berechnet mit Quad-Floats, erreicht. Der Grund für das Abknicken der Diagonale nach unten rechts liegt daran, dass der Fehler für die Solver mit WP `float128` für $h = 10^{-8}$ bei exakt Null liegt, was auf einer Logarithmischen Achse nicht dargestellt werden kann.

Wie man an der Linie von `float64 float32` mit $p = 2$ Korrekturschritten sehen kann, wird dieser "Knickpunkt" an dem der Rundungsfehler dominiert sozusagen nach links verschoben: Während ohne Korrektur die Konvergenzordnung ab einer Schrittweite von 10^{-5} nicht mehr garantiert ist, so ist dies mit Korrekturschritten erst ab 10^{-7} der Fall. Hierbei sei noch angemerkt, dass ein Abweichen von der Konvergenzordnung $\mathcal{O}(h^2)$ natürlich nicht nur dem Rundungsfehler ϵ des Reduced-Precision Typs geschuldet ist, sondern auch vom Work-Precision Typen und dessen Genauigkeit abhängt. Dies lässt sich für `float32 float32` und `float64 float64` erkennen, die beide einen Verlust von der Konvergenzordnung ab den Punkten 10^{-3} respektive 10^{-7} aufzeigen, obwohl hier keine fehlerbelasteten Typkonvertierungen vorliegen. Diese Abweichungen stammen stattdessen aus den Ungenauigkeiten des WP-Typen, der für kleine h diese nicht mehr richtig darstellen kann. Dieser Fall macht sich bei den naiven Mixed-Precision Varianten ($p = 0$) nur nicht bemerkbar, da dort der größere Rundungsfehler von RP dominiert.

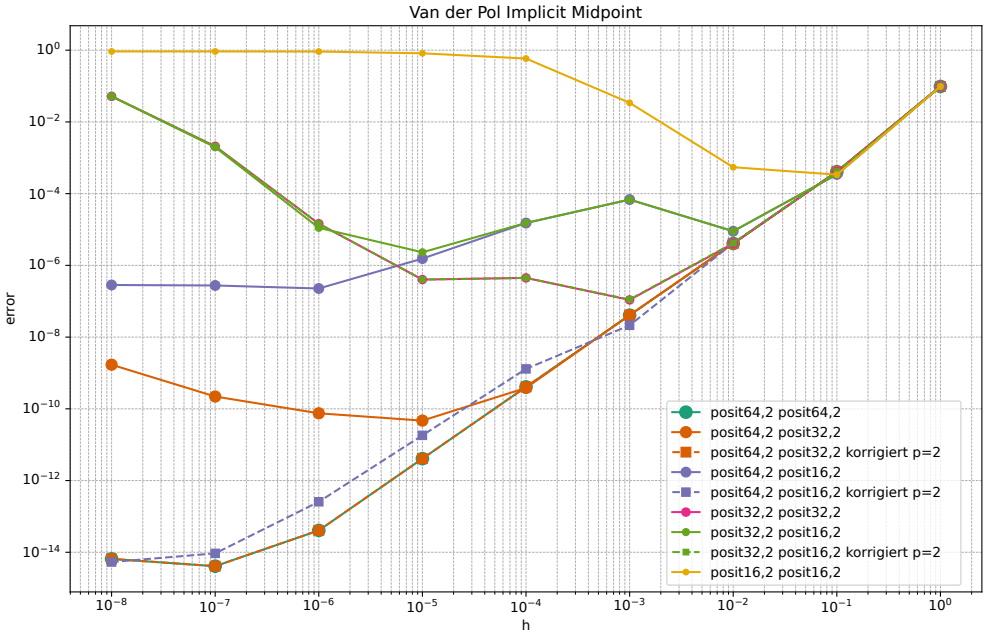


Abbildung 4.2: Allgemeiner Test des Mixed-Precision Implicit-Midpoint Verfahrens mit Posits.

All diese Feststellungen erfüllen die Erwartungen aus den Ergebnissen der Paper von Grant [1] und Burnett [6]. Diese Erkenntnis, dass die eigene Implementierung des Lösungsverfahrens diese Ergebnisse für Float-Datentypen reproduzieren kann, gepaart mit den Korrektheitstests der Positsoftwaresimulation soll für die folgenden Tests als Garantie ihrer Gültigkeit gelten.

4.3.2 Evaluation von Posits

Wir betrachten zum Einstieg ähnliche Testfälle wie gerade für die Floats nun für Posits. Da die Softwareposits keine 128 Bit großen Datentypen unterstützen, wurde hier stattdessen ein 16 Bit Typ hinzugefügt und alle Datentypen um eine Stufe verkleinert, um ein qualitativ ähnliches Bild zu erhalten. Der Einfachheit halber wurden zu Beginn für alle $es = 2$ gewählt, was in weiteren Betrachtungen noch verfeinert wird.

Für dasselbe AWP und denselben Solver ergibt sich für korrigierende und nichtkorigierende Verfahren ein Fehlerverlauf wie in Grafik 4.2 dargestellt. Hier lassen sich dieselben Entwicklungen wie für die Floats in Grafik 4.1 betrachten: Die korrigierenden Verfahren bewahren für kleiner werdende Schrittweiten länger ihre Konvergenzordnung als die nichtkorrigierenden; alle Verfahren, auch korrigierende, weichen ab dem Punkt ab dem der Rundungsfehler von WP zu groß wird von der Diagonalen ab. Das Verhalten ist hier offensichtlich analog zu dem der Floats. Einzige bemerkenswerte Eigenheit ist das Abweichen des korrigierenden Lösers für `posit64,2 posit16,2` zwischen den Schrittweiten $h = 10^{-4}$ und $h = 10^{-7}$. Außerhalb dieses Intervalls folgt sein Fehlerverlauf dem der Löser für `posit64,2 posit32,2` (korrigiert $p = 2$) und `posit64,2 posit64,2` mit demselben Typen für WP, und alle knicken für 10^{-8} gemeinsam ab, sobald der Rundungsfehler von WP überhand gewinnt. Dass auf diesem spe-

Tabelle 4.1: Posit-Datentypen der Längen $16, 32, 64$ angepasst an den Zahlenbereich der Float-Datentypen derselben Bitlänge.

Bitlänge	Exp.länge Float	Ungefährer Float Zahlenbereich	Exp.länge Posit	Ungefährer Posit Zahlenbereich
16	5	$6 \cdot 10^{-8}$ bis $7 \cdot 10^{4}$	1	$4 \cdot 10^{-9}$ bis $3 \cdot 10^{8}$
32	8	$1 \cdot 10^{-45}$ bis $3 \cdot 10^{38}$	3	$6 \cdot 10^{-73}$ bis $2 \cdot 10^{72}$
64	11	$5 \cdot 10^{-324}$ bis $2 \cdot 10^{308}$	5	$6 \cdot 10^{-598}$ bis $4 \cdot 10^{587}$

ziellen Intervall die Abweichung jedoch größer ist, kann demnach nur am Zahlenformat für RP und seinem Rundungsfehler ϵ liegen, dessen Beitrag von $\mathcal{O}(h^2\epsilon)$ hier größer ist als der von `posit32,2`.

Posits mit gleichem Zahlenbereich wie Floats

Was jedoch jetzt interessant ist, ist der Vergleich der Posits zu Floats. Um vergleichbare Datentypen zu haben, wird zu jedem der Float-Formate ein Posit-Format mit der gleichen Bitlänge konstruiert, dessen *es* so gewählt ist, dass der Posit den dynamischen Zahlenbereich des Floats abdeckt oder übersteigt. In der Theorie bedeutet das, dass beide Formate nur Zahlen aus dem gleichen Intervall beinhalten, die Posits aber aufgrund ihrer *Tapered-Precision* anders verteilt sind und durch die Kodierung des Regimes nahe der Eins eine höhere Genauigkeit als Floats aufweisen, wie bereits in Kapitel 2.1.2 angesprochen und in Grafik 2.3 aufgezeigt. Die entsprechenden Posit-Formate werden in Tabelle 4.1 gelistet und sind der Herleitung von Gustafson [3] entnommen, wobei für den 64-bit Typen die Exponentelänge angepasst wurde, sodass der gessamte Zahlenbereich von Double-Floats abgedeckt wird.

Aus dieser Tabelle wird deutlich, dass die Exponenten der Posits im Allgemeinen weniger Bits benötigen. Diese Exponentenlänge in Kombination mit der mininmalen Regimelänge von 2 führt dazu, dass mehr Bits für die Mantisse übrig sind, was den Rundungsfehler ϵ im Vergleich zu Floats verringert. Andererseits wird für betragsmäßig große Exponenten die Mantisse länger und damit die Mantisse potentiell kürzer was sich negativ auf den Rundungsfehler auswirkt. Es gilt nun, durch Experimente herauszufinden, ob sich diese Eigenschaft der Posits positiv auf das Fehlerverhalten der Lösungsverfahren auswirkt.

Da die Betrachtung aller Formate für alle Varianten mit Korrektur und ohne in einem einzelnen Diagramm überfordernd wäre, werden in Diagramm 4.3 zuerst nur Verfahren ohne Mixed-Precision betrachtet. Hier lassen sich trotz der gleichen abgedeckten Zahlenbereiche maßgebliche Unterschiede in den erreichten Fehlern erkennen. So halten beidermaßen `posit16,2` und `posit32,3` um etwa eine Zehnerpotenz länger die Konvergenzordnung als ihre Float-Gegenstücke, bevor sie aufgrund interner Rundungsfehler davon abweichen. Ähnliches lässt sich auch für `posit64,5` feststellen, wenn auch nicht im selben Ausmaß. Hier bleiben zwar beide Varianten bis zur gleichen Schrittweite in zweiter Ordnung konvergent, allerdings wächst der Fehler des Posit-Verfahrens nach dem Abknicken des des Graphen hier langsamer an als der seines Gegenspielers.

Mit den Diagrammen in Abbildung 4.4 werden nun die Mixed-Precision Varianten des ImpMid Solvers unter Verwendung der Posits betrachtet. Beide Seiten zeigen dabei dieselben Kombinationen von Datentypen bei den gleichen Schrittweiten, links werden keine Korrekturschritte gemacht, rechts $p = 2$ Stück. Hier lässt sich eine Reihe von Feststellungen machen:

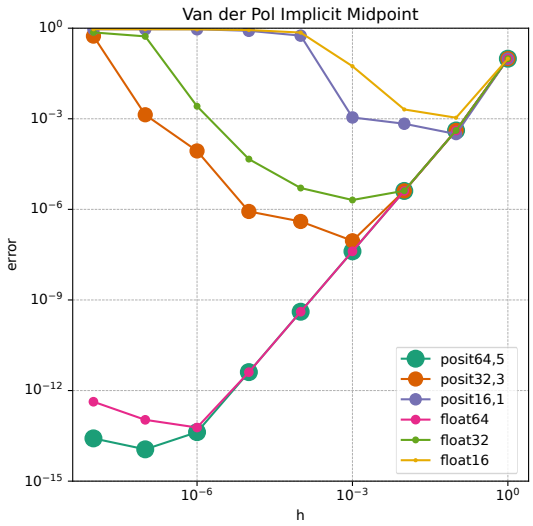


Abbildung 4.3: Vergleich von Float- und im Zahlenbereich angepassten Positformaten für das Implizite Mittelpunktsverfahren ohne gemischter Genauigkeit.

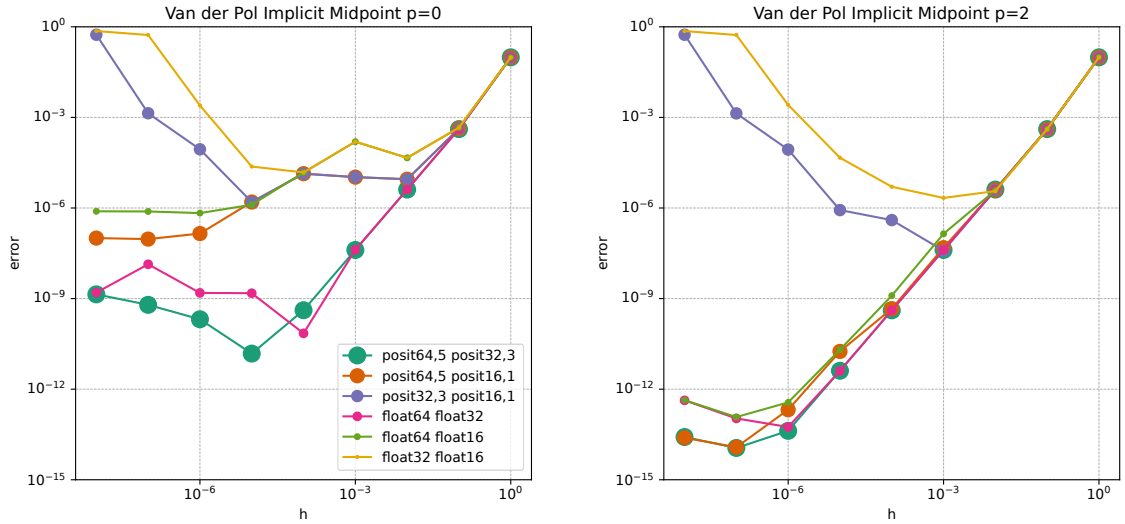


Abbildung 4.4: Vergleich von Float und Posits in der Mixed-Precision Impliziten Mittelpunktsregel, einmal mit und ohne Korrekturschritten.

Tabelle 4.2: Werte für Bitlänge W, Exponentenlänge es (W_E) der konstruierten Posit-Formate mit gleichem Zahlenbereich und Mantissenlänge.

Allg.	Float			Posit		
Mantissen-länge W_M	W	W_E	Ungefährer Float Zahlenbereich	W	es	Ungefährer Posit Zahlenbereich
10	16	5	$6 \cdot 10^{-8}$ bis $7 \cdot 10^{4}$	14	1	$6 \cdot 10^{-8}$ bis $4 \cdot 10^{6}$
23	32	8	$1 \cdot 10^{-45}$ bis $3 \cdot 10^{38}$	29	3	$9 \cdot 10^{-66}$ bis $4 \cdot 10^{62}$
52	64	11	$5 \cdot 10^{-324}$ bis $2 \cdot 10^{308}$	60	5	$2 \cdot 10^{-559}$ bis $1 \cdot 10^{549}$

Die größeren Mantissenlängen und daraus resultierenden kleineren Rundungsfehlern verschaffen den Posits beidermaßen für korrigierende und nichtkorrigierende Verfahren einen Vorteil gegenüber der Float-Formate. So verlieren in beiden Fällen die Löser mit Float-Formaten für größere Schrittweiten bereits ihre Konvergenz als Löser mit Posits.

Besonders hervorheben kann man hier den korrigierenden Solver für die Kombination der Formate `posit64,5 posit16,2`, der für Schrittweiten $h \leq 10^{-7}$ einen kleineren Fehler als der Solver mit `float64 float32` erreicht, obwohl er ganze zwei Bytes weniger Speicher für seinen Reduced-Precision Datentypen zur Verfügung stehen hat. Auch wenn beide hier nicht mehr die Konvergenzordnung einhalten ist dies trotzdem ein positiv überraschendes Ergebnis.

Posits mit gleicher Genauigkeit wie Floats

Aus Abbildung 4.4 wurde erkenntlich, dass Posits bei gleichen Bitlängen gegenüber ihrer Float-Äquivalente eine Steigerung der Genauigkeit bewirken können. Aus den Resultaten des letzten Abschnitts kommt nun die Idee, den umgekehrten Fall zu betrachten: Schaffen es Posits, mit kleineren Bitlängen und reduziertem Speicherverbrauch, Ergebnisse derselben Genauigkeit wie Floats zu erzeugen?

Hierzu kann realistisch wieder nur eine Auswahl von Posit-Formaten betrachtet werden, zu diesem Zwecke werden in Tabelle erneut äquivalente Formate gewählt, die den Floats in ihrem Zahlenbereich, sowie ihrer maximalen Anzahl an Mantissenbits angeglichen sind. Zur Berechnung des Zahlenbereichs dient Gleichung 2.3.

Mit den resultierenden Posit-Formaten `posit14,1`, `posit29,3` und `posit60,5` werden nun die Tests wiederholt. Diagramme 4.5 und 4.6 wie sich die in Genauigkeit angepassten Posits im Vergleich zu den korrespondierenden Floats auf den Löser der impliziten Mittelpunktsregel auswirkt. Dabei werden wie zuvor die Fälle ohne gemischter Genauigkeit, mit Korrekturschritten und ohne betrachtet. Wie die Grafiken zeigen, erreichen diese weniger Speicher nutzenden Posits die exakt gleichen Ergebnisse wie die Floats. Bis auf minimal größere Fehler der Posits, die an Stellen entstehen können, an denen sie aufgrund eines größeren Regimes nicht die volle vorgesehene Mantissenlänge ausnutzen können, gleichen sich die resultierenden Abweichungen für beide Zahlenformate ziemlich exakt, trotz ihrer Einsparung an Bits.

Diese Ergebnisse lassen darauf schließen, dass die Posits hier häufig ihre maximale Mantissenlänge ausnutzen können da die Regimes nicht groß werden, also sich das Problem auf einem Intervall nahe der Eins abspielt. In der Tat zeigt sich, dass sich der Van-der-Pol Oszillator im nichtsteifen Fall nur auf einem Intervall innerhalb von $[-2,5; 2,5]$ bewegt (das zusätzlich dazu

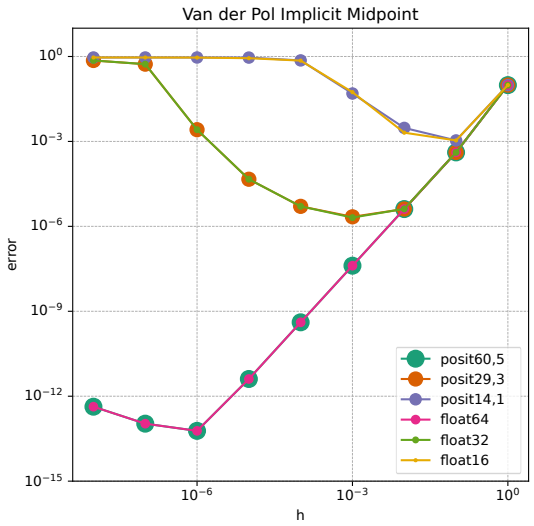


Abbildung 4.5: Genauigkeit-angeglichene Posits im Vergleich zu Floats im ImpMid Löser ohne Mixed-Precision.

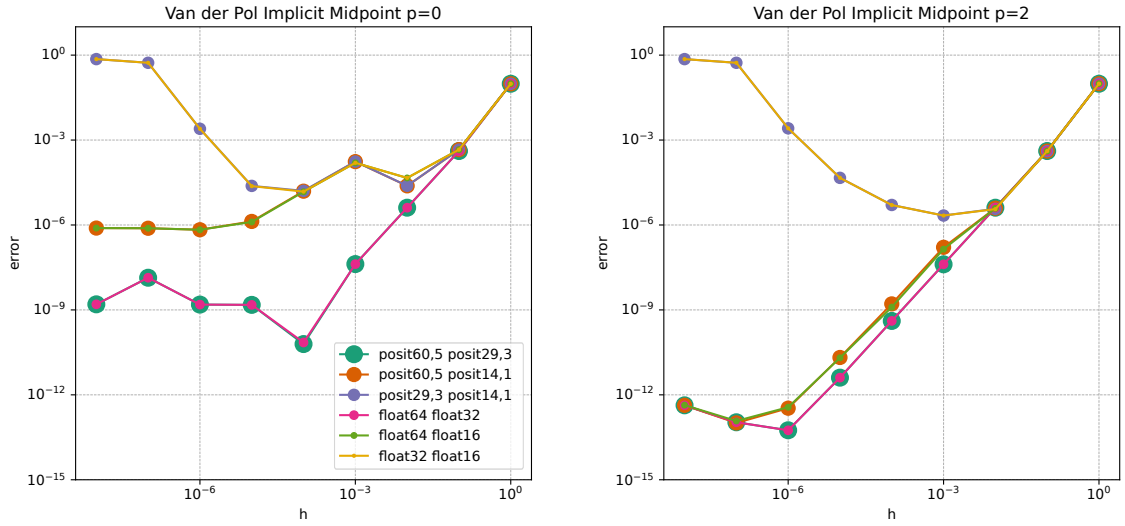


Abbildung 4.6: Genauigkeits-angeglichene Posits im vergleich zu Floats im Mixed-Precision ImpMid Löser. Links ohne und rechts mit Korrektuschritten.

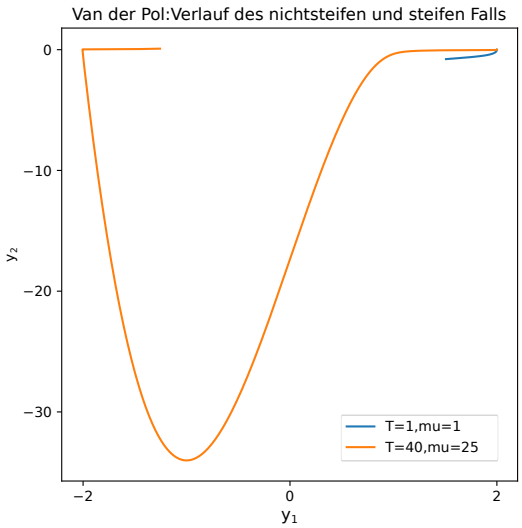


Abbildung 4.7: Vergleichsergebnis des nichtsteifen und steifen Van-Der-Pol-Systems; zeitlicher Verlauf auf y_1, y_2 Koordinaten.

um die 0 eine Lücke hat), was natürlich den Posits mit ihrer Tapered-Precision besonders zu Gute kommt.

Posits für steife ODEs

Allerdings bietet bei einem numerischen Problem auf einem fest bekannten Zahlenintervall eine Fixkommaarithmethik um einiges mehr Vorteile als es die Posits tun können, weswegen dies kein ausreichendes Kriterium ist um die Posits als den Floats überlegen hervorzuheben. Zusätzlich zu diesem begünstigendem Szenario soll jetzt noch der steife Fall der ODE beleuchtet werden, um festzustellen, ob die Posits ihre Vorteile auch auf größeren Zahlenbereichen ausspielen können.

Um die Steifigkeit zu erhöhen wird für das Van-der-Pol System aus Gl. 4.1 der Parameter $\mu = 25,0$ gesetzt und das Zeitintervall von $T = 1$ auf $T = 40$ erweitert. Abb. 4.7 gibt einen Überblick über den zeitlichen Verlauf des numerischen Ergebnisses der Differentialgleichung für den nichtsteifen und steifen Fall. Der steife Fall vergrößert also das Intervall auf dem sich das Schrittverfahren bewegt und fordert damit die Posits mehr heraus, zusätzlich dazu stellt die erhöhte Steifigkeit des Problems eine Herausforderung an den Solver im Allgemeinen dar. Dies ist gut für den Testlauf, da in der Praxis implizite Verfahren häufig zum Lösen steifer Probleme genutzt werden, wofür ein gutes Abschneiden der Posits natürlich auch wünschenswert wäre.

Grafik 4.9 zeigt die Diagramme zu den Testfällen für die steife Differentialgleichung. Da Aufgrund der Steifigkeit vor allem größere Schrittweiten zu schlechter Konvergenz führen, wurde die Fehlerachse im Vergleich zu den vorher betrachteten Diagrammen nach oben erweitert. Für große Schrittweiten ist dieses Verhalten für alle Formate gleichermaßen beobachtbar, da dieses am Löser und der Differentialgleichung liegt und nicht am Zahlenformat. Allerdings zeigt sich bei Schrittweiten von $h = 1$ und $h = 0,1$, dass die Löser die half-Floats verwenden, verstärkt

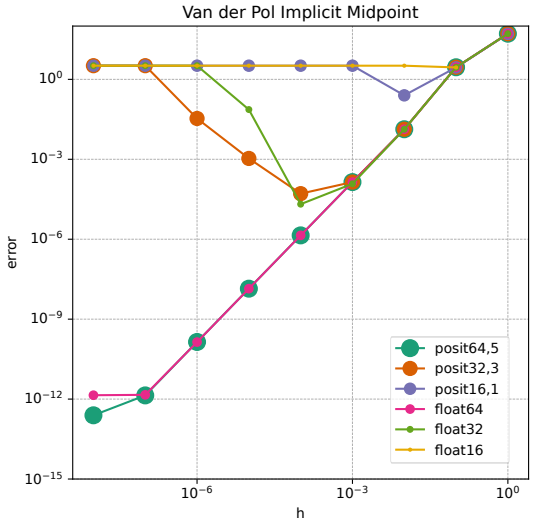


Abbildung 4.8: Steifes Van der Pol AWP mit $\mu = 25$ $T = 40$. Gelöst mit Impliziter Mittelpunktsregel ohne gemischter Genauigkeit.

unter der Steifigkeit der ODE leiden und hier zu NaN als Ergebnis divergieren, während die Löser, die Posits verwenden, noch auf einen Ergebniswert konvergieren. Hier zahlt sich also die Eigenschaften der Posits, dank ihrer Regimes variabler Länge einen großen Zahlenbereich abdecken zu können und nicht nach ∞ zu runden aus.

Für kleiner werdende Schrittweiten h kann man dieselben Phänomene beobachten, die auch bei der nicht steifen ODE bereits genannt wurden: Ab bestimmten Schrittweiten, knicken die Verfahren ein und verlieren ihre Konvergenzordnung, die Korrekturschritte wirken diesem entgegen. Im Vergleich zu den Betrachtungen aus Abb. 4.4 kann man hier feststellen, dass die Posits nicht immer erst später als die Floats ihre Konvergenz verlieren, sondern die beiden Formate, vor allem die 16 Bit Typen, ab der gleichen Schrittweite beginnen stärker von der Referenzlösung abzuweichen. Dies ergibt auch aus theoretischer Betrachtung Sinn, da zur Darstellung von Zahlen > 32 das `posit16,1` Format das Regime so lange wird, dass für die Mantisse nur noch 10 Bits übrig sind, was sie gleich genau wie die Mantisse eines Half-Floats macht.

Arbeitsaufwand von Posits

Als letzte Untersuchung zur Qualität der Posits für die Implziten Runge-Kutta-Verfahren mit gemischter Genauigkeit soll nun noch der Aufwand verglichen werden. Trotz ihrer numerischen Vorteile wären Posits ziemlich nutzlos, wenn dieses Ergebnis mit einem inhärent höheren Aufwand verbunden wäre. Da es sich aufgrund der Softwaresimulation der Posits nicht anbietet die Laufzeit direkt als Metrik für Aufwand zu betrachten, wird stattdessen die Anzahl ausgeführter Rechenoperationen gezählt. Zudem lag bei der Entwicklung des Frameworks der Fokus mehr auf der Vollständigkeit als auf bis ins Letzte optimierten BLAS-Funktionen, weswegen Algorithmen wie die das Gaußverfahren in der naiven Variante implementiert sind, was die Zeitmessung als Metrik sowieso nicht repräsentativ im Vergleich mit anderen Lösern macht.

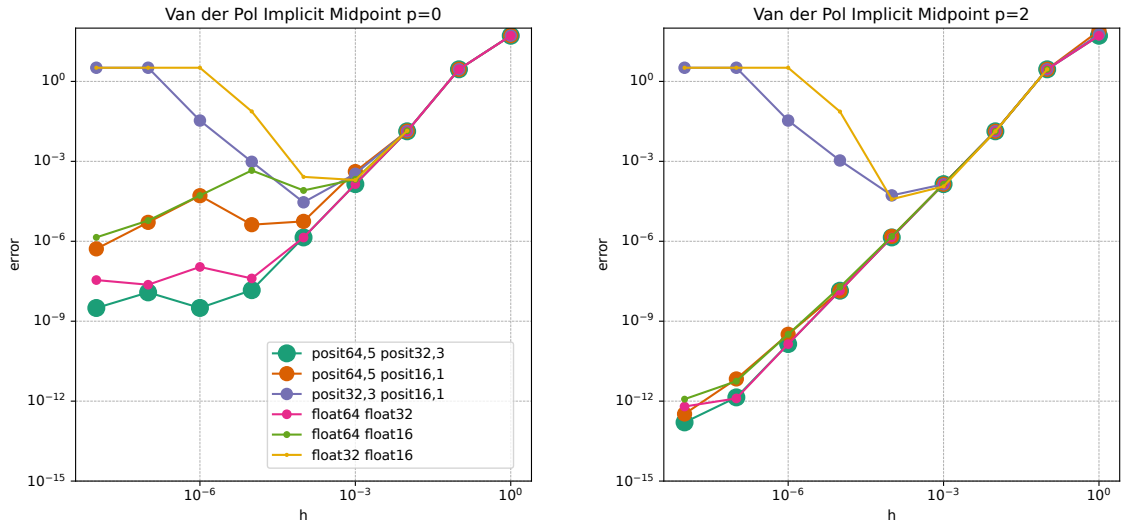


Abbildung 4.9: Steifes Van der Pol AWP mit $\mu = 25$ $T = 40$. Gelöst mit Impliziter Mittelpunktsregel mit gemischter Genauigkeit, links ohne und rechts mit Korrekturschritten.

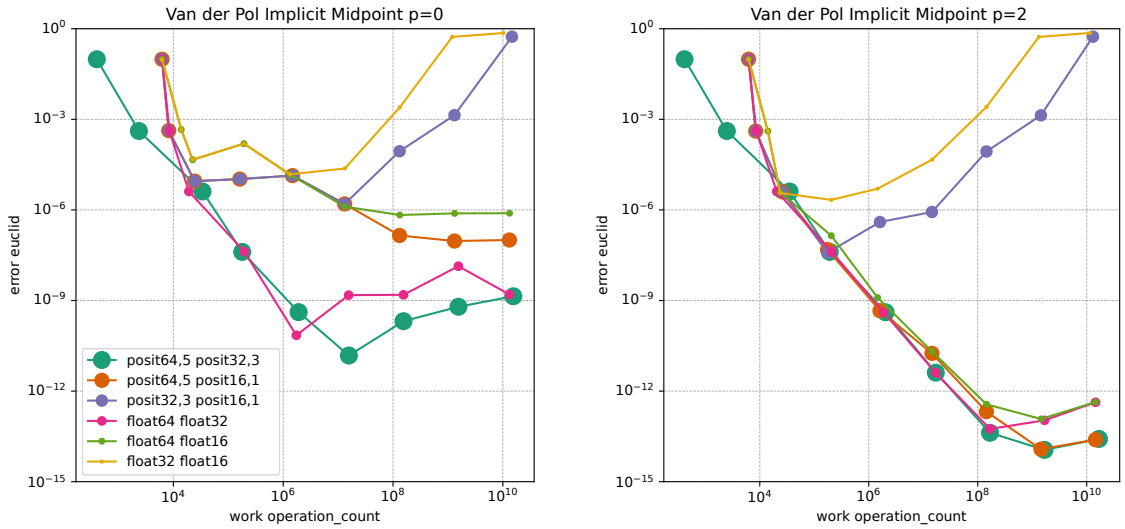


Abbildung 4.10: Work-Precision Diagramm, Vergleich von Fehler und durchgeführter Rechenoperationen für verschiedene Float und Positformate. Durchgeführt auf ImpMid Solver, links ohne und rechts mit Korrekturschritten.

Zur Evaluation der Operationszahl wurden für die beiden Mixed-Precision Fälle ohne und mit Korrekturschritten je ein Work-Precision-Diagramm erstellt, die in Abbildung 4.10 gezeigt sind. Die Parameter für die Solver sind hierfür wieder dieselben wie aus der Betrachtung der im Zahlenbereich angeglichenen Posits, genauso die gewählten Schrittweiten von 10^0 bis 10^{-8}

Aufgrund der Gleichheit der Parameter, kann der einzige Verursacher von mehr oder weniger Rechenoperationen die Schrittzahl des Newton-Raphson Verfahrens sein, welches immer bis zu einer festen Toleranz seine iterative Berechnung wiederholt. Würden sich nun ein Zahlenformat negativ auf die Konvergenzgeschwindigkeit dessen auswirken, so könnte man dies anhand der Gesamtzahl der Operationen erkennen. Betrachtet man die gemessenen Operationen, so zeigt sich, dass die Posits an keiner Stelle groß im Nachteil, also weiter rechts in x-Richtung, gegenüber der Floats liegen. Einzig sticht hier der Graph von `posit64,5` `posit32,3` für große Schrittweiten hervor, für die das Verfahren mit deutlich weniger Rechenoperationen als die anderen auskommt, was wie unter vorheriger Erläuterung nur an einem schnelleren Abbruch des Newton-Raphson Verfahrens liegen kann.

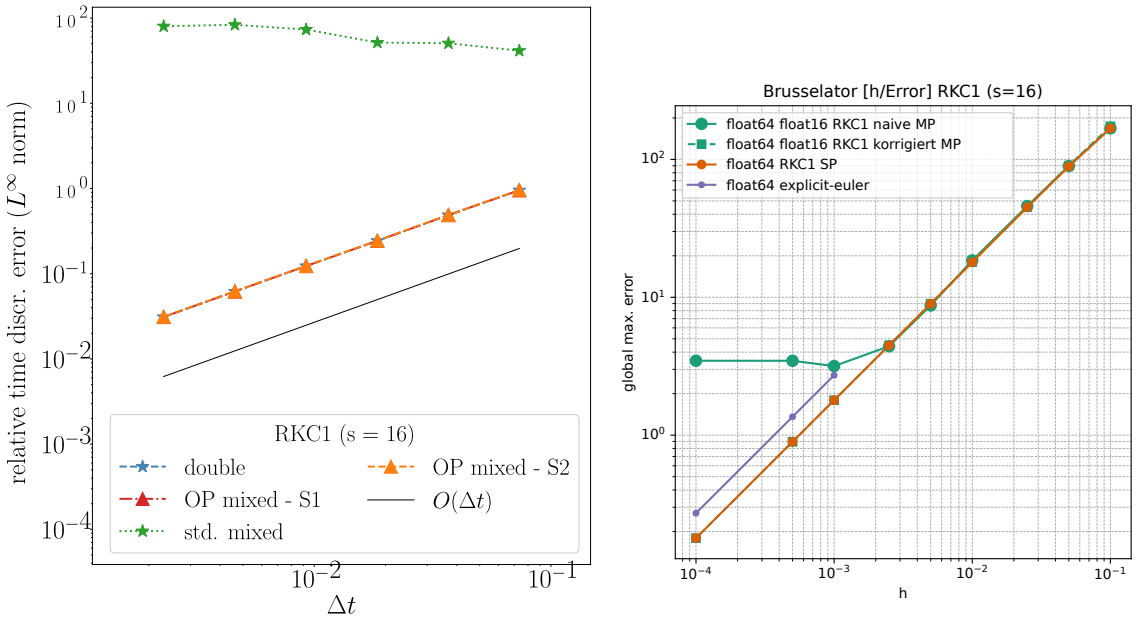
4.4 Mixed-Precision Runge-Kutta-Chebyshev

Als zweite Klasse an Solvern sollen nun noch die expliziten Mixed-Precision-Runge-Kutta-Chebyshev Verfahren betrachtet werden, genauer davon der im Theorieteil beschriebene RKC1-Löser.

Analog zum Vorgehen für die impliziten Verfahren soll zuerst die eigene Implementierung mit den Ergebnissen der Publikation verglichen werden. Grafik 4.11 zeigt dabei das Fehlerdiagramm mit den Ergebnissen des Papers [2] auf der linken Seite und den Ergebnissen der eigenen Implementierung rechts.

Der hier betrachtete Fehler ist der globale maximale Fehler (vgl. Tab. 3.4) dividiert durch den Rundungsfehler ϵ des Reduced-Precision Typs. Im linken Diagramm bezeichnet "OP mixed - S2" die ordnungsbewahrende bzw. korrigierende Mixed-Precision Variante die zur Ermittlung der $\widehat{\Delta f_j}$ Terme die Auswertung der Jacobimatrix verwendet, der Graph zu Szenario S1 nutzt eine alternaitve Methode dafür und ist hier nicht weiter von Belangen. Weiter beschreibt "std. mixed" die naive Variante des Mixed-Precision Verfahrens, die nur Funktionsauswertungen in verringerter Genauigkeit nutzt, während "double" einen Löser der ohne Mixed-Precision nur mit double als Datentyp arbeitet, beschreibt. Auf der rechten Seite findet sich noch zusätzlich ein Graph des Euler-Verfahrens, das als explizites Verfahren mit einer Konvergenzordnung von $q = 1$ einen guten Vergleichswert darstellt, was die Größenordnung der zu erwartenden Ergebnisse angeht.

Wie in der Grafik erkennbar konnten die Ergebnisse der Publikation nicht exakt reproduziert werden. Dabei wurde großen Wert auf die Gleichheit der Umstände gelegt, also dasselbe Anfangswertproblem, die Brusselator Differentialgleichung mit derselben Systemgröße $m = 64$ auf $T = 10$ untersucht. Der Löser wurde ebenfalls unter denselben Bedingungen mit $s = 16$ Stufen ausgeführt. Einzig und allein im Reduced-Precision Typen besteht ein Unterschied, da die Referenz Brain-Floats anstatt normaler IEEE Half-Floats verwendet, allerdings kann dieser auch nicht so groß ausfallen um die Größenordnungen an Unterschied zwischen den beiden Diagrammen zu erklären. Um dessen Einfluss trotzdem so gut wie möglich zu beseitigen und die Größenordnungen der Skalen anzugleichen, wurden die Fehler trotzdem durch den Rundungsfehler ϵ der Brain-Floats anstatt des der Half-Floats dividiert. Trotzdem kann augenscheinlich



(a) Referenz aus [2]

(b) Eigene Implementierung

Abbildung 4.11: Vergleich des Globalen maximalen Fehlers der Publikation mit der eigenen Implementierung des RKC1 Lösers für die Brusselator ODE

das gezeigte Verhalten auch für die Löser die gemeinsam Double-Floats verwenden nicht exakt nachgestellt werden. Hinsichtlich des globalen Fehlers sind die korrigierenden Verfahren auch nicht besser als die nichtkorrigierenden, was auch fraglich erscheint.

Um diese Ergebnisse zu analysieren, betrachten wir zuerst das Fehlerverhalten des Eulerverfahren: Für dieses ist der Graph wie erwartet, bei zu großen Schrittweiten liegt die steife ODE des Brusselators nicht im kleinen Stabilitätsbereich des expliziten Verfahrens, weswegen das Ergebnis dort divergiert, und erst ab einer Schrittweite die klein genug ist wie erwartet konvergiert. Hier zeigt sich, dass die eigene Implementierung anscheinend den Stabilitätsbereich trotzdem erweitert, da sie im Gegensatz zum Eulerverfahren für größere Schrittweiten doch noch konvergiert. Ferner folgt das eigene RKC1-Verfahren derselben Konvergenzordnung des Grades $q = 1$ wie das Eulerverfahren, was auch den Eigenschaften des Lösers entspricht. Für kleinere Schrittweiten kann ebenfalls ein Anstieg des Fehlers für das nichtkorrigerende Verfahren observiert werden, wenn auch dieser nicht in demselben Maß wie im Vergleichsergebnis ausfällt. Man sieht also, dass trotz einer Abweichung der tatsächlichen Zahlenwerte durchaus eine qualitative Ähnlichkeit vorherrscht und die Löser auch die erwarteten Kriterien erfüllen.

Einen vergleichbaren Unterschied zwischen den korrigierenden und nichtkorrigierenden Varianten erhält man erst bei Untersuchung des lokalen maximalen Fehlers anstatt des globalen. Abbildung 4.12 zeigt wie das nichtkorrigerende Verfahren einen deutlich größeren Fehler im letzten Zeitschritt erreicht, der auch von der erwarteten Größenordnung, die das Eulerverfahren darstellt, weit nach abweicht, während die korrigierenden Verfahren die Konvergenzordnung halten können.

Schlussendlich weichen jedoch die tatsächlichen Fehlerwerte immer noch von den Ergebnissen des Papers ab. Auch nachdem das Verfahren und die Koeffizientenberechnung mit mehreren

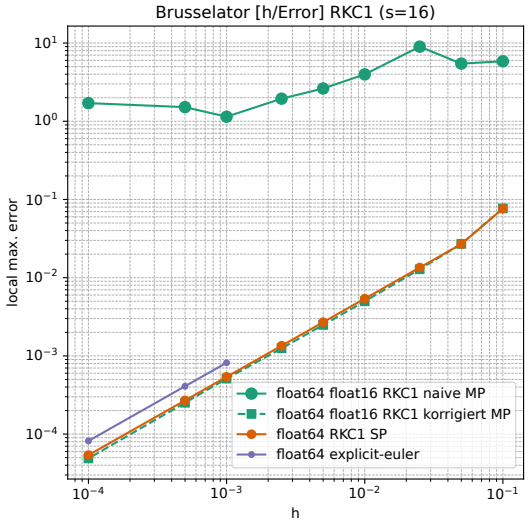


Abbildung 4.12: Vergleich des lokalen Maximalen Fehlers für das nichtkorrigierende und korrigierende Mixed-Precision RKC1-Verfahren

anderen Quellen wie [21], [22] abgeglichen wurde, konnten die Ergebnisse nie exakt repliziert werden. Erschwert wurde die Implementierung durch das Fehlen von direkt verfügbarem Quellcode zu [2] sowie den Mangel an weiterführenden, zitierenden Arbeiten, die Aufschluss darüber geben welche Teile des Verfahrens im Quellcode genau in verringerter Genauigkeit ausgeführt werden sollen. Schlussendlich wirken die Ergebnisse noch zusätzlich verwirrend, da sie zwar das aus der Theorie erwartete qualitative Verhalten aufzeigen, aber trotzdem zahlenmäßig nicht das gleiche Ergebnis liefern können.

Aufgrund dieser gewissen Unsicherheit soll sich die Betrachtung der Posits für dieses Verfahren auf ein Minimum beschränken. In Diagramm 4.13 werden die Floats gegen ihre in Zahlenbereich äquivalenten Posit-Formate (aus Tabelle 4.1) ausgetauscht, und, um etwas mehr Vergleichsfälle zu haben zu dem Paar aus 64 und 16 Bit Typen noch die anderen Vergleichstypen der vorherigen Evaluation von ImpMid übernommen.

Interessanterweise sind hier zum ersten Mal die Posits sichtbar schlechter als die Float-Formate, spezifisch die Verfahren die `posit16,1` als Reduced-Precision Typen verwenden. Beidermaßen die naive/nichtkorrigierende sowie die korrigierende Variante liefern eine größere Abweichung als die äquivalenten Float-Typen für diese Verfahren. Da diese Umstände für RP Typen höherer Genauigkeit nicht gegeben sind, liegt die Vermutung nahe, dass dieses Verhalten `posit16,2` geschuldet sein muss. Damit `posit16,2` einen solchen Genauigkeitsverlust im Vergleich zu `float16` bewirkt, muss an einer Stelle im Löser oder Differentialgleichungssystem mit sehr kleinen oder sehr großen Zahlen gerechnet worden sein. Eine mögliche Erklärung dafür könnte die vergleichsweise große Stufenzahl sein, die sehr kleine Koeffizienten und Teilschritte hervorruft, welche diese kleinen Posits aufgrund wachsender Regime Mantissenbits und damit an Genauigkeit verlieren lässt.

Bei genauerem Betrachten des rechten Diagramms aus Abb. 4.13 fällt auf, dass interessanterweise der Punkt ab dem `posit32,2` `posit16,1`(lila) von der Konvergenzordnung abkommt

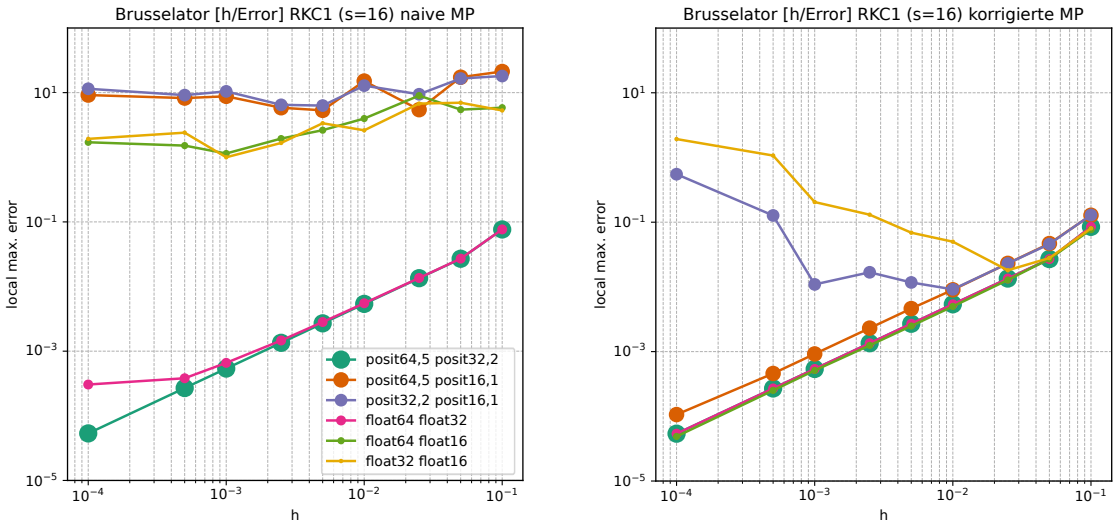


Abbildung 4.13: Fehlerbetrachtung von im Zahlenbereich angeglichenen Posits für den Mixed-Precision RKC1 Löser.

erst für kleinere Schrittweiten eintritt, als für `float32 float16` (gelb), obwohl für größere Schrittweiten die Posits zuerst größere Fehler verursachen als die Floats. Hier kann man die Hypothese, dass dieser ursprüngliche Fehler für große h am Genauigkeitsverlust von `posit16,1` liegt, darauf erweitern, dass sich dieser bei `posit32,2` im Vergleich zu `float32` anscheinend nicht bemerkbar macht. Stattdessen scheinen die Posits für 32 Bit mehr Genauigkeit zu bieten, als ihre Float-Äquivalente. Dies könnte am größeren $es = 2$ liegen, weswegen `posit32,2` nur ein halb so langes Regime benötigt um die gleichen Größenordnungen wie `posit16,1` darzustellen. Anstatt wie dieses nun aufgrund der kleinen Zwischenschritte an Genauigkeit einzubüßen, könnten diese Werte als für `posit32,2` noch groß genug sein, dass das Format noch ein kurzes Regime hat, also mehr Mantissebits als `float32` für die Genauigkeit des Ergebnisses ausnutzen kann.

4.5 Nutzen komprimierter Vektoren

Nach der Betrachtung der Solver, soll hier noch ein Blick auf eines der sie umgebenden Systeme geworfen werden. Das größte davon ist das in Abschnitt 3.5 vorgestellte zur Komprimierung von Vektoren. Für alle bisher betrachteten Tests wurden dabei stets unkomprimierte Vektoren betrachtet, so würden also die Posits für die Untersuchungen in Abschnitt 4.3.2 zu den kürzeren Bitlängen trotzdem so viel Speicher wie die Floats benötigen, da die übrigen Bits ihres Containers frei und ungenutzt bleiben.

Im Folgenden soll der Nutzen dieser verdichteten Vektoren untersucht werden. Besonders interessant neben dem Ausmaß der Speicherplatzersparnis ist die Frage, ob die Reduzierung der übertragenen Speichermenge eine Beschleunigung des Programms erwirken kann. Zur Untersuchung dessen wird ein Testfall konstruiert, für den der Speichergewinn optimal wäre: Posits die gerade um 1 Bit größer sind als die nächstkleinere Containergröße, also 9, 17, oder

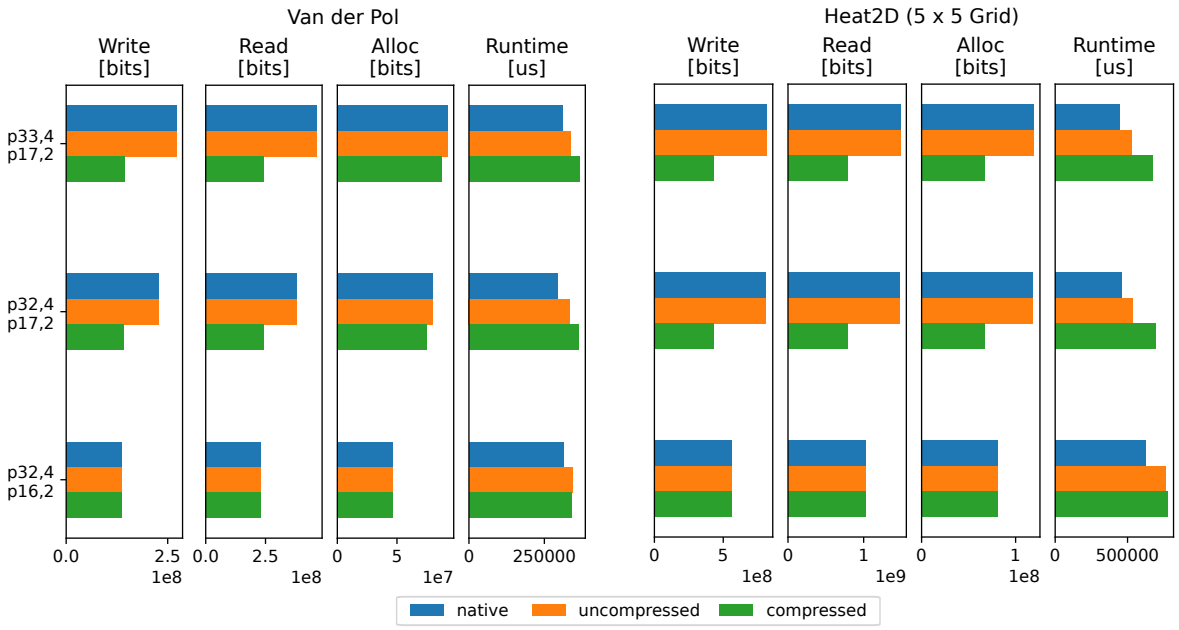


Abbildung 4.14: Vergleich von Lese- / Schreibezugriffen, Allokationen und Laufzeit für verschiedene Datentypen mit verschiedenen Varianten der Vektorkomprimierung, verglichen zwischen den AWP Van-der-Pol und Heat2D, gelöst vom Mixed-Precision Implicit-Mitpoint Löser mit $p = 2$ und Schrittweiten $h = 10^{-5}$ bzw. $h = 10^{-3}$ für Heat2d.

33 Bit. Um vergleichbare Ergebnisse zu bekommen und die Auswirkungen auf die Laufzeit besser einschätzen zu können, werden alle drei Varianten der Vektorkomprimierung betrachtet: Native Arrazugriffe, unkomprimierte Zugriffe durch eine Funktion, um den Overhead der Funktionsaufrufe bemessen zu können, sowie zuletzt die Zugriffe auf das kompaktierte Array.

Diagramm 4.14 zeigt die Statistiken der drei Varianten für das Van-der-Pol Problem bzw. die Wärmeleitungsgleichung, gelöst vom Löser der Impliziten Mittelpunktsregel. *Write* und *Read* bezeichnen dabei die Zahl der im Idealfall geschriebenen bzw. gelesenen Bits, also wenn die Kompaktierung keine weiteren Zugriffe bräuchte. D.h. in der komprimierten Variante werden für einen Lesezugriff auf einen `posit17` auch 17 Bits gezählt ungeachtet etwaiger Zusatzzugriffe für die Adressberechung oder auf weitere Speicherzellen, wenn der 17 Bit Posit zwei Container überlappt. Mit *Alloc* wird stattdessen der tatsächlich allokierte Speicher gezählt, und damit auch die leeren Bits die am Ende der Kompaktierung anfallen. Die letzte Spalte zeigt die Laufzeiten der Solver je nach Variante. Da sich keine Parameter für die Solver ändern, sie also alle dieselbe Anzahl an Operationen durchführen, müssten diese Laufzeiten exakt gleich sein, wenn jeder Speicherzugriff exakt gleich lang wäre. Damit zählt also die Gesamtlaufzeit als Maß dafür die lange die einzelnen Speicherzugriffe der Varianten im Vergleich zueinander brauchen. Die Messungen wurden durchgeführt für die drei Konfigurationen WP RP `posit32,4 posit16,2`, `posit32,4 posit17,2` und `posit33,4 posit17,2` deren Kurzschreibweisen die Achsen beschriften.

An den Diagrammen für beide Problemfälle lässt sich wie zu erwarten eine Speicherersparnis bei den Lese- und Schreibmengen feststellen. Durch Komprimierung müssen für einen `posit17` nur noch 17 Bits statt den 32 der Containergröße geladen werden, was einer Ersparnis von

etwas unter der Hälfte entspricht. Genauso kann man betrachten, dass die Lese- und Schreibwerte für die komprimierten Varianten etwa die Hälfte der nichtkomprimierten Variante betragen, wobei die Ersparnis für `posit32 posit17` schon recht nahe an der von `posit33 posit17` liegt, was darin begründet ist, dass die Speicherintensiven Operanden wie die Matrizen zur Lösung des impliziten Gleichungssystems des Solvers nur in Reduced-Precision vorliegen. Durch kompaktierte Zugriffe des Work-Precision Typs lässt sich also keine große Speicherersparnis erzielen.

Geht man von Lese- und Schreibzahlen zu den allokierten Bits über, so fällt auf, dass für Vander-Pol kaum Speicher eingespart werden kann. Das liegt daran, dass dieses AWP nur eine kleine Dimension $d = 2$ hat, weswegen die Vektoren zur Lösung des Problems aus zwei und die Matrizen aus 4 Elementen bestehen. Diese können nur in zwei bzw. drei Container kompaktiert werden ($2 \cdot 17 = 34 > 32$ bzw. $4 \cdot 17 = 68 > 64$) wobei noch immer viele Bits leer bleiben (30 für Vektoren, 28 für Matrizen), weswegen hier die Menge allokierten Speichers nicht wirklich abgenommen hat. Für speicheraufwändigere Probleme hingegen, wie die Wärmeleitgleichung die hier mit $m = 5$ ein AWP der Dimension 25 darstellt, kann man maßgebliche Einsparungen der Speicherallokationen feststellen, die proportional den Lese- und Schreibmengen ähnlich sind.

Der Blick auf die Laufzeiten fällt dagegen eher ernüchternd aus: Die komprimierten Varianten benötigen in allen Fällen mehr Zeit um ihr Ergebnis zu berechnen, was Aufgrund der Parametergleichheit nur auf den zusätzlichen Aufwand bei Speicherzugriffen zurückführbar ist. Dabei kommt ein kleiner Anteil aus dem zusätzlichen Funktionsaufruf, dieser Overhead lässt sich in den Laufzeiten der *uncompressed* Variante auch beim Vergleichsfall `posit32 posit16` beobachten. Dazu kommt noch der Aufwand der Indexberechnung der sich in einer weiteren Verlangsamung bemerkbar macht. Für `posit32 posit16` entfällt dieser komplett, da die Indexberechnung vom Präprozessor entfernt wird, sobald die Bitlänge der Containerlänge entspricht, weswegen die Laufzeiten hier der *uncompressed* Variante so gleich sind.

Beim Rückblick auf die Ergebnisse der Performanceevaluation der Posits in Abschnitt 3.2.1 zeigt sich, dass diese Ergebnisse so zu erwarten sind. Bei einer Rechengeschwindigkeit von etwa 70 MPOPs benötigt eine Berechnung $1/(70 \cdot 10^6) \approx 14,3\text{ns}$ was über der Speicherlatenzzeit von etwa $10,6\text{ns}$ des Testrechners liegt. Folglich ist die Posit-Softwaresimulation Compute-bound, weswegen sich durch sparsamere und dadurch schnellere Speicherzugriffe keine schnellere Berechnung der Ergebnisse erreichen lässt, bzw. der Zusatzaufwand für die Indexberechnung auch nicht in diesem optimalen betrachteten Fall durch schnellere Speicherzugriffe kompensiert werden kann. Hier sei noch angemerkt, dass dieses Verhalten in diesen Betrachtungen strikt an die Eigenschaften der Softwaresimulation gekoppelt ist und bei einer Hardwareunterstützung der Posits wohl andere Ergebnisse zu erwarten wären.

Die Vektorkomprimierung bietet also eine Speicherersparnis für große ODE-Systeme, die allerdings mit zusätzlicher Ausführungsdauer bezahlt wird. Sie könnte also dann nutzen, wenn die ODE-Systeme sehr groß werden, sodass sie ohne Kompaktierung nicht mehr in den Speicher passen würden.

5 Zusammenfassung

5.1 Bewertung

Wie in mehreren Auswertungen exemplarisch gezeigt, können Posits durchaus zu einem Vorteil bei der Verwendung in Runge-Kutta-Verfahren verhelfen. Dank ihrer Tapered-Precision ermöglichen sie es, mehr Bits der Gesamtbitlänge für die Mantisse zu nutzen und damit den Rundungsfehler zu senken und die allgemeine Genauigkeit zu erhöhen. Diese Eigenschaft verhilft ihnen auch dazu, mit weniger Bits als Floats dieselbe Ergebnisgenauigkeit zu erreichen.

Posits haben sich auch im Allgemeinen als tauglich für Mixed-Precision Ansätze erwiesen und konnten mit diesen Verfahren und Korrekturmethoden dieselbe Verbesserung wie auch Floats verzeichnen, und zusätzlich davon noch von ihrer gesteigerten Genauigkeit profitieren. Auch für steife Differentialgleichungen konnte ein etwa gleichgutes, wenn nicht besseres Fehlerverhalten beobachtet werden.

Trotz ihrer guten Eignung kann man Posits jedoch nicht als allgemein besser pauschalisieren. So wird der Gewinn an Genauigkeit auf einem passenden Intervall von Zahlen nahe der Eins durch einen Genauigkeitsverlust für betragsmäßig sehr große und sehr kleine Zahlen erkauft. Dieser kann sich besonders bei Formaten kleiner Bitlängen schnell bemerkbar machen, wie bei der Auswertung der Runge-Kutta-Chebyshev-Verfahren aufgefallen ist.

Des Weiteren fällt dieser Gewinn an Genaugkeit natürlich relativ zu den verschiedenen Bitlängen aus. Während es für `posit16,2` eine relativ größere Verbesserung ist, im besten Falle 2 Mantissenbis mehr zur Verfügung stehen zu haben als `float16` mit 10, so wirkt sich eine Verbesserung um maximal 4 weitere Mantissenbits für `posit64,5` relativ betrachtet nicht so stark, im Vergleich zu `float64` die sowieso schon 52 Mantissenbits zur Verfügung stehen haben, aus.

Was die Geschwindigkeit angeht, so lässt sich aus dieser Arbeit heraus keine gute Schlussfolgerung ziehen, da die Softwaresimulation direkte Zeitmessungen aussagenlos macht. Im direkten Vergleich der benötigten Rechenoperationen zum Lösen der Runge-Kutta-Verfahren konnten für die Posits keine nennenswerten Vor- oder Nachteile festgestellt werden.

Rundum lässt sich schlussfolgern, dass Posits durchaus eine ernsthafte Alternative zu den alteingesessenen IEEE-754 Floats darstellen können und in der Regel vergleichbare oder gar bessere Ergebnisse liefern können. Schlussendlich wird die Güte des Ergebnisses durch die genaue Wahl des Posit-Formats und den Umgebungsbedingugen des numerischen Verfahrens beeinflusst, was durch eine günstige Wahl von Formaten durchaus zu Vorteilen führen kann.

5.2 Ausblick

Die wohl maßgeblichste Einschränkung des untersuchten Frameworks ist die Softwaresimulation der Posits, die Laufzeitmessungen nicht repräsentativ macht. Da für solche Laufzeitmessun-

gen vor allem die Abhängigkeit der Speicherzugriffsgeschwindigkeit, die man durch sparsamere Datentypen verringern kann, von großem Interesse ist, sind diese Evaluationen besser mit einer Hardwareunterstützung oder schnellen FPGA-Simulation zu tätigen, um ein Programm zu erhalten, dass nicht Compute-Bound ist. Die in den Betrachtungen festgestellte Fähigkeit der Posits, mit weniger Bits dieselbe Ergebnisgenauigkeit erzeugen zu können, deutet darauf hin, dass sich hier durch die Speicherersparnis eventuell deutliche Verbesserungen verzeichnen ließen, wenn das Programm Memory-Bound ist, wie es z.B. die impliziten RKV für große ODE-Systeme in der Regel sind.

Für die praktische Verwendung von Posits außerhalb einer Softwaresimulation eröffnen sich wieder neue Probleme: Für eine Hardwareimplementierung ist die Wahl konkreter Bit- und Exponentenlängen notwendig, welche maßgeblich das Fehlerverhalten der Formate bestimmt und dem Endnutzer die freie Wahl des zum Problem am besten passenden Formates nimmt. Zudem zeigen andere Betrachtungen [8], dass die Hardwareimplementierung von Posits trotz der Einsparung zusätzlicher Logik für Sonderfälle im Allgemeinen mehr Transistoren und Platz auf einem Prozessordie benötigt, was unter anderem einen gesteigerten Energieverbrauch bedeuten kann. Dies sind beides Metriken die in dieser Arbeit aufgrund der Softwaresimulation nicht betrachtet werden konnten.

An sich hat das Framework mit der Softwaresimulation auch noch Potential zur Verbesserung, da der Hautpaugenmerk auf einer Bemessung des Fehlerverhaltens lag. Ein erster Anhaltspunkt wäre eine Beschleunigung der BLAS-Funktion und Löser durch bessere Numerische Algorithmen oder eine Parallelsierung. Um die Posit-Operationen an sich weiter zu beschleunigen, könnte von den SIMD-Instruktionen der CPU Gebrauch gemacht werden um z.B. Skalarprodukte für Posits zu beschleunigen. Dies wurde in anderen Publikationen [31] schon auf der RISC-V Architektur aufgegriffen und würde sich eventuell durch Ausnutzung der neuen Befehlssatzerweiterungen von AVX512 auch auf X86-Architekturen anbieten.

Im Rahmen der Erweiterung der Softwaresimulation würde sich wohl auch der Wechsel auf eine höhere Programmiersprache als C anbieten, da vor allem der Mangel modernerer Features wie generischer Templates, Operatorenüberladung und geschachtelter Funktionen bei der Programmierung immer wieder ein Nachhelfen mit dem Präprozessor erfordert haben, was den gesamten Build-Prozess auf Dauer immer weiter verkompliziert hat.

Weiterhin gäbe es noch Optionen die verfügbaren Posit-Formate besser zu evaluieren. Während hier bei der manuellen Auswertung der Fokus nur auf einer kleinen Auswahl an konstruierten Formaten liegen konnte, so ließe sich beispielsweise mittels automatischer Optimierungsalgorithmen maschinell aus der Vielzahl einstellbarer Parameter die beste Kombination von WP und RP Formaten ermitteln. Neben den Lösungsverfahren für Differentialgleichungen gibt es aber auch noch andere Anwendungen in denen Posits im Rahmen von Mixed-Precision-Verfahren untersucht werden könnten. Dazu existieren bereits Untersuchungen zu Lösungsverfahren von Gleichungssystemen mittels ”Iterative Refinement” [10].

Zusammenfassend sind Posits aktuell ein aktiv weiterentwickeltes Feld der Wissenschaft, in dem noch viele Dinge zur Untersuchung offen stehen, zu denen diese Arbeit hoffentlich einen kleinen Anteil geschafft hat beizutragen.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wie auch Stellen in der Arbeit, welche aus anderen Quellen genutzt wurden, kenntlich gemacht sind. Zudem bestätige ich hiermit, dass diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht wurde.

David Schuberth, Bayreuth den 11. Juli 2025

Literatur

[1] Z. J. Grant, „Perturbed Runge–Kutta methods for mixed precision applications," *Journal of Scientific Computing*, Jg. 92, Nr. 1, S. 6, 2022.

[2] M. Croci und G. R. de Souza, „Mixed-precision explicit stabilized Runge–Kutta methods for single-and multi-scale differential equations," *Journal of Computational Physics*, Jg. 464, S. 111 349, 2022.

[3] J. L. Gustafson und I. T. Yonemoto, „Beating floating point at its own game: Posit arithmetic," *Supercomputing frontiers and innovations*, Jg. 4, Nr. 2, S. 71–86, 2017.

[4] A. Abdelfattah u. a., „A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, Jg. 35, Nr. 4, S. 344–369, 2021.

[5] B. Burnett, S. Gottlieb und Z. J. Grant, „Stability analysis and performance evaluation of additive mixed-precision Runge-Kutta methods," *Communications on Applied Mathematics and Computation*, Jg. 6, Nr. 1, S. 705–738, 2024.

[6] B. Burnett, S. Gottlieb, Z. J. Grant und A. Heryudono, „Performance evaluation of mixed-precision Runge-Kutta methods," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2021, S. 1–6.

[7] I. Dravins, M. Koch, V. Griehl und K. Kormann, „Performance evaluation of mixed-precision Runge-Kutta methods for the solution of partial differential equations," *arXiv preprint arXiv:2412.16638*, 2024.

[8] R. Murillo Montero, „Posit arithmetic and its applications," 2025.

[9] Z. Carmichael, H. F. Langroudi, C. Khazanov, J. Lillie, J. L. Gustafson und D. Kudithipudi, „Deep positron: A deep neural network using the posit number system," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019, S. 1421–1426.

[10] J. Quinlan und E. T. L. Omtzigt, „Iterative Refinement with Low-Precision Posit Arithmetic," in *Conference on Next Generation Arithmetic*, Springer, 2024, S. 74–90.

[11] F. Netsch, „Experimentelle Analyse und Evaluation von Runge-Kutta-Verfahren mit gemischter Genauigkeit," 2022.

[12] F. Netsch, „Parallele Implementierung und Performanz-Optimierung von Runge-Kutta-Verfahren mit gemischter Genauigkeit unter Berücksichtigung numerischer Parameter," 2024.

[13] E. Kögler, „Implementierung und Nutzenanalyse des Posit-Zahlenformats im Vergleich zu Floating Point," 2021.

[14] J.-M. Muller u. a., *Handbook of floating-point arithmetic*. Springer, 2018.

[15] „IEEE Standard for Floating-Point Arithmetic,“ IEEE Computer Society, New York, NY, USA, Standard IEEE Std 754-2008, Aug. 2008. Adresse: https://web.archive.org/web/20160806053349/http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf.

[16] J. L. Gustafson, *The end of error: Unum computing*. Chapman und Hall/CRC, 2017.

[17] P. W. Group, *Standard for Posit Arithmetic*, Posit Working Group, 2022.

[18] S. H. Leong, *SoftPosit*, März 2020. DOI: 10.5281/zenodo.3709035. Adresse: https://doi.org/10.5281/zenodo.3709035.

[19] J. Gustafson, *Understanding Bankers' Rounding for Posits*, 2019. Adresse: https://groups.google.com/g/unum-computing/c/Qhx87YUFSP8/m/j0X232lVBAAJ.

[20] L. Grüne. „Numerische Methoden für Differentialgleichungen. “Adresse: https://num.math.uni-bayreuth.de/de/team/lars-gruene/skripten/num2/num2_7.pdf.

[21] A. Abdulle, „Explicit stabilized runge-kutta methods,“ *Encyclopedia of Applied and Computational Mathematics*, S. 460–468, 2015.

[22] J. G. Verwer, W. H. Hundsdorfer und B. P. Sommeijer, „Convergence properties of the Runge-Kutta-Chebyshev method,“ *Numerische Mathematik*, Jg. 57, S. 157–178, 1990.

[23] J. C. Mason und D. C. Handscomb, *Chebyshev polynomials*. Chapman und Hall/CRC, 2002.

[24] M. K. Jaiswal und H. K.-H. So, „PACoGen: A Hardware Posit Arithmetic Core Generator,“ *IEEE Access*, Jg. 7, S. 74 586–74 601, 2019. DOI: 10.1109/ACCESS.2019.2920936.

[25] N. Team, *Survey of Posit Hardware and Software Development Efforts (July 2019)*, Posit Working Group, 2019. Adresse: https://posithub.org/docs/PDS/PositEffortsSurvey.html.

[26] E. T. L. Omtzigt und J. Quinlan, „Universal Numbers Library: Multi-format Variable Precision Arithmetic Library,“ *Journal of Open Source Software*, Jg. 8, Nr. 83, S. 5072, 2023. DOI: 10.21105/joss.05072. Adresse: https://doi.org/10.21105/joss.05072.

[27] E. Anderson u. a., *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial und Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback).

[28] L. Grüne und M. H. Baumann. „Numerische Mathematik für Naturwissenschaftler, Ingenieure und Informatiker. “Adresse: https://num.math.uni-bayreuth.de/de/team/lars-gruene/skripten/num_ing/nni23.pdf.

[29] N. Henning, *utest.h*, 2015. Adresse: https://github.com/sheredom/utest.h.

[30] E. Hairer und G. Wanner, „Stability Analysis for Explicit RK Methods,“ in *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, S. 15–39, ISBN: 978-3-642-05221-7. DOI: 10.1007/978-3-642-05221-7_2. Adresse: https://doi.org/10.1007/978-3-642-05221-7_2.

[31] M. Cococcioni, F. Rossi, E. Ruffaldi und S. Saponara, „Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE,“ *Neural Computing and Applications*, Jg. 33, S. 10 575–10 585, 2021.

Abbildungsverzeichnis