Universität Stuttgart

Fakultät 5 / IPVS-SC

Universitätsstraße 38
70569 Stuttgart

Bachelorarbeit

Hierarchische Funktionsdarstellung mit Posits

Nils Georg Hoyler

Studiengang:	Informatik
1. Prüfer:	Prof. Dr. Dirk Pflüger
2. Prüfer:	
Betreuer:	Dr. Stefan Zimmer
begonnen am:	26.04.2023
beendet am:	26.10.2023

Zusammenfassung

Durch die Verbreitung der Dünnen Gitter zur Diskretisierung haben hierarchische Basen zunehmend mehr Anwendung gefunden. Wir haben in dieser Arbeit Methoden erarbeitet, welche den Speicheraufwand für eine Approximation einer zweifach stetig differenzierbaren Funktion um mehrere Größungsordnungen reduzieren kann. Dazu nutzen wir die Eigenschaften von zweimal stetig differenzierbaren Funktionen in hierarchischen Basen und eine IEEE754-Alternative namens Posits. Dies bildet den Grundstein für die Reduzierung des Speicherbedarfs von Simulationen und Berechnungen mit Dünnen Gitter, was dann neue Probleme berechenbar macht.

Inhaltsverzeichnis

1 Einleitung

Im Bereich des Wissenschaftlichen Rechnens haben Dünne Gitter, eine spezielle Art der Diskretisierung, Einzug erhalten. Diese Methode erlaubt es, im Gegensatz zu regulären Gittern, den Einfluss des Fluchs der Dimensionen zu reduzieren und so Probleme berechenbar zu machen. Dabei werden hierarchischen Basen zur Approximation der Funktionen genutzt. Obwohl die Methode durch die Verwendung von hierarchischen Basen den Speicherbedarf für d Dimensionen und n Auswertungstellen von $\mathcal{O}(n^d)$ auf $\mathcal{O}(n * log(n)^{d-1})$ deutlich reduziert, benötigt man für viele relevante Probleme viel Speicher [2]. Deshalb beschäftigt sich diese Arbeit mit Methoden zur Reduktion dieses Speicheraufwands. Dazu nutzen wir die Eigenschaften von zweimal stetig differenzierbaren Funktionen in hierarchischen Basen und eine IEEE754-Alternative namens Posits. Diese bieten bei gleichem Speicherbedarf eine größere Dynamik und für den Großteil des Zahlenbereichs eine höhere Genauigkeit [3].

1.1 Hierarchische Basen

1.1.1 Definition

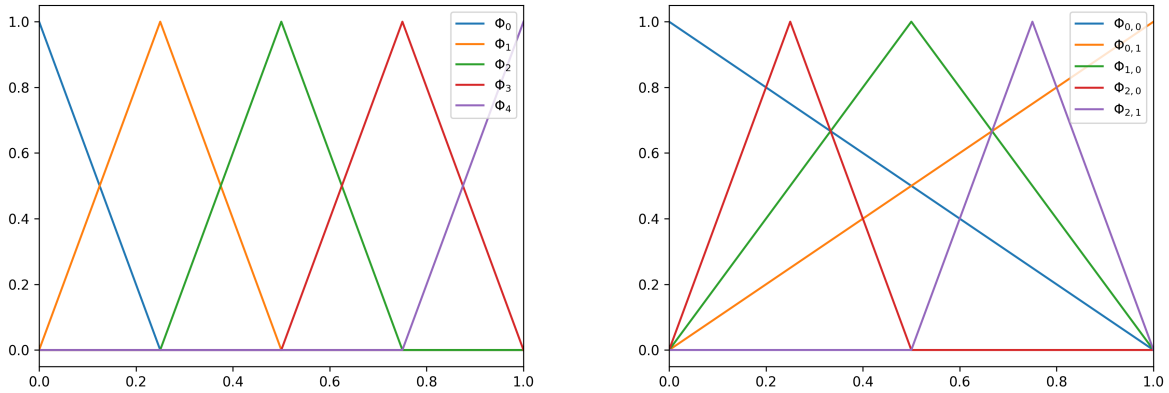
Da diese Arbeit sich auf die Approximation einer eindimensionalen, reellen Funktion im Intervall zwischen 0 und 1 beschränkt, sind die Ansatzfunktionen für eine Tiefe d und einem Index j wie folgt definiert:

$$\Phi_{d,j} = \begin{cases} 1 - |x| & \text{wenn } d = 0,\ j = 0 \text{ und } x \in [0,1] \\ 1 - |x - 1| & \text{wenn } d = 0,\ j = 1 \text{ und } x \in [0,1] \\ 1 - |2^d x - 2j - 1| & \text{wenn } d \neq 0 \text{ und } x \in [\frac{j}{2^{d-1}}, \frac{j+1}{2^{d-1}}] \cap [0,1] \\ 0 & \text{sonst} \end{cases} \tag{1.1.1}$$

Für eine Tiefe $d > 0$ gibt es 2^{d-1} Ansatzfunktionen, welche durch die Variation des Index j von 0 bis 2^{d-1} bestimmt werden können. Somit erhält man folgende Basis für eine Tiefe d:

$$(\Phi_{0,0}, \Phi_{0,1}, \Phi_{1,0}, \Phi_{2,0}, \Phi_{2,1}, \Phi_{3,0}, \ \dots\ , \Phi_{d,0}, \ \dots\ \Phi_{d,2^{d-1}}) \tag{1.1.2}$$

Diese Basis spannt den gleichen Funktionsraum auf, wie eine nodale Basis gleicher Intervalllänge. In Abbildung 1.1 sind die Ansatzfunktionen für diese beiden Arten von Basen zum Vergleich dargestellt.



(a) Nodale Basis (b) hierarchische Basis

Abbildung 1.1: Basen für eine Intervalllänge von 0,25

Die Approximation einer Funktion f ist dann eine Linearkombination der Ansatzfunktionen, wobei $a_{d,j}$ die entsprechenden Koeffizienten sind:

$$f_{\text{approx}} = a_{0,0}\Phi_{0,0} + a_{0,1}\Phi_{0,1} + \sum_{d=1}^{N} \sum_{i=0}^{2^{d-1}-1} a_{d,i}\Phi_{d,i} \tag{1.1.3}$$

Diese Koeffizienten lassen sich leicht berechnen, da sie die Abweichung zur Linearinterpolation der beiden Nachbarauswertestellen aus niedrigeren Auswertungstiefen sind.
Aufgrund dieser hierarchischen Auswertung ergibt es Sinn die Koeffizienten nach den Abhängigkeiten der entsprechende Basen, wie in Abbildung 1.2 zu ordnen.

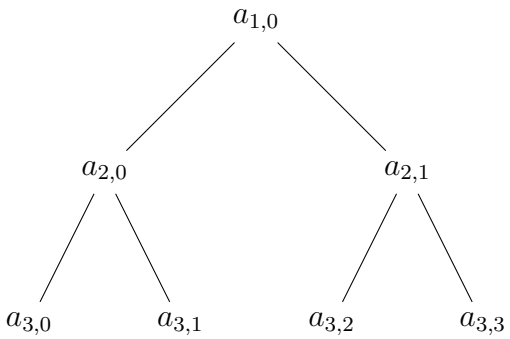


Abbildung 1.2: Baumstruktur der Koeffizienten mit einer Auswertungstiefe von 3

Dabei bezeichnen wir den Abstand eines Koeffizienten zu der Wurzel plus 1 als Tiefe eines Koeffizienten und die maximale Tiefe der Koeffizienten als Auswertungstiefe. Wenn wir in dieser Arbeit vom Level eines Koeffizienten sprechen, meinen wir damit die Auswertungstiefe minus die Tiefe des Koeffizienten. [2]

1.1.2 Eigenschaften

Die hierarchische Basen besitzen mehrere Eigenschaften, mit denen sie sich von den nodalen Basen unterscheiden. So bleiben alle Ansatzfunktionen bestehen, wenn man die Intervalllänge halbiert. Es werden nur neue Ansatzfunktionen der Basis hinzugefügt. Dies ermöglicht ein iteratives Vorgehen. Des Weiteren gilt, wie in der Literatur [2] hinlänglich bekannt, dass bei der Approximation von eindimensionalen, zweimal stetig differenzierbaren Funktionen die Koeffizienten ab einer Auswertungstiefe mit jeder Tiefe um $\frac{1}{4}$ abfallen.
Die Koeffizienten der nächsten Auswertungstiefe können generell als Abschätzung des Fehlers nach unten dienen, da sie die Abweichung an einer Stelle der bisherigen Approximation sind. Wenn man nun annimmt, dass die zu approximierende Funktion hinreichend glatt ist, fallen alle weiteren Koeffizienten ab und sind somit kleiner. In diesem Fall können die Koeffizienten auch zu einer Abschätzung des Fehlers nach oben dienen.
Die Möglichkeit den Fehler abschätzen zu können und die Funktion iterativ auszuwerten, ermöglicht ein adaptives Vorgehen bei der Approximation der Funktion.

1.2 Posit

Ein Posit ist ein Datentyp, welcher 2017 von John L. Gustafson und Isaac Yonemoto als Drop-in-replacement für IEEE754 Floats entwickelt wurde. Dieser bietet bei gleichem Speicherbedarf eine größere Dynamik und für den Großteil des Zahlenbereichs eine bessere Genauigkeit. Da er aber noch nicht von gängiger Hardware unterstützt wird, verläuft die Verbreitung noch stockend.

1.2.1 Definition

Posits haben eine feste Länge und bestehen im Gegensatz zu IEEE754 Floats aus 4 Teilen: dem Vorzeichenbit (s), den Regimbits (R), den Exponentenbits (E) und den Mantissenbits (F). Diese sind dabei wie in Abbildung 1.3 angeordnet.

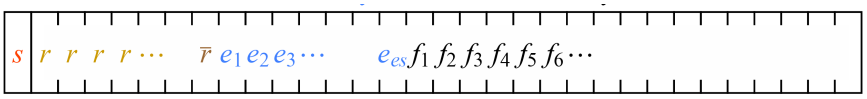


Abbildung 1.3: Bitaufteilung von Posits, mit Vorzeichen s, Regimbits r, Exponentenbits e und Mantissenbits f [3]

Das Vorzeichenbit gibt das Vorzeichen an, wobei eine Eins für ein negatives Vorzeichen steht. Die Exponentenbits haben eine feste Länge es und beschreiben eine positive Zahl e im Binärformat. Die Regimbits bestehen aus einer Folge von Einsen gefolgt von einer Null, oder aus einer Folge von Nullen gefolgt von einer Eins. Diese kodieren eine Zahl k. Bei einer Folge von n Nullen und einer Eins ist k gleich -n und bei einer Folge von n Einsen gefolgt von einer Null ist k gleich $n-1$.
Die Mantissenbits $f_1 f_2 f_3 f_4 f_5$... beschreiben eine rationale Zahl f, welche den Wert der Binärzahl $1, f_1 f_2 f_3 f_4 f_5$... hat.

Der Zahlenwert x eines Posits ist wie folgt definiert:

$$x = \begin{cases} 0 & \text{wenn alle Bits gleich 0 sind} \\ \pm\infty & \text{wenn alle Bits gleich 1 sind} \\ -1^s * \text{useed}^k * 2^e * f & \text{sonst} \end{cases} \tag{1.2.1}$$

Dabei hängt die Konstante *useed* von der Größe des Exponenten *es* ab und ist wie folgt definiert:

$$ussed = 2^{2^{\text{es}}} \tag{1.2.2}$$

Somit nutzt man die Regimbits, um größere Exponenten zu speichern. Jedes Mal wenn die positive Zahl e einen Overflow produziert, werden Regimebitfolgen aus Einsen eine Eins hinzugefügt und von Regimebitfolgen aus Nullen eine Null entfernt. Äquivalent dazu wird jedes Mal, wenn die positive Zahl e einen Underflow produziert, von Regimebitfolgen aus Einsen eine Eins entfernt und Regimebitfolgen aus Nullen eine Null hinzugefügt. Deshalb variiert bei Posits die Länge der Mantisse anhand des Werts von dem Exponenten. Wenn der Betrag des Exponenten groß ist, dann benötigt man viele Regimbits und die Anzahl der Mantissenbits nimmt ab. Wenn dagegen der Betrag des Exponenten klein ist, dann benötigt man weniger Regimbits und die Anzahl an Mantissenbits nimmt zu. Somit erhält man mit Posits eine höhere Genauigkeit für Werte um die Eins. [3]

2 Methoden

Um die verschiedenen Methoden zur Reduktion des Speicherbedarfs zu testen und zu vergleichen, verwenden wir Beispielfunktionen. Diese approximieren wir mit hierarchischen Basen und reduzieren dann systematisch den Speicherbedarf und die Genauigkeit der Koeffizienten. Anschließend vergleichen wir die Methoden zur Reduzierung anhand des Fehlers der Approximation und dem dafür benötigten Speicherbedarf.

2.1 Beispielfunktionen

Die gewählten Beispielfunktionen sind alle zweimal stetig differenzierbar, da so die Koeffizienten der hierarchischen Basen ab einer bestimmten Tiefe mit einem Viertel abfallen (siehe Kapitel 1.1.2). Des Weiteren bilden die Funktionen vom Intervall von 0 bis 1 ab. Funktionen, welche von einem anderen Intervall aus abbilden, können durch Skalierung der Definitionsmenge und Verschiebung in eine Funktion mit dem Intervall von 0 bis 1 umgeformt werden. Für die Werte der Randwerte der Funktionen gilt, dass sie 0 sind. Falls eine gegebene Funktion das nicht erfüllt, kann diese durch Addition einer Geraden in eine solche Funktion überführt werden (siehe beispielsweise Gleichung 2.1.3). Somit müssen die Koeffizienten am Rand nicht gespeichert werden und benötigen damit keinen Speicher.
Als erste Beispielfunktion verwenden wir eine um 0,1 skalierte Normalparabel, da die Koeffizienten der hierarchischen Basen von Beginn an mit zunehmender Auswertungstiefe immer um ein Viertel abfallen. Wir skalieren diese mit 0,1. So sind die Koeffizienten nicht alle Zweierpotenzen. Außerdem kann die Mantisse im Binärformat nicht genau dargestellt werden kann, weil 0,1 periodisch ist. Somit können wir anhand dieser Funktion die Auswirkungen unserer Methoden auf die Mantisse testen.

$$\text{Parabel} : [0,1] \to \mathbb{R},\ x \mapsto 0{,}4 \cdot x \cdot (1-x) \tag{2.1.1}$$

Um die Effekte, welche erst bei großen Auswertungstiefen und damit kleinen Koeffizienten auftreten, schon bei geringen Tiefen zu erkennen, nutzen wir dieselbe Funktion mit kleinerem Vorfaktor

$$\text{kleineParabel} : [0,1] \to \mathbb{R},\ x \mapsto 0{,}000004 \cdot x \cdot (1-x) \tag{2.1.2}$$

Um zu erkennen, wie die Methoden sich verhalten, wenn alle Koeffizienten einer Tiefe nicht gleich sind, verwenden wir folgende Exponentialfunktion.

$$\text{Exponential} : [0,1] \to \mathbb{R},\ x \mapsto e^{x^{10}} + (1-e)x - 1 \tag{2.1.3}$$

2.2 Fehler

Für eine Approximation a und eine Abbildung f, bezeichnen wir $f - a$ als Differenz.
Mit dieser definieren wir nun den Fehler für eine Approximation a für eine Abbildung f wie folgt.

$$\mathrm{err}_f^a = \frac{||f - a||_2}{||f||_2} \tag{2.2.1}$$

Dabei ist die quadratische Norm für Funktionen wie folgt definiert:

$$||f||_2 = \sqrt{\sum_{k=0}^{N} f(\frac{k}{N})^2} \tag{2.2.2}$$

Hierbei ist N die Anzahl an Auswertungsstellen zwischen null und eins. Diese wird so gewählt, dass die Intervalllänge der Auswertungsstellen bei der Fehlerberechnung ein Vielfaches der Intervalllänge der Auswertungsstellen bei der Approximationen ist.
In dieser Arbeit ist die Auflösung der Auswertungsstellen für die Fehlerberechnung viermal so groß, wie die Auflösung der Auswertungsstellen für die Approximation. Wenn eine unterschiedliche Anzahl an Auswertungsstellen für die Approximation miteinander verglichen werden, wird die größte Anzahl als Referenz für die Auflösung bei der Fehlerberechnung genommen.

3 Ergebnisse

3.1 Ausgangslage

Als Ausgangslage, gegen welche wir unsere neue Methoden testen, verwenden wir, aufgrund ihrer großen Verbreitung, Doubleprecision Floats (64bit-Floats nach IEEE754[1]). Dennoch können die nachfolgenden Methoden auch äquivalent für zum Beispiel 32bit-Floats oder 128bit-Floats definiert werden.
Durch die Verwendung von Doubleprecision Floats benötigt das Speichern eines Koeffizienten 64 Bits. Da wir die Speicherung der Randwerte nicht betrachten, benötigt man $2^N - 1$ Koeffizienten für eine gleichmäßige Auswertung mit hierarchischen Basen bis zur Tiefe N. Daraus folgt für eine Auswertungstiefe N, dass man $64 * (2^N - 1)$ Bits benötigt.
Wir betrachten bei den nachfolgenden Methoden zunächst die Verbesserung des Speicherbedarfs von Mantisse und Exponenten getrennt. Diese Möglichkeit ergibt sich, weil der Exponenten exakt gespeichert werden muss, da sonst der relative Fehler größer wäre, als wenn wir die Mantisse nicht speichern.

3.2 Reduktion der Mantisse

Um nun die Anzahl der benötigten Bits zu reduzieren, reduzieren wir systematisch die Genauigkeit der einzelnen Koeffizienten. Dabei reduzieren wir die Genauigkeit der Koeffizienten, welche einen geringen Einfluss auf die Gesamtgenauigkeit der Approximation haben. Da nach Kapitel 1.1.2 die Koeffizienten der hierarchischen Basen bei der Approximation einer zweimal stetig differenzierbaren Funktion irgendwann immer um ein Viertel mit jeder zunehmenden Tiefe abfallen, benötigt man für jede nächst tiefere Auswertung nur noch ein Viertel der relativen Genauigkeit. Dies entspricht für Zahlenformate mit Exponenten und Mantisse einer Reduktion der Mantissenlänge um zwei Bits. Diese Reduktion der Mantissenlänge kann man für jede weitere Tiefe fortsetzten bis man bei einer Mantissenlänge von null oder eins ankommt. Wenn die Mantissenlänge auf der untersten Ebene Eins ist, dann sind alle Mantissenlängen ungerade und wir nennen dann eine entsprechende Methode ungerade. Äquivalent dazu nennen wir Methoden mit gerader Mantissenlänge gerade.
Somit lassen sich die Anzahl an benötigten Mantissenbits S für die Stützwerte, welche nicht am Rand liegen, für eine Auswertungstiefe N folgend beschreiben:

Für eine gerade Methode:

$$S_{\text{even}}(N) = \sum_{k=1}^{N} 2(k-1)2^{N-k} \tag{3.2.1}$$

$$= 2^{1+N} - 2N - 2 \tag{3.2.2}$$

Für eine ungerade Methode:

$$S_{\text{uneven}}(N) = \sum_{k=1}^{N} (2(k-1)+1)2^{N-k} \tag{3.2.3}$$

$$= 3 * 2^N - 2N - 3 \tag{3.2.4}$$

Dies ist deutlich weniger als die Verwendung von 52 Mantissenbits pro Koeffizient der 64bit-Floats nach IEEE754, welche insgesamt $52 * (2^N - 1)$ Mantissenbits benötigen.
In den folgenden zwei Kapitel gehen wir auf drei Methoden zur Reduktion der Mantissenlänge ein.

3.2.1 Runden

Für die Methode *Runden* besteht eine Repräsentation des Koeffizienten aus 3 Bitfolgen S, E und M. Hierbei ist S ein Bit und E 11 Bit lang. Die Mantisse M hingegen benötigt für den geraden Fall $2l$ Bits und für den ungerade $2l + 1$ Bits, wobei l das Level ist.
Der Zahlenwert dieser Repräsentation ist dann $s * 2^e * m$ für gerade Methoden mit:

$$s = (-1)^S \tag{3.2.5}$$

$$e = E - 1023 \tag{3.2.6}$$

$$m = 1 + \frac{M}{2^{2l}} \qquad \text{mit } 0 \leq M < 2^{2l} \tag{3.2.7}$$

$$\text{oder } m = 0 \qquad \text{wenn } M = 0 \text{ und } E = 0 \tag{3.2.8}$$

Hingegen für ungerade Methoden definieren wir m

$$m = 1 + \frac{M}{2^{2l+1}} \qquad \text{mit } 0 \leq M < 2^{2l+1} \tag{3.2.9}$$

$$\text{oder } m = 0 \qquad \text{wenn } M = 0 \text{ und } E = 0 \tag{3.2.10}$$

Für die Methode *Runden* wählen wir s, e, m zur Beschreibung des Koeffizienten so, dass der resultierende Wert möglichst nah an dem zuvor exakt bestimmten Koeffizientenwert liegt. Dies entspricht einer Rundung der Mantisse auf $2l$ (oder $2l + 1$) Nachkommastellen.

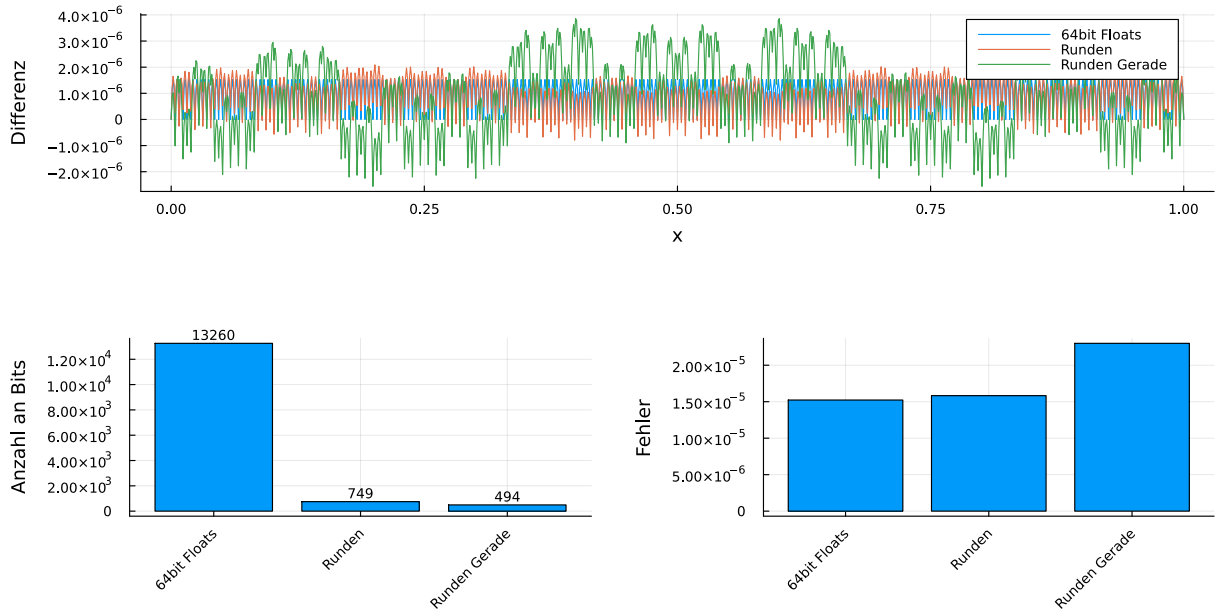


Abbildung 3.1: *oben:* Differenz, *links:* Mantissenbits, *rechts:* Fehler für die Abbildung *Parabel* (2.1.1) und einer Auswertungstiefe von 8

Im oberen Graphen der Abbildung 3.1 ist die Differenz zwischen der Funktion und ihrer Approximation dargestellt. In blau ist die Differenz für eine exakte Speicherung der Koeffizienten dargestellt. Dabei erkennt man die zu erwartenden skalierten Parabeln mit einer Höhe von $0.1 * \frac{1}{4}^8$, da die zu approximierende Funktion die Normalparabel ist und bei dieser sich der Fehler mit jeder Tiefe um den Faktor $\frac{1}{4}$ verkleinert.
Bei den Methoden, welche die Bits reduzieren, kann man diese Symmetrie nicht erkennen. Hier ist lediglich eine Symmetrie um den mittleren Wert 0.5 zu sehen. Dies ist zu erklären, da der Fehler eines Koeffizienten einen größeren Einfluss auf den Gesamtfehler einer Stelle hat, wenn die Auswertungsstelle näher am Hochpunkt der entsprechenden Basis ist. Somit erkennen wir die größten Fehler in der Mitte. Zusätzlich kann man erkennen, dass das Runden auf eine gerade Mantissenlänge immer dann eine positive Differenz hat, wenn das Runden auf eine ungerade Mantissenlänge eher negativ ist. Dies ist durch den periodischen Vorfaktor der Funktion zu erklären. Somit runden wir meist bei der einen Methode auf, wenn wir bei der anderen abrunden.
Bezüglich der genutzten Bits kann man eine deutliche Reduktion an Bits erkennen, wie durch die Gleichungen 3.2.1 und 3.2.3 vorhergesagt.

$$S_{\text{even}}(8) = 2^{1+8} - 2 * 8 - 2 = 494 \tag{3.2.11}$$

$$S_{\text{uneven}}(8) = 3 * 2^8 - 2 * 8 - 3 = 749 \tag{3.2.12}$$

Für die Methode des Runden auf eine ungerade Mantissenlänge sehen wir erwartungsgemäß trotz der deutlichen Reduzierung in der Anzahl an benötigten Bit, nur eine leichte Verschlechterung der Approximation gegenüber den 64bit-Floats. Beim Runden auf eine gerade Mantissenlänge ist dieser Fehler größer. Es wurden aber weniger Bits dafür benötigt. Somit kann die schlechtere Genauigkeit eventuell durch größere Auswertungstiefe kompensiert werden. Deshalb betrachten wir nun Abbil-

dung 3.2.

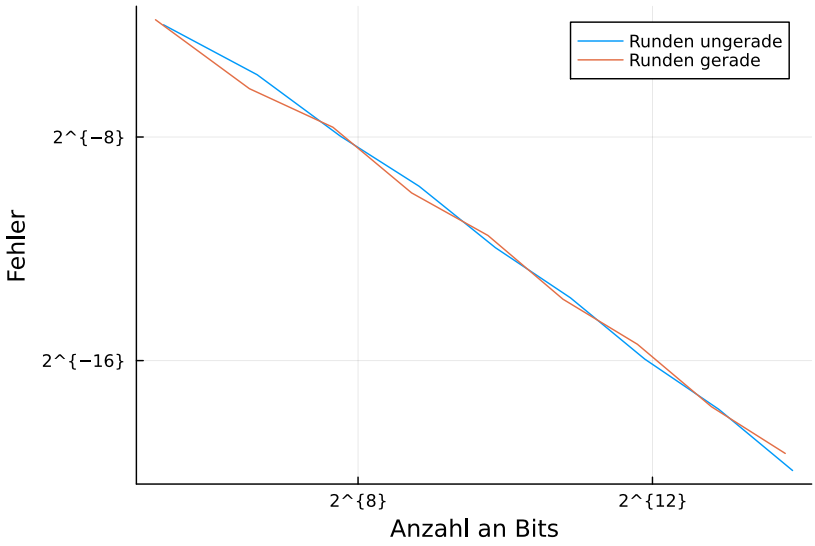


Abbildung 3.2: Vergleich der Methode *Runden* mit gerader und ungerader Mantissenlänge

In der Abbildung ist der Fehler gegen die Anzahl an benötigten Bits verschiedener Auswertungstiefen dargestellt. Man kann erkennen, dass die zwei verschiedene Methoden mit gleicher Anzahl an Bits gleich gut approximieren.
Da aber die Methode *Runden ungerade* bei gleicher Auswertungstiefe bereits mehr Bits hat, werden in allen weiteren Methoden eine ungerade Anzahl an Mantissenbits verwendet.

3.2.2 Abschneiden

Nun betrachten wir eine Methode, welche im Gegensatz zu der vorherigen keine von dem Koeffizientenwert abhängige Operation zu Reduzierung benötigt.
Hierbei ist die Darstellung des Werts der Koeffizienten wie in Kapitel 3.2.1 definiert. Wir wählen dabei s und e, wie bei 64bit-Floats nach IEEE754 und m so, dass es ein Präfix der Mantissendarstellung des korrespondierenden 64bit-Floats ist. Dies entspricht einem Abrunden der Mantisse auf $2l$ (oder $2l + 1$) Nachkommastellen.
Technisch entspricht dies einem Abschneiden und kann durch die Anwendung einer Bitmaske auf den 64bit-Float umgesetzt werden. Somit müssen keine Bits des 64bit-Floats verändert werden. Wir erwarten für das Abschneiden auf eine ungerade Mantissenlänge, eine Genauigkeit der Approximation zwischen dem Runden auf ungerader und gerader Mantissenlänge, da wir gegenüber der ungeraden Methode ein halbes Bit verlieren und gegenüber der geraden Methode ein halbes gewinnen.

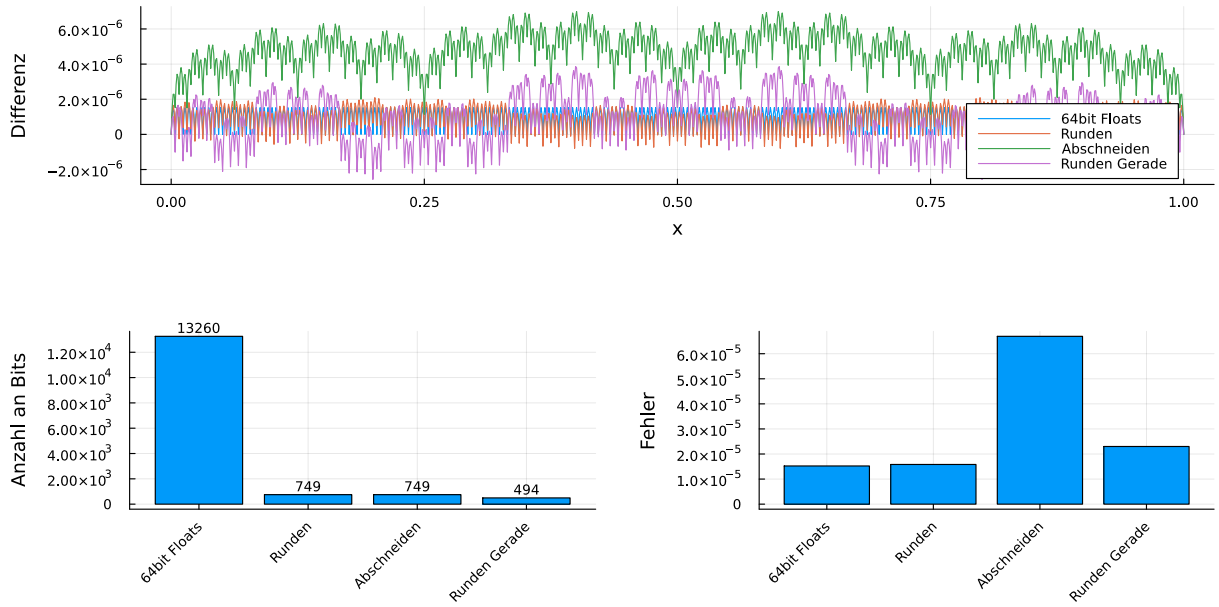


Abbildung 3.3: *oben:* Differenz, *links:* Mantissenbits, *rechts:* Fehler für die Abbildung *Parabel* und einer Auswertungstiefe von 8

Wenn wir uns nun die Abbildung 3.3 anschauen, erkennen wir, dass die Methode *Abschneiden* schlechter als erwartet ist. Die Approximation mit der Methode *Abschneiden* ist schlechter als die Approximation der Methoden *Runden gerade* und *Runden ungerade*. Dieses Verhalten kann man durch die Betrachtung des oberen Graphs in Abbildung 3.3 erklären. Man erkennt anhand der Differenz, dass die Funktion selbst immer größer gleich der Approximation ist. Dies liegt daran, dass wir die Koeffizienten durch das Abschneiden immer abrunden und sich so die Fehler aller Koeffizienten zwingend aufaddieren. Bei der Methode *Runden* wird dagegen manchmal aufgerundet und manchmal abgerundet, da die Mantisse an den zu rundenden Stellen sowohl Einsen als auch Nullen hat. Somit können sich zum Teil die Fehler der Koeffizienten gegenseitig aufheben.
Aufgrund dieses Nachteils muss man für die Verwendung *Abschneiden* eine Notwendigkeit für das Fehlen einer Operation auf den Bits haben. Da die eventuell gesparte Rechenzeit, dann für größere Auswertungstiefen verwendet wird. Somit erhält man mit dieser Methode eine schlechtere Reduktion der benötigten Bits für eine gegebene Genauigkeit.

3.2.3 Festkommazahl

Für die Methode *Festkommazahl* richtet sich die Länge der Mantisse M nach der Größe des Exponenten, um eine konstante Anzahl an Nachkommastellen in der Festkommadarstellung zu beschreiben. Die Repräsentation des Koeffizienten besteht, wie bei den vorherigen Methoden, aus drei Ganzzahlen S, E und M. Hierbei ist S ein Bit und E elf Bit lang. Die Länge der Mantisse M richtet sich nach dem größten Exponenten der untersten Auswertungsebene $e_{\max}$. Dabei wird die Länge so gewählt, dass alle Koeffizienten die gleiche absolute Genauigkeit haben, wie der Koeffizient mit $e_{\max}$, wenn dieser 1 Mantissenbit hätte. Somit ergibt sich eine Länge von M für ein Level l als das

Maximum von 0 und $(e - e_{\max} + 1)$

$$s = (-1)^S \tag{3.2.13}$$

$$e = E - 1023 \tag{3.2.14}$$

$$m = 1 + \frac{M}{2^{e-e_{\max}+1}} \qquad \text{mit } 0 \leq M < 2^{e-e_{\max}+1} \tag{3.2.15}$$

$$\text{oder } m = 0 \qquad \text{wenn } M = 0 \text{ und } E = 0 \tag{3.2.16}$$

Wenn man nun die Differenz in der Abbildung 3.4 betrachtet, kann man keinen Unterschied zwischen der Approximation mit der Methode *Runden* und der Methode *Festkommazahl* finden. Dies liegt daran, dass die Exponenten der Koeffizienten für die Abbildung *Parabel* (siehe 2.1.1) mit zunehmender Auswertungstiefe jeweils um zwei abnehmen, da die Koeffizienten selbst um den Faktor ein Viertel kleiner werden. Somit entspricht dann die jeweilige Mantissenlänge auf die gerundet wird, exakt derer der Methode *Runden*. Auch in den anderen zwei Graphen ist kein Unterschied zwischen diesen Methoden festzustellen.

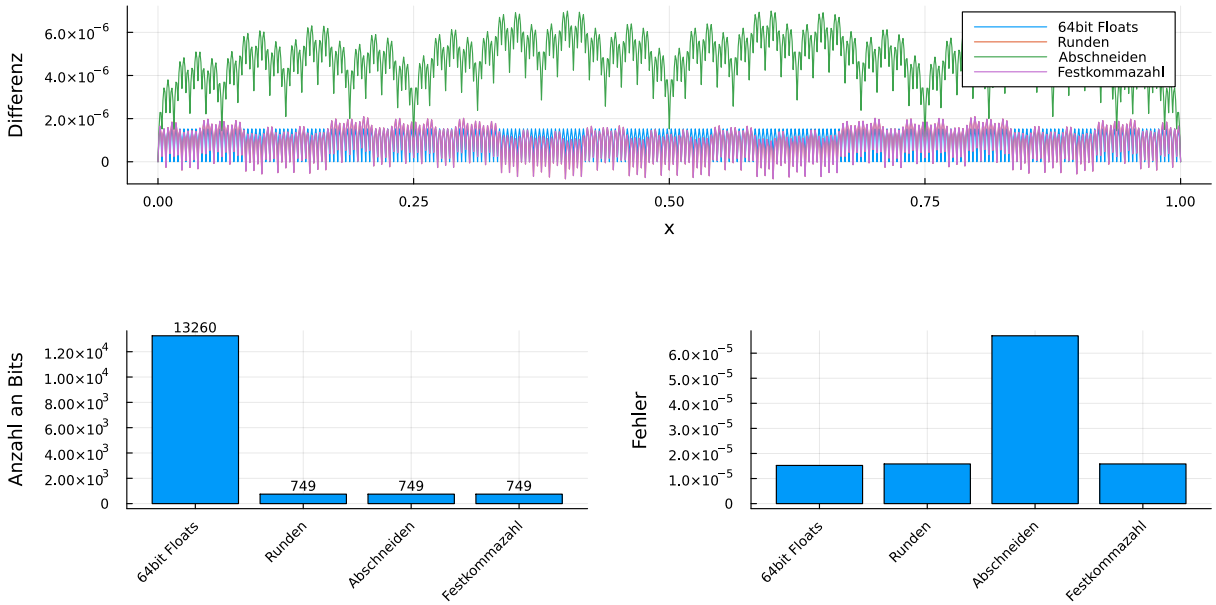


Abbildung 3.4: *oben:* Differenz, *links:* Mantissenbits, *rechts:* Fehler für die Abbildung *Parabel* (siehe Gleichung 2.1.1) und einer Auswertungstiefe von 8

Nun betrachten wir die Abbildung 3.5, welche die Approximationen der Funktion *Exponential* (siehe 2.1.3) mit den verschiedenen Methoden zeigt. Man kann erkennen, dass Methoden *64bit-Floats*, *Runden* und *Abschneiden* fast keinen Fehler im Intervall von 0 bis 0,75 im Vergleich zum Intervall 0,75 bis 1 machen. Trotzdem verwendet diese Approximation $\frac{3}{4}$ der Bits für die Approximation des ersten Intervalls, weil die Auswertungstiefe uniform ist. Wenn wir uns nun die Methode *Festkommazahl* anschauen, können wir erkennen, dass Approximation im letzten Viertel ähnlich gut wie die Methode *Runden* ist. Es werden dann im Intervall von 0 bis 0.75 jedoch nur so viele Bits verwendet, um eine ähnliche Genauigkeit wie im letzte Viertel zu erreichen. Dies führt zu einem

größeren Fehler, aber auch zu einer geringen Anzahl an Mantissenbits. Folglich werden mit dieser Methode die Mantissenbits so verteilt, um eine möglichst gleichmäßige Reduktion des Fehlers zu erreichen.

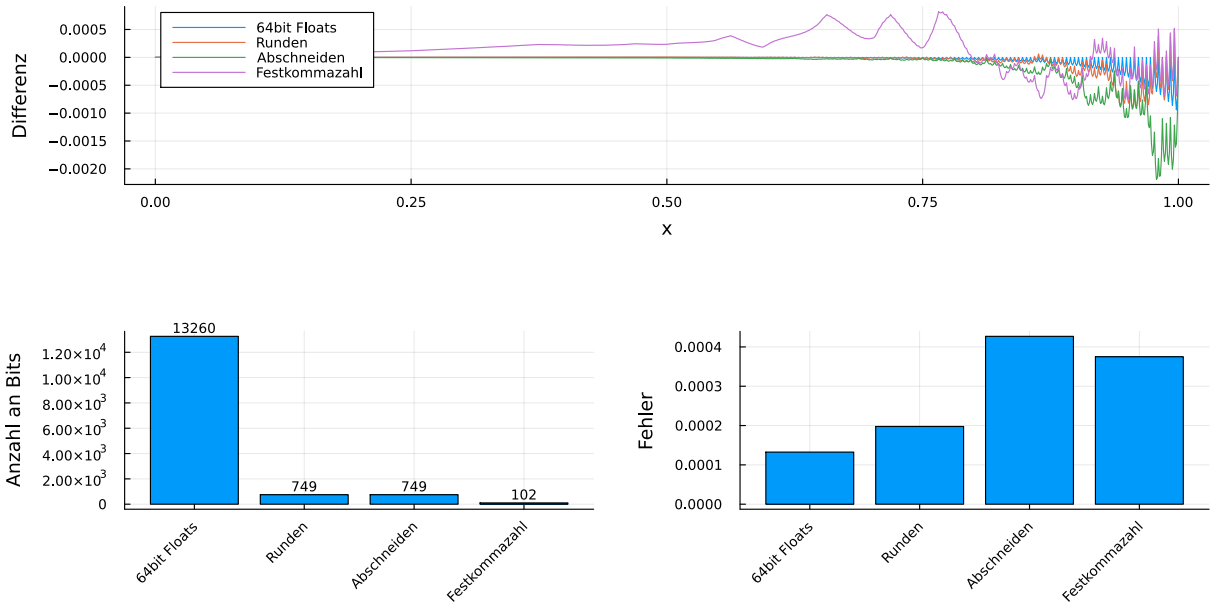


Abbildung 3.5: *oben:* Differenz, *links:* Mantissenbits, *rechts:* Fehler für die Abbildung *Exponential* (siehe Gleichung 2.1.3) und einer Auswertungstiefe von 8

3.2.4 Vergleich

Nun vergleichen wir in der Abbildung 3.6 die verschiedenen Methoden zur Reduktion der Mantissenbits indem wir die Gesamtanzahl der Bits gegen den Fehler der Approximation auftragen.

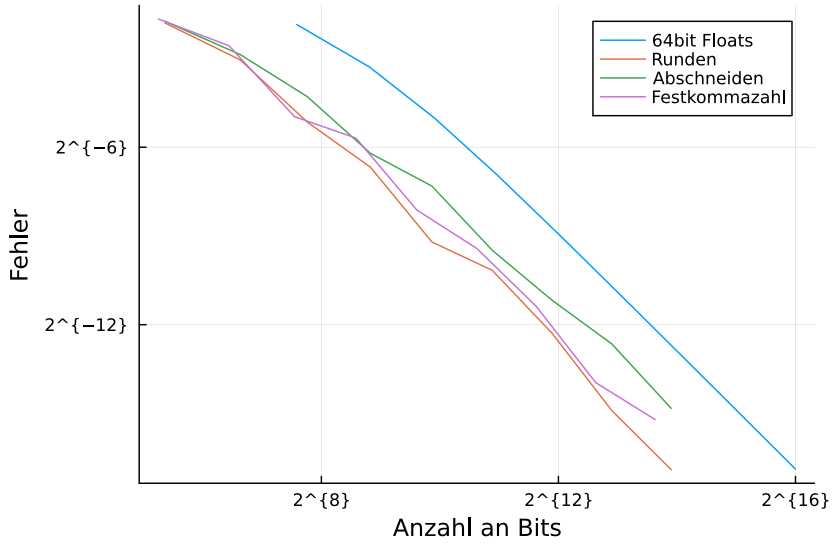


Abbildung 3.6: Vergleich der Methoden zur Reduktion der Mantissenbits an der Approximation für die Abbildung *Exponential* (siehe Gleichung 2.1.3)

Man kann erkennen, dass für eine gewünschte Genauigkeit alle beschriebenen Methoden zu einer deutlichen Reduktion in der Anzahl benötigter Bits führen. Dabei sieht man auch gut, den in Kapitel 3.2.2 beschriebenen Nachteil der Methode *Abschneiden* gegenüber den Methoden *Runden* und *Festkommazahl*. Der Gesamtfehler der Approximation mit der Methode *Festkommazahl* ist nach dem Graphen größer als bei der Methode *Runden*, aber kleiner als bei der Methode *Abschneiden* und der Fehler ist dabei gleichmäßiger über das Intervall verteilt.

3.3 Reduktion des Exponenten

Nun widmen wir uns Methoden zur Reduktion des Exponenten. Hierbei soll der Exponent des 64bit-Floats nach IEEE754 anders dargestellt werden, so dass es zu einer Reduktion an Bits in den häufig auftretenden Größenordnung kommt. Wir reduzieren dabei nicht die Genauigkeit des gespeicherten Exponenten, da wir sonst einen größeren Fehler machen, als durch das Weglassen explizit gespeicherten Mantissenbits und dies unsere unterste Stufe der Genauigkeit ist.

3.3.1 Posit mit fester Mantissenlänge

Für die Darstellung des Exponenten nehmen wir die Darstellung dieser aus der Definition der Posits (siehe Kapitel 1.2). E ist hierbei die positive Ganzzahl mit einer festen Länge *es* und k die durch die Regimbits R kodierte Zahl. Die Konstante *useed* ergibt sich direkt aus der Wahl von *es* durch useed $= 2^{2^{es}}$. Somit erhalten wir mit dieser Darstellung für den Exponenten und die Darstellung der Mantisse wie in IEEE754 mit 64 Bits folgende Werte für Vorzeichen s, Exponent e und Mantisse m:

$$s = (-1)^S \tag{3.3.1}$$

$$e = \text{useed}^k * 2^E \tag{3.3.2}$$

$$m = 1 + \frac{M}{2^{52}} \qquad \text{mit } 0 < M < 2^{52} \tag{3.3.3}$$

$$\text{oder } m = 0 \qquad \text{wenn } M = 0,\ E = 0,\ R = 0 \text{ und } S = 0 \tag{3.3.4}$$

Dabei lässt sich m durch die Definition der Mantisse aus einer der Methoden zur Reduktion der Mantissenbits ersetzen.
Die Wahl von *es* hängt hier von der zu approximierenden Funktion ab. Je näher die Exponenten der Koeffizienten beieinander liegen, desto eher können sie so skaliert werden, dass die Exponenten nahe der Null sind und *es* kann klein gewählt werden. Je weiter die Exponenten der Koeffizienten auseinanderliegenden, desto größer sollte *es* gewählt werden und desto mehr nähert sich die Exponentendarstellung von Posits in den Eigenschaften der Darstellung in IEEE754 an.
Wir wählen für die nachfolgende Ergebnisse in dieser Arbeit *es* als 2, um den Vorteil von Posits zu demonstrieren. Sie ist dabei aber auch nicht zu angepasst an die gewählten Beispielfunktionen.

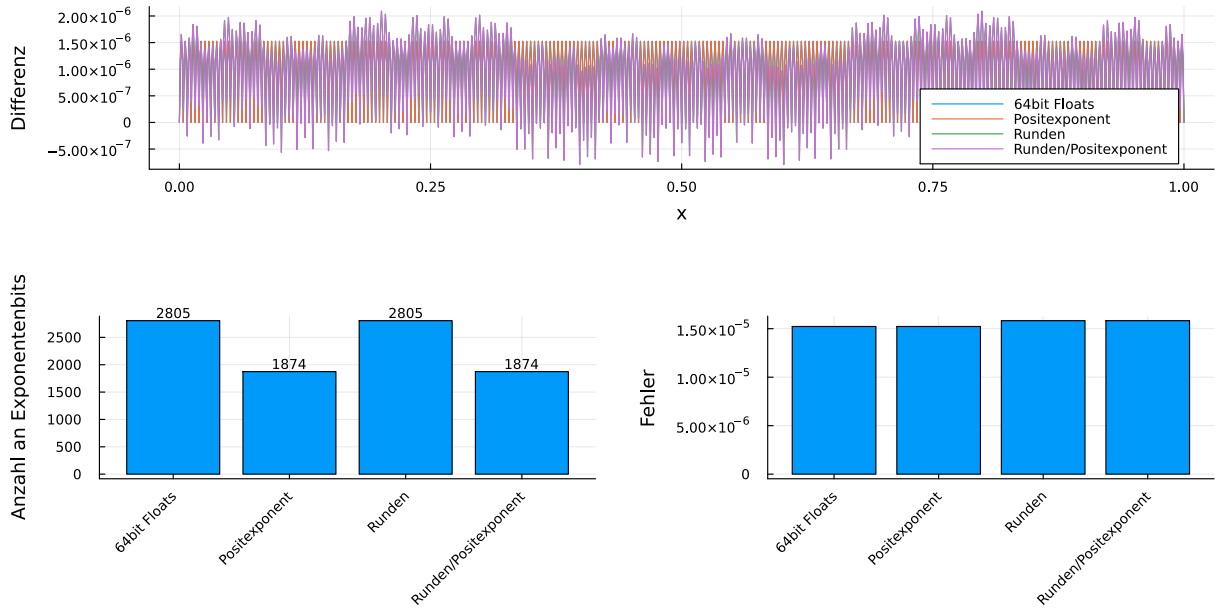


Abbildung 3.7: *oben:* Differenz, *links:* Exponentenbits, *rechts:* Fehler für die Abbildung *Parabel* (siehe Gleichung 2.1.1) und einer Auswertungstiefe von 8

Wenn wir nun die Abbildung 3.7 betrachten, können wir feststellen, dass sich die Approximation durch geänderte Darstellung des Exponenten nicht verändert. Dies ist vorhersehbar, da sich der Wert des Exponenten durch die Darstellung nicht ändert.
Um möglichst viele Bits einzusparen, müssen die Exponenten der Koeffizienten möglichst nah an 1 liegen. Dies kann durch Skalierung, wie auch in der Veröffentlichung über Posits [3] beschrieben, erreicht werden, wenn man die Verteilung der Größenordnungen der Koeffizient kennt.

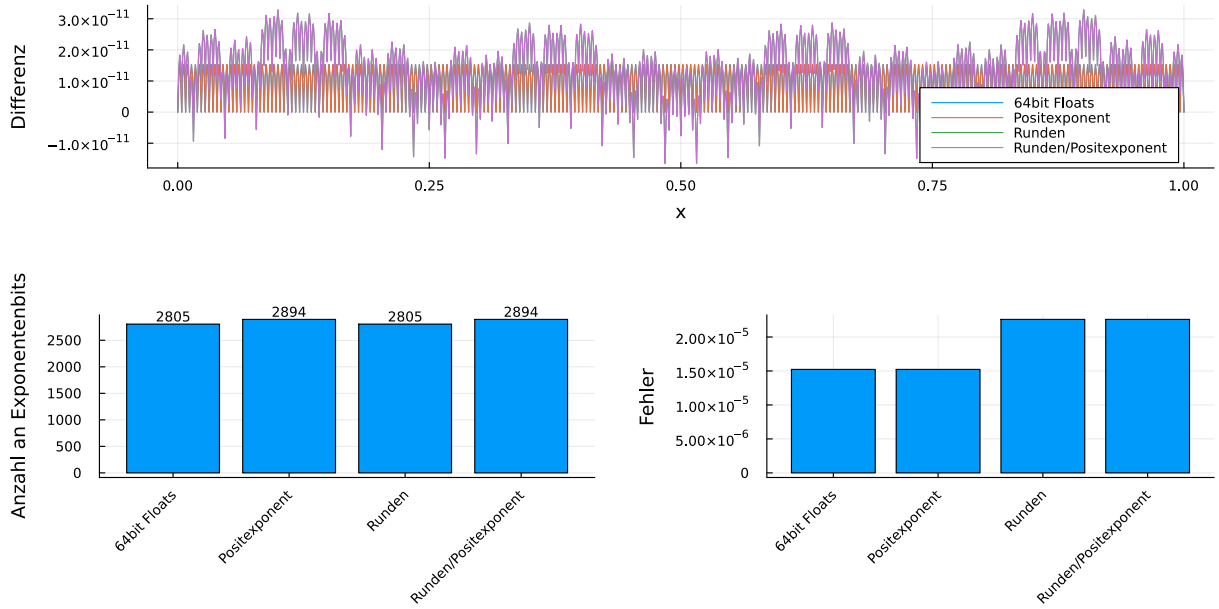


Abbildung 3.8: *oben:* Differenz, *links:* Exponentenbits, *rechts:* Fehler für die Abbildung *kleineParabel* (siehe Gleichung 2.1.2) und einer Auswertungstiefe von 8

Würden wir nun aber tiefer auswerten, werden die Exponenten immer kleiner (siehe Kapitel 1.1.2). Wir benötigen dann für das Speichern der Exponenten nahe an null viele Regimbits. Da die kleinen Exponenten unten in der Auswertung auftreten, haben wir von ihnen auch am meisten und es kann passieren, dass die neue Darstellung des Exponenten mehr Bits benötigt als das Äquivalent nach IEEE754.
Dies kann man in der Abbildung 3.8 erkennen, welche die Funktion *kleineParabel* (siehe Gleichung 2.1.2) approximiert. Es ist eine skalierte Parabel und somit äquivalent zu einem Abschnitt der Differenz einer Parabelapproxmation mit hierarchischen Basen, Deshalb sieht man diesen Effekt so schon bei geringeren Auswertungstiefen.
Um dem entgegen zu wirken kann man *es* vergrößern. Dann tritt das Problem erst bei noch größeren Auswertungstiefen auf. Da es aber irgendwann immer auftreten wird und man sich so nur der Darstellung von IEEE754 annähert, ist dies keine gute Lösung, denn im Grenzfall hat man keinen Vorteil aus der neuen Darstellung. Deshalb wird im nächsten Kapitel eine bessere Lösung des Problems beschrieben.

3.3.2 Skalierter Posit mit fester Mantissenlänge

Wir benötigen für das Speichern des Exponenten bei großen Auswertungstiefen mit der Darstellung vorherigem Kapitel viele Regimbits und brauchen so eventuell sogar mehr Bits um den Exponenten darzustellen als IEEE754.

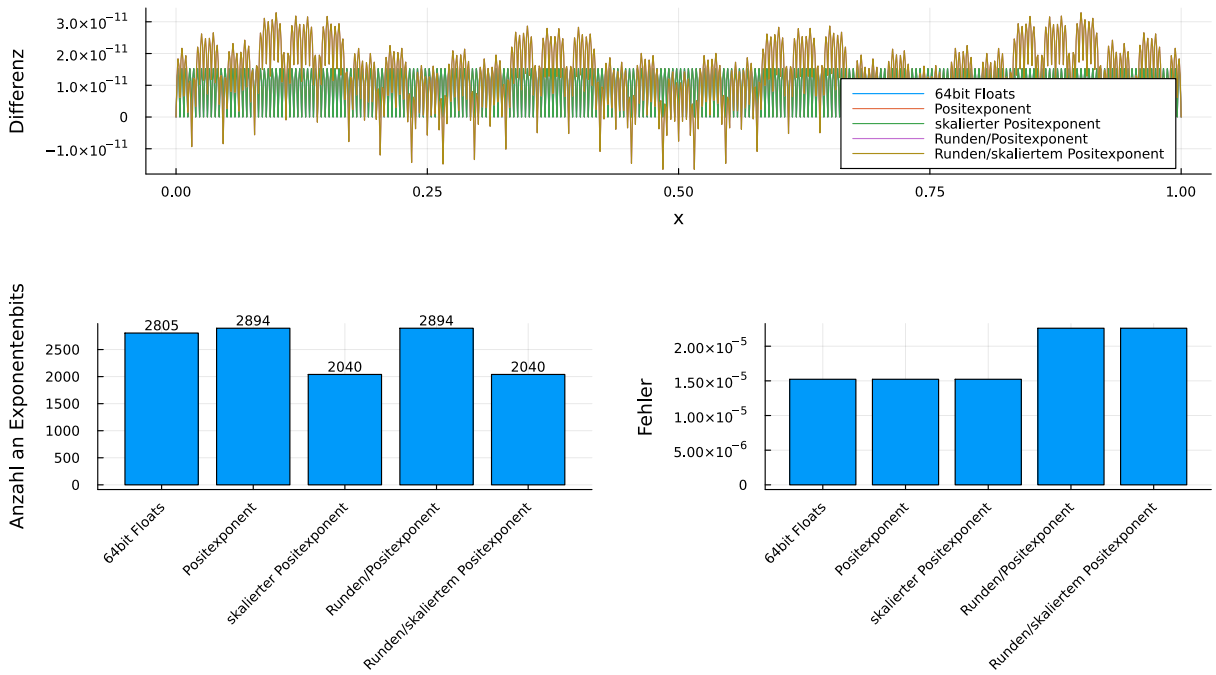


Abbildung 3.9: *oben:* Differenz, *links:* Exponentenbits, *rechts:* Fehler für die Abbildung *kleineParabel* (siehe Gleichung 2.1.2) und einer Auswertungstiefe von 8

Wir wissen um welchen Faktor die Koeffizienten irgendwann einmal kleiner werden, da die zu approximierende Funktionen zweimal stetig differenzierbar sind (siehe Kapitel 1.1.2). Somit

können wir diesem Kleinerwerden der Exponenten durch eine von der Auswertungstiefe abhängige Skalierung entgegenwirken. Also skalieren wir die Koeffizienten vor dem Speichern wie in Kapitel 3.3.1 jeweils, um den folgenden Faktor abhängig von ihrer Position:

$$4^{\text{Tiefe}} \tag{3.3.5}$$

Es führt dazu, dass für Funktionen, bei welchen die Koeffizienten direkt und gleichmäßig um ein Viertel abfallen, nun alle Koeffizienten einen Exponenten in der gleichen Größenordnung haben. Dies führt zu Vermeidung von Regimbits und einer Wiederherstellung des Vorteils der Exponenten in Positdarstellung, was man auch in der Abbildung 3.9 erkennt.

3.3.3 Vergleich

Wenn wir nun die verschiedenen Methoden zur Reduktion von Exponentenbits nach Abbildung 3.10 vergleichen, können wir erkennen, dass für eine geringe Genauigkeit und somit geringe Auswertungstiefe erwartungsgemäß die Darstellung des Exponenten wie bei Posits zu einer Einsparung an Bits bei gleicher Genauigkeit führt. Während die Speicherung des skaliertem Exponenten seinen Vorteil gegenüber der Darstellung nach IEEE754 beibehält, nimmt für die Exponentendarstellung ohne Skalierung der Bedarf an Bits zu.

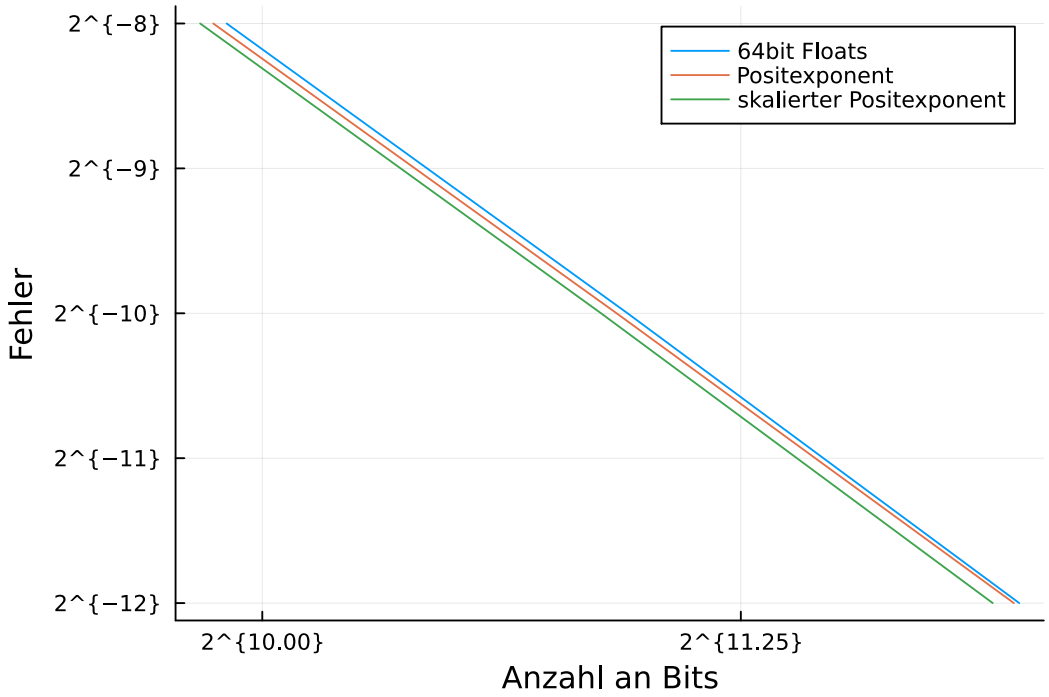


Abbildung 3.10: Vergleich der Methoden zur Reduktion der Exponentenbits bei geringer Auswertungstiefe für die Abbildung *kleineParabel* (siehe Gleichung 2.1.2)

In der Abbildung 3.11, in welchem die Methoden bei hoher Auswertungstiefe verglichen werden, erkennt man, dass die Darstellung als Positexponent ohne Skalierung schlechter als bei der Dar-

stellung nach IEEE754 ist. Die Darstellung wird auch zunehmend schlechter, da die Gerade eine geringere Steigung hat.

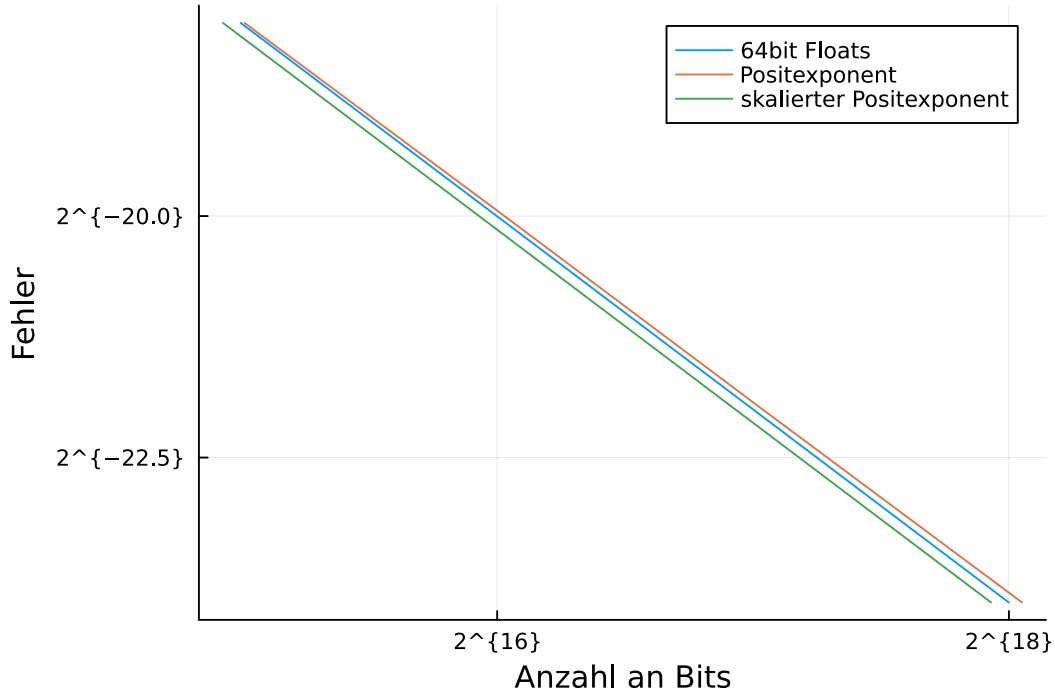


Abbildung 3.11: Vergleich der Methoden zur Reduktion der Exponentenbits bei großer Auswertungstiefe für die Abbildung *kleineParabel* (siehe Gleichung 2.1.2)

3.4 Reduktion der Anzahl an Knoten

In diesem Kapitel wollen wir nun die Adaptivität von hierarchischen Basen nutzen, um Funktionen, bei welchen die Koeffizient nicht gleichmäßige abfallen, wie zum Beispiel die Abbildung 2.1.3, besser zu approximieren.
Dazu werten wir die Funktion, um ein Level mehr aus. Aus dem untersten Auswertungslevel wählen wir den betragsmäßig größten Koeffizienten. Diesen nennen wir ϵ. Er gibt eine untere Schranke für die betragsmäßig maximale Differenz zwischen Approximation und Funktion. Falls danach die Koeffizienten mit $\frac{1}{4}$ abfallen, weil die Funktion hinreichend glatt ist, ist ϵ auch eine obere Schranke für diese Differenz. Somit gibt ϵ uns die Genauigkeit der Approximation und es kann auch so lange ausgewertet werden bis man eine gewünschte Genauigkeit, beziehungsweise ein gewünschtes ϵ erhält.
Nun nutzen wir ϵ und entfernen in der Baumdarstellung alle Kindknoten, wenn diese kleiner gleich ϵ sind und alle deren Kindknoten bereits entfernt wurden. Der resultierende Baum ist nun aber nicht mehr vollständig. Deshalb speichern wir seine Struktur explizit. Dazu fügen wir jedem dann noch zu speicherndem Koeffizient zwei Bits hinzu, welche angeben, ob der Koeffizient im Baum ein rechtes oder ein linkes Kind hat. Wenn man nun diesen Baum speichern möchte, kann man einfach die Koeffizienten in preorder Traversierung hintereinanderschreiben.

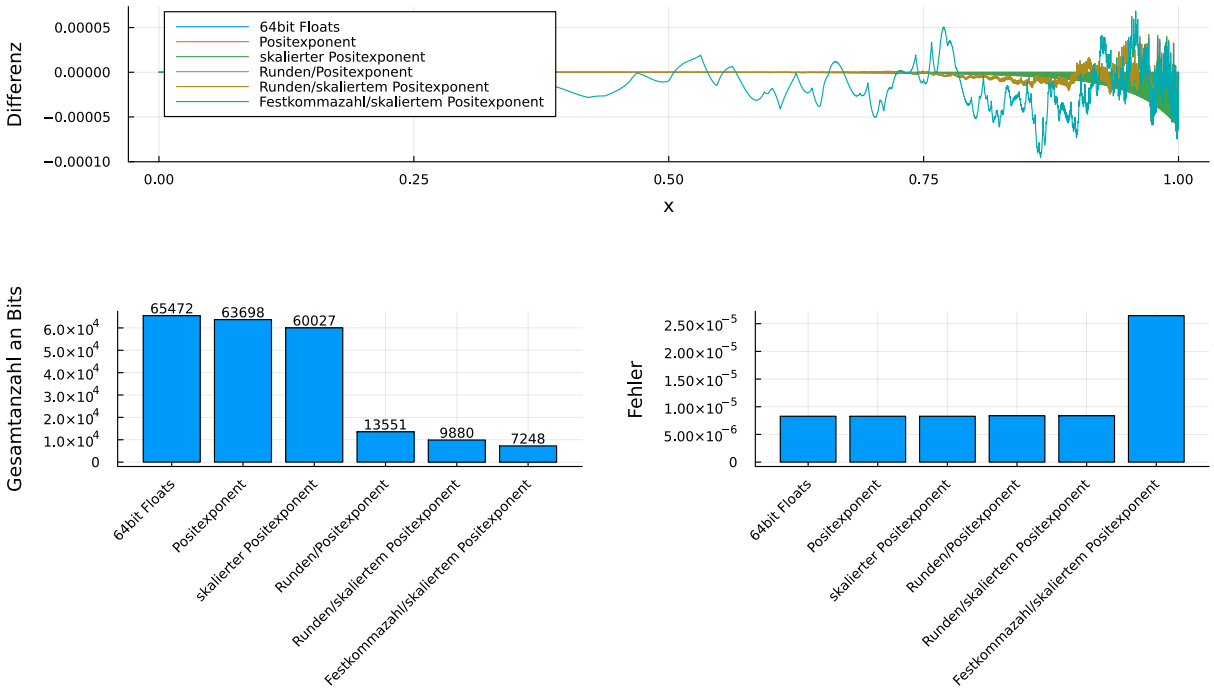


Abbildung 3.12: *oben:* Differenz, *links:* Bitanzahl, *rechts:* Fehler für die Abbildung *Exponential* (siehe Gleichungen 2.1.3) und einer maximalen Auswertungstiefe von 10

Betrachten wir nun die Abbildung 3.12. Wir können erkennen, dass alle Methoden außer *Festkommazahl* sehr viele Bits verwenden, um Intervalle mit sehr kleinen Fehlern noch weiter zu verbessern. Währenddessen ist der Betrag der Differenz im letzten Viertel dagegen sehr groß ist.
Die Methode *Festkommazahl* nutzt hingegen, mehr Mantissenbit in Abschnitten bei denen der Fehler größer ist. Da jedoch aber auch der Exponent gespeichert wird, ist die Ersparnis an Bits für den gestiegenen Fehler noch zu verbessern.
Dies sieht man dann in Abbildung 3.13. Dort hat die nicht uniforme Auswertungstiefe mit der Methode *Runden/skaliertem Positexponent* eine ähnliche Genauigkeit wie die Methode *Festkommazahl*. Dafür benötigt sie aber deutlich weniger Bits. Man sieht auch die zu erwartende Verbesserung bei 64bit-Floats nach IEEE754, wie in [2] bereits beschrieben.
Die Methode *Festkommazahl* profitiert von einer nicht uniformen Auswertungstiefe, besitzt aber einen deutlich größeren Fehler. In der Abbildung 3.14 kann man erkennen, dass so die Methode *Festkommazahl* für eine gegebene Genauigkeit am wenigsten Bits benötigt. Dies lässt sich dadurch erklären, dass die Koeffizienten, welche kaum benötigt werden, nun vermehrt nicht gespeichert werden, anstatt noch den Exponenten vollständig zu speichern.
Es führt dazu, dass alle Methoden bei der Approximation einer Funktion, bei welcher die Koeffizienten nicht gleichmäßig abfallen, sich durch eine nicht uniforme Auswertungstiefe verbessern.
Im Kontrast dazu ist eine nicht uniforme Auswertungstiefe schlechter, wenn die Funktion wie zum Beispiel die *Parabel* (siehe Gleichung 2.1.1) gleichmäßig abfallende Koeffizienten hat. Hier wird jeder Koeffizient gespeichert. Da aber bei dieser Methode jeder Koeffizient zwei zusätzliche Bits speichert, ist der Speicheraufwand größer ohne das die Genauigkeit zunimmt. Dies kann man in Abbildung 3.15 erkennen.

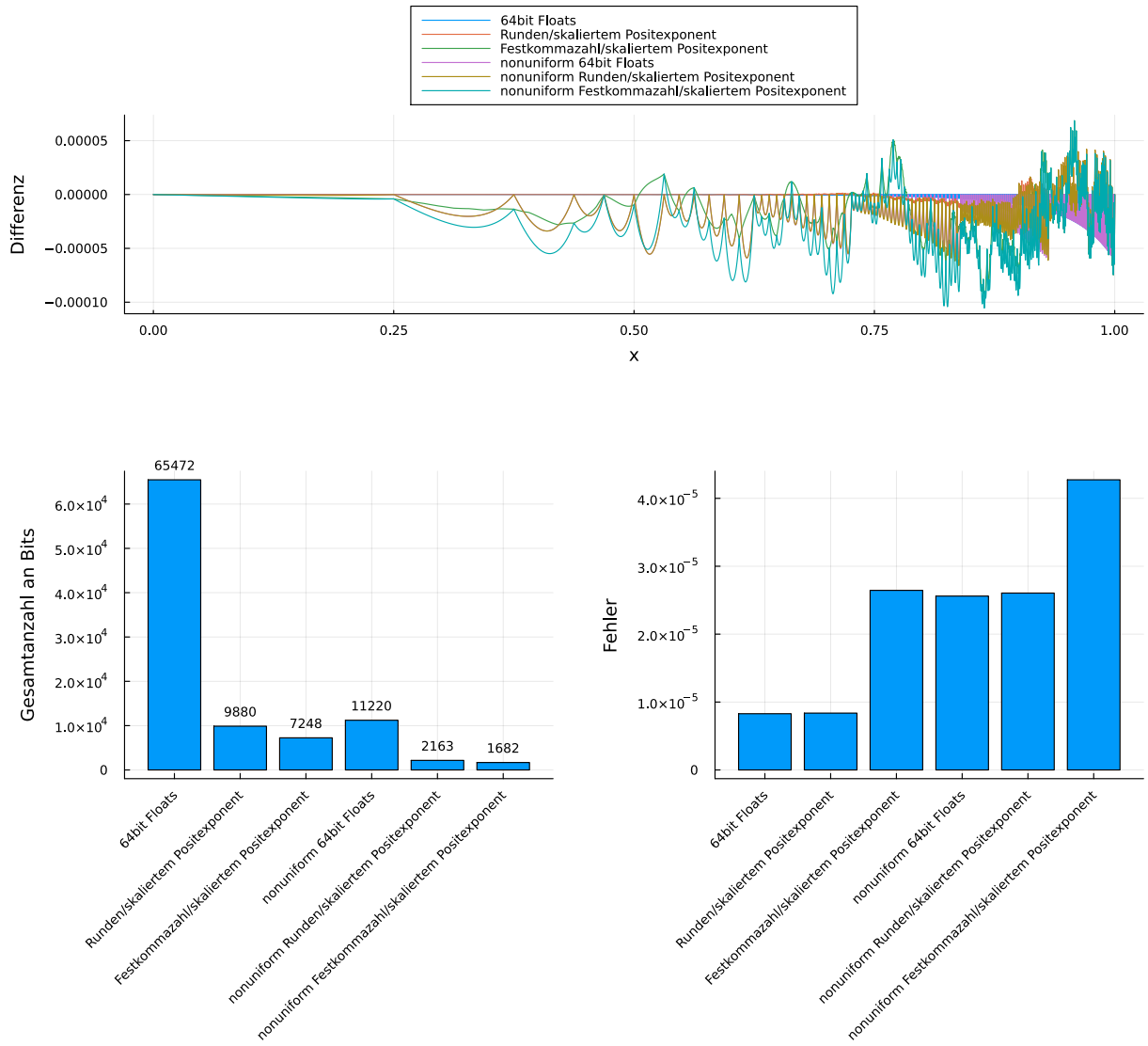


Abbildung 3.13: *oben:* Differenz, *links:* Bitanzahl, *rechts:* Fehler für die Abbildung *Exponential* (siehe Gleichungen 2.1.3) und einer maximalen Auswertungstiefe von 10

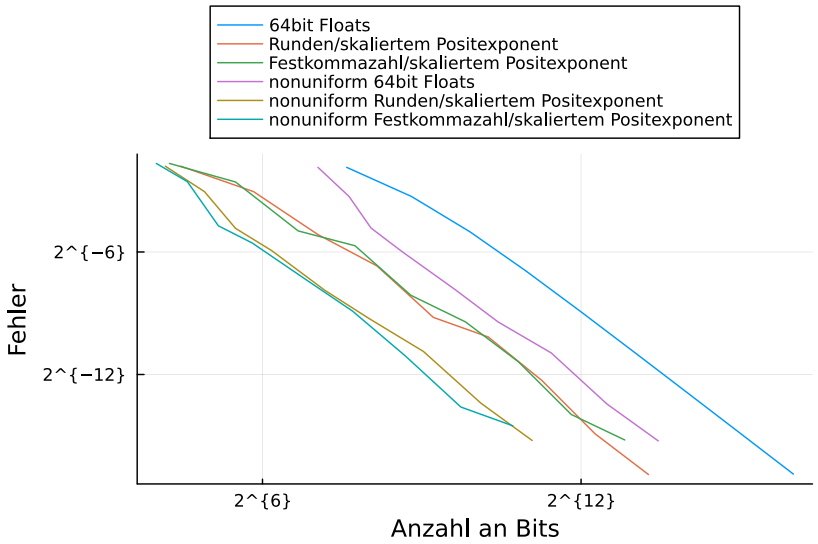


Abbildung 3.14: Vergleich der Methoden zur Reduktion der Gesamtanzahl an Bits für die Abbildung *Exponential*

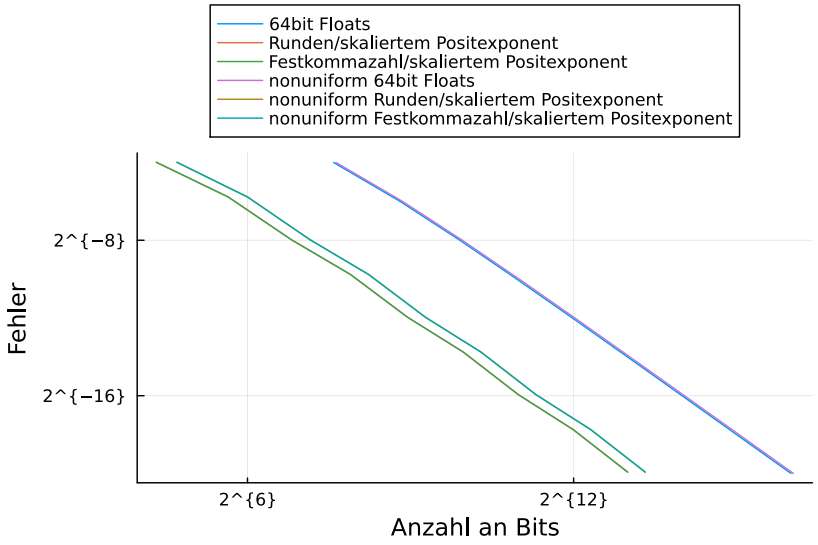


Abbildung 3.15: Vergleich der Methoden zur Reduktion der Gesamtanzahl an Bits für die Abbildung *Parabel*

3.5 Verwendung von Standardposits

Wir versuchen die Ideen der Methode *Runden* für die Mantisse und der Methode *Positexponent* oder *skalierter Positexponent* auf Posits nach dem Standard [3] zu übertragen. Nach dem Standard ist ein Posit durch die Wahl von *es* und der Länge definiert. Als *es* wählen wir wegen der Vergleichbarkeit 2. Um nun die Gesamtlänge der Koeffizienten als Posits sinnvoll zu wählen, bestimmen wir die maximale Länge zur Speicherung des Exponenten nach der Methode *Positexponent* oder *skalierter*

Positexponent. Diesen nennen wir MaxExp. Die Gesamtlänge eines Posits sei für ein gegebenes Level l wie folgt definiert:

$$1 + \text{MaxExp} + (2 * l + 1) \tag{3.5.1}$$

Der erste Summand ist das Vorzeichen. Der zweite Summand ist die obere Grenze des benötigten Speichers für den Exponenten. Wenn ein Koeffizient diese Bits nicht zum Speichern des Exponenten benötigt, werden diese, wie im Positstandard beschrieben, für Mantisse zu verwendet. Der dritte Summand beschriebt die benötigten Mantissenbits nach der Methode *Runden*, also einer Abnahme von 2 Bits pro zunehmender Tiefe.

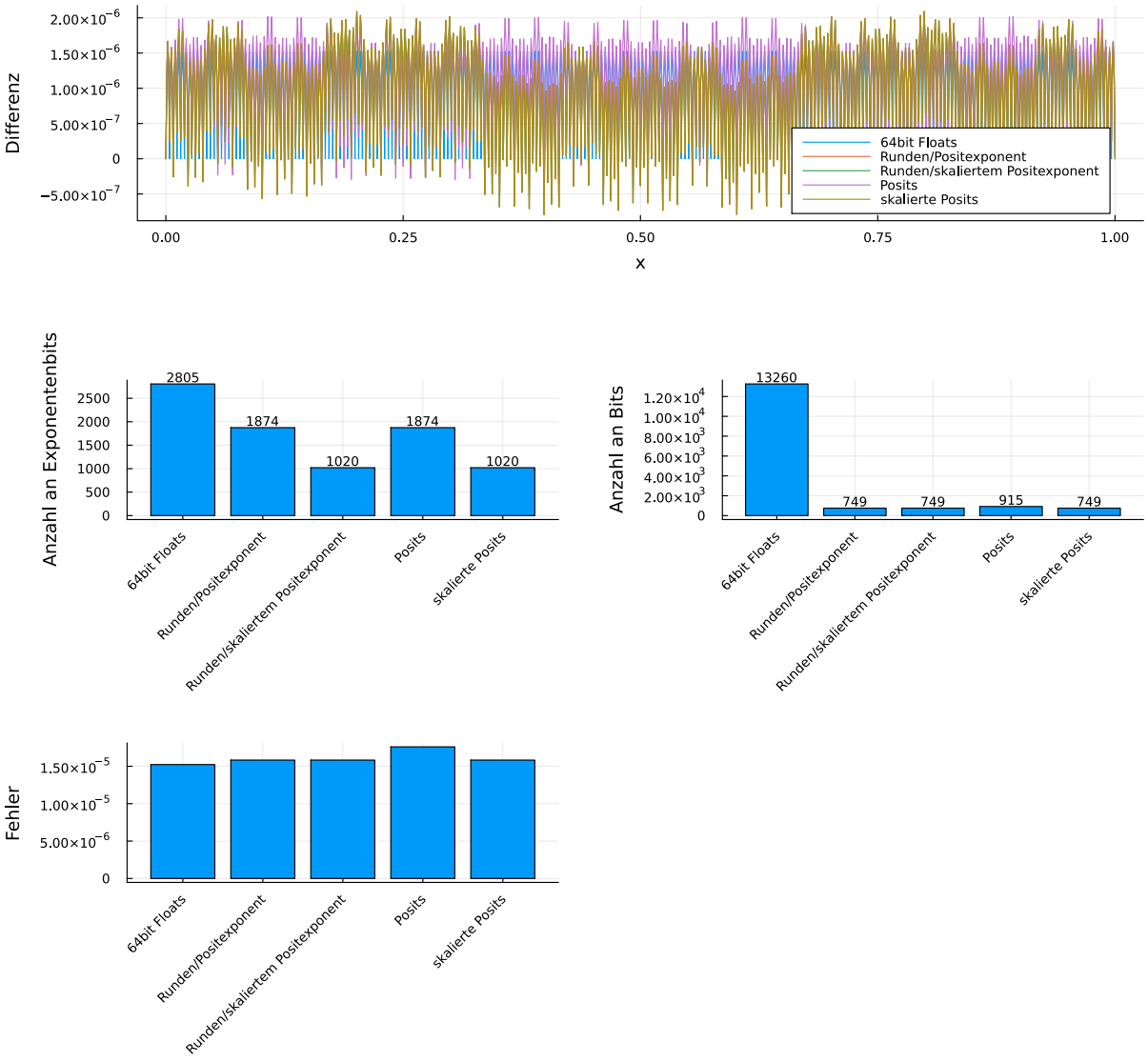


Abbildung 3.16: *oben:* Differenz, *mittelinks:* Exponentenbits, *mitterechts:* Mantissenbits, *unten:* Fehler für die Abbildung *Parabel* (siehe Gleichung 2.1.1) und einer Auswertungstiefe von 8

In der Abbildung 3.16 können wir erkennen, dass wie erwartet die Anzahl verwendeten Bits zum Speichern des Exponenten, sich gegenüber den äquivalenten Methoden mit den Positexponenten, nicht verändert. Die Anzahl der verwendeter Mantissenbits nimmt hingegen bei der Methode *Posits*

zu. Dies sind Bits, welche nicht unbedingt benötigt werden, aber durch die Abschätzung der Anzahl an Exponentenbits nun für die Mantisse verwendet werden. Bei der Methode *skalierte Posits* nimmt die Anzahl an Mantissenbits nicht zu, da die Exponenten der Koeffizienten durch die Skalierung alle die gleiche Größe benötigen.
Bei der Approximation der Funktion *Exponential* (siehe Gleichung 2.1.3) ist die Abschätzung der benötigten Exponentenbits für die meisten Koeffizienten sehr grob. Dies führt wie in Abbildung 3.17 zu einer deutlichen Zunahme an verwendeten Mantissenbits, was auch zu einer Reduzierung des Fehlers führt.

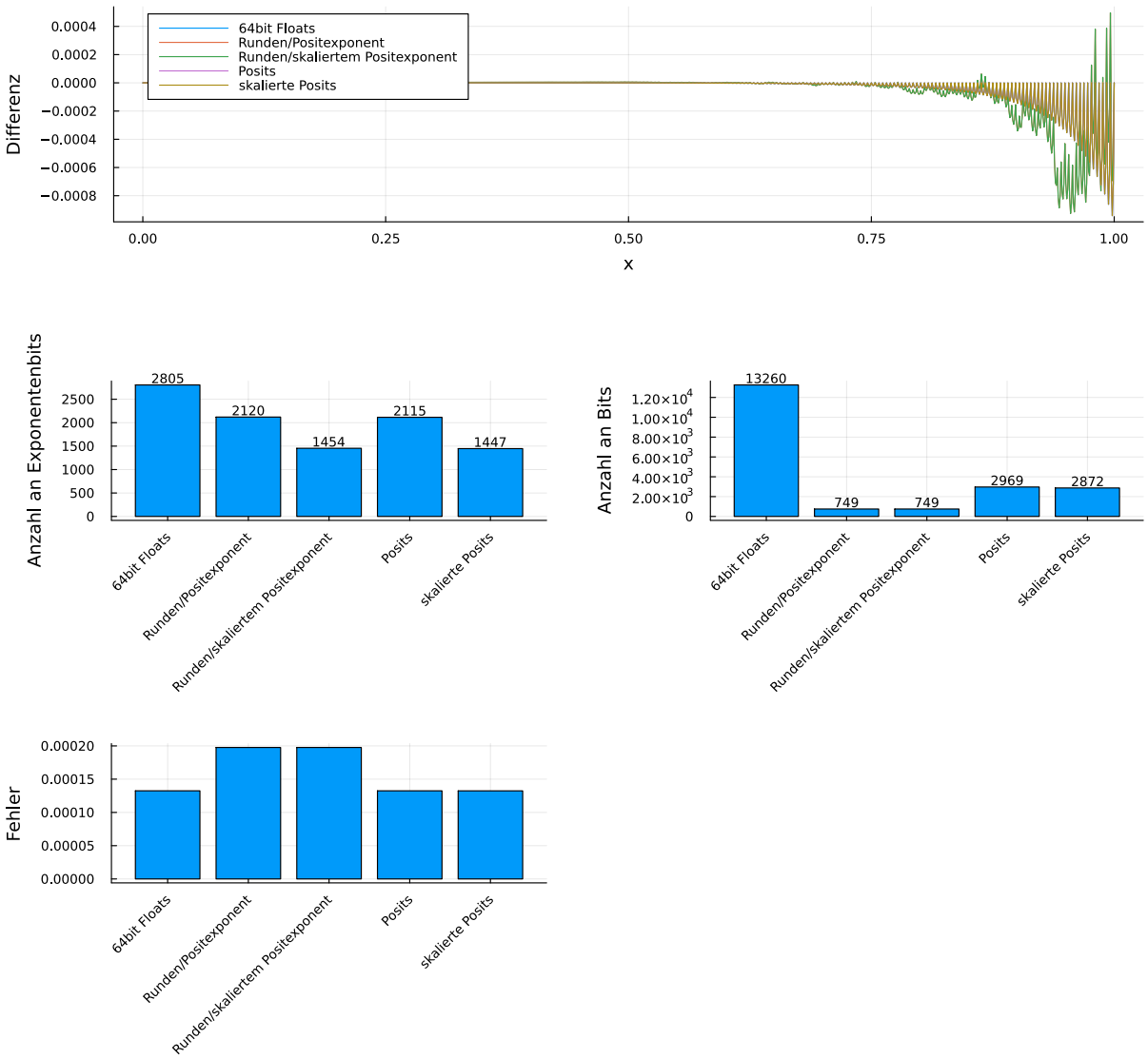


Abbildung 3.17: *oben:* Differenz, *mittelinks:* Exponentenbits, *mitterechts:* Mantissenbits, *unten:* Fehler für die Abbildung *Exponential* (siehe Gleichung 2.1.3) und einer Auswertungstiefe von 8

Wenn wir nun für diese Funktion den Fehler für eine gegebene Anzahl an Bits anschauen, sehen wir, dass alle beschriebenen Methoden eine deutliche Verbesserung gegenüber der Verwendung von IEEE754 erreichen. Die Verwendung von Posits nach Standard ist jedoch aufgrund der höheren

Anzahl an Mantissenbits, leicht schlechter als die Methode *Runden/Positexponent*. Dagegen sind die skalierten Posits nach Standard exakt gleich gut, wie die Methode *Runden/skaliertem Positexponent*, weil die Koeffizienten gleichmäßig abfallen.

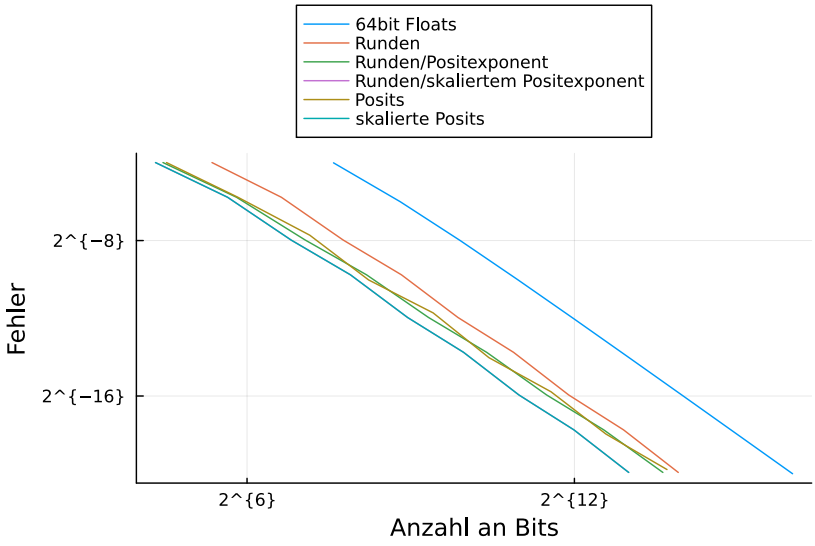


Abbildung 3.18: Vergleich der Methoden zur Reduktion der Gesamtanzahl an Bits für die Abbildung *Parabel* (siehe Gleichung 2.1.1). Hierbei ist der Graph für die Methoden *Runden/skaliertem Positexponent* und *skalierte Posits* identisch

3.6 Schlussfolgerung

Wenn die Koeffizienten bei der zu approximierenden Funktion nicht gleichmäßig abfallen, bietet es sich an, die Funktion adaptiv auszuwerten. Dadurch werden Koeffizienten mit relativ zum Gesamtfehler minimalem Einfluss nicht gespeichert.
Wenn man keine echte Posits benötigt, bietet sich die Methode *Runden/skaliertem Positexponent* an. Diese erzeugt im Gegensatz Posits nach Standard keinen Overhead durch Abschätzen der Exponentenbits.
Wenn man echte Posits benötigt, sollte man *skalierte Posits* verwenden. Da man dann auch bei tiefen Auswertungen weniger Speicher für den Exponenten benötigt, und durch die gleiche Größenordnung der Exponenten bei gleichmäßig abfallenden Koeffizienten der Overhead durch Abschätzen der Exponentenbits gering ist.
Die Methode *Abschneiden* kann genutzt werden, wenn man keine Operationen auf den Bits selbst, sondern nur eine Bitmaske verwenden möchte. Dafür sollte man jedoch einen guten Grund haben, da die Operation zum Runden sich für den Zugewinn an Genauigkeit lohnt.

4 Verwandte Arbeiten und Ausblick

4.1 Verwandte Arbeiten

Hierarchische Methoden zur Kompression und Approximation haben in Form von Wavelets schon in verschiedenen Bereichen der Signalverarbeitung Einzug erhalten. [5]. Insbesondere zur Kompression von Bild- und Videodaten werden Wavelets schon sehr lange verwendet [4]. Da das Ziel dieser Arbeit die Approximation von zweimal stetig differenzierbaren Funktionen ist und die meisten Verfahren mit Wavelets sich auf die Kompression von Bilddaten oder ähnlichen Daten konzentrieren, sind die Ideen nur bedingt übertragbar.

4.2 Ausblick

Diese Arbeit zeigt, dass man bei der Funktionsapproxmierung mit hierarchischen Basen in einer Dimension den Speicheraufwand der Koeffizienten unter und ohne Zuhilfenahme der Datentypen Posits deutlich reduzieren kann. Nun kann man versuchen, die in dieser Arbeit vorgestellten Methoden an höhere Dimensionen anzupassen. In höheren Dimensionen können dann die Methoden im Forschungsfeld der Dünnen Gitter [2] mehr Anwendung finden.
Da diese Arbeit die genaue technische Umsetzung der Methode weitestgehend ausblendet, ergeben sich hier weitere Forschungsfragen. So kann eine Implementierung für Hardware mit und ohne Unterstützung von Posits untersucht werden, um eine effektive Methode zur Reduktion des Arbeitsspeicherbedarfs von Wissenschaftlichen Rechnungen zu entwickeln. Des Weiteren gilt es den Trade-off bezüglich Speicherbedarf und Rechenaufwand genauer zu untersuchen, insbesondere wenn die Auswertung der Funktion hohe Kosten verursacht.

Literaturverzeichnis

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta numerica*, 13:147–269, 2004.

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

[4] A. Lewis and G. Knowles. Image compression using the 2-d wavelet transform. *IEEE Transactions on Image Processing*, 1(2):244–250, 1992.

[5] M. Vetterli. Wavelets, approximation, and compression. *IEEE Signal Processing Magazine*, 18(5):59–73, 2001.

Selbsständigkeitserklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Datum	**Unterschrift**