View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 261

# Kompression von numerischen Datensätzen mittels mehrdimensionaler hierarchischer Teilraumschemata

Tim Würtele

**Studiengang:** Informatik

**Prüfer/in:** JP Dr. Dirk Pflüger

**Betreuer/in:** Dr. Stefan Zimmer

**Beginn am:** 21. September 2015

**Beendet am:** 22. März 2016

**CR-Nummer:** E.4, G.1.1, G.1.2, H.1.1

## Zusammenfassung

Bei der Arbeit mit großen numerischen Datensätzen ist es oft wünschenswert, diese zur Übertragung und Speicherung zu komprimieren. Häufig sind solche Datensätze das Ergebnis einer Simulation von physikalischen Zusammenhängen, zum Beispiel Strömungen. Im Endeffekt sind diese Datensätze daher eine diskrete Darstellung einer (unbekannten) Funktion, die einen physikalischen Zusammenhang beschreibt. Da Funktionen, die physikalische Zusammenhänge beschreiben, in der Regel relativ glatt sind, bietet sich als Vorstufe zur Kompression eine Transformation der Daten an, bei der Differenzen zwischen den Datenpunkten gebildet werden. Wegen der glatten Ausgangsdaten sind diese Differenzen im Betrag kleiner und unterscheiden sich weniger als die ursprünglichen Werte, was die Kompression erleichtert.

Eine Möglichkeit, solche Differenzen zu bilden, stellt die Hierarchisierung der Ausgangsdaten dar. Neben den erwähnten kleineren Beträgen der Differenzen bietet die hierarchisierte Darstellung weitere Eigenschaften, die bei der Kompression ausgenutzt werden können. Auf Basis von bzip2, einem allgemeinen Datenkompressionsverfahren, wird in dieser Arbeit ein Verfahren zur Kompression von hierarchisierten numerischen Datensätzen entwickelt und der Einfluss verschiedener Parameter auf die Kompressionsleistung untersucht.

# Inhaltsverzeichnis

**1. Einleitung** **1**
1.1. Verwandte Arbeiten . . . 1

**2. Verfahren** **5**
2.1. Hierarchisierung . . . 5
2.2. Verlustbehaftete Vorbehandlung . . . 7
2.3. Gleitkommazahlen und ihre Darstellung nach IEEE 754 . . . 9
2.4. bzip2- und pyFPbzip-Kompressionspipeline . . . 11
2.4.1. Lauflängencodierung . . . 12
2.4.2. Burrows-Wheeler-Transformation . . . 13
2.4.3. Move-to-Front-Transformation . . . 14
2.4.4. Codierung von Nullläufen . . . 15
2.4.5. Huffmancodierung . . . 17
2.4.6. Alphabetcodierung . . . 19
2.4.7. Unterschiede zwischen bzip2- und pyFPbzip-Kompressionspipeline . . . 21

**3. Implementierung** **23**
3.1. Erzeugung und Hierarchisierung der Datensätze . . . 23
3.2. Kompressionspipeline . . . 24
3.3. Vorbereitung der Daten für die Kompressionspipeline . . . 25
3.4. Behandlung der Vorzeichen . . . 26

**4. Numerische Ergebnisse** **27**
4.1. Testdaten & -parameter . . . 27
4.2. Ergebnisse . . . 29
4.2.1. Verlustfreie Kompression . . . 32
4.2.2. Verlustbehaftete Kompression . . . 32

**5. Ausblick** **35**

**Literatur** **37**

**A. Beispiel: Vorbereitung eines Datensatzes für die Kompressionspipeline** **39**

# 1. Einleitung

In vielen Bereichen der naturwissenschaftlichen Forschung muss regelmäßig mit großen numerischen Datensätzen umgegangen werden. Diese Datensätze stammen meist aus Simulationen oder Messungen und werden in der Regel als lange Liste von Gleitkommazahlen gespeichert. Dabei können die Datensätze so groß sein, dass die Übertragung oder auch nur Speicherung ein Problem darstellt. Darum wurden und werden Verfahren entwickelt, um solche Datensätze zu komprimieren, diese Verfahren können grob in zwei Gruppen eingeteilt werden; die verlustfreien und die verlustbehafteten Verfahren. Verlustfreie Verfahren erlauben eine exakte Rekonstruktion des ursprünglichen Datensatzes aus den komprimierten Daten, bei verlustbehafteten Verfahren werden kleine Abweichungen hingenommen, um bessere Kompressionsraten oder höhere Kompressionsgeschwindigkeit zu erreichen.

Neben solchen spezialisierten Kompressionsverfahren gibt es auch Verfahren, die für jede Art von Daten verwendet werden können, Beispiele dafür sind DEFLATE [4] – das Standardkompressionsverfahren für das Zip-Containerformat – und bzip2 [9]. In dieser Arbeit wurde auf Basis von bzip2 ein für hierarchisierte numerische Datensätze angepasstes Kompressionsverfahren entwickelt und untersucht. Die Hierarchisierung der Datensätze vor der eigentlichen Kompression führt dazu, dass bei der Kompression einige Eigenschaften der hierarchischen Darstellung ausgenutzt werden können. Die Kompression bei bzip2 basiert auf mehreren hintereinander ausgeführten Transformationen der Eingabe und einer anschließenden Entropiecodierung, diese *Kompressionspipeline* wurde übernommen und für hierarchisierte Datensätze als Eingabe angepasst.

Implementiert wurde die angepasste Kompressionspipeline in Python/C++ und anschließend mit unterschiedlich vorbereiteten Datensätzen evaluiert. Die dabei erzielten Kompressionsraten sind zwischen 1.5- und 18mal besser als sie mit bzip2 erreicht werden können, allerdings auf Kosten höherer Laufzeit.

## 1.1. Verwandte Arbeiten

Die Kompression von Daten allgemein und numerischen Datensätzen speziell ist schon länger Gegenstand der Forschung. Im Folgenden werden einige der Arbeiten vorgestellt, die sich speziell mit der Kompression von numerischen Datensätzen beschäftigen.

Zur verlustfreien Kompression von zweidimensionalen geologischen Datensätzen, die aus 16 Bits langen Ganzzahlen bestehen, hat Wessel 2003 ein Verfahren vorgestellt [10], bei dem die Daten zunächst differenziert, dann codiert und abschließend mit bzip2 komprimiert werden.

Differenziert wird dabei zunächst entlang der ”Zeilen” des Datensatzes und anschließend entlang der ”Spalten”, d. h. es wird quasi $\frac{\partial^2 z}{\partial x \partial y}$ diskret berechnet, wobei pro Zeile und Spalte noch ein ”Startwert” zur Rekonstruktion der Ausgangsdaten behalten wird. Diese (zweiten) Differenzen und die Startwerte werden dann nach einem statischen Schema codiert und anschließend mit bzip2 nochmals weiter komprimiert. Durch die statische Codierung und die relativ einfache Differenzenbildung ist das Verfahren insgesamt algorithmisch einfach, komprimiert relativ glatte Datensätze aber dennoch deutlich effizienter, als es durch direktes Komprimieren mit bzip2 möglich ist. Die erzielbare (De-)Kompressionsgeschwindigkeit dürfte recht hoch sein, z. B. können alle ”Zeilen” bzw. ”Spalten” parallel differenziert werden, allerdings ist das Verfahren auf Ganz- bzw. Festkommazahlen der Länge 16 Bits beschränkt und damit für Anwendungen, die höhere Genauigkeit und/oder Gleitkommazahlen erfordern, ungeeignet.

Ein Verfahren zur Kompression von beliebig-dimensionalen Datensätzen, die aus Gleitkommazahlen bestehen, wurde von Lindstrom und Isenburg [8] entwickelt. Dabei wird auf Basis der bisher verarbeiteten Werte der nächste Wert ”geraten” und dann die Differenz zwischen dem tatsächlichen und dem ”geratenen” Wert gebildet. Diese Differenzen werden dann in Intervalle einsortiert, so dass jede der Differenzen durch eine Intervallnummer und einen Offset innerhalb des Intervalls dargestellt werden kann. Die Intervallnummern werden dann Entropiecodiert und anschließend werden alle Nummern mit einer Bereichscodierung komprimiert. Damit konnten, verglichen mit dem für beliebige Daten entworfenen DEFLATE-Algorithmus, deutlich bessere Kompressionsraten bei kürzeren Laufzeiten erzielt werden.

Ein weiteres Verfahren zur verlustfreien Kompression von beliebigen Sätzen von Gleitkommazahlen im IEEE 754-Format (doppelte Genauigkeit) stammt von Burtscher und Ratanaworabhan [3], dabei wird – ähnlich wie beim Verfahren von Lindstrom und Isenburg – der nächste Wert basierend auf den bisher verarbeiteten Werten ”geraten”, hier allerdings durch zwei verschiedene Algorithmen. Für jede Gleitkommazahl wird dann der bessere Tipp mit dem tatsächlichen Wert XOR-verknüpft und die Anzahl der führenden Nullbytes im Ergebnis dieser Verknüpfung durch drei Bits codiert, zusätzlich gibt ein Bit an, welcher der beiden ”Ratealgorithmen” verwendet wurde; danach folgen ”roh” die restlichen Bytes der Eingabezahl. Das Besondere an diesem Verfahren ist, dass – je nach Implementierung – nach einer kurzen Anlaufzeit alle Daten außer der eigentlichen Ein- und Ausgabe im CPU-Cache liegen, dadurch ist die Kompressionsgeschwindigkeit sehr hoch (im Bereich von GBit/s). Außerdem wird es dadurch möglich, die Kompression effizient als Teil einer Simulation zu implementieren, so dass wertvolle Speicherbandbreite gespart wird. Die erreichten Kompressionsraten sind bis zu dreimal besser als die von bzip2 (dabei ist bzip2 allerdings signifikant langsamer).

Ein Verfahren zur verlustbehafteten Kompression von Gleitkomma-Datensätzen stammt von Fischer et al. [5], dabei wird die Eingabe – wie beim in dieser Arbeit vorgestellten Verfahren – zunächst hierarchisiert. Durch die Hierarchisierung ist es möglich, ohne weiteres Wissen über die Daten den vorhandenen Diskretisierungsfehler abzuschätzen. Mit dieser Schätzung können die hierarchisierten Daten ”ein wenig verändert” werden, so dass möglichst viele Nullbits entstehen, ohne dass ein Fehler eingebracht wird, der größer als der ohnehin vorhandene Diskretisierungsfehler ist. Zusätzlich induziert die Hierarchisierung eine Aufteilung der Eingabe in Levels, innerhalb dieser Levels liegen alle Werte in etwa in derselben Größenordnung.

Das wird ausgenutzt, indem die hierarchisierten Werte nach Levels sortiert werden und anschließend jeder Wert mit seinem Vorgänger XOR-Verknüpft wird. Dadurch entstehen – weil aufeinanderfolgende Werte oft in derselben Größenordnung liegen – führende Nullbytes, die nach einem festen Schema, ähnlich dem vom Burtscher und Ratanaworabhan, codiert werden. Die erreichten Kompressionsraten sind verglichen mit den Ergebnissen von Lindstrom und Isenburg sehr gut, allerdings ist der Vergleich nur begrenzt aussagekräftig, weil das Verfahren von Lindstrom und Isenburg verlustfreie Kompression bietet. Das von Fischer et al. beschriebene Verfahren zum Abschätzen des Diskretisierungsfehlers und der nachfolgenden ”kleinen Änderungen” an den hierarchisierten Werten wird auch in dieser Arbeit benutzt, um neben verlustfreier auch verlustbehaftete Kompression zu ermöglichen.

# 2. Verfahren

## 2.1. Hierarchisierung

Vor der eigentlichen Kompression, wie sie in Abschnitt 2.4 beschrieben ist, müssen die Eingabedaten hierarchisiert werden. Insbesondere ist das Hierarchisieren selbst keine Kompression, sondern nur eine Transformation der Daten; die hierarchisierten Daten weisen aber einige Eigenschaften auf, die für die Kompression ausgenutzt werden können. In der folgenden Beschreibung wird davon ausgegangen, dass der Datensatz, der verarbeitet wird, die Diskretisierung einer (unbekannten) Funktion auf einem $d$-dimensionalen, regulären Gitter mit $2^n + 1$ Punkten in jeder Dimension darstellt[1]. Dabei gilt:

$$G_n^d := \left\{ \left( \frac{k_{x_1}}{2^n}, \frac{k_{x_2}}{2^n}, \dots, \frac{k_{x_d}}{2^n} \right) \middle| k_{x_i} \in \{0, 1, 2, ..., 2^n\} \right\}$$

Grundlage der Hierarchisierung ist eine in jeder Dimension lineare Interpolation der Funktion, die dem Datensatz zugrunde liegt. Gespeichert werden müssen dann nur noch Differenzen zwischen der Interpolation und den tatsächlichen Werten, diese Differenzen werden im Folgenden als *Überschüsse* bezeichnet. Bei Bungartz [1] finden sich mathematische Grundlagen und weitere Details zur Hierarchisierung.

Zur einfacheren Erklärung wird hier zunächst neben $d = 1$ angenommen, dass die Funktion $f : \mathbb{R} \to \mathbb{R}$, die hierarchisiert werden soll, bekannt ist und dass das Intervall, auf dem hierarchisiert wird, der Einheitswürfel (also $[0, 1]$) ist. Das Gitter, auf dem $f$ dann diskretisiert wird, ist dementsprechend

$$G_n^1 := \left\{ \left( \frac{k_x}{2^n} \right) \middle| k_x \in \{0, 1, 2, ..., 2^n\} \right\}$$

Dieses Gitter kann auch als die Vereinigung von mehreren Teilgittern $l_i$ betrachtet werden, innerhalb derer alle Punkte den selben Abstand haben, wobei dieser Abstand in $l_{i+1}$ gerade halb so groß wie in $l_i$ ist. Erreicht wird das, indem $l_{i+1}$ gerade die Punkte enthält, die genau zwischen zwei Punkten von $\bigcup_{j=0}^{i} l_j$ liegen, im Folgenden wird $l_i$ auch als Level $i$ bezeichnet, die ersten vier Levels für den eindimensionalen Fall sind in Abbildung 2.1 veranschaulicht. Zugleich werden Bezeichner $x_{i,j}$ eingeführt, die im Endeffekt Namen für die Gitterpunkte des Gitters $G_n^1$ sind (es kann mehrere Bezeichner für den selben Gitterpunkt geben!):

$$x_{i,j} := j \cdot 2^{-i} \quad j \in \{0, ..., 2^i\}, i \in \{0, 1, ..., n\}$$

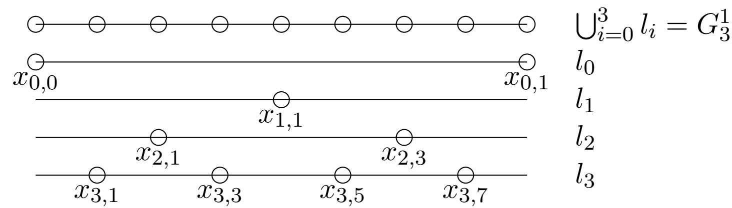



Abbildung 2.1.: Levels 0 bis 3 in einer Dimension, Idee zur Grafik aus [5]

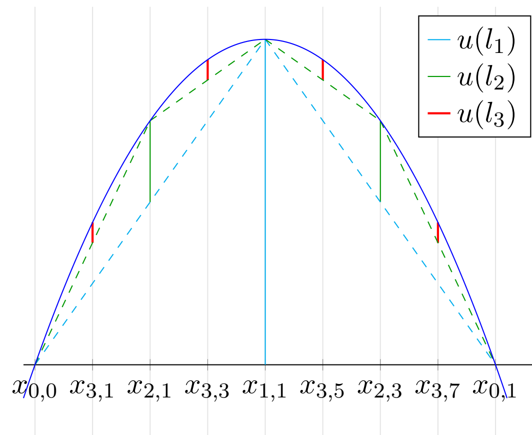



Abbildung 2.2.: Darstellung der Hierarchisierung für $x \mapsto 4x(1-x)$

Dabei steht $x_{i,j}$ für den $j$-ten Gitterpunkt, wenn alle Levels bis einschließlich $i$ berücksichtigt werden.

Die Überschüsse (also das Ergebnis der Hierarchisierung) für jeden Punkt $x_{i,j}$ werden dann wie folgt berechnet:

$$\begin{aligned} u(x_{i,j}) &:= f(x_{i,j}) - \frac{f(x_{i-1,j-1}) + f(x_{i-1,j+1})}{2} \\ u(x_{i,0}) &:= f(0) \\ u(x_{i,2^i}) &:= f(1) \end{aligned}$$

Das bedeutet, dass eine lineare Interpolation zwischen den Funktionswerten der Gitterpunkte links und rechts von $x_{i,j}$ auf den gröberen Gittern ($l_j$ mit $j < i$) vorgenommen wird, diese Interpolation an der Stelle $x_{i,j}$ ausgewertet wird und die Differenz zwischen der Interpolation und dem tatsächlichen Funktionswert an der Stelle $x_{i,j}$ "ausgegeben" wird.

Ein Beispiel ist in Abbildung 2.2 zu sehen, die Funktion, die hierarchisiert werden soll, ist blau dargestellt, die Überschüsse sind durchgezogene Linien und die (gedachte) interpolierende Funktion auf jedem Level ist gestrichelt eingezeichnet. Gut zu sehen ist hier, dass die

[1]Falls der Datensatz andere Dimensionen hat, muss ggf. mit Nullen aufgefüllt werden.

Überschüsse die Differenz zwischen der Interpolation und der tatsächlichen Funktion sind und dass die Überschüsse mit steigendem Level schnell kleiner werden. Tatsächlich kann gezeigt werden, dass falls $|f''|$ beschränkt ist, die Überschüsse (betragsmäßig) mit jedem Level um ungefähr Faktor vier kleiner werden (also $|u(x_{i,j})| \propto 4^{-i}$ für ungerade $j$), Bungartz und Griebel [1] haben diesen Zusammenhang auch für beliebig viele Dimensionen gezeigt.

Für die Erweiterung auf zwei Dimensionen betrachten wir den Datensatz (also die Funktionswerte der zu hierarchisierenden Funktion auf $G_n^2$) als zweidimensionales Feld. Weil die Interpolation pro Dimension linear sein soll, kann als erstes jede Zeile des Feldes hierarchisiert werden und danach jede Spalte (die Reihenfolge ist dabei egal). Analog ist das Vorgehen für beliebig viele Dimensionen, es wird in Richtung jeder Dimension einzeln nacheinander hierarchisiert.

Die erste Beobachtung beim Übergang von einer auf mehr Dimensionen ist, dass Levels nicht mehr mit einem Index auskommen; aus der im letzten Absatz beschriebenen Hierarchisierung mehrdimensionaler Datensätze folgt, dass es Levels in Richtung jeder Dimension geben muss. Für zwei Dimensionen zeigt Abbildung 2.3 die Levels $l_{i,j}$ für $0 \leq i, j \leq 3$, die folgendermaßen definiert sind ($l_k$ entspricht der $k$-ten Diagonale in Abbildung 2.3):

$$
\begin{aligned}
l_{0,0} &:= \{ (x, y) |\, x \in \{0, 1\}, y \in \{0, 1\} \} \\
l_{0,j} &:= \left\{ \left( \frac{k_x}{2^j}, y \right) \middle|\, k_x \in \left\{ 1, 3, 5, ..., 2^j - 1 \right\}, y \in \{0, 1\} \right\} \\
l_{i,0} &:= \left\{ \left( x, \frac{k_y}{2^i} \right) \middle|\, x \in \{0, 1\}, k_y \in \left\{ 1, 3, 5, ..., 2^i - 1 \right\} \right\} \\
l_{i,j} &:= \left\{ \left( \frac{k_x}{2^j}, \frac{k_y}{2^i} \right) \middle|\, k_x \in \left\{ 1, 3, 5, ..., 2^j - 1 \right\}, k_y \in \left\{ 1, 3, 5, ..., 2^i - 1 \right\} \right\} \\
l_k &:= \bigcup_{\substack{i+j=k \\ i,j \in \mathbb{N}_0}} l_{i,j}
\end{aligned}
$$

Wie in Abbildung 2.4 zu sehen ist, werden die Überschüsse auch in zwei Dimensionen schnell kleiner.

Genau diese Eigenschaft, dass die Absolutwerte der Überschüsse mit zunehmendem Level sehr schnell kleiner werden, macht die Hierarchisierung so interessant, um Datensätze zur Kompression vorzubereiten: Kleinere Werte können bei gleicher Genauigkeit mit weniger Bits gespeichert werden. Für die in Abschnitt 2.2 beschriebene verlustbehaftete Kompression ist außerdem die bereits erwähnte Eigenschaft wichtig, dass die Überschüsse mit jedem Level um ca. $\frac{1}{4}$ kleiner werden.

## 2.2. Verlustbehaftete Vorbehandlung

Mit dem Wissen, dass die Überschüsse pro Level um etwa den Faktor vier kleiner werden, kann – interessanterweise ohne weiteres Wissen über die Daten – der Diskretisierungsfehler eines

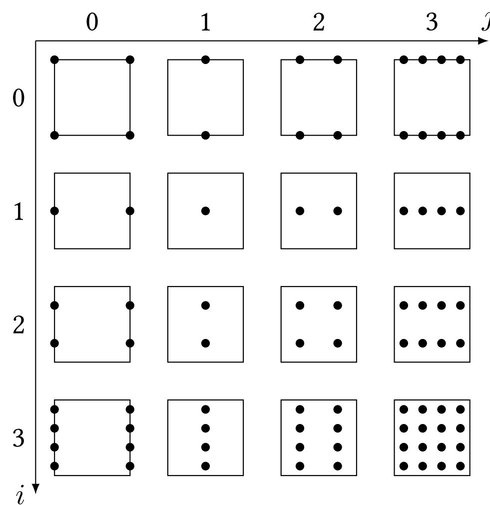



Abbildung 2.3.: Levels $l_{i,j}$ in 2D

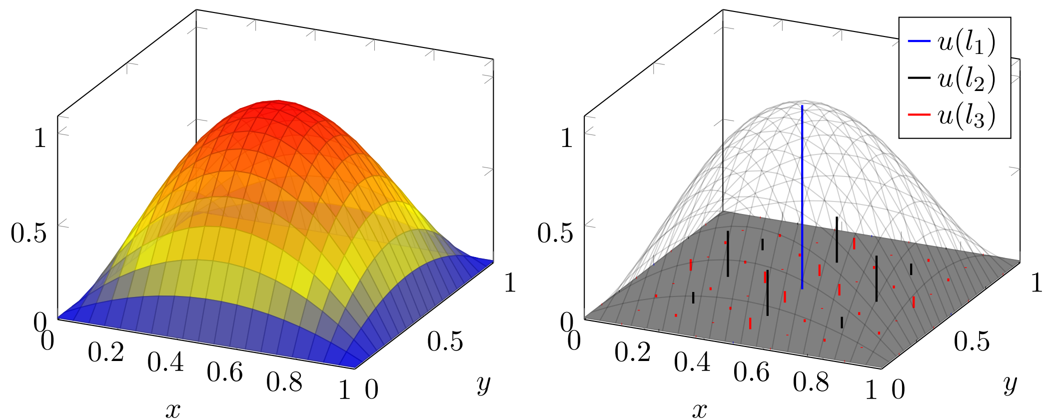



Abbildung 2.4.: Plot und einige Überschüsse für $16x(x-1)y(y-1)$

Datensatzes abgeschätzt werden. Der Grund dafür ist, dass bei der Hierarchisierung "Randlevels" auftreten, bei den in Abbildung 2.3 gezeigten Levels wären das z. B. $l_{3,0}$ und $l_{0,3}$. Auf den Levels $l_{4,0}$ bzw. $l_{0,4}$ würde man Überschüsse erwarten, deren Beträge ungefähr $\frac{1}{4}$mal so groß sind wie die Beträge der Überschüsse auf den Levels $l_{3,0}$ und $l_{0,3}$. Weil der Datensatz diese Werte aber nicht enthält, kann angenommen werden, dass der Diskretisierungsfehler ungefähr $\frac{1}{4}$ des maximalen Überschussbetrags auf den Randlevels[2] ist.

Wenn der Diskretisierungsfehler $\varepsilon$ bekannt ist, können die Überschüsse ein wenig verändert werden, so dass sie sich besser komprimieren lassen – die Idee und ein darauf basierendes

[2] Auf den Levels $l_{3,0}$ und $l_{0,3}$ würde man diese Schätzung in der Realität noch nicht wagen, weil die Abstände zwischen den Gitterpunkten noch recht groß sind – bei einem Datensatz mit nur 32 Werten ist die Sinnhaftigkeit von verlustbehafteter Kompression aber ohnehin fragwürdig.

Codierungs- und Kompressionsschema werden von Fischer et al. [5] beschrieben.

Etwas formaler formuliert schätzt man also den Diskretisierungsfehler $\varepsilon$ eines Datensatzes auf dem Gitter $G_n^2$ ab durch:

$$\varepsilon = \frac{\max\{|u(\mathbf{x})| : \mathbf{x} \in \overbrace{l_{0,n} \cup l_{n,0}}^{\text{Randlevels}}\}}{4}$$

Damit können dann in der Darstellung aller Überschusswerte alle Stellen "abgeschnitten" werden, deren Wert $\leq \varepsilon$ ist. Das bedeutet zugleich, dass auf den gröberen Levels meist keine oder nur wenige Stellen abgeschnitten werden können, auf den feineren Levels – die deutlich mehr Punkte enthalten und daher einen größeren Teil der Daten ausmachen – dagegen oft viele Stellen abgeschnitten werden können. Da durch das Abschneiden von signifikanten Stellen der zu komprimierende Informationsgehalt der Daten kleiner wird, lassen sich diese in der Regel besser komprimieren. Wenn die Überschüsse im IEEE 754-Format vorliegen, bedeutet "abschneiden" nichts anderes, als dass Mantissenbits auf Null gesetzt werden (oder die komplette Zahl, falls $|u(\mathbf{x})| \leq \varepsilon$).

Der dabei eingebrachte Fehler ist in der Größenordnung des ohnehin vorhandenen Diskretisierungsfehlers, die Datenqualität wird also nicht signifikant verschlechtert – vorausgesetzt, dass die Schätzung des Diskretisierungsfehlers ausreichend genau war. Es muss dabei allerdings darauf geachtet werden, dass sich die eingebrachten Fehler bei der Rekonstruktion der Daten aus der hierarchisierten Darstellung nicht aufsummieren. Das kann erreicht werden, indem die Überschüsse $u$ vor Anwendung der Abschneidefunktion cut vorbehandelt werden und dann das Ergebnis der Vorbehandlung statt die "rohen" Überschüsse abgeschnitten wird, hier für eine Dimension:

$$\tilde{u}(x_{i,j}) := u(x_{i,j}) + \underbrace{\frac{u(x_{i-1,j-1}) + u(x_{i-1,j+1})}{2} - \frac{\text{cut}(u(x_{i-1,j-1})) + \text{cut}(u(x_{i-1,j+1}))}{2}}_{\text{Auf Level } i-1 \text{ eingebrachter Abschneidefehler}}$$

Der Sinn dieser Vorbehandlung besteht darin, die auf Level $i-1$ eingebrachten Abschneidefehler auf Level $i$ vor dem Abschneiden wieder auszugleichen, so dass sich die Abschneidefehler nicht akkumulieren.

## 2.3. Gleitkommazahlen und ihre Darstellung nach IEEE 754

Die Darstellung von reellen Zahlen in endlichem, und diskretem Speicher ist immer auf eine abzählbare, endliche Teilmenge der reellen Zahlen beschränkt. Gleitkommazahlen sind eine Möglichkeit, einen Teil der reellen Zahlen diskret darzustellen. Eine Gleitkommazahl setzt sich aus einer *Mantisse* $M$ fester Länge und einem ganzzahligen *Exponenten* $E$ zusammen, für eine eindeutige Darstellung wird die Mantisse üblicherweise *normalisiert.* D. h. dass wertgleiche

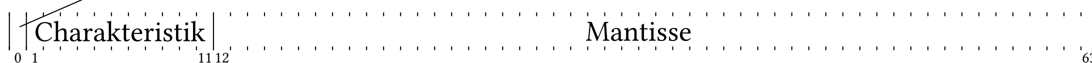



Abbildung 2.5.: Format für die Darstellung von Gleitkommazahlen doppelter Genauigkeit nach IEEE 754

Zahlen ungleich Null immer gleich dargestellt werden, was den Vergleich von Zahlen vereinfacht. Die Menge der normalisierten Gleitkommazahlen zur Basis $B \in \mathbb{N}_\emptyset \setminus \{1\}$ und Mantissenlänge $t \in \mathbb{N}_\emptyset$ ist dabei wie folgt definiert:

$$\mathbb{F}_{B,t} := \left\{ M \cdot B^E : |M| = 0 \text{ oder } B^{t-1} \leq |M| < B^t, M \in \mathbb{Z}, E \in \mathbb{Z} \right\}$$

Diese Menge ist abzählbar, jedoch nicht endlich, weil der Exponent beliebige ganzzahlige Werte annehmen kann. Für die Darstellung im Speicher von Computern muss also neben der Mantisse auch der Exponent eingeschränkt werden, so kommt man auf die *Maschinenzahlen*:

$$\mathbb{F}_{B,t,\alpha,\beta} := \{\mathbb{F}_{B,t} : \alpha \leq E \leq \beta\}$$

Für eine plattform- oder auch nur programmübergreifende Kompatibilität bezüglich Maschinenzahlen gibt es neben anderen die Norm IEEE 754 [7], die weite Verbreitung gefunden hat. Die Norm regelt unter anderem, wie Exponent und Mantisse im Speicher repräsentiert werden und wie mit Ausnahmefällen (zu große / zu kleine Zahlen, ungültige Werte) umgegangen wird. Im Folgenden wird die Darstellung von "Gleitkommazahlen doppelter Genauigkeit" beschrieben, der dadurch abbildbare Zahlenbereich ist etwas größer[3] als $\mathbb{F}_{2,53,-1074,971}$.

Die Darstellung ist 64 Bit lang, wie Abbildung 2.5 zeigt, sind diese aufgeteilt in ein Vorzeichenbit, elf Bits zur Darstellung der Charakteristik und 52 Bits zur Darstellung der Mantisse. Die Charakteristik ist gleich dem Exponenten verschoben um einen *Bias* von $1023_{10}$. Durch diesen Bias ist die Charakteristik immer positiv, es muss also kein Vorzeichen der Charakteristik beachtet werden. Da die Mantisse – von Sonderfällen abgesehen – normalisiert ist, beginnt sie in binärer Darstellung immer mit einer `1`, daher wird dieses Bit nicht explizit gespeichert, sondern implizit angenommen. Das führt dazu, dass die Mantisse in den meisten Fällen effektiv ein Bit länger ist (die Sonderfälle werden gleich behandelt). Der Betrag einer so gespeicherten Zahl ist dann $1{,}\text{Mantisse}_2 \cdot 2^{\text{Charakteristik}-1023}$.

Falls eine Zahl dargestellt werden soll, deren Betrag kleiner als $2^{-1022}$ ist, wird die Charakteristik auf Null gesetzt und die Mantisse *ohne* implizite `1` am Anfang interpretiert. Der Betrag einer solchen denormalisierten Zahl ist dann $0{,}\text{Mantisse}_2 \cdot 2^{-1022}$, die beiden Darstellungen der Null ($\pm 0$) gehören ebenfalls zu den denormalisierten Zahlen.

Zahlen, deren Betrag zu groß ist, um dargestellt zu werden, werden als "unendlich" gespeichert, dafür wird die Charakteristik auf ihren maximalen Wert ($2^{12} - 1$) und alle Mantissenbits auf

[3]"Etwas größer", weil mit denormalisierten Zahlen ein Sonderfall definiert ist, in dem $0 < |M| \leq 2^{52}$ erlaubt ist.

| Charakteristik $C$ | Mantisse | Bedeutung |
|---|---|---|
| $0 < C < 2^{11} - 1$ | $M \geq 0$ | Zahl mit Wert $(-1)^s \cdot (1 + \frac{M}{2^{52}}) \cdot 2^{C-1023}$ |
| $C = 0$ | $M = 0$ | $\pm 0$ (vorzeichenbehaftete Null) |
| $C = 0$ | $M > 0$ | Zahl mit Wert $(-1)^s \cdot \frac{M}{2^{52}} \cdot 2^{C-1023}$ |
| $C = 2^{11} - 1$ | $M = 0$ | "Unendlich" |
| $C = 2^{11} - 1$ | $M > 0$ | Keine Zahl |

Tabelle 2.1.: Interpretation von Gleitkommazahlen nach IEEE 754, $M$ entspricht dabei den hinteren 52 Bits, interpretiert als Ganzzahl, $s$ ist das Vorzeichen

Null gesetzt. Falls das Ergebnis einer Berechnung nicht definiert ist (z. B. $\frac{0}{0}$), wird die Charakteristik auf ihren maximalen Wert und die Mantisse auf einen beliebigen Wert ungleich Null gesetzt. Tabelle 2.1 fasst die Darstellung und Sonderfälle nochmals zusammen.

Wenn im Folgenden von der "Mantisse" die Rede ist, sind damit die 52 Bits gemeint, die tatsächlich gespeichert werden. Ebenso bezieht sich im Folgenden "Exponent" auf die elf Bits, mit denen die Charakteristik gespeichert wird. Der Grund dafür ist, dass beides innerhalb der Kompressionspipeline einfach als Folge von Bits verarbeitet wird, unabhängig von der Bedeutung dieser Bits.

## 2.4. bzip2- und pyFPbzip-Kompressionspipeline

Als Grundlage für pyFPbzip dient bzip2 [9], ein quelloffenes Programm zur Kompression beliebiger Daten. Die erste Version wurde 1996 von Julian Seward veröffentlicht. Bzip2 ist ein blockbasiertes Kompressionsverfahren, d. h. die Eingabedaten werden in Blöcken fester Größe verarbeitet, pyFPbzip dagegen verarbeitet immer die gesamte Eingabe auf einmal. Im Fall von bzip2 werden diese Blöcke komplett unabhängig voneinander verarbeitet, bei anderen blockbasierten Verfahren wie z. B. DEFLATE sind komprimierte Blöcke unter bestimmten Umständen von vorhergehenden Blöcken abhängig.

Jeder Block wird durch eine Kompressionspipeline verarbeitet, eine leicht modifizierte Version dieser Kompressionspipeline kommt auch beim im Rahmen dieser Arbeit entwickelten Verfahren pyFPbzip zum Einsatz. Abbildung 2.6 zeigt schematisch den Ablauf der bzip2-Kompressionspipeline, die einzelnen Schritte werden im Folgenden näher erläutert. Dabei wird auch auf die Unterschiede zwischen der bzip2-Kompressionspipeline und der pyFPbzip-Kompressionspipeline eingegangen, sowie die Umkehrbarkeit der einzelnen Stufen gezeigt.

Anmerkung zur Notation: In den folgenden Unterabschnitten werden einzelne Bits mit `0` und `1`, die Dezimalzahlen Null und Eins dagegen mit $0$ und $1$ notiert, falls aus dem Kontext nicht eindeutig hervorgeht, was gemeint ist.

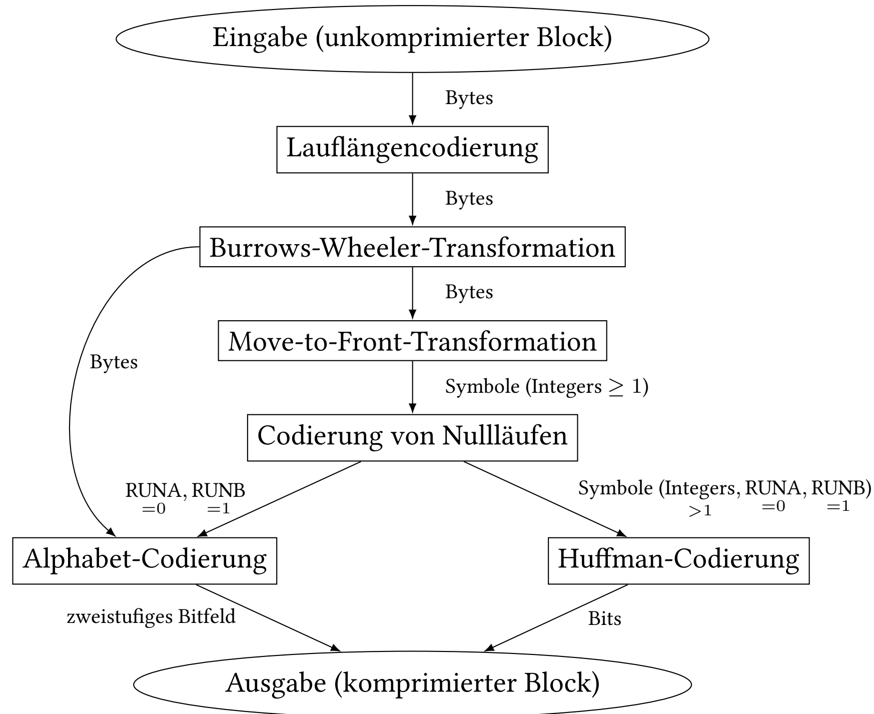



Abbildung 2.6.: bzip2-Kompressionspipeline

### 2.4.1. Lauflängencodierung

Die erste Stufe der Kompressionspipeline ist eine einfache Lauflängencodierung, die Eingabe dieser Stufe sind die "rohen" Symbole des Eingabeblocks, diese sind bei bzip2 immer genau ein Byte groß, bei pyFPbzip ist die Symbolgröße einstellbar. Dabei werden Läufe von vier oder mehr gleichen Symbolen in Folge codiert, indem viermal das Symbol selbst und danach ein Wiederholungszähler ausgegeben wird. Läufe von weniger als vier Symbolen werden unverändert ausgegeben.

D. h. in `ABBBBDAAAAAAB` würde nach den vier `B`s eine Null eingefügt, weil nach diesen vier `B`s direkt ein anderes Symbol steht, d. h. es muss noch Null mal wiederholt werden. Die sechs `A`s würden ersetzt durch vier `A`s und die Wiederholungszahl zwei, weil nach den vier `A`s nochmals zwei stehen. Aus `ABBBBDAAAAAAB` würde also `ABBBB0DAAAA2B`.

Weil die bzip2-Kompressionspipeline mit 8-Bit-Symbolen arbeitet, kann der Wiederholungszähler nicht größer als 255 werden, in der pyFPbzip-Implementierung wird als Länge für den Wiederholungszähler die Länge der Eingabesymbole (ggf. die Länge des längsten Eingabesymbols), mindestens aber acht Bits verwendet.

Ausgegeben werden Symbole und Zahlen, wobei die Zahlen binär dargestellt sowohl bei bzip2 als auch bei pyFPbzip im Bereich der ohnehin vorhandenen Symbole liegen, d. h. sie müssen in den nachfolgenden Stufen der Kompressionspipeline nicht gesondert behandelt werden. Dass

| | | | | |
|---|---|---|---|---|
| BANANA | | 0: ABANA**N** | | |
| ANANAB | | 1: ANABA**N** | | |
| NANABA | $\Longrightarrow$ | 2: ANANA**B** | $\Longrightarrow$ | Ausgabe: **NNBAAA**, 3 |
| ANABAN | Sortieren | 3: BANAN**A** | | |
| NABANA | | 4: NABAN**A** | | |
| ABANAN | | 5: NANAB**A** | | |

Abbildung 2.7.: Burrows-Wheeler-Transformation am Beispiel "BANANA"

diese Lauflängencodierung reversibel ist, indem das erste Symbol nach vier aufeinanderfolgenden gleichen Symbolen als Wiederholungszähler interpretiert und entsprechend expandiert wird, ist offensichtlich.

### 2.4.2. Burrows-Wheeler-Transformation

Die zweite Stufe der Kompressionspipeline ist eine Burrows-Wheeler-Transformation, in dieser Stufe findet keine Kompression statt, sondern nur eine (umkehrbare) Umsortierung der Eingabe, d. h. die Ausgabegröße ist gleich der Eingabegröße bzw. sogar um einen konstanten Wert größer. Die Umsortierung sorgt dabei dafür, dass sich wiederholende Symbolabfolgen in der Eingabe (z. B. `AN` in `BANANA`) zu Läufen gleicher Symbole in der Ausgabe werden. Solche Läufe werden durch die beiden nachfolgenden Stufen der Kompressionspipeline effizient komprimiert. Entwickelt wurde diese Transformation 1994 von Burrows und Wheeler [2].

Listing 2.1 zeigt eine einfache Python-Implementierung der Burrows-Wheeler-Transformation[4]. Zunächst wird eine Tabelle aufgebaut, deren Zeilen einfache Rotationen der Eingabe sind. Die Zeilen dieser Tabelle werden dann lexikographisch sortiert und die letzte Spalte der sortierten Tabelle ausgegeben. Um die Transformation beim Entpacken wieder umkehren zu können, wird zusätzlich die Zeilennummer der Eingabe in der sortierten Tabelle benötigt, diese Zeilennummer ist es, die die Ausgabe der Burrows-Wheeler-Transformation etwas länger macht als die Eingabe. In Abbildung 2.7 ist ein Beispiel für die Burrows-Wheeler-Transformation der Symbolfolge "BANANA" gezeigt.

Die Umkehrung der Burrows-Wheeler-Transformation ist zunächst nicht unbedingt intuitiv klar, im Prinzip aber einfach.

Für die inverse Burrows-Wheeler-Transformation wird die Tabelle mit Rotationen wie folgt von rechts nach links aus der Ausgabe last_column, orig_index aufgebaut: Zunächst wird eine leere, quadratische Tabelle erstellt, die Anzahl der Zeilen/Spalten entspricht dabei gerade der Länge von last_column. Dann wird last_column in die letzte Spalte der (leeren) Tabelle kopiert und die Zeilen der Tabelle sortiert. Anschließend wird last_column in die vorletzte Spalte kopiert (die Reihenfolge der gerade sortierten Zeilen bleibt dabei gleich), die dadurch entstandenen Zeilen der Länge zwei werden auch wieder sortiert. Dann wird ein weiteres mal last_column eingefügt,

[4] Die gezeigte Implementierung ist sehr ineffizient, insbesondere beim Speicherbedarf – der ist quadratisch, mit Zeigern kann der Speicherbedarf recht einfach auf linear reduziert werden.

```python
def bwt(symbols: list):
    # Tabelle mit allen Rotationen der Eingabe erstellen,
    # in der ersten Zeile dieser Tabelle steht die Eingabe selbst
    table = [symbols[i:] + symbols[:i] for i in range(len(symbols))]
    # Rotationen lexikographisch sortieren
    table = sorted(table)
    # Letzte Spalte der sortierten Tabelle ermitteln
    last_column = [row[-1:][0] for row in table]
    # Zeilennummer der Eingabe in der sortierten Tabelle finden
    orig_index = table.index(symbols)
    return last_column, orig_index
```

Listing 2.1: Burrows-Wheeler-Transformation in Python

```python
def ibwt(bwt_result: list, orig_index: int):
    # Anzahl der Symbole pro Zeile, zugleich Anzahl der Zeilen
    sym_count = len(bwt_result)
    # Leere Tabelle mit sym_count Zeilen erstellen
    table = [[] for _ in range(sym_count)]
    # Tabelle von rechts auffüllen
    for _ in range(sym_count):
        # In jeder Zeile i der Tabelle...
        for i in range(sym_count):
            # ... vorne das ite Zeichen aus bwt_result einfügen
            table[i] = [bwt_result[i]] + table[i]
        # Nachdem jede Zeile links um ein Zeichen erweitert wurde: Tabelle sortieren
        table.sort()
    # Zum Schluss: Die richtige Zeile der Tabelle ausgeben
    return table[orig_index]
```

Listing 2.2: Inverse Burrows-Wheeler-Transformation in Python

sortiert und so weiter, bis die Tabelle voll ist. Dann muss nur noch Zeile Nummer orig_index der Tabelle ausgegeben werden. Eine Beispielimplementierung in Python ist in Listing 2.2 angegeben[5]. In Abbildung 2.8 ist zudem die inverse Burrows-Wheeler-Transformation für das eben schon verwendete Beispiel ”BANANA” dargestellt.

### 2.4.3. Move-to-Front-Transformation

Die dritte Stufe der Kompressionspipeline wandelt die Symbolfolge aus der Burrows-Wheeler-Transformation in eine Folge von Zahlen um, die in den folgenden Stufen komprimiert wird. Für die Move-to-Front-Transformation wird zunächst das Eingabealphabet $\Gamma$, also die Menge verschiedener Symbole in der Eingabe, ermittelt und sortiert in einer Liste $L$ gespeichert. Dann wird die Eingabe Symbol für Symbol gelesen und für jedes Symbol $c$ der Index von $c$ in

[5] Auch diese Implementierung ist sehr ineffizient, mit Hilfe von Abkürzungslisten kann auch hier der Speicher- und Zeitbedarf deutlich reduziert werden.

Aus Burrows-Wheeler-Transformation: NNBAAA, 3

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $\Longrightarrow$ Einfügen | - - - - - N<br>- - - - - N<br>- - - - - B<br>- - - - - A<br>- - - - - A<br>- - - - - A | $\Longrightarrow$ Sortieren | - - - - - A<br>- - - - - A<br>- - - - - A<br>- - - - - B<br>- - - - - N<br>- - - - - N | $\Longrightarrow$ Einfügen | - - - - NA<br>- - - - NA<br>- - - - BA<br>- - - - AB<br>- - - - AN<br>- - - - AN | $\Longrightarrow$ Sortieren | - - - - AB<br>- - - - AN<br>- - - - AN<br>- - - - BA<br>- - - - NA<br>- - - - NA | $\Longrightarrow$ Einfügen | - - - NAB<br>- - - NAN<br>- - - BAN<br>- - - ABA<br>- - - ANA<br>- - - ANA | |

| | | | | | | |
|---|---|---|---|---|---|---|
| $\Longrightarrow$ Sortieren | - - - ABA<br>- - - ANA<br>- - - ANA<br>- - - BAN<br>- - - NAB<br>- - - NAN | $\Longrightarrow$ Einfügen $\cdots$ | $\Longrightarrow$ Einfügen | NABANA<br>NANABA<br>BANANA<br>ABANAN<br>ANABAN<br>ANANAB | $\Longrightarrow$ Sortieren | 0: ABANAN<br>1: ANABAN<br>2: ANANAB<br>3: BANANA<br>4: NABANA<br>5: NANABA |

Abbildung 2.8.: Inverse Burrows-Wheeler-Transformation am Beispiel von "BANANA"

$L$ ausgegeben, bevor jedoch das nächste Zeichen gelesen wird, wird $c$ in $L$ an den Anfang verschoben, daher auch der Name "Move-to-Front". Wenn ein Symbol mehrfach hintereinander auftritt, führt das zu Läufen von Nullen in der Ausgabe, die in der nächsten Stufe sehr effizient codiert werden können.

Auch hier ist in Listing 2.3 eine Python-Implementierung[6] gegeben, mit der man aus der Ausgabe "NNBAAA" der Burrows-Wheeler-Transformation die Zahlen $2, 0, 2, 2, 0, 0$ erhält. Die Worst-Case-Laufzeit bei $n$ Eingabesymbolen aus dem Alphabet $\Gamma$ ist in $\mathcal{O}(|\Gamma| \cdot n)$. Bei bzip2 fällt das nicht sehr ins Gewicht, weil das Alphabet ohnehin maximal 255 Zeichen und ein Block maximal $900kB$ umfasst, bei pyFPbzip können Alphabet und Blöcke aber beliebig groß gewählt werden, die Laufzeit ist dann entsprechend hoch.

Damit die Move-to-Front-Transformation rückgängig gemacht werden kann, müssen das sortierte Eingabealphabet $\Gamma$ und die Ausgabe $I$ der Move-to-Front-Transformation vorliegen, auf die Codierung des Eingabealphabets wird in Unterabschnitt 2.4.6 eingegangen. Auch beim Umkehren der Move-to-Front-Transformation wird eine Liste $L$ benötigt, die initial das sortierte Eingabealphabet $\Gamma$ enthält. Dann wird für jeden Index $i$ in $I$ zuerst $c = L[i]$ ausgegeben und dann $c$ in $L$ ganz nach vorne verschoben. Die Laufzeit der inversen Move-to-Front-Transformation ist linear in $|I|$, deshalb sind bzip2 und pyFPbzip asymmetrische Kompressionsverfahren, bei denen die Dekompression schneller als die Kompression ist.

### 2.4.4. Codierung von Nullläufen

In der vierten Stufe der Kompressionspipeline werden ausschließlich Nullen in der Ausgabe der vorhergehenden Move-to-Front-Transformation codiert. Zur Codierung werden zwei neue Symbole bzw. Zahlen, RUNA und RUNB, verwendet. Weil jede vorkommende Null (also insbesondere auch einzelne Nullen) codiert wird, kann eines dieser Symbole durch die Zahl Null dargestellt werden, sowohl bzip2 als auch pyFPbzip verwenden RUNA $= 0$ und RUNB $= 1$. Damit RUNB dabei nicht mit bereits vorhandenen Zahlen "kollidiert", werden zusätzlich zur Codierung der Nullen alle Zahlen um Eins erhöht. Codiert werden die Nullen mittels einer

[6] Auch diese Implementierung ist bezüglich der Performance nicht optimal.

```
def mtf(symbols: list):
    # Alphabet bestimmen (set) und sortieren
    L = sorted(list(set(symbols)))
    # Leere Ausgabeliste erstellen
    output = list()
    # Nacheinander alle Eingabesymbole abarbeiten
    for c in symbols:
        # Index des aktuellen Symbols in L ausgeben
        output.append(L.index(c))
        # c in L ganz nach vorne verschieben
        L.remove(c)
        L.insert(0, c)
    return output
```

Listing 2.3: Move-to-Front-Transformation in Python

*Standardnummerierung* über dem Alphabet $\Gamma := (\text{RUNA}, \text{RUNB})$ (dieses $\Gamma$ hat nichts mit dem Eingabealphabet aus Unterabschnitt 2.4.3 zu tun).

Die Standardnummerierung der Menge $\Gamma^*$ über dem (total geordneten, endlichen) Alphabet $\Gamma = (\gamma_1, ..., \gamma_m)$ ist eine bijektive Abbildung $\nu_\Gamma : \Gamma^* \to \mathbb{N}$, für die gilt:

$$\nu_\Gamma(\underbrace{\gamma_{i_1}\gamma_{i_2}\dots\gamma_{i_l}}_{\text{Wort aus } \Gamma^*}) = \sum_{k=1}^{l} m^{l-k} \cdot i_k$$

Anschaulicher wird das, wenn man die folgende Aufzählung von $\Gamma^*$ für $\Gamma = (1, 2)$ betrachtet, unter jedem Wort $\alpha_j$ ist dabei $\nu_\Gamma(\alpha_j)$ angegeben. $\varepsilon$ ist das leere Wort und es gilt in diesem Beispiel $\gamma_i = i$ und $m = 2$:

$$\Gamma^* = \left\{ \underset{0}{\varepsilon}, \underset{1}{1}, \underset{2}{2}, \underset{3}{11}, \underset{4}{12}, \underset{5}{21}, \underset{6}{22}, \underset{7}{111}, \underset{8}{112}, \underset{9}{121}, \underset{10}{122}, \dots \right\}$$

Die Standardnummerierung unterscheidet sich von der gewöhnlichen Darstellung in Stellenwertsystemen vor allem dadurch, dass es keine Ziffern gibt, die den Wert Null haben, d. h. es sind keine führenden und abschließenden Nullen möglich.

Um nun einen Nulllauf der Länge $j$ zu codieren, muss $\nu^{-1}_{(\text{RUNA},\text{RUNB})}(j)$ berechnet werden, das ist am einfachsten von rechts nach links möglich, wie die Implementierung in Listing 2.4 zeigt. Weil $\nu^{-1}_{(\text{RUNA},\text{RUNB})}(j)$ ohnehin von rechts nach links berechnet wird, wird es auch in dieser Reihenfolge gespeichert – für die nachfolgende Stufe ist die Reihenfolge nicht relevant und beim Dekomprimieren ist es algorithmisch ebenfalls einfacher, die höchstwertige Stelle als letztes zu lesen.

Die Ausgabe dieser Stufe ist eine Folge von Zahlen und den Symbolen RUNA und RUNB. Diese Stufe ist umkehrbar, indem alle Folgen von RUNA und RUNB durch entsprechende Nullläufe ersetzt werden, außerdem müssen wegen $\text{RUNB} = 1$ alle anderen Zahlen um Eins reduziert werden.

```python
# rle steht für Run Length Encoding (Lauflängencodierung)
def rle(j: int, RUNA: int, RUNB: int):
    while j > 0:
        # Wenn j ungerade ist, muss die aktuelle Stelle RUNA sein (RUNB kann nur gerade Zahlen codieren)
        if j % 2 == 1:
            yield RUNA
        else:
            yield RUNB
        # Letzte Stelle von j (bezüglich der Standardnummerierung) abschneiden
        j = (j-1) // 2
```

Listing 2.4: Berechnung von $v^{-1}_{(\text{RUNA,RUNB})}(j)$ in Python

### 2.4.5. Huffmancodierung

Die letzte Stufe der Kompressionspipeline ist eine Huffmancodierung [6] der Symbole. Dabei wird für jedes Symbol ein (binärer) Code bestimmt, dessen Länge von der Häufigkeit des Symbols abhängt, häufige Symbole haben kurze Codes, während seltenere Symbole längere Codes haben. Um aus einer Symbolfolge (z. B."BANANA") eine Codetabelle mit Huffmancodes für jedes Symbol zu erhalten, kann folgender Algorithmus verwendet werden, bei dem ein binärer Codebaum von den Blättern aus zur Wurzel hin erzeugt wird:

1. Zähle für jedes Symbol $c$, wie oft es in der Eingabe vorkommt und bestimme daraus die relative Häufigkeit $p(c)$ für jedes Symbol.

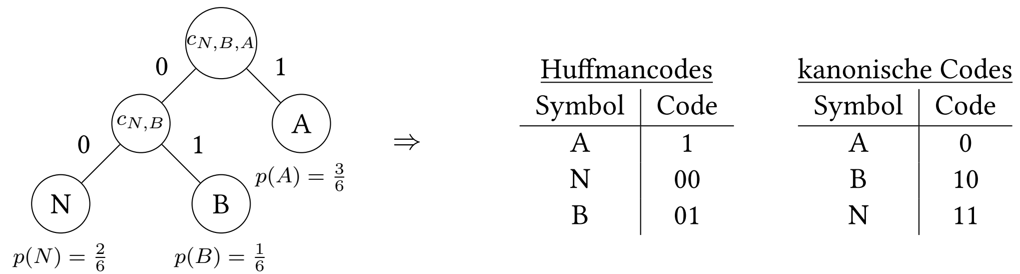
2. Erstelle eine aufsteigend nach $p$ sortierte Liste $L$ der Symbole, d. h. im Beispiel aus Abbildung 2.9 wäre $L$ = (B, N, A). Die einzelnen Symbole entsprechen im entstehenden Codebaum den Blättern.

3. Fasse die ersten beiden Elemente $c_i, c_j$ aus $L$ zu einem neuen Element $c_{i,j}$ zusammen und füge dieses mit $p(c_{i,j}) = p(c_i) + p(c_j)$ wieder entsprechend der Sortierung nach $p$ in $L$ ein. Im Codebaum entsprechen diesen zusammenfassenden Elemente den inneren Knoten.

4. Wiederhole die Schritte 2 und 3, bis nur noch ein Element in $L$ enthalten ist – dieses Element repräsentiert dann die Wurzel des Codebaums.

5. Für jedes Symbol $c$ kann nun ein Code bestimmt werden, indem für jede Kante auf dem Pfad von der Wurzel zum Blatt $c$ ein `0`-Bit bei einer Kante nach links bzw. ein `1`-Bit bei einer Kante nach rechts gesetzt wird.

Abbildung 2.9 zeigt den Codebaum und die resultierenden Codetabellen für die Eingabe "BANANA". Mit dieser Codetabelle kann die Eingabe "BANANA" als Bitfolge `011001001` codiert werden – um diese Bitfolge wieder decodieren zu können, muss aber neben der Bitfolge auch die verwendete Codetabelle bekannt sein.




Huffmancodes

| Symbol | Code |
|---|---|
| A | 1 |
| N | 00 |
| B | 01 |

kanonische Codes

| Symbol | Code |
|---|---|
| A | 0 |
| B | 10 |
| N | 11 |

Abbildung 2.9.: Huffman-Codebaum und Codetabellen für die Eingabe ”BANANA”

Die Speicherung des kompletten Codebaumes wäre nicht sehr effizient, weil entweder die Symbole an den Blättern und die Kanten des Baumes oder die Symbole zusammen mit ihren Häufigkeiten gespeichert werden müssten. Um das zu vermeiden, werden die verwendeten Symbole unabhängig vom Codebaum gespeichert (siehe Unterabschnitt 2.4.6) und die Codetabelle vor der Codierung der Eingabe in eine *kanonische* Codetabelle umgewandelt. In einer kanonischen Codetabelle sind die Codes für die Symbole des Alphabets in der Reihenfolge der lexikographisch sortierten Symbole abgelegt, d. h. der Code für das Symbol ”A” steht vor dem Code für das Symbol ”B” in der Codetabelle. Zusätzlich gilt für alle Codes gleicher Länge, dass sie im Wert monoton wachsen, d. h. der Code `1100` kann erst nach dem Code `1011` in der Codetabelle stehen.

Der Vorteil von kanonischen Codetabellen besteht darin, dass bei bekanntem Alphabet nur noch die Längen der Codes gespeichert werden müssen, um die Codetabelle vollständig zu beschreiben. Ein Algorithmus zur Umwandlung einer (Huffman-)Codetabelle in eine kanonische Codetabelle ist in RFC 1951 [4] beschrieben, dabei bleibt die Codelänge für jedes Symbol gleich. In Abbildung 2.9 ist neben der Huffman-Codetabelle auch die kanonische Codetabelle für die Eingabe ”BANANA” gezeigt. Wenn nun bekannt ist, dass die Symbole ”A”, ”B” und ”N” codiert werden, genügt die Angabe der Codelängen $1, 2, 2$, um die Tabelle rekonstruieren zu können.

Tatsächlich gespeichert werden aber nicht die Codelängen selbst, sondern die erste Codelänge und danach Differenzen zwischen aufeinanderfolgenden Codelängen, im Beispiel würde also der Startwert $1$ und die Differenzen $0, +1, 0$ gespeichert werden – die erste Null wäre nicht unbedingt nötig in bzip2, erspart aber Ausnahmebehandlung und ”kostet” nur ein Bit. Der Startwert ist bei bzip2 immer fünf Bits lang, das ist möglich, weil die Codelängen ohnehin auf 20 Bits beschränkt sind. Bei pyFPbzip ist der Startwert ein Byte groß, das beschränkt den Startwert auf 255 Bits, es können zwar theoretisch Codes mit mehr als 255 Bits Länge auftreten, weil auch die Differenz vom Startwert zur ersten Codelänge gespeichert wird, stellt das aber keine Einschränkung dar, da diese Differenz beliebig groß sein kann. Die Differenzen zwischen den Längen von aufeinanderfolgenden Codes werden bei beiden Verfahren folgendermaßen codiert:

- Der nächste Code ist um $n$ Bits länger als der aktuelle: Es wird $n$-mal `10` und abschließend `0` ausgegeben

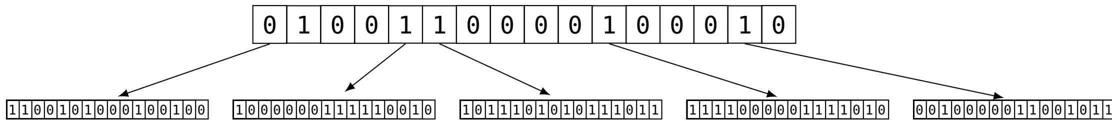



Abbildung 2.10.: Zweistufiges hierarchisches Bitfeld zur Alphabetcodierung in bzip2

- Der nächste Code ist um $n$ Bits kürzer als der aktuelle: Es wird $n$-mal `11` und abschließend `0` ausgegeben
- Der nächste Code ist gleich lang wie der aktuelle: Es wird `0` ausgegeben

Die Codetabelle aus Abbildung 2.9 würde also von pyFPbzip gespeichert als $\underbrace{\texttt{00000001}}_{\text{Startwert}}\underbrace{\texttt{0 10 0 0}}_{\text{Differenzen}}$.

## 2.4.6. Alphabetcodierung

Weil von den Huffmantabellen nur die Codelängen gespeichert werden, muss das verwendete Eingabealphabet an anderer Stelle gespeichert werden. ”Eingabealphabet” bezeichnet dabei die Menge der unterschiedlichen Symbole vor der Move-to-Front-Transformation plus die Symbole RUNA und RUNB aus der Codierung von Nullläufen. Weil sich bzip2 und pyFPbzip an dieser Stelle stark unterscheiden, werden die verwendeten Codierungsverfahren getrennt erläutert.

### ”Sparse Bit Array” bei bzip2

Bei bzip2 sind die Eingabesymbole immer genau ein Byte lang, daher kann es maximal 256 verschiedene Symbole in der Eingabe geben. Codiert wird das Alphabet in einem hierarchischen Bitfeld mit zwei Stufen, ein Beispiel dafür zeigt Abbildung 2.10. Die erste Stufe unterteilt den Bereich der 256 möglichen Symbole in 16 Teile zu je 16 Symbolen. Wenn ein Bit der ersten Stufe Null ist, bedeutet das, dass alle 16 Bits der zweiten Stufe Null wären, d. h. keines der 16 Symbole wird benutzt, dementsprechend werden die 16 Bits der zweiten Stufe erst gar nicht gespeichert. Ist dagegen ein Bit der ersten Stufe Eins, werden die entsprechenden 16 Bits der zweiten Stufe gespeichert (für jedes verwendete Symbol eine `1`, für jedes nicht verwendete eine `0`). Gespeichert wird immer zuerst die erste Stufe und dann in aufsteigender Reihenfolge die Felder der zweiten Stufe. Dieser zweistufige Ansatz ermöglicht vor allem bei ASCII-codierten Textdateien eine deutlich kürzere Alphabetcodierung, weil bei diesen in der Regel nur die Symbole $32_{10}$ bis $127_{10}$ vorkommen.

### Differentielle Alphabetcodierung bei pyFPbzip

Weil die Länge (in Bits) der Eingabesymbole bei pyFPbzip nicht festgelegt ist, muss die Codierung des Eingabealphabets anders aufgebaut werden. Da die Eingabesymbole in der pyFPbzip-Kompressionspipeline als Integers interpretiert werden, ist im Folgenden mit ”Symbol” und

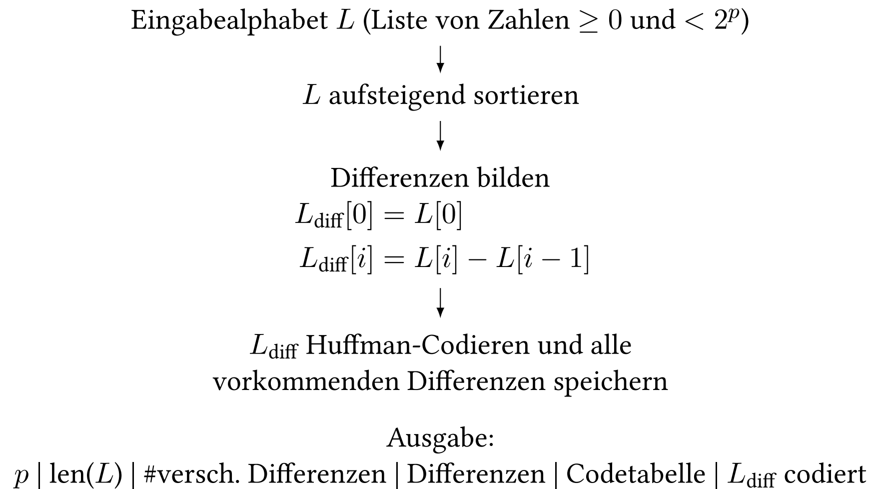



Abbildung 2.11.: Ablauf der differentiellen Alphabetcodierung bei pyFPbzip

"Zahl" dasselbe gemeint, nämlich einfach eine Folge von $p$ Bits, dabei ist $p$ die gewählte Länge der Eingabesymbole. Bei den Versuchen haben sich zwei- und auch mehrstufige hierarchische Bitfelder mit verschiedenen Größen der Stufen als nicht praktikabel gezeigt, vor allem bei langen Eingabesymbolen ($p \geq 20$) war das so codierte Alphabet oft größer als das "roh" gespeicherte Alphabet.

Bei näherer Untersuchung insbesondere der großen Alphabete stellte sich heraus, dass die tatsächlich vorkommenden Symbole relativ gleichmäßig über die Menge der möglichen Symbole (alle Zahlen $\geq 0$ und $< 2^p$) verteilt sind, was die schlechten Ergebnisse mit hierarchischen Bitfeldern erklärt. Die relativ gleichmäßige Verteilung der Symbole legt nahe, die Differenzen zwischen den aufsteigend sortierten Symbolen (die ja ohnehin als Zahlen interpretiert werden) zu betrachten. Diese Differenzen sind betragsmäßig kleiner als die Symbole selbst und es gibt im Vergleich zur Alphabetgröße wegen der Verteilung der Symbole nur wenige verschiedene Differenzen.

Darum wird zur Alphabetcodierung das in Abbildung 2.11 gezeigte Verfahren benutzt, mit einer Ausnahme: Wenn das Alphabet "voll" ist, also alle möglichen $2^p$ Symbole vorkommen, wird nur ein entsprechendes Flag gesetzt und $p$ gespeichert. Dabei werden die Differenzen zwischen den einzelnen Symbolen nicht direkt gespeichert, sondern zuvor mit einem kanonischen Huffman-Code (siehe Unterabschnitt 2.4.5) codiert. Wegen der kanonischen Huffmancodierung müssen neben der Codetabelle und den codierten Differenzen noch die vorkommenden Differenzen gespeichert werden – quasi das "Alphabet der Differenzen". Da es von diesen Differenzen im Vergleich zur Alphabetgröße nur wenige gibt, werden sie "roh" als $p$-Bit-Zahlen gespeichert. Um das alles wieder entpacken zu können, müssen noch $p$, die Größe des Eingabealphabets und die Anzahl der Differenzen gespeichert werden, die brauchen mit jeweils 8 Bytes aber sehr wenig Platz.

### 2.4.7. Unterschiede zwischen bzip2- und pyFPbzip-Kompressionspipeline

Bereits beschrieben wurden die folgenden Unterschiede zwischen der bzip2- und der pyFPbzip-Kompressionspipeline:

- bzip2 ist blockbasiert, die Blockgröße ist dabei $i \cdot 100kB$ mit $i \in \{1, ..., 9\}$, pyFPbzip verarbeitet immer die ganze Eingabe auf einmal.
- Zur Speicherung von Wiederholungszählern in der ersten Stufe wird bei bzip2 immer genau ein Byte verwendet, bei pyFPbzip werden so viele Bits verwendet, wie jedes Eingabesymbol lang ist, mindestens aber acht.
- bzip2 codiert das Eingabealphabet in einem zweistufigen hierarchischen Bitfeld, in pyFPbzip wird ein differentielles Codierungsverfahren verwendet.

Neben diesen Unterschieden gibt es einen weiteren grundlegenden Unterschied zwischen den Kompressionspipelines: bzip2 nutzt pro Block nicht nur eine Codetabelle, sondern mehrere, die iterativ verbessert werden. Für jeden 50 Bytes langen Abschnitt wird dann die beste Codetabelle gewählt; die Reihenfolge, in der die Tabellen genutzt werden, wird am Anfang eines Blocks zusammen mit den Codetabellen gespeichert.

# 3. Implementierung

Der eigentliche Gegenstand der Implementierung, pyFPbzip, besteht nur aus Kompressionspipeline (Abschnitt 2.4) und Vorzeichencodierung (Abschnitt 3.4). Um pyFPbzip sinnvoll evaluieren zu können, waren jedoch noch weitere Programme nötig, die hier zusammen mit der Implementierung von pyFPbzip beschrieben werden, das Zusammenspiel der einzelnen Teile ist in Abbildung 3.1 schematisch dargestellt.

## 3.1. Erzeugung und Hierarchisierung der Datensätze

Um überhaupt Versuchsdaten für die Kompression zu haben, wurden zunächst zweidimensionale Datensätze für mehrere Funktionen auf verschiedenen Gittern mit einem Pythonprogramm erzeugt, die gewählten Funktionen und Gitter werden in Kapitel 4 diskutiert. Da das

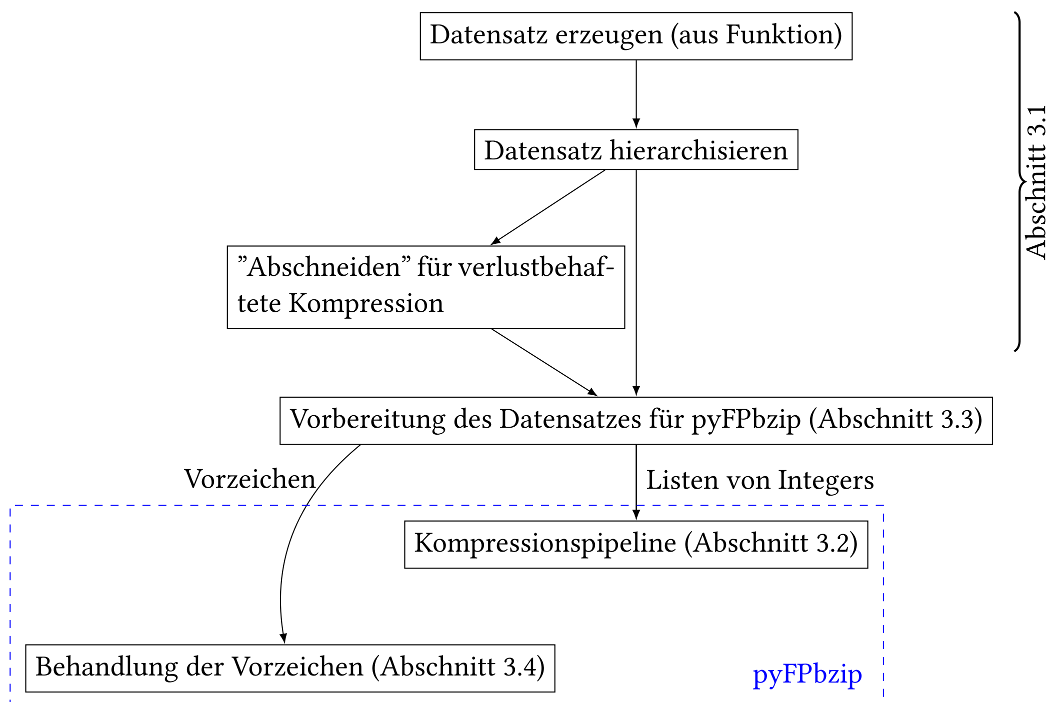



Abbildung 3.1.: Zusammenspiel der einzelnen Programme/Programmteile

Verhalten der Kompressionspipeline für hierarchisierte Daten untersucht werden sollte, wurden die so generierten Datensätze anschließend – ebenfalls mit einem Pythonprogramm – hierarchisiert und das Ergebnis als binäre Datei gespeichert.

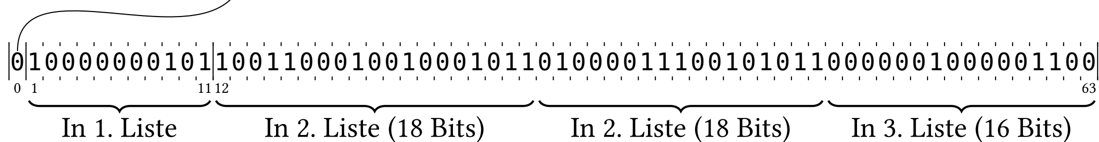
Für die verlustbehaftete Kompression wurde das in Abschnitt 2.2 beschriebene Verfahren zum "Abschneiden" der Überschüsse unter Zuhilfenahme der Arbeit von Fischer et al. [5] in Python implementiert und das Ergebnis (unkomprimiert, nur abgeschnitten) ebenfalls als binäre Datei gespeichert. Die eigentliche Kompression fand dann wie in Abschnitt 3.2 und 3.3 beschrieben statt. Dieser Implementierung für das Abschneiden fehlt allerdings die in Abschnitt 2.2 beschriebene Korrektur der Überschüsse, um akkumulierte Fehler beim Dehierarchisieren zu vermeiden. Der Grund dafür ist, dass das Problem der akkumulierten Fehler erst entdeckt wurde, als die Experimente bereits größtenteils abgeschlossen waren, Fischer et al. erwähnen die Problematik nicht.

## 3.2. Kompressionspipeline

Die eigentliche Kompressionspipeline wurde zunächst komplett in Python implementiert, dabei sind die einzelnen Stufen der Pipeline komplett unabhängig voneinander. Diese Trennung ist nicht unbedingt notwendig, so könnten z. B. die Move-to-Front-Transformation und die Codierung der Nullläufe sinnvoll zu einem Schritt zusammengefasst werden, davon wurde hier abgesehen, um die Implementierung so flexibel wie möglich zu halten. Bei den Experimenten zeigte sich, dass die Move-to-Front-Transformation, implementiert in Python, relativ viel Zeit in Anspruch nimmt. Darum wurde diese Transformation in C++ implementiert und mittels des `ctypes`-Moduls in die Kompressionspipeline eingebunden, die Move-to-Front-Transformation konnte dadurch selbst bei kleinen Datensätzen um das 30fache, häufig auch deutlich mehr, beschleunigt werden.

Als Eingabe erwartet die Kompressionspipeline eine Liste nicht-negativer Ganzzahlen, d. h. die Eingabesymbole werden innerhalb der Pipeline, unabhängig von ihrer Bedeutung im Datensatz, einfach als Ganzzahlen behandelt. Diese Darstellung wurde gewählt, weil Ganzzahlen in Python nicht durch eine feste Bitlänge beschränkt sind[1] und die Implementierung dadurch sehr flexibel ist, was die Größe der Eingabesymbole angeht. Falls in der Eingabe Zahlen größer als $2^{64} - 1$ vorkommen, kann jedoch die beschriebene C++-Implementierung der Move-to-Front-Transformation nicht mehr verwendet werden, eine entsprechende Fallback-Funktion ist zwar implementiert, wurde in den Experimenten zu dieser Arbeit aber nie gebraucht. Die Verwendung von Ganzzahlen als Eingabesymbole erlaubt es insbesondere, jeden beliebigen Datentyp mit der implementierten Pipeline zu komprimieren, wenn er passend vorbereitet (also in eine Folge von Ganzzahlen umgewandelt) wird. Die Umwandlung von numerischen Datensätzen in Listen von nicht-negativen Ganzzahlen ist in Abschnitt 3.3 beschrieben.

[1] Verwendet wurde Python3, aus der Dokumentation, Abschnitt 4.4: *"Integers have unlimited precision."* Die einzige Beschränkung ist die Speichergröße der Maschine, auf der das Programm läuft.




In die Vorzeichenliste wird `0` eingefügt.
In Liste 1 wird $\texttt{10000000101}_2$, also die Zahl $1029_{10}$ eingefügt.
In Liste 2 werden $\texttt{100110001001000101}_2 = 156229_{10}$ und $\texttt{101000011100101011}_2 = 165675_{10}$ eingefügt.
In Liste 3 wird $\texttt{0000001000001100}_2 = 524_{10}$ eingefügt.

Abbildung 3.2.: Beispiel für die Aufteilung einer IEEE 754-Gleitkommazahl auf drei Listen plus Vorzeichen

## 3.3. Vorbereitung der Daten für die Kompressionspipeline

Wie in Abschnitt 3.2 beschrieben, erwartet die Kompressionspipeline als Eingabe eine Liste von nicht-negativen Ganzzahlen, d. h. die binär gespeicherten, hierarchisierten Datensätze müssen zunächst in dieses Format gebracht werden. Nicht-negative Ganzzahlen werden in der Programmierung häufig als *unsigned integers* bezeichnet, im Folgenden werden sie einfach *Integers* genannt.

Zur Vorbereitung der Datensätze für die Kompressionspipeline werden die 64 Bits langen IEEE 754-Repräsentationen der Überschüsse in mehrere kürzere Teile zerlegt und diese Teile unabhängig voneinander weiterverarbeitet.

Der Grund für das Zerlegen ist, dass die Hierarchisierung zwar zu kleineren Absolutwerten führt, die Größe $|\Gamma|$ des Eingabealphabets[2] $\Gamma$ aber nicht unbedingt kleiner wird. Wenn nun jede IEEE 754-Repräsentation im hierarchisierten Datensatz als ein einzelnes Eingabesymbol der Länge 64 Bits betrachtet wird, muss die Kompressionspipeline ungefähr $|\Gamma|$ verschiedene Symbole verarbeiten. In diesem Fall ist die Kompressionspipeline beinahe äquivalent zu einer einfachen Huffmancodierung der Eingabe (nur deutlich langsamer). Weil zusätzlich zu den codierten Daten und der Codetabelle auch das Eingabealphabet $\Gamma$ gespeichert werden muss, um die Daten wieder decodieren zu können, sind die so erreichbaren Kompressionsraten relativ schlecht, häufig wird die Ausgabe sogar größer als die Eingabe sein. Durch das Zerlegen der Überschüsse entstehen zwar mehr Symbole, die codiert werden müssen, diese wiederholen sich jedoch öfter und können so in vielen Fällen durch die Entropiecodierung effizient komprimiert werden. Insbesondere bei den Exponenten ist dies zu erwarten, weil durch die Hierarchisierung auf jedem Level Überschüsse ähnlicher Größenordnung stehen, die dementsprechend ähnliche oder sogar gleiche Exponenten haben.

[2] Das Eingabealphabet ist die Menge aller paarweise verschiedenen Symbole in der Eingabe – wenn jeder Überschuss als ein einzelnes Eingabesymbol interpretiert wird, sind das alle paarweise verschiedenen Überschusswerte.

Abbildung 3.2 zeigt ein Beispiel, in dem die IEEE 754-Repräsentation eines Überschusswerts in vier Teile zerlegt wird: Das Vorzeichen (1 Bit), den Exponenten (11 Bits) und drei Mantissenteile (18, 18 und 16 Bits). Diese Teile, die ja nichts anderes als Bitfolgen sind, werden dann – interpretiert als Integers – in verschiedene Listen eingefügt, danach wird die nächste IEEE 754-Repräsentation zerlegt, die Teile in die jeweiligen Listen eingefügt usw., bis die komplette Eingabe gelesen, zerteilt und auf Integer-Listen verteilt ist. Diese Listen werden dann unabhängig voneinander durch die Kompressionspipeline verarbeitet[3]. Nach dem Erstellen der Listen wäre es möglich, die Verarbeitung zu parallelisieren, die Implementierung zu dieser Arbeit nutzt diese Möglichkeit jedoch nicht.

Implementiert wurde die beschriebene Vorbereitung der Daten in Python und – um die Experimente schneller durchführen zu können – C++. Ein ausführliches Beispiel zur Vorbereitung eines Datensatzes ist in Anhang A beschrieben.

## 3.4. Behandlung der Vorzeichen

Bei der Untersuchung der Überschüsse zeigte sich, dass die Vorzeichen von aufeinanderfolgenden Überschüssen häufig gleich sind, darum werden die Vorzeichen gesondert durch eine Lauflängencodierung codiert. Verwendet wird dabei ein Schema, dass dem in Unterabschnitt 2.4.4 beschriebenen sehr ähnlich ist: Für Läufe von `1`-Bits werden die Bitfolgen `10` als RUNA und `11` als RUNB verwendet, für Nullläufe die Bitfolgen `00` und `01`. Im schlechtesten Fall (das Vorzeichen wechselt bei jedem Überschuss) würde das die Anzahl der zur Speicherung der Vorzeichen nötigen Bits verdoppeln. Das kann verhindert werden, indem beim Komprimieren verglichen wird, ob die Codierung der Vorzeichen oder die ”rohen” Vorzeichen mehr Bits benötigen und dann die bessere Variante gewählt wird – die hier vorgestellte Implementierung macht diesen Vergleich nicht und speichert immer die codierten Vorzeichen.

[3]Die Liste der Vorzeichen wird gesondert behandelt, siehe Abschnitt 3.4

# 4. Numerische Ergebnisse

## 4.1. Testdaten & -parameter

Als Testdatensätze wurden mehrere Funktionen auf einem zweidimensionalen Gitter über $[0,1]^2$ vorberechnet und hierarchisiert. Der Vorteil gegenüber Datensätzen aus Simulationen und ähnlichem, wie sie z. B. von Fischer et al. [5] zur Evaluation verwendet werden, ist, dass die Testbedingungen auf diese Weise sehr kontrolliert sind und damit auch Vergleiche zwischen den einzelnen Testfällen sinnvoll möglich sind. Nachteilig ist dabei, dass die "künstlichen" Datensätze möglicherweise zu wenig aussagekräftigen Ergebnissen führen, wenn Funktionen verwendet werden, die dem verwendeten Verfahren "besonders gut liegen". Um das zu vermeiden, wurden zur Evaluation verschiedene Varianten von Sinusprodukten und -summen verwendet, da diese relativ "schlecht" linear interpoliert werden können – bei Funktionen, die sich "gut" linear interpolieren lassen, entstehen durch die Hierarchisierung viele Nullen, die sich sehr gut komprimieren lassen.

Die verwendeten Funktionen sind:

$$
\begin{aligned}
g_1(x,y) &:= \sum_{k_1=1}^{3}\sum_{k_2=1}^{3} \sin(k_1\pi x)\sin(k_2\pi y) && \text{Kurz: } \sum_{1}^{3} \sin(k_1\pi x)\sin(k_2\pi y)\\
g_2(x,y) &:= \sum_{k_1=1}^{5}\sum_{k_2=1}^{5} \sin(k_1\pi x)\sin(k_2\pi y) && \text{Kurz: } \sum_{1}^{5} \sin(k_1\pi x)\sin(k_2\pi y)\\
g_3(x,y) &:= \sin(10\pi x)\sin(10\pi y)\\
g_4(x,y) &:= \sin(20\pi x)\sin(20\pi y)\\
g_5(x,y) &:= \sin(5\pi x)\sin(5\pi y)\\
g_6(x,y) &:= \sin(7\pi x)\sin(3\pi y)\\
g_7(x,y) &:= \sin(\pi x^3)\sin(y)\\
g_8(x,y) &:= \sin(\pi x)\sin(\pi y)\\
g_9(x,y) &:= \sin(\pi x)\sin(\pi y) + 2y
\end{aligned}
$$

Zunächst wurden jeweils die Funktionswerte $g_i(p)$ auf dem regulären Gitter $G_{11}^2$ (Größe $(2^{11}+1)\times(2^{11}+1)$ Punkte) berechnet und hierarchisiert. Anschließend wurden die so erhaltenen Überschusswerte mit verschiedenen Aufteilungen der 64-Bit-IEEE 754-Darstellung jeweils

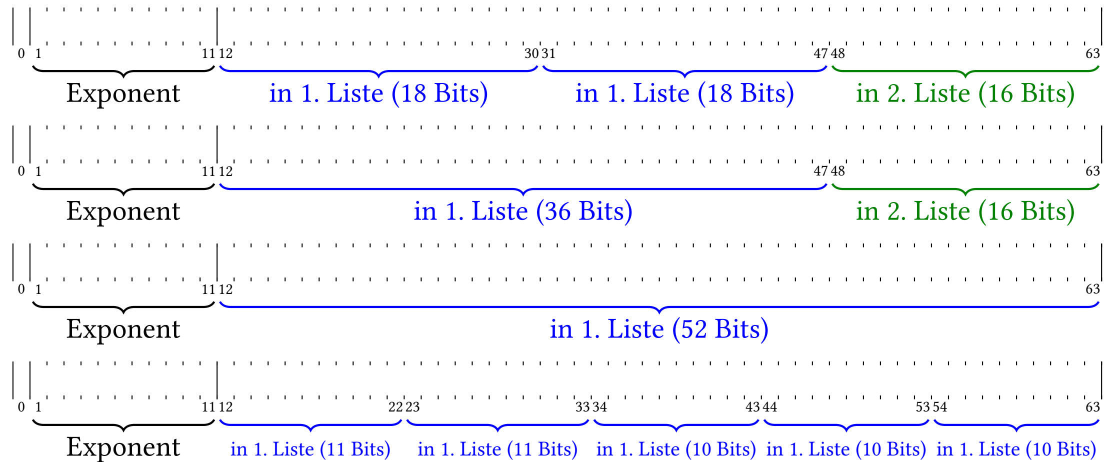



Abbildung 4.1.: Die verwendeten Aufteilungen der 52 Mantissenbits auf Listen

verlustfrei und verlustbehaftet komprimiert. Zum Vergleich wurden die hierarchisierten Datensätze, gespeichert als Binärdateien, mit bzip2 und DEFLATE64[1] bei maximaler Blockgröße komprimiert. Da die DEFLATE64-Ergebnisse in allen Fällen schlechter als die bzip2-Ergebnisse waren, werden sie im Folgenden nicht weiter betrachtet.

Beim Aufteilen der IEEE 754-Repräsentationen der Überschüsse auf Integer-Listen wurden die Vorzeichen gesondert behandelt (Kapitel 3); zudem bildeten die Exponenten immer eine der Integer-Listen. Die Verarbeitung der Exponenten in einer "eigenen" Liste ist sinnvoll, weil durch die Hierarchisierung viele Überschüsse gleicher Größenordnung, also mit gleichen oder ähnlichen Exponenten, entstehen. Tatsächlich zeigte sich in den Experimenten, dass oft nur sehr wenige ($< 20$) verschiedene Exponenten auftreten, die dann entsprechend effizient komprimiert werden konnten. Übrig sind dann noch die 52 Bits der Mantisse, die verschiedenen Aufteilungen, die verwendet wurden, zeigt Abbildung 4.1.

Bei der Wahl der Aufteilungen und Listen muss beachtet werden, dass für jede Liste eine Codetabelle und das Eingabealphabet bezüglich dieser Liste gespeichert werden muss, d. h. mehr Listen führen zwar dazu, dass die einzelnen Symbole effizienter codiert werden können, weil das Eingabealphabet kleiner ist und damit kürzere Codes verwendet werden können, zugleich wächst aber auch der Speicherbedarf für Alphabete und Codetabellen.

Um das Verhalten von pyFPbzip für andere Datensatzgrößen abschätzen zu können, wurden die gleichen Schritte mit den gleichen Funktionen auch auf dem Gitter $G_{10}^{2}$ ausgeführt.

[1]DELFATE64 ist eine Erweiterung des DEFLATE-Algorithmus' (beschrieben in RFC 1951 [4]) und einer der Kompressionsalgorithmen, die für das zip-Containerformat verwendet werden.

```
1 >>> from math import sin, pi
2 >>> sin(pi)*sin(pi)
3 1.4997597826618576e-32   # Wäre bei exakter Arithmetik Null, g_8 wird also in (1, 1) "falsch" berechnet
4 >>> sin(pi)*sin(pi)+2
5 2.0   # Stimmt mit dem exakten Ergebnis überein, g_9(1, 1) wird also korrekt berechnet
```

Listing 4.1: Gleitkommaarithmetik in Python (aus einem interaktiven Python-Interpreter)

## 4.2. Ergebnisse

In Tabelle 4.1 und Tabelle 4.2 sind die Ergebnisse der verlustfreien bzw. verlustbehafteten Kompression für Datensätze auf $G_{11}^2$ zusammengefasst, in Tabelle 4.3 und Tabelle 4.4 dasselbe für Datensätze auf $G_{10}^2$. In den Doppelspalten steht für jede Funktion links die komprimierte Größe *ohne* Codierung des Alphabets und rechts die Größe des codierten Alphabets. Fett hervorgehoben ist in jeder Zeile das Ergebnis, bei dem die Summe aus Datencodierung und Alphabetcodierung am kleinsten ist. Zusätzlich sind zu diesem besten Ergebnis die Kompressionsrate (Eingabegröße durch Ausgabegröße), die ungefähre Rechenzeit (CPU-Zeit, nur Kompressionspipeline) und der Anteil der Burrows-Wheeler-Transformation an der Rechenzeit angegeben. Die Rechenzeit ist nicht zum Vergleich mit anderen Verfahren geeignet, weil bei der Implementierung viele Optimierungsmöglichkeiten zugunsten höherer Flexibilität nicht genutzt wurden – insbesondere bei der Burrows-Wheeler-Transformation, darum ist deren Anteil an der Rechenzeit mit angegeben. Als Vergleichsgröße ist außerdem die Kompressionsrate angegeben, die bzip2 mit maximaler Blockgröße auf derselben Eingabe erzielt.

Die deutlichen Unterschiede – besonders bei der verlustbehafteten Kompression – zwischen den Funktionen $g_8$ und $g_9$ sind eine Folge der Gleitkommaarithmetik: Bei exakter Arithmetik sind die Überschüsse für $g_8$ und $g_9$ außer auf dem nullten Level (also den vier Eckpunkten) gleich, weil die Differenz von $+2y$ in einem Schritt exakt linear interpoliert werden kann. Darum wäre anzunehmen, dass sich die Datensätze beider Funktionen sehr ähnlich komprimieren lassen. Mit dem `math`-Modul von Python erhält man aber schon vor der Hierarchisierung Fehler durch Gleitkommaarithmetik, wie Listing 4.1 zeigt. Weil der Funktionswert von $g_8$ an einer Ecke des Gitters nicht genau stimmt, werden Überschüsse auf den feineren Levels ebenfalls verfälscht, insbesondere auch die, die auf dem Rand des Gitters liegen und bei exakter Rechnung gleich Null wären. Deshalb enthält der hierarchisierte Datensatz zu $g_9$ viel mehr (korrekte) Nullen als der hierarchisierte Datensatz zu $g_8$ und ist dadurch sehr viel besser komprimierbar.

| Mantissenteile | 11, 11, 10, 10, 10 | | 18, 18, 16 | | 36, 16 | | 52 | | Beste Rate | bzip2 Rate | Dauer Beste | Anteil BWT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Daten | Alph. | Daten | Alph. | Daten | Alph. | Daten | Alph. | | | | |
| $\sum_1^3 \sin(k_1\pi x)\sin(k_2\pi y)$ | 10946 | 0.2 | **10755** | **34.8** | 11532 | 3132.3 | 11490 | 3568.4 | 3.113 | 1.758 | 11m 41s | 80% |
| $\sum_1^5 \sin(k_1\pi x)\sin(k_2\pi y)$ | 13078 | 0.2 | **12998** | **34.9** | 11999 | 4937.0 | 12249 | 5879.2 | 2.577 | 1.606 | 7m 23s | 52% |
| $\sin(10\pi x)\sin(10\pi y)$ | **6878** | **0.2** | 8103 | 34.3 | 6650 | 1028.3 | 6701 | 1334.9 | 4.883 | 2.011 | 4h 0m | 98% |
| $\sin(20\pi x)\sin(20\pi y)$ | **3702** | **0.2** | 4904 | 35.9 | 4443 | 299.5 | 4333 | 564.6 | 9.072 | 2.399 | 11h 32m | 99% |
| $\sin(5\pi x)\sin(5\pi y)$ | **10742** | **0.2** | 11102 | 32.9 | 9875 | 2009.6 | 9854 | 2349.7 | 3.127 | 1.903 | 30m 11s | 87% |
| $\sin(7\pi x)\sin(3\pi y)$ | **10853** | **0.2** | 11275 | 33.7 | 10067 | 2887.8 | 10042 | 3308.8 | 3.095 | 1.911 | 36m 54s | 88% |
| $\sin(\pi x^3)\sin(y)$ | 6950 | 0.2 | **5910** | **15.5** | 8448 | 1331.6 | 8442 | 1419.7 | 5.668 | 2.276 | 16m 37s | 80% |
| $\sin(\pi x)\sin(\pi y)$ | 6195 | 0.2 | **4528** | **14.5** | 6975 | 547.6 | 6972 | 613.8 | 7.394 | 2.601 | 39m 32s | 93% |
| $\sin(\pi x)\sin(\pi y) + 2y$ | 5166 | 0.2 | **4428** | **9.6** | 6610 | 375.9 | 6609 | 396.0 | 7.569 | 2.996 | 36m 42s | 93% |

Tabelle 4.1.: Ergebnisse verlustfreie Kompression (in KB, Eingabegröße 33587 KB)

| Mantissenteile | 11, 11, 10, 10, 10 | | 18, 18, 16 | | 36, 16 | | 52 | | Beste Rate | bzip2 Rate | Dauer Beste | Anteil BWT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Daten | Alph. | Daten | Alph. | Daten | Alph. | Daten | Alph. | | | | |
| $\sum_1^3 \sin(k_1\pi x)\sin(k_2\pi y)$ | 10385 | 0.2 | **10010** | **24.4** | 11358 | 2402.0 | 11345 | 2569.8 | 3.347 | 2.141 | 11m 57s | 67% |
| $\sum_1^5 \sin(k_1\pi x)\sin(k_2\pi y)$ | **11574** | **0.2** | 11868 | 32.4 | 11899 | 3282.2 | 11869 | 3638.4 | 2.902 | 1.950 | 4m 4s | 44% |
| $\sin(10\pi x)\sin(10\pi y)$ | **4227** | **0.2** | 4493 | 20.3 | 3602 | 653.7 | 3564 | 800.1 | 7.945 | 3.249 | 29m 9s | 86% |
| $\sin(20\pi x)\sin(20\pi y)$ | **1594** | **0.2** | 1718 | 14.4 | 1487 | 226.8 | 1410 | 295.8 | 21.07 | 6.511 | 38m 53s | 89% |
| $\sin(5\pi x)\sin(5\pi y)$ | **9346** | **0.2** | 9683 | 23.9 | 8680 | 1575.8 | 8675 | 1760.9 | 3.594 | 2.544 | 21m 21s | 81% |
| $\sin(7\pi x)\sin(3\pi y)$ | **8540** | **0.2** | 8663 | 24.1 | 7908 | 1709.0 | 7898 | 1897.7 | 3.933 | 2.675 | 20m 44s | 83% |
| $\sin(\pi x^3)\sin(y)$ | **93** | **0.0** | 93 | 0.0 | 93 | 0.0 | 93 | 0.0 | 359.8 | 400.6 | 15m 44s | 88% |
| $\sin(\pi x)\sin(\pi y)$ | 5696 | 0.2 | **4235** | **10.5** | 7069 | 448.8 | 7069 | 479.1 | 7.911 | 3.837 | 11m 0s | 74% |
| $\sin(\pi x)\sin(\pi y) + 2y$ | **3** | **0.0** | 3 | 0.0 | 3 | 0.0 | 3 | 0.0 | 10536 | 568.6 | 13m 51s | 87% |

Tabelle 4.2.: Ergebnisse verlustbehaftete Kompression (in KB, Eingabegröße 33587 KB)

| Mantissenteile | 11, 11, 10, 10, 10 | | 18, 18, 16 | | 36, 16 | | 52 | | Beste Rate | bzip2 Rate | Dauer Beste | Anteil BWT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Daten | Alph. | Daten | Alph. | Daten | Alph. | Daten | Alph. | | | | |
| $\sum_1^3 \sin(k_1 \pi x) \sin(k_2 \pi y)$ | **3465** | **0.2** | 3500 | 29.4 | 2752 | 1830.7 | 2716 | 2168.9 | 2.425 | 1.567 | 5m 42s | 79% |
| $\sum_1^5 \sin(k_1 \pi x) \sin(k_2 \pi y)$ | **3648** | **0.2** | 3955 | 34.9 | 2811 | 2327.6 | 2723 | 3011.9 | 2.304 | 1.444 | 5m 7s | 79% |
| $\sin(10\pi x) \sin(10\pi y)$ | **1656** | **0.2** | 1906 | 31.2 | 1641 | 277.8 | 1671 | 431.5 | 5.074 | 2.010 | 1h 0m | 98% |
| $\sin(20\pi x) \sin(20\pi y)$ | 1316 | 0.2 | **1255** | **20.7** | 1212 | 67.7 | 1226 | 172.6 | 6.587 | 2.570 | 2h 5m | 99% |
| $\sin(5\pi x) \sin(5\pi y)$ | **2794** | **0.2** | 2995 | 27.7 | 2342 | 872.7 | 2316 | 1127.4 | 3.008 | 1.809 | 5m 57s | 81% |
| $\sin(7\pi x) \sin(3\pi y)$ | **2892** | **0.2** | 3154 | 33.0 | 2364 | 1232.4 | 2334 | 1562.5 | 2.906 | 1.782 | 5m 41s | 81% |
| $\sin(\pi x^3) \sin(y)$ | 2490 | 0.2 | **2376** | **15.8** | 2591 | 988.9 | 2586 | 1080.1 | 3.514 | 1.898 | 4m 12s | 75% |
| $\sin(\pi x) \sin(\pi y)$ | 2016 | 0.2 | **1847** | **13.5** | 1976 | 406.3 | 1974 | 458.9 | 4.517 | 2.281 | 11m 41s | 93% |
| $\sin(\pi x) \sin(\pi y) + 2y$ | 1983 | 0.2 | **1697** | **9.4** | 2081 | 316.7 | 2080 | 337.5 | 4.926 | 2.454 | 10m 30s | 92% |

Tabelle 4.3.: Ergebnisse verlustfreie Kompression (in KB, Eingabegröße 8405 KB)

| Mantissenteile | 11, 11, 10, 10, 10 | | 18, 18, 16 | | 36, 16 | | 52 | | Beste Rate | bzip2 Rate | Dauer Beste | Anteil BWT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Daten | Alph. | Daten | Alph. | Daten | Alph. | Daten | Alph. | | | | |
| $\sum_1^3 \sin(k_1 \pi x) \sin(k_2 \pi y)$ | **3180** | **0.2** | 3305 | 23.7 | 2730 | 1502.6 | 2716 | 1695.9 | 2.643 | 1.919 | 6m 14s | 78% |
| $\sum_1^5 \sin(k_1 \pi x) \sin(k_2 \pi y)$ | **3490** | **0.2** | 3619 | 27.6 | 2762 | 1804.6 | 2732 | 2129.6 | 2.408 | 1.786 | 4m 21s | 77% |
| $\sin(10\pi x) \sin(10\pi y)$ | **986** | **0.2** | 1093 | 19.1 | 915 | 233.9 | 902 | 305.2 | 8.525 | 4.175 | 6m 32s | 82% |
| $\sin(20\pi x) \sin(20\pi y)$ | 505 | 0.2 | 496 | 10.9 | 464 | 56.1 | **420** | **78.2** | 16.87 | 9.842 | 4m 9s | 89% |
| $\sin(5\pi x) \sin(5\pi y)$ | **2436** | **0.2** | 2600 | 22.1 | 2043 | 701.8 | 2031 | 874.2 | 3.451 | 2.770 | 6m 7s | 80% |
| $\sin(7\pi x) \sin(3\pi y)$ | **2383** | **0.2** | 2454 | 24.3 | 1881 | 916.1 | 1871 | 1107.1 | 3.527 | 2.790 | 5m 32s | 80% |
| $\sin(\pi x^3) \sin(y)$ | **38** | **0.0** | 38 | 0.0 | 38 | 0.0 | 38 | 0.0 | 223.7 | 228.0 | 59s | 53% |
| $\sin(\pi x) \sin(\pi y)$ | 1918 | 0.2 | **1746** | **10.2** | 1942 | 353.4 | 1942 | 386.8 | 4.787 | 3.429 | 2m 29s | 67% |
| $\sin(\pi x) \sin(\pi y) + 2y$ | **2** | **0.0** | 2 | 0.0 | 2 | 0.0 | 2 | 0.0 | 4596 | 381.2 | 55s | 50% |

Tabelle 4.4.: Ergebnisse verlustbehaftete Kompression (in KB, Eingabegröße 8405 KB)

### 4.2.1. Verlustfreie Kompression

Beim Betrachten der Tabellen 4.1 und 4.3 fällt schnell auf, dass die besten Ergebnisse bei "feinerer" Aufteilung der Mantissenbits erreicht wurden. Das kann zum einen damit erklärt werden, dass die Codierung der Eingabealphabete viel weniger Bits benötigt, weil kürzere Eingabesymbole zu kleineren Alphabeten führen. Zum anderen können in vielen Fällen auch die Daten selbst mit weniger Bits codiert werden, weil die kürzeren Symbole zu mehr Wiederholungen von Symbolen und damit zu einer "besseren" Entropiecodierung führen. Dabei spielt ebenfalls eine Rolle, dass bei einem kürzeren Eingabealphabet auch die Codetabelle kürzer ist. Weitere Experimente mit noch feinerer Aufteilung der Mantissen waren auf den Gittern $G^2_{10}$ und $G^2_{11}$ nicht möglich, weil die Laufzeit der Burrows-Wheeler-Transformation zu groß wurde. Entsprechende "schnelle" Versuche auf $G^2_9$ zeigten, dass eine möglichst gleichmäßige Aufteilung der Mantissen – während Vorzeichen und Exponenten gesondert behandelt werden – zu guten Kompressionsraten führt, wenn die Teile nicht zu klein werden. Dabei ist "zu klein" vor allem von der Größe des Datensatzes abhängig, der komprimiert werden soll, für größere Datensätze "lohnen" sich etwas längere Symbole, weil bei entsprechend großer Eingabe auch längere Symbole oft vorkommen. Ein weiterer Nachteil von längeren Eingabesymbolen und daraus resultierend längeren Eingabealphabeten ist, dass die Laufzeit der Move-to-Front-Transformation ungefähr quadratisch mit der Alphabetgröße wächst.

Weiter ist erkennbar, dass der benötige Platz zur Alphabetcodierung bei kurzen Eingabesymbolen kaum von der Größe des zu komprimierenden Datensatzes abhängt, bei längeren Eingabesymbolen wächst der benötigte Platz zur Alphabetcodierung etwa mit der Wurzel der Eingabegröße. Der zur Datencodierung benötigte Platz ist dagegen unabhängig von der Länge der Eingabesymbole ungefähr linear abhängig von der Größe des zu komprimierenden Datensatzes. Beides zusammen führt zu der Vermutung, dass sich lange Eingabesymbole ($>$ 20 Bits) auch für sehr große Datensätze nicht "lohnen", d. h. dass auch für große Datensätze gilt, dass "viele kurze" Symbole eine bessere Kompression ermöglichen als "wenige lange" Symbole.

Zwischen den verschiedenen Funktionen gibt es bezüglich der erreichten Kompressionsraten deutliche Unterschiede, die auf dem feineren Gitter $G^2_{11}$ deutlicher ausfallen als auf dem gröberen Gitter $G^2_{10}$, was darauf schließen lässt, dass die Unterschiede vor allem von den feineren Levels der Hierarchisierung herrühren und sich die Überschüsse dieser Levels besser komprimieren lassen.

### 4.2.2. Verlustbehaftete Kompression

Der bereits in Unterabschnitt 4.2.1 behandelte Zusammenhang zwischen der Größe der Datencodierung, der Länge der Eingabesymbole und der Größe des zu komprimierenden Datensatzes ist auch bei der verlustbehafteten Kompression zu beobachten; die Gründe werden hier nicht erneut diskutiert. Ebenso wächst auch bei der verlustbehafteten Kompression die Größe der Alphabetcodierung ungefähr mit der Wurzel der Eingabegröße.

Interessant ist daher vor allem der Vergleich zwischen den Ergebnissen der verlustfreien und den Ergebnissen der verlustbehafteten Kompression. Durch die Vorbehandlung der Überschüsse, die die verlustbehaftete Kompression von der verlustfreien unterscheidet, sollten vor allem auf den feinen Levels relativ viele signifikante Stellen "abgeschnitten" werden bzw. Überschüsse komplett verschwinden. Da diese feinen Levels einen großen Teil der zu komprimierenden Daten ausmachen, liegt die Erwartung nahe, damit deutlich bessere Kompressionsraten erzielen zu können. Die Ergebnisse zeigen jedoch, dass die verlustbehaftete Vorbereitung der Überschüsse bei den meisten Testdatensätzen kaum einen Vorteil bringt, das normale bzip2-Verfahren "profitiert" oft stärker von der Vorbehandlung als die angepasste Kompressionspipeline. Die auffallend guten Ergebnisse für $g_7$ und $g_9$ lassen sich damit erklären, dass die maximalen Überschüsse auf den Randlevels bei diesen Funktionen sehr groß sind und darum ein relativ großer Teil der Überschüsse durch die Vorbereitung zu Nullen wird.

# 5. Ausblick

Mit der im Rahmen dieser Arbeit entwickelten flexiblen Implementierung der Kompressionspipeline können zwar recht einfach Versuche durchgeführt werden, bei der Performance mussten zum Erreichen dieser Flexibilität aber große Abstriche gemacht werden. Besonders die Burrows-Wheeler-Transformation, die in der vorliegenden Implementierung meist über 80% der Laufzeit ausmacht, kann deutlich effizienter implementiert werden, wenn die Länge der Eingabesymbole bekannt ist. Zudem ist Python als Programmiersprache zwar gut für Experimente geeignet, verglichen mit kompilierten Sprachen wie C ist die Ausführung aber relativ langsam.

Zusätzlich könnte die Eingabe in Blöcken verarbeitet werden, wodurch zwar die Kompressionsrate etwas verschlechtert wird, die Laufzeit aber – je nach Blockgröße – stark reduziert werden kann. Sinnvoll wäre möglicherweise, die hierarchisierten Datensätze nach Levels zu sortieren und je ein oder mehrere Levels als einen Block zu verarbeiten.

Die Codierung der Eingabealphabete ist wegen der gewünschten Flexibilität bewusst sehr generisch entworfen, auch hier wäre eine spezialisierte und damit wahrscheinlich sowohl bei der Laufzeit als auch bei der Kompression deutlich effizientere Lösung möglich, wenn die Größe der Eingabesymbole festgelegt ist. Möglich wäre bei kürzeren Eingabesymbolen z. B. ein hierarchisches Bitfeld, wie es auch bei bzip2 zum Einsatz kommt, für lange Eingabesymbole könnte die differentielle Codierung angepasst werden.

Bei der Codierung der Vorzeichen kann zum einen die bereits in Abschnitt 3.4 beschriebene Verbesserung verwendet werden, um zu verhindern, dass die codierten Vorzeichen mehr Platz belegen als die ”rohen” Bits. Zum anderen wäre es aber auch möglich, die Vorzeichen zusammen mit den Mantissen zu behandeln oder einen ganz anderen Ansatz zur Speicherung der Vorzeichen zu wählen. Eine vielversprechende Möglichkeit dafür wäre, die Vorzeichen zu Gruppen zusammenzufassen und diese Gruppen mit einer Kombination aus LZ77 und Huffmancodierung zu verpacken, weil sich in der Abfolge der Vorzeichen öfter wiederholte Muster fanden.

# Literatur

[1] Hans-Joachim Bungartz und Michael Griebel. „Sparse grids“. In: *Acta Numerica* 13 (Mai 2004), S. 147–269. ISSN: 1474-0508. DOI: 10.1017/S0962492904000182.

[2] Michael Burrows und David Wheeler. *A Block-Sorting Lossless Data Compression Algorithm.* Techn. Ber. Digital SRC Research Report, 1994.

[3] Martin Burtscher und Paruj Ratanaworabhan. „FPC: A High-Speed Compressor for Double-Precision Floating-Point Data“. In: *IEEE Transactions on Computers* 58.1 (2009), S. 18–31. DOI: 10.1109/TC.2008.131.

[4] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3.* RFC 1951. http://www.rfc-editor.org/rfc/rfc1951.txt. Aladdin Enterprises, Mai 1996.

[5] Michael Fischer u. a. *Compression of Floating Point Numbers in Hierarchical Subspace Schemes.* Universität Stuttgart. Apr. 2015.

[6] David A. Huffman. „A method for the construction of minimum-redundancy codes“. In: *Proceedings of the IRE* 40.9 (1952), S. 1098–1101.

[7] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2008* (Aug. 2008), S. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[8] Peter Lindstrom und Martin Isenburg. „Fast and Efficient Compression of Floating-Point Data“. In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (2006), S. 1245–1250. DOI: 10.1109/TVCG.2006.143.

[9] Julian Seward. *bzip2.* 2016. URL: http://bzip.org/index.html.

[10] Paul Wessel. „Compression of large data grids for Internet transmission“. In: *Computers & Geosciences* 29.5 (2003), S. 665–671. ISSN: 0098-3004. DOI: 10.1016/S0098-3004(03)00038-4.

# A. Beispiel: Vorbereitung eines Datensatzes für die Kompressionspipeline

Als Ausgangspunkt für die Vorbereitung eines Datensatzes für die Kompressionspipeline dient ein hierarchisierter Datensatz – in diesem Beispiel ist der Datensatz die Diskretisierung der Funktion $f(x) = -1.3x^2 + 1.3x$ auf dem eindimensionalen Gitter $G_3^1$. Die Überschüsse, die bei der Hierarchisierung entstehen, sind im Speicher als IEEE 754-Gleitkommazahlen dargestellt, d. h. nach der Hierarchisierung hat der Datensatz folgende Repräsentation:

```
0 00000000000 0000000000000000000000000000000000000000000000000000
0 01111111001 0100110011001100110011001100110011001100110011001000
0 01111111011 0100110011001100110011001100110011001100110011001110
0 01111111001 0100110011001100110011001100110011001100110011000000
0 01111111101 0100110011001100110011001100110011001100110011001101
0 01111111001 0100110011001100110011001100110011001100110011000000
0 01111111011 0100110011001100110011001100110011001100110011001110
0 01111111001 0100110011001100110011001100110011001100110010110000
0 00000000000 0000000000000000000000000000000000000000000000000000
```

Diese einzelnen IEEE 754-Gleitkommazahlen werden nun aufgeteilt – für dieses Beispiel in Vorzeichen, Exponent und drei Mantissenteile (18, 18 und 16 Bits) – und auf Listen verteilt:

```
|0|00000000000|000000000000000000|000000000000000000|0000000000000000|
 0 1        11 12              30 31              47 48            63
```
$0_{10} \to L_1$ $\quad 0_{10} \to L_2$ $\quad 0_{10} \to L_2$ $\quad 0_{10} \to L_3$

```
|0|01111111001|010011001100110011|001100110011001100|1100110011001000|
 0 1        11 12              30 31              47 48            63
```
$1017_{10} \to L_1$ $\quad 78643_{10} \to L_2$ $\quad 52428_{10} \to L_2$ $\quad 52424_{10} \to L_3$

```
|0|01111111011|010011001100110011|001100110011001100|1100110011001110|
 0 1        11 12              30 31              47 48            63
```
$1019_{10} \to L_1$ $\quad 78643_{10} \to L_2$ $\quad 52428_{10} \to L_2$ $\quad 52430_{10} \to L_3$

```
|0|01111111001|010011001100110011|001100110011001100|1100110011000000|
 0 1        11 12              30 31              47 48            63
```
$1017_{10} \to L_1$ $\quad 78643_{10} \to L_2$ $\quad 52428_{10} \to L_2$ $\quad 52416_{10} \to L_3$

```
|0|01111111101|010011001100110011|001100110011001100|1100110011001101|
 0 1        11 12              30 31              47 48            63
```
$1021_{10} \to L_1$ $\quad 78643_{10} \to L_2$ $\quad 52428_{10} \to L_2$ $\quad 52429_{10} \to L_3$

# A. Beispiel: Vorbereitung eines Datensatzes für die Kompressionspipeline

```
|0|01111111001|0100110011001100110|01100110011001100|1100110011000000|
 0 1         11 12              30 31             47 48            63
```

$1017_{10} \to L_1$ $78643_{10} \to L_2$ $52428_{10} \to L_2$ $52416_{10} \to L_3$

```
|0|01111111011|0100110011001100110|01100110011001100|1100110011001110|
 0 1         11 12              30 31             47 48            63
```

$1019_{10} \to L_1$ $78643_{10} \to L_2$ $52428_{10} \to L_2$ $52430_{10} \to L_3$

```
|0|01111111001|0100110011001100110|01100110011001100|1100110010110000|
 0 1         11 12              30 31             47 48            63
```

$1017_{10} \to L_1$ $78643_{10} \to L_2$ $52428_{10} \to L_2$ $52400_{10} \to L_3$

```
|0|00000000000|0000000000000000000|00000000000000000|0000000000000000|
 0 1         11 12              30 31             47 48            63
```

$0_{10} \to L_1$ $0_{10} \to L_2$ $0_{10} \to L_2$ $0_{10} \to L_3$

Die Listen, die dann anschließend durch die Kompressionspipeline verarbeitet werden können, sind:

$$
\begin{aligned}
L_0 &= (0, 0, 0, 0, 0, 0, 0, 0, 0) && \text{Vorzeichen} \\
L_1 &= (0, 1017, 1019, 1017, 1021, 1017, 1019, 1017, 0) && \text{Exponenten} \\
L_2 &= (0, 78643, 78643, 78643, 78643, 78643, 78643, 78643, 0) \\
L_3 &= (0, 52428, 52428, 52428, 52428, 52428, 52428, 52428, 0) \\
L_4 &= (0, 52424, 52430, 52416, 52429, 52416, 52430, 52400, 0)
\end{aligned}
$$

**Erklärung**

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.

---

Ort, Datum, Unterschrift