Datenkompression*

Maciej Liśkiewicz[1], Henning Fernau[2]

[1] Institut für Theoretische Informatik, Medizinische Universität zu Lübeck
Wallstr. 40, D-23560 Lübeck
und
[2] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen
liskiewi@tcs.mu-luebeck.de, fernau@informatik.uni-tuebingen.de

*LaTeXund sprachliche Bearbeitung unter Mithilfe von Mark Dokoupil und Thomas Arand

Inhaltsverzeichnis

Vorwort

Fast jeder, der Computer verwendet, benutzt bewusst oder unbewusst Komprimierungsverfahren: bewusst etwa zur Archivierung von Dateien, um deren Speicherbedarf zu senken, unbewusst durch Verwendung von Standarddatenformaten, die vordefinierterweise Datenkompression vorsehen. Wir wollen hier versuchen, die (oft recht mathematischen) Grundlagen und Grundgedanken der dabei verwendeten Algorithmen zu verstehen.

Diese Schrift wurde als Skriptum für eine zweistündige Vorlesung an einer Hochschule mit Semestersystem entworfen. Grundsätzlich sind die einzelnen Kapitel als eine „Doppelstunde" dieser Vorlesung gedacht, wobei kürzere Kapitel weniger als eine Doppelstunde beanspruchen dürften.

Dieses hängt natürlich auch davon ab, wieviel insbesondere mathematische Grundlagen die Studierenden mitbringen. So wird im Kapitel 13 die Fouriertransformation nur in Erinnerung gebracht, aber nicht grundlegend eingeführt. Die Dozierenden werden daher von Fall zu Fall Dinge gründlicher als im Buch darlegen müssen.

Auf Beweise mathematischer Sachverhalte wurde im Buch fast durchgehend verzichtet, um den Umfang nicht zu sprengen, zumal deren Durchführung im Rahmen einer zweistündigen Vorlesung nur auf Kosten der Breite zu erreichen ist.

Unserer Erfahrung nach (nach diesem Skript wurden bislang — nach unseren Kenntnissen — etwa ein halbes Dutzend Vorlesungen an unterschiedlichen Universitäten in Deutschland gehalten) reicht der vorgestellte Stoffumfang gut aus, um 12-14 Vorlesungswochen auszugestalten. Wer weitere Hilfen benötigt, etwa vorbereitete LaTeX-Folien, kann gerne mit den Verfassern Kontakt aufnehmen.

Grundsätzlich ist dies Buch auch zum Selbststudium gedacht und geeignet. Mitzubringen sind, neben Interesse am Thema, eine gewisse mathematische Grundbildung, wie sie im Rahmen jedes Grundstudiums in Informatik und Mathematik, aber auch in jedem naturwissenschaftlichen oder ingenieurswissenschaftlichen Grundstudium erworben wird.

Wer weiter gehende Informationen benötigt, der sei auf das mittlerweile in zweiter erweiterter Auflage erschienene englischsprachige Buch von Sayood [17] verwiesen. Dieses ist auch die wesentliche Grundlage des vorliegenden Skriptums. Insbesondere auf intensive Literaturverweise haben wir in diesem Skipt weitgehend verzichtet und verweisen auf [17].

Darüber hinaus gibt es viele Hinweise zu aktuellen Themen der Informatik, und zu diesen zählt zweifelsohne die Datenkompression, im Internet. Exem-

plarisch verweisen wir an dieser Stelle auf die Adressen 2 und 3 aus unserem Webadressenanhang.

1 Einleitung

Dieses Kapitel soll motivieren, warum wir uns mit dem Thema Datenkompression beschäftigen wollen. Daneben werden wir einige grundsätzliche Begriffe klären.

Unter *Datenkompression* (oder *Datenkomprimierung*; wir verwenden beide Wörter im Folgenden unterschiedslos) verstehen wir ein Paar von Algorithmen, ausführlich auch *Datenkompressionschema* genannt: Der erste Algorithmus —die (eigentliche) *Kompression*— konstruiert für die Eingabe $\mathcal{X}$ eine *Repräsentation* $\mathcal{X}_c$, die (möglichst) weniger Bits als $\mathcal{X}$ braucht, und der zweite Algorithmus —die *Dekompression*— generiert für gegebenes $\mathcal{X}_c$ die *Rekonstruktion* $\mathcal{Y}$. Ein Datenkompressionschema heißt *verlustfrei* (engl.: lossless), wenn $\mathcal{X} = \mathcal{Y}$; sonst nennen wir das Schema *verlustbehaftet* (engl.: lossy).

1.1 Wozu Datenkompression?

Ein natürliches Maß für die Qualität des Komprimierungsschemas ist der Quotient „Anzahl der Bits von $\mathcal{X}_c$“ durch „Anzahl der Bits von $\mathcal{X}$“, den wir *Kompressionsquotienten* (engl.: compression ratio) nennen. Der Kompressionsquotient wird auch in bpb (bit per bit) gemessen.[1] In der Praxis vergleichen wir die Komprimierungsschemata bezüglich des Kompressionsquotienten so, dass bestimmte Eingaben $\mathcal{X}$ betrachtet werden. Zum Beispiel benutzt man acht bestimmte Schwarz-Weiß-Testbilder, *CCITT fax test images* genannt, um verschiedene Methoden der Komprimierung von Bildern zu vergleichen (Benchmarks). Die Tabelle unten zeigt den globalen Kompressionsquotienten für verschiedene verlustfreie Methoden, das heißt wir teilen die Größe aller Bildern durch die gesamte Größe der komprimierten Repräsentationen (mehr über *CCITT fax test* finden Sie z. B. in Webadresse 4).

Jahr	Schema	Kompressionsfaktor
1980	CCITT Group 3, T.4	7,7
1984	CCITT Group 4, T.6	15,5
1988	IBM's Q-Coder	19,0
1991	2-level coding	21,4
1993	JBIG	19,7
1995	TIC	22,3

[1] Der Kehrwert des Kompressionsquotienten wird auch *Kompressionsfaktor* genannt. Leider ist die Verwendung dieser Begriffe in der Literatur uneinheitlich, und dies gilt wohl auch für unser Skript; aus dem Zusammenhang sollte aber stets klar werden, welche Definition gemeint ist.

Ein weiteres Maß für die Qualität ist die durchschnittliche Anzahl von Bits für die Repräsentation eines einzigen Grundbestandteils (z. B. Pixel, Zeichen).

Für die verlustbehaftete Komprimierung braucht man noch ein Maß, um die Qualität der Rekonstruktion $\mathcal{Y}$ zu bewerten. Den Unterschied zwischen $\mathcal{Y}$ und dem Eingabebild $\mathcal{X}$ nennen wir die *Entstellung* oder *Verzerrung* (engl.: distortion) und den Begriff werden wir in Kapitel 7 diskutieren.

Als Motivation für Datenkompression betrachten wir zwei Beispiele:

Beispiel 1.1 Wir zeigen, wieviel Speicherplatz ein Bewegtbild-Video ohne irgendwelche Kompression benötigt. Das Video habe folgende Parameter:

Auflösung:	720×480 Pixel
Farbtiefe:	2 Bytes
Bildwiederholfrequenz:	30 fps (frames per second = Bilder je Sekunde)
Länge:	2 Stunden

Das ergibt:

	720	
×	480	
	345600	Pixels
×	2	Bytes
	691200	Bytes je Bild
×	30	fps
	20736000	Bps (Bytes per second)
×	7200	s
	142383 MB	≈ 139 GB

Das heißt, es werden 219 CDs benötigt, um ein solches Video auf CD-ROMs mit einer Kapazität von 650 MB zu speichern. Eine CD kann also knapp 33 s Video speichern.

Beispiel 1.2 Betrachten wir nun die Zeit und die Geschwindigkeit, die man braucht, um die Daten in unkomprimierter Form zu senden.

1. Faxgerät:
   Seitengröße: $8,5 \times 11$ inch $= 93,5$ inch^2
   Abtasten mit 200 dpi:^{2} $93,5 \times 200^2 = 3740000$ b (bits)

Ein Modemgerät mit einer Übertragungsrate von 14,4 kbps (Kilobit je Sekunde) benötigt damit

$$3740 \text{ kb}/14,4 \text{ kbps} = 4 \text{ min } 20 \text{ s}$$

um eine solche Seite zu senden. Um eine komprimierte Seite zu speichern braucht man etwa 250 kb und das gibt uns eine Übertragungszeit von

$$250 \text{ kb}/14,4 \text{ kb/s} = 17 \text{ s}.$$

2. Video: Wir haben gezeigt, daß für ein Video in der Qualität wie oben in jeder Sekunde 20736000 B = 19,775 MB übertragen werden müssen. Das heißt, um ein Video online zu senden, braucht man eine Verbindung mit einer Bandbreite von mehr als 165 Mbps! Ein Video im MPEG2 Format lässt sich bereits mit 3 Mbps online übertragen.

Ist Datenkompression überhaupt möglich? Diese Frage, gestellt in einer Schrift zu eben diesem Thema, erscheint geradezu ketzerisch. Vestehen wir aber unter Datenkompression (im engeren Sinne) ein Verfahren, das eine bestimmte Repräsentation in eine andere überführt, die stets weniger Bits als die urspüngliche benötigt, so zeigt ein einfaches Abzählargument, dass es ein solches Verfahren *nicht* geben kann. Diese Argumentation wird in Arbeiten zum Thema *Kolmogorov-Komplexität* als sog. *Inkompressibilitätsargument* weitergeführt. Im Übrigen ist das angesprochene Gebiet der Kolmogorov-Komplexität für die Theorie der Datenkompression von daher von eminenter Bedeutung, als es in gewissem Sinne begründet, warum sich bestimmte Bitfolgen *nicht* komprimieren lassen: Die Kolmogorov-Komplexität einer Bitfolge t ist nämlich definiert als die Länge des kürzesten „Programms", welches t erzeugt.[3] Eine Bitfolge heißt dann *unkomprimierbar*, falls sie selbst quasi ihre kürzeste Beschreibung ist.

Trotzdem funktioniert, wie wir alle wissen, Datenkompression in der Praxis hervorragend. Woran liegt dies? Die naiv angewendete Inkompressibilitäts-Überlegung übersieht die folgenden Grundbeobachtungen bei menschlich erzeugten Informationen (um die es sich im Folgenden durchgehend handelt):

- Die einzelnen Zeichen, d. h. die Grundelemente des Alphabets, tauchen nicht mit der selben Wahrscheinlichkeit auf. Die sich daraus ableitende

[2] dpi (dots per inch)= Punkte je Zoll ist eine bei Druckern und Scannern übliches Qualitätsmaß.

[3] Man kann zeigen, dass —bei geeigneter Formalisierung des Begriffes „Programm"— der Begriff der Kolmogorov-Komplexität bis auf eine additive Konstante wohldefiniert ist. Interessierte finden Näheres bei [10].

Idee, „wahrscheinlichere" Zeichen mit weniger Bit als unwahrscheinlichere zu codieren, ist bereits aus dem Morse-Alphabet bekannt und wird im folgenden Abschnitt 1.2 eingehender behandelt.

- Blöcke von Zeichen lassen sich aufgrund der Abhängigkeiten aufeinander folgender Ereignisse geschickter gemeinsam als lose unabhängige Folge einzelner Zeichen verschlüsseln.

- Information ist nicht „zufällig" (im Sinne der Kolmogorov-Komplexität), sondern enthält vielmehr zahllose Regelmäßigkeiten. Eine Grundstrategie von Kompressionsverfahren beruht darauf, solche Regeln zu entdecken. In Kapitel 4 werden wir den einfachsten Umsetzungen dieser Idee, den sog. Wörterbuchtechniken, begegnen. Häufig ist es jedoch schwierig, solche Regeln zu finden. Manchmal ist es hilfreich, die ursprüngliche Bitfolge zunächst geeignet umzuformen.

- Für oder von Menschen erzeugte Information ist nie zusammenhangslos. Vielfach ist es möglich und sinnvoll, verschiedene Gattungen von Information zu unterscheiden.

  Ist bekannt, dass die vorliegende Bitfolge einen ASCII-Text der englischen Sprache darstellt, so suggeriert dies andere Kompressionsverfahren (nämlich solche, die auf Byte-Ebene arbeiten) als wenn die Bitfolge eine zeilenweise Darstellung eines „sinnvollen" Schwarz-Weiß-Bildes ist (dann sind starke Abhängigkeiten von Bit zu Bit zu erwarten, und dies natürlich nicht nur zeilen- sondern auch spaltenweise).

- Schließlich —wie schon angedeutet— kann oft auf eine exakte Rekonstruierbarkeit des Originals verzichtet werden, da gewisse Feinheiten vom menschlichen Auge oder Ohr gar nicht wahrgenommen werden (können). Bei verlustbehafteter Kompression sind Inkompressibilitätsargumente direkt nicht anwendbar.

  Im Übrigen muss eine verlustbehaftete Kompression nicht störend wahrgenommen werden. So können bei Audiodaten, die für das menschliche Ohr gedacht sind, Frequenzen, die gar nicht wahrgenommen werden können, von vornherein vernachlässigt werden.

Dieser Abriss legt nahe, dass unser Thema in der Praxis sehr vielschichtig und wichtig ist. Steigen wir also weiter ein!

1.2 Ein bisschen Informationstheorie

Betrachten wir das folgende Problem: Wie lautet eine sinnvolle Definition für das quantitative Maß der Information?

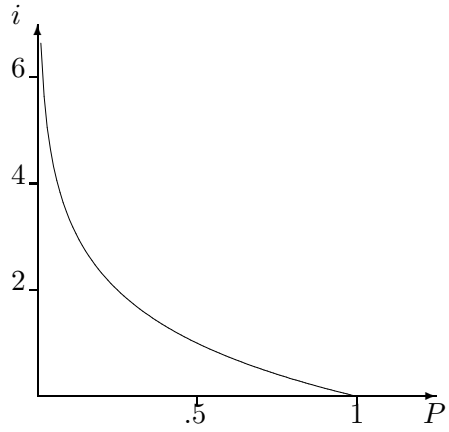


Abbildung 1: Die Selbstinformation in Abhängigkeit von $P(A)$

Elwood Shannon hat dieses Maß wie folgt definiert: Sei A ein Ereignis und $P(A)$ die Wahrscheinlichkeit, dass A eintritt; definiere dann

$$i(A) = \log_2 \frac{1}{P(A)}.$$

Shannon hat $i(A)$ *Selbstinformation* (engl.: self-information) genannt. Wir wollen $i(A)$ auch als *Informationsgehalt* ansprechen. Das Verhalten der Selbstinformation in Abhängigkeit von der Wahrscheinlichkeit P des Ereignisses A ist in Bild 1 grafisch dargestellt.

Intuitiv ist der Begriff klar: Seien A und B zwei Ereignisse (z. B. $A =$ „Der Verbrecher ist kleiner als 2m.“ und $B =$ „Der Verbrecher ist größer als 2m.“) mit den Wahrscheinlichkeiten

$$P(A) = 0,999 \quad \text{und} \quad P(B) = 0,001.$$

Eine Zeugenaussage, A treffe zu, ist keine große Überraschung, und deshalb enthält sie nicht viel Information ($i(A)$ ist sehr klein). Eine Nachricht, daß B wahr ist, hat dagegen einen großen Informationsgehalt.

Sei S eine bestimmte Informationsquelle. Die *Entropie* ist der „mittlere Informationsgehalt pro Nachricht aus S”:

$$H(S) = \sum_{A \in S} P(A) \cdot i(A),$$

wobei wir für $P(A) = 0$ das Produkt $P(A) \cdot i(A)$ als 0 betrachten (dies ist sinnvoll, weil $\lim_{x \to 0} x \cdot \log \frac{1}{x} = 0$). Wir werden meistens S als endliches Quellenalphabet $\Sigma = \{a_1, \ldots, a_n\}$ betrachten. Dann ist $P(a_i)$ die Wahrscheinlichkeit, daß a_i eintritt und der Informationsgehalt $i(a_i) = \log_2 1/P(a_i)$ beschreibt die Anzahl von Bits (deshalb Basis 2 für den Logarithmus), die für

die Kodierung des Zeichens a_i nötig ist (wobei theoretisch Bruchteile von Bits zugelassen werden). Der Einfachheit halber lassen wir im Folgenden die Basisangabe bei Logarithmen weg und treffen die Übereinkunft, 2 sei unsere Standardbasis. Die Entropie

$$H(\Sigma) = \sum_{i=1}^{n} P(a_i) \cdot \log \frac{1}{P(a_i)}$$

ist die mittlere Anzahl von Bits um eine Nachricht aus Σ zu codieren.

	$P(a)$	$P(b)$	$P(c)$	$P(d)$	$P(e)$	H
1.	$0,2$	$0,2$	$0,2$	$0,2$	$0,2$	$2,322$
2.	$0,5$	$0,25$	$0,125$	$0,0625$	$0,0625$	$1,875$
3.	$0,75$	$0,0625$	$0,0625$	$0,0625$	$0,0625$	$1,3$
4.	$0,94$	$0,01$	$0,01$	$0,01$	$0,01$	$0,322$

Tabelle 1: Beispiele für Entropien

Beispiele zur Entropie: Sei $\Sigma = \{a, b, c, d, e\}$ mit $P(a) = P(b) = P(c) = 0,25$ und $P(d) = P(e) = 0,125$. Dann ist

$$H = 3 \cdot 0,25 \cdot \log 4 + 2 \cdot 0,125 \cdot \log 8 = 1,5 + 0,75 = 2,25.$$

Weitere Beispiele enthält Tabelle 1.

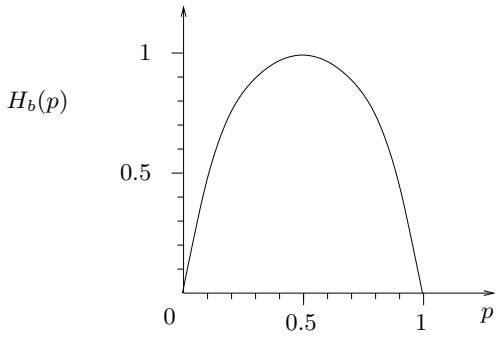


Abbildung 2: Die binäre Entropie-Funktion H_b

Eine interessante Eigenschaft ist, daß die Entropie für ungleichmäßigere Verteilungen kleiner wird. Als weiteres Beispiel, das diese Eigenschaft illustriert, betrachten wir die *binäre Entropie-Funktion*:

$$H_b(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p},$$

das heißt, die Entropie-Funktion für zwei Quellensymbole mit Wahrscheinlichkeiten p und $1-p$. Die Werte für $H_b(p)$, wobei p zwischen 0 und 1 variiert, zeigt das Diagramm 2.

Weitere interessante Eigenschaften der Entropie-Funktion (im Allgemeinen) sind:

1. H ist symmetrisch, das heißt für jede Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ gilt:

   $$H(p_1, p_2, \ldots, p_n) = H(p_{\pi(1)}, p_{\pi(2)}, \ldots, p_{\pi(n)})$$

   wobei $H(p_1, p_2, \ldots, p_n)$ die Entropie für das Quellenalphabet $\Sigma = a_1, a_2, \ldots, a_n$ mit $P(a_1) = p_1, \ldots, P(a_n) = p_n$ bezeichnet.

2. Untere und obere Grenzen für H lassen sich angeben:

   $$0 = H(1, 0, \ldots, 0) \leq H(p_1, \ldots, p_n) \leq H\left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right) = \log n$$

1.3 Benchmarks für die englische Sprache

Um die Leistung eines Komprimierungsschemas für Dateien mit Texten in Englisch abzuschätzen, muss man wissen, wieviel Informationsgehalt ein Zeichen der englischen Sprache besitzt. Ähnliche Überlegungen sind auch für andere (mit phonem- oder silbenorientierten Buchstaben geschriebenen) Sprachen gemacht worden; diese liefern im Wesentlichen die gleichen Ergebnisse. Nur im Englischen hat sich jedoch ein Standardkorpus für Benchmarks entwickelt. Da dies für die Einschätzung von Kompressionsverfahren wesentlich ist, beziehen wir uns im Folgenden auf die englische Sprache.

Nehmen wir an, dass wir die Information in bps (Bits je Zeichen) messen und daß wir 96 druckfähige ASCII Zeichen benutzen. Dann bekommen wir bei Annahme einer Gleichverteilung der Zeichen heraus, daß die Entropie gleich $\log 96 = 6,6$ bps, also recht groß ist. Verwenden wir jedoch eine Verteilung, die aus den Häufigkeiten der Vorkommen von Zeichen in einer großen Sammlung englischer Texte empirisch ermittelt wurde, so bekommen wir für die Entropie etwa $4,5$ bps. Für die Codierung, bei der jedes Zeichen separat codiert wird, ist die Huffman-Methode optimal (siehe Abschnitt 2.4) und in dem Fall benötigt sie etwa $4,7$ bps, das ist nur wenig schlechter als die Entropie.

Dass die Entropie $4,5$ beträgt, heißt allerdings nicht, dass man englische Texte nicht besser komprimieren könnte als mit $4,5$ Bits je Zeichen. Nehmen wir an, dass wir Blöcke von Zeichen der Länge 8 betrachten. Dann bekommen

wir 96^8 verschiedene Blöcke, und diese versuchen wir jetzt zu codieren. Eine interessante Beobachtung ist, dass das für die uniforme Verteilung nicht hilft: Die Entropie für die neue Quelle ist jetzt $\log 96^8 = 8 \times 6,6$ Bits je Block $= 6,6$ Bits je Zeichen, also genau so groß wie früher. Für die Verteilung, die der englischen Sprache entspricht, bekommen wir aber eine viel bessere untere Schranke für die Anzahl von Bits je Zeichen; die Entropie ist hier nämlich etwa 19 Bits je Block und das heißt $2,4$ bps. Wenn wir die Länge der Blöcke weiter vergrößern, dann werden wir eine immer bessere Entropie bekommen. Wegen der enormen Anzahl von Blöcken mit großer Länge ist es leider unmöglich, eine präzise Statistik zu bekommen. Man vermutet, daß die untere Schranke für die Entropie etwa bei $1,3$ bps liegt. Das bedeutet, man könnte verlustfrei von 96 druckfähigen ASCII-Zeichen zu 3 druckfähigen Zeichen übergehen, ohne deshalb längere Texte drucken zu müssen. Das Lesen solch eines Textes wäre aber für Menschen sehr mühevoll.

	Bytes	Beschreibung
bib	111261	Bibliographic files (refer format)
book1	768771	Hardy: Far from the madding crowd
book2	610856	Witten: Principles of computer speech
geo	102400	Geophysical data
news	377109	News batch file
obj1	21504	Compiled code for Vax: compilation of progp
obj2	246814	Compiled code for Apple Macintosh: Knowledge support system
paper1	53161	Witten, Neal and Cleary: Arithmetic coding for data compression
paper2	82199	Witten: Computer (in)security
paper3	46526	Witten: In search of "autonomy"
paper4	13286	Cleary: Programming by example revisited
paper5	11954	Cleary: A logical implementation of arithmetic
paper6	38105	Cleary: Compact hash tables using bidirectional linear probing
pic	513216	Picture number 5 from the CCITT Facsimile test files (text and drawings)
progc	39611	C source code: compress version 4.0
progl	71646	Lisp source code: system software
progp	49379	Pascal source code: prediction by partial matching evaluation program
trans	93695	Transcript of a session on a terminal

Tabelle 2: Details zum Calgary-Corpus

In der Praxis entwickelt man Komprimierungsmethoden, die für englische Texte verschiedener Art effizient sind. Als Benchmark wird für diese Zwecke oft der sogenannte *Calgary-Corpus* verwendet. Diese Sammlung enthält 2 Bücher, 5 Artikel, 1 Literaturliste, 1 Sammlung von Zeitungsartikeln, 3 Pro-

gramme, 1 Protokoll einer Sitzung am Computerterminal, 2 Binärcodes von Programmen, 1 geographische Datei und 1 Bit-Map eines Schwarz-Weiß-Bildes (Einzelheiten finden Sie in Tabelle 2; noch mehr Informationen hierzu enthält [1]). Zugegebenermaßen ist die letztere Beispieldatei eben keine Textdatei, wurde aber wohl hinzugenommen um typische Anwendungen von Kompressionsverfahren zu simulieren.

Datum	bps	Schema	Autoren
May 1977	3,94	LZ77	Ziv, Lempel
1984	3,32	LZMW	Miller and Wegman
1987	3,30	LZH	Brent
1987	3,24	MTF	Moffat
1987	3,18	LZB	Bell
.	2,71	GZIP	.
1988	2,48	PPMC	Moffat
.	2,47	SAKDC	Williams
.	2,47	PPMD	Howard
Nov 1993	2,34	PPMC	Moffat
Oct 1994	2,34	PPM*	Cleary, Teahan, Witten
18 Nov 1994	2,33	PPMD	Moffat
1995	2,29	BW	Burrows, Wheeler
31 Jan 1995	2,27	PPMD	Teahan
1997	1,99	BOA	

Tabelle 3: Die Entwicklung der Leistung von Kompressionsverfahren, gezeigt am Calgary-Corpus

Die Tabelle 3 zeigt die Entwicklung der Komprimierungsmethoden durch die Jahre für den Calgary-Corpus als Benchmark.

2 Grundlegende Codes

2.1 Präfixcodes

Betrachten wir ein Codierungverfahren, das jedem Symbol aus dem Quellenalphabet genau ein binäres Codewort zuordnet. Die Codierung hat zusätzlich folgende Eigenschaft: Kein Codewort ist ein Präfix des anderen Codeworts. Die Codes, die diese Eigenschaft haben, nennen wir *Präfixcodes*. Mit Hilfe von Präfixcodes können wir eine Folge von Symbolen so codieren, dass wir einfach die Codewörter hintereinander schreiben, ohne zusätzliche Trennsymbole zu benutzen. Wie man sich leicht überlegt, führt die Einführung von Trennsymbolen nach Codewörtern dazu, einen Präfixcode zu erzeugen.

In diesem Kapitel zeigen wir den Satz von Kraft und beschreiben dann die grundlegenden Präfixcodes.

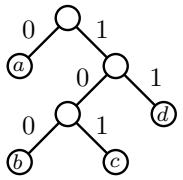


Abbildung 3: Der Baum eines Präfixcodes

Eine einfache Kennzeichnung von Präfixcodes ist die folgende: Im Präfixcode ist jedes Symbol durch ein Blatt in einem Binärbaum repräsentiert, und das Symbol wird durch den Weg von der Wurzel bis zum Blatt binärcodiert (links = 0, rechts = 1). Als Beispiel betrachte man Bild 3.

Satz 2.1 (Kraft 1949): *Es existiert ein Präfixcode* $K : \{a_1, \ldots, a_n\} \to \{0,1\}^+$ *mit* $|K(a_1)| = \ell_1, \ldots, |K(a_n)| = \ell_n$ *genau dann wenn*

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1. \tag{1}$$

Die Relation (1) heißt auch *Kraftsche Ungleichung*.

Beweis: Nehmen wir an, es gibt einen Präfixcode $K : \Sigma \to \{0,1\}^+$, wobei $\Sigma = \{a_1, \ldots, a_n\}$ ein Quellenalphabet ist. Dann sei $\ell_i = |K(a_i)|$ für $i = 1, \ldots, n$ und es sei

$$m = \max\{\ell_1, \ldots, \ell_n\}.$$

Betrachten wir jetzt den Binärbaum T für K, das heißt T hat n Blätter und jedes Blatt repräsentiert ein Codewort. Jetzt verlängern wir T zu einem vollständigen Binärbaum T' der Höhe m. Ein Beispiel für $n = 6$, $m = 4$ und einen Code $K(a_i) = c_i$ findet sich in Bild 4.

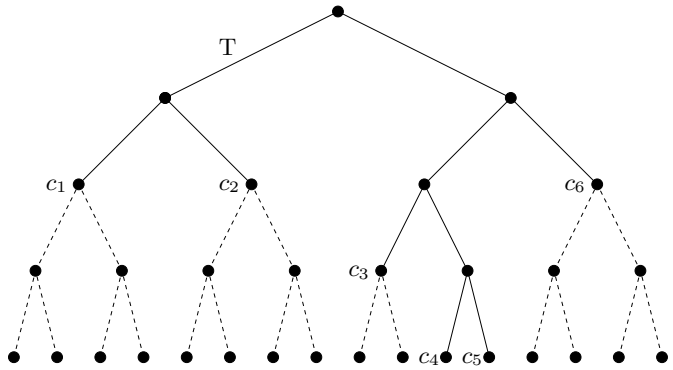


Abbildung 4: Ein (vervollständigter) Binärbaum T'

Jetzt betrachten wir die Teilbäume mit Wurzeln $c_1, c_2, \ldots, c_n$ und stellen fest daß die Teilbäume disjunkt sind. Es ist einfach zu sehen, daß ein Baum mit Wurzel c_i $2^{m-\ell_i}$ Blätter hat. Weil alle Teilbäume disjunkt sind und T' insgesamt 2^m Blätter hat, ergibt sich $\sum_{i=1}^{n} 2^{m-\ell_i} \leq 2^m$. Das impliziert die Kraftsche Ungleichung.

Jetzt nehmen wir an, dass die Kraftsche Ungleichung $\sum_{i=1}^{n} 2^{-\ell_i} \leq 1$ gilt. Wir wollen einen Code K konstruieren mit $|K(a_i)| = \ell_i$ für $i = 1, \ldots, n$. Es sei $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n = m$. Betrachten wir nun einen vollständigen Binärbaum T der Höhe m, wie beispielweise in Bild 5. Um den Code K zu

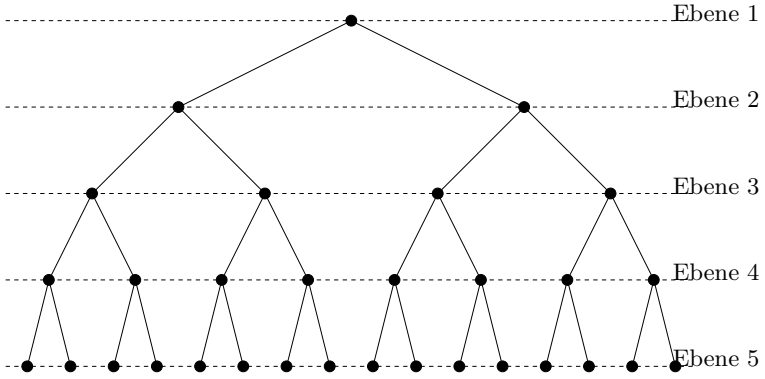


Abbildung 5: Ein Binärbaum zur Illustration

erzeugen, benutzen wir folgenden Algorithmus:

```
for k := 1 to n do
      1. in Ebene ℓ_k + 1 nimm Knoten
            v als c_k mit v ≠ c_i
            für alle i < k mit ℓ_i = ℓ_k
      2. entferne alle Nachfolger von v in T.
```

Da (wegen Punkt 2. des Schleifenrumpfes) stets nur Blätter in dem gleichzeitig konstruierten Baum T gewählt werden, erzeugt der Algorithmus einen Präfixcode. Die Frage ist nun, ob er für jedes $k = 1, \ldots, n$ in Punkt 1. des Schleifenrumpfes immer einen Knoten v in Ebene $\ell_k + 1$ findet. Um dies zu zeigen, betrachten wir die folgenden Ungleichungen:

$$2^m \geq 2^m \cdot \sum_{i=1}^{n} 2^{-\ell_i} > 2^m \cdot \sum_{i=1}^{k-1} 2^{-\ell_i} = \sum_{i=1}^{k-1} 2^{m-\ell_i}$$

Die rechte Größe beschreibt die Anzahl der Blätter des vollständigen Binärbaumes, die wir in den Schleifendurchläufen $1, 2, \ldots, k-1$ entfernt haben. Weil am Anfang T 2^m Blätter hat und weil $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n$ gilt, folgt aus den obigen Ungleichungen, dass wir in Schritt k mindesten ein Blatt in Ebene $m + 1$ haben. Deshalb gibt es einen Knoten v wie in Punkt 1. des Schleifenrumpfes erforderlich. □

Aus dem Beweis des vorigen Satzes lesen wir ab:

Folgerung 2.2 *Jeder Präfixcode genügt der Kraftschen Ungleichung.*

Für das Alphabet $\Sigma = \{a_1, ..., a_k\}$, wobei jedes Symbol a_i mit Wahrscheinlichkeit $P(a_i)$ auftritt, und die Codierung $K : \Sigma \to \{0,1\}^+$ definieren wir die *erwartete Länge* von K als

$$L_K := \sum_{i=1}^{k} P(a_i) \cdot |K(a_i)|.$$

Unten erklären wir Shannon-, Shannon-Fano- und Huffman-Codierungen. Huffman-Codes sind optimale Präfixcodes (also Codes mit kleinster erwarteter Länge L_K) und die ersten beiden Codierungen erzeugen für manche Eingaben —wie wir sehen werden— nicht optimale Lösungen. Wir geben aber diese Verfahren an, weil sie interessante und nützliche Strategien für Datenkompression enthalten.

2.2 Shannon-Algorithmus

Es seien $p_1 = P(a_1)$, ..., $p_n = P(a_n)$ mit $p_1 \geq p_2 \geq \ldots \geq p_n$. Dann sei $P_1 := 0$, und für $i > 1$ sei $P_i := p_1 + \ldots + p_{i-1}$.

Shannon-Algorithmus:

```
for i := 1 to n do
        ℓ_i := ⌈− log p_i⌉;
        Sei P_i := 0, b_1 b_2 b_3 ...
                K(a_i) := b_1 b_2 ... b_{ℓ_i};
```

Beispiel 2.3 Betrachten wir einen Text über dem Alphabet $\Sigma = \{a, b, c, d, e\}$, wobei die Verteilung der Symbole durch $P(a) = 0,35$, $P(b) = 0,17$, $P(c) = 0,17$, $P(d) = 0,16$ und $P(e) = 0,15$ gegeben ist. Folgende Tabelle stellt den Code dar, wie er durch den Shannon-Algorithmus generiert wird:

	p_i	ℓ_i	P_i	P_i (Binär)	Code
a	0,35	2	0,0	0,0000000...	00
b	0,17	3	0,35	0,0101100...	010
c	0,17	3	0,52	0,1000010...	100
d	0,16	3	0,69	0,1011000...	101
e	0,15	3	0,85	0,1101100...	110

Damit ergibt sich

$$L_S = 0,35 \cdot 2 + 0,17 \cdot 3 + 0,17 \cdot 3 + 0,16 \cdot 3 + 0,15 \cdot 3 = 2,65$$

Es ist leicht zu sehen, dass die Shannon-Codierung nicht optimal ist.

Die Shannon-Codierung zeigt in einfacher Weise Bezüge zwischen Zahldarstellungen von Wahrscheinlichkeiten (bzw. deren Summen) und Binärcodes. Dieser Gedanke wird in Kapitel 3 weiter entwickelt werden.

Satz 2.4 *Der Shannon-Algorithmus generiert einen Präfixcode.*

Beweis: Es sei $P_i = 0, b_1 b_2 b_3 \ldots = \frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots$ Nach Konstruktion gilt

$$\log \frac{1}{p_i} \leq \ell_i,$$

womit für jedes $j \geq i + 1$

$$P_j - P_i \geq P_{i+1} - P_i = p_i \geq \frac{1}{2^{\ell_i}}$$

gilt. Wegen $p_1 \geq p_2 \geq \ldots \geq p_n$ gilt $\ell_1 \leq \ell_2 \leq \ldots \leq \ell_n$. Angenommen, es gibt ein i und ein j mit $i < j$ und

$$K(a_i) = b_1 b_2 \ldots b_{\ell_i}, \quad K(a_j) = c_1 c_2 \ldots c_{\ell_j}$$

wobei $\ell_j \geq \ell_i$. Ist $b_1 = c_1, \ldots, b_{\ell_i} = c_{\ell_i}$, so gilt

$$\begin{aligned} & P_j - P_i \\ = & \left(\frac{b_1}{2^1} + \cdots + \frac{b_{\ell_i}}{2^{\ell_i}} + \frac{c_{\ell_i+1}}{2^{\ell_i+1}} + \cdots\right) - \left(\frac{b_1}{2^1} + \cdots + \frac{b_{\ell_i}}{2^{\ell_i}} + \frac{b_{\ell_i+1}}{2^{\ell_i+1}} + \cdots\right) \\ < & \ 2^{-\ell_i}, \end{aligned}$$

also ein Widerspruch. □

2.3 Shannon-Fano-Codierung

Ein weiteres Beispiel für einen Präfixcode ist die *Shannon-Fano-Codierung*:

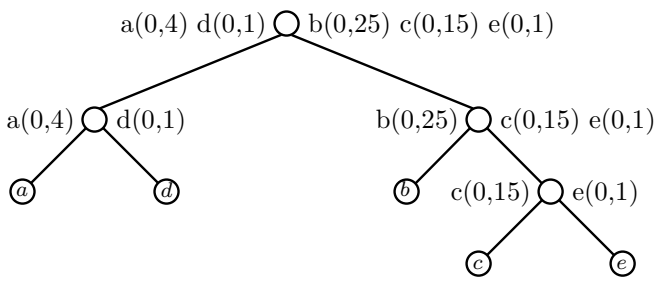
```
make-code(Σ)
    if |Σ| = 1 then for a ∈ Σ return node(a)
    else
        teile Σ auf in Σ1 und Σ2 mit Σ_{a∈Σ1} P(a) ≈ Σ_{a∈Σ2} P(a);
        return node(make-code(Σ1),make-code(Σ2)).
```

Aus dem rekursiven Aufbau des Algorithmus (Rekursionsbaum) folgt sofort:

Satz 2.5 *Der Shannon-Fano-Algorithmus liefert einen Präfixcode.*

Die Codierung zeigen wir nur an einem Beispiel. Im Vergleich mit der Huffman-Codierung ist die Methode von Shannon-Fano viel schlechter. Sie konstruiert nicht immer optimale Präfixcodes.

Beispiel 2.6 Betrachten wir die folgende Eingabe: $P(a) = 0,4,\ P(b) = 0,25,\ P(c) = 0,15$ und $P(d) = P(e) = 0,1$. Der Shannon-Fano-Algorithmus arbeitet folgendermaßen



Die Ausgabecode ist:

	P	Code	Länge
a	$0,4$	00	2
b	$0,25$	10	2
c	$0,15$	110	3
d	$0,1$	01	2
e	$0,1$	111	3

Für diese Codierung ist die erwartete Länge

$$L_{S\text{-}F} = 2 \cdot 0,4 + 2 \cdot 0,25 + 3 \cdot 0,15 + 2 \cdot 0,1 + 3 \cdot 0,1 = 2,25$$

während die (optimale) erwartete Länge $2,15$ ist. Wir werden später sehen, daß die erwartete Länge für die Huffman-Codierung

$$L_H = 1 \cdot 0,4 + 2 \cdot 0,25 + 3 \cdot 0,15 + 2 \cdot 4 \cdot 0,1 = 2,15$$

beträgt, also optimal ist.

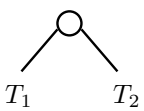
Wie man leicht nachrechnet, liefert die Shannon-Fano-Codierung im Beispiel aus dem letzten Abschnitt eine erwartete Länge $L_{S-F} = 2,5$, ist also in dem Fall etwas besser als die Shannon-Codierung.

2.4 Huffman-Algorithmus

Huffman-Codes sind optimale Präfixcodes (also Codes mit kleinster erwarteter Länge L_K).

Der Algorithmus für die *Huffman-Codierung* arbeitet wie folgt:

1. `Starte mit dem Wald aus Bäumen, in dem jeder Baum ein Symbol darstellt und` $w_i = P(a_i)$ `das Gewicht des Baumes ist.`
2. `Repeat until Wald besteht aus nur einem Baum:`
   - `wähle die zwei Bäume` T_1 `und` T_2 `mit den kleinsten Gewichten` w_1 `und` w_2`;`
   - `nimm nun statt` T_1 `und` T_2 `den Baum`

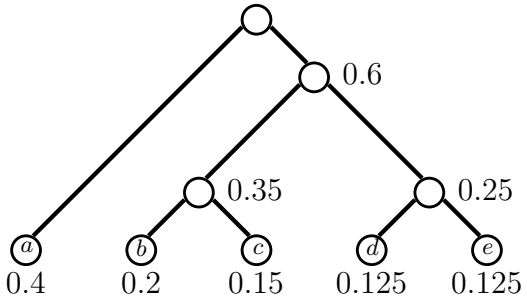
     

     `mit dem Gewicht` $w_1 + w_2$`.`

Beispiel 2.7 Betrachten wir das Eingabealphabet $\Sigma = \{a, b, c, d, e\}$ und die Wahrscheinlichkeiten:

$P(a)$	$P(b)$	$P(c)$	$P(d)$	$P(e)$
$0,4$	$0,2$	$0,15$	$0,125$	$0,125$

Der Algorithmus konstruiert den Ausgabebaum folgendermaßen:



Der Ausgabecode ist:

a	b	c	d	e
0	100	101	110	111

Man kann induktiv das folgende Ergebnis beweisen:

Satz 2.8 *Die erwartete Codelänge ist für Huffman-Codierung optimal.*

Betrachten wir jetzt die Qualität der Huffman-Codierung in Vergleich mit der Entropie, oder —anders gesagt— vergleichen wir nun die optimale erwartete Codelänge mit der Entropie.

Satz 2.9 *Sei Σ ein Quellenalphabet und $K : \Sigma \to \{0,1\}^+$ ein beliebiger Präfixcode. Dann gilt:*

$$L_K \geq H(\Sigma).$$

Satz 2.10 *Für jedes Quellenalphabet Σ ist die minimale erwartete Codelänge für Präfixcodes höchstens $H(\Sigma) + 1$.*

Dann bekommen wir als Korollar die folgende Abschätzung für die erwartete Länge der Huffman-Codierung:

Korollar 2.11 *Für jedes Quellenalphabet Σ gilt:*

$$H(\Sigma) \leq L_H \leq H(\Sigma) + 1.$$

Wir beschließen dieses Kapitel mit der Skizze einer Implementierung des Huffman-Algorithmus.

Codierer:

```
/* --- first pass --- */
initialize_frequencies;
while (ch != eof)
{
     ch = getchar(input);
     update_frequence(ch);
}
construct_tree(T);
puttree(output,T);
/* --- second pass --- */
initialize_input;
while (ch != eof)
{
     ch = getchar(input);
     put(output,encode(ch));
}
put(output,encode(eof));
```

Decodierer:

```
gettree(input,T);
while ((ch=decode(input)) != eof)
     putchar(output,ch);
```

2.5 Adaptive Huffman-Codierung

Um die Huffman-Codierung zu implementieren, braucht man vollständige Kenntnisse über die Wahrscheinlichkeitverteilung beziehungsweise über die Häufigkeit der einzelnen Symbole. Die *adaptive Huffman-Codierung* generiert hingegen die entsprechende Statistiken dynamisch. Ein solcher Zugang ist besonders nützlich bei einer on-line Übertragung von Daten. Außerdem braucht bei der adaptiven Huffman-Codierung der Codierer nicht den Baum in Ausgabecode zu codieren. Das allgemeine Szenario für die adaptive Huffman-Codierung sieht wie folgt:

Adaptive Huffman-Codierung

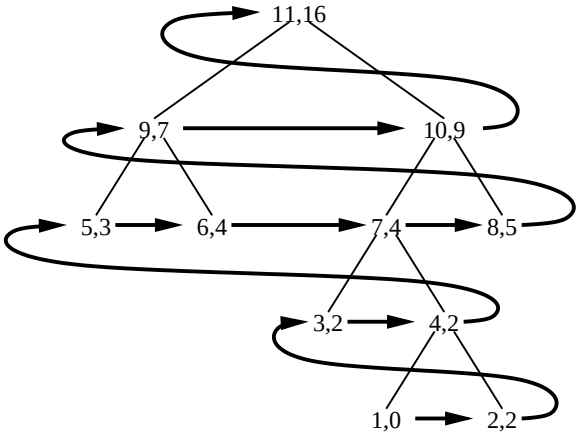
Codierer:

```
initialize_tree(T);
while (ch != eof)
{
     ch = getchar(input);
     put(output, encode(ch));
     update_tree(ch);
}
```

Decodierer:

```
initialize_tree(T);
while (ch != eof)
{
     ch = decode(input);
     putchar(output,ch);
     update_tree(ch);
}
```

Wir zeigen jezt eine effiziente Implementierung für `update_tree`. Es sei Σ mit $|\Sigma| = n-1$. Wir betrachten $\Sigma \cup \{\texttt{NYT}\}$ (`NYT` - not yet transmitted). Jeder Knoten eines Binärbaumes (wobei für jeden Knoten gilt: *outdeg* $= 2$ oder 0) hat Knotennumer $\ell_i \in \{1, 2, \ldots 2n-1\}$ und Knotengewicht $w_i \in \mathbb{N}_0$. Wir definieren die *Geschwister-Eigenschaft* wie folgt: Für $j = 1, .., n$ sind ℓ_{2j-1} und ℓ_{2j} Geschwister eines Vaters mit $\ell_k > \ell_{2j-1}, \ell_{2j}$ und $w_k = w_{2j-1} + w_{2j}$ und $w_1 \leq w_2 \leq w_3 \ldots \leq w_{2n-1}$.



Ein *Block* sind alle Knoten mit gleichem Gewicht. Man kann zeigen: Jeder Baum mit Geschwister-Eigenschaft ist ein Huffman-Baum.

Initially: `current_number = 2n-1; root.number = 2n-1; root.weight=0;`

```
void update_tree(char ch)
{
     if first_appearance(ch)  {
          p = leaf(NYT); p->weight++;
          q = new node(ch);
          q->father = p; q->weight = 1; q->number = --current_number;
          r = new node(NYT);
          r->father = p; r->weight = 0; r->number = --current_number;
          p = p->father;
     }
     else p = leaf(ch);
     while (!root(p))  {
          if (!(sibling(p, leaf(NYT))) && !max_number_in_block(p))
          { let q highest numbered node in block; switch(p,q); }
          p->weight++;
          p = p->father;
     }
}
```

2.6 Erweiterte Huffman-Codierung

Betrachten wir motivierend folgendes Beispiel: Sei $\Sigma = \{a, b\}$ und $P(a) = 0,9$; $P(b) = 0,1$. Dann liefert der Huffman-Algorithmus folgende Codierung:

	P	Code
a	$0,9$	0
b	$0,1$	1

mit der erwarteten Länge $1 \cdot 0,9 + 1 \cdot 0,1 = 1$ Bits/Symbol. Die Entropie für unser Alphabet (also die mittlere Anzahl von Bits, um das Symbol aus Σ zu codieren) ist

$$H(\Sigma) = 0,9 \cdot \log \frac{10}{9} + 0,1 \cdot \log 10 = 0,47 \ .$$

Die *Redundanz* —die Differenz zwischen der erwarteten Länge und der Entropie— ist $0,53$ Bits je Symbol, also fast 113% der Entropie. Das heißt, daß (1) unser Code praktisch nicht komprimiert (er „übersetzt“ lediglich $a \to 0$ und $b \to 1$) und (2) um einen Text zu codieren, benutzen wir um 113% mehr Bits, als eine minimale Codierung bräuchte. Um diesen Unterschied zu verkleinern, betrachten wir das neue Alphabet

$$\Sigma^2 = \{aa, ab, ba, bb\}$$

mit der Wahrscheinlichkeiten: $P(aa) = [P(a)]^2$, $P(bb) = [P(b)]^2$ und $P(ab) = P(ba) = P(a) \cdot P(b)$. Der Huffman-Algorithmus konstruiert folgende Codierung:

	P	Code	Länge
aa	$0,81$	0	1
ab	$0,09$	10	2
ba	$0,09$	110	3
bb	$0,01$	111	3

Die erwartete Länge ist $1 \cdot 0,81 + 2 \cdot 0,09 + 3 \cdot 0,09 + 3 \cdot 0,01 = 1,29$ Bits je Symbol im Alphabet Σ^2, also $0,645$ Bits je Symbol in Σ und jetzt benutzt unsere Codierung nur noch 37% mehr Bits als eine minimale Codierung. Für das Alphabet

$$\Sigma^3 = \{aaa, aab, aba, baa, abb, bab, bba, bbb\}$$

ist der Unterschied noch kleiner:

	P	Code	Länge
aaa	$0,729$	0	1
aab	$0,081$	100	3
aba	$0,081$	101	3
baa	$0,081$	110	3
abb	$0,009$	11100	5
bab	$0,009$	11101	5
bba	$0,009$	11110	5
bbb	$0,001$	11111	5

und die erwartete Länge ist $0,52$ bps: nur um 11% mehr als die Entropie $H(\Sigma)$.

Sei H^m die *erweiterte Huffman-Codierung* für die Blöcke der Länge m. Dann bekommen wir die folgende Ungleichung.

Satz 2.12 *Für jedes Quellenalphabet Σ gilt:*

$$H(\Sigma) \leq L_{H^m} \leq H(\Sigma) + \frac{1}{m}.$$

Beweisidee: Wir zeigen, daß $H(\Sigma^m) = mH(\Sigma)$. Daraufhin verwenden wir Korollar 2.11. □

Wir haben gesehen, daß die Wahrscheinlichkeit für jedes neue „Zeichen" $a_{i_1}a_{i_2}\ldots a_{i_m}$ des Alphabets Σ^m das Produkt von Wahrscheinlichkeiten

$$P(a_1) \times P(a_2) \times \ldots P(a_m)$$

ist. Das heißt, daß die Wahrscheinlichkeiten für die Zeichen innerhalb eines Blocks unabhängig sind. In der Praxis haben wir aber sehr selten mit einer solchen Situation zu tun. Für die englische Sprache z. B. ist die Wahrscheinlichkeit, dass das Zeichen 'a' im U. S. Grundgesetz vorkommt, gleich $0,057$, aber die Wahrscheinlichkeit für die Reihenfolge von 10 'a's ist nicht $0,057^{10}$ sondern 0. In der Praxis muß man für jedes m eine neue Statistik für Σ^m konstruieren.

Das ist aufgrund des exponentiellen Wachstums kaum machbar. Einen praktikablen Ausweg beschreibt das nächste Kapitel.

3 Arithmetische Codes

Im vorherigen Kapitel haben wir gezeigt, dass die erweiterte Huffman-Codierung eine Ausgabe mit sehr kleiner Redundanz produziert. Die Idee war die Folgende: Statt einzelne Symbole des Quellenalphabets codieren wir die Sequenzen der Symbole mit Hilfe des grundlegenden Huffman-Algorithmus. Der Nachteil des Verfahrens ist aber, dass der Algorithmus Huffman-Codes für alle möglichen Folgen der Quellensymbole konstruieren muss, um einen Eingabetext zu codieren. Das heißt beispielsweise, dass für ein Quellenalphabet mit 256 Symbolen und für Folgen der Länge 3 der Algorithmus 16 777 216 neue Symbole betrachten muss. Deshalb ist der erweiterte Huffman-Algorithmus schon für den Fall der Länge 3 einfach unbrauchbar. In diesem Kapitel werden wir die *arithmetischen Codes* diskutieren. Für diese Codes betrachten wir auch statt einzelner Symbole Folgen von Symbolen, und dann codieren wir diese Folgen. Der Vorteil des Verfahrens ist, dass jede Folge separat codiert werden kann. Das Schema des Verfahrens ist folgendes: Es sei $u_1u_2u_3u_4\dots u_m \in \Sigma^*$ ein Text, wobei Σ das Quellenalphabet ist. Dann berechne für die Folge eine numerische Repräsentation – einen Bruch zwischen 0 und 1, und für diese Repräsentation berechne den binären Code. Der Algorithmus, der Arithmetische Codes konstruiert, produziert für die Folge direkt den binären Code, aber um die Methode leichter zu verstehen, werden wir sie weiterhin als zweistufiges Verfahren darstellen:

Folge der Symbole $\longrightarrow$ numerische Repräsentation $\longrightarrow$ binärer Code.

3.1 Numerische Repräsentation

Sei $\Sigma = \{a_1, a_2, a_3, \dots, a_n\}$ das Quellenalphabet mit Wahrscheinlichkeiten $P(a_1), \dots, P(a_n)$. Wir definieren $F(0) = 0$ und für $i = 1, 2, \dots, n$

$$F(i) = \sum_{k=1}^{i} P(a_k).$$

Sei $a_{i_1}a_{i_2}a_{i_3}\dots a_{i_m}$ eine Folge. Eine *numerische Repräsentation* für diese Folge ist ein Bruch im Intervall $[l^{(m)}, u^{(m)})$ (engl.: lower bzw. upper). Die Grenzen des Intervalls definieren wir rekursiv folgendermaßen:

$$l^{(1)} = F\left(i_1 - 1\right) \qquad u^{(1)} = F\left(i_1\right)$$

und für alle $k = 2, 3, \dots, m$

$$\begin{aligned} l^{(k)} &= l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F\left(i_k - 1\right) \\ u^{(k)} &= l^{(k-1)} + \left(u^{(k-1)} - l^{(k-1)}\right) F\left(i_k\right). \end{aligned}$$

Wie man leicht durch Betrachten von „Teleskopprodukten" sieht, gilt der folgende Zusammenhang (unter Beibehaltung der eben eingeführten Bezeichnungsweisen):

Lemma 3.1 $u^{(m)} - l^{(m)} = \prod_{k=1}^{m} P(a_{i_k})$.

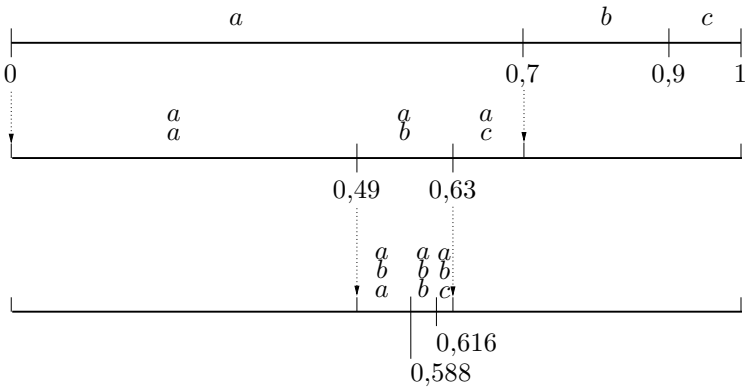


Abbildung 6: Intervallrepräsentation

Als Beispiel für diese Konstruktion betrachten wir:

Beispiel 3.2 $\Sigma = \{a, b, c\}$ mit Wahrscheinlichkeiten $P(a) = 0,7, P(b) = 0,2, P(c) = 0,1$ und $m = 3$. Das Intervall z. B. für die Folge abb ist $[0,588; 0,616)$, siehe Bild 6.

Die numerische Repräsentation kann man jetzt beispielsweise folgendermaßen definieren:

$$\overline{T}(a_{i_1} a_{i_2} \ldots a_{i_m}) = \frac{l^{(m)} + u^{(m)}}{2},$$

also als den Mittelpunkt des Intervalls. Wir sehen, dass die einzige Information, die der Algorithmus braucht, um die numerische Repräsentation für eine gegebene Folge zu berechnen, die Funktion F ist. Insbesondere braucht er nicht die Repräsentationen für die anderen Folgen zu kennen.

Um die numerische Repräsentation zu finden, berechnen wir die Grenzen $l^{(k)}$ und $u^{(k)}$, für $k = 1, 2, \ldots, m$ und dann wählen wir einen Wert in $[l^{(m)}, u^{(m)})$. Der umgekehrte Prozess —eine numerische Repräsentation zu entziffern— läuft sehr ähnlich: Es sei T die gegebene numerische Repräsentation. Der folgende Algorithmus konstruiert die Folge für T.

```
Sei l^(0) = 0 und u^(0) = 1;
for k := 1 to m do
begin
```

```
        T* := (T − l^(k−1))/(u^(k−1) − l^(k−1));
        Finde i_k so dass F(i_k − 1) ≤ T* < F(i_k);
        Return(a_{i_k});
        Berechne l^(k) und u^(k);
end.
```

3.2 Binärer Code

Nehmen wir an, dass $\overline{T}(a_{i_1}a_{i_2}\dots a_{i_m}) = (u^{(m)} - l^{(m)})/2$ eine numerische Repräsentation für die Folge $a_{i_1}a_{i_2}\dots a_{i_m}$ ist. Die binäre Darstellung der numerischen Repräsentation kann natürlich beliebig lang bzw. unendlich sein. In diesem Kapitel zeigen wir, wie man die Repräsentationen mit Hilfe einer kleinen Anzahl von Bits codieren kann.

Es sei, wie früher, $P(a_{i_1}a_{i_2}\dots a_{i_m}) = P(a_{i_1})\cdot P(a_{i_1})\dots P(a_{i_m})$. Wir definieren

$$\ell(a_{i_1}a_{i_2}\dots a_{i_m}) \;=\; \left\lceil \log \frac{1}{P(a_{i_1}a_{i_2}\dots a_{i_m})} \right\rceil + 1.$$

Den binären Code der Repräsentationen definieren wir als $\ell(a_{i_1}a_{i_2}\dots a_{i_m})$ höchstwertige Bits des Bruchs $\overline{T}(a_{i_1}a_{i_2}\dots a_{i_m})$.

Beispiel 3.3 Sei $\Sigma = \{a, b\}$, $P(a) = 0,9$, $P(b) = 0,1$ und die Länge $m = 2$. Dann bekommen wir folgende Codierung für alle Folgen:

x	$\overline{T}(x)$	binär	$\ell(x)$	Code
aa	$0,405$	$0,0110011110...$	2	01
ab	$0,855$	$0,1101101011...$	5	11011
ba	$0,945$	$0,1111000111...$	5	11110
bb	$0,995$	$0,1111111010...$	8	11111110

Wir zeigen jetzt, dass man die Codierung *eindeutig* codieren und decodieren kann. Dann geben wir den Algorithmus für die Codierung bzw. die Decodierung an.

Es sei $\lfloor y \rfloor_\ell$ ein Bruch y, der nach dem ℓ-ten Bit abgeschnitten ist. Wir wollen zeigen, dass für die Folge x die Zahl $\lfloor \overline{T}(x) \rfloor_{\ell(x)}$ ein eindeutiger Repräsentant für die numerische Repräsentation von x ist. Hinreichend dazu werden wir zeigen, dass sich $\lfloor \overline{T}(x) \rfloor_{\ell(x)}$ im Intervall $\left[l^{(m)}, u^{(m)}\right)$ der Folge x befindet.

Satz 3.4 *Sei $\left[l^{(m)}, u^{(m)}\right)$ ein Intervall für die Folge $x \in \Sigma^m$. Dann gilt:*

$$\left\lfloor \overline{T}(x) \right\rfloor_{\ell(x)} \in \left[l^{(m)}, u^{(m)}\right).$$

Beweis: Wir haben $\lfloor \overline{T}(x) \rfloor_{\ell(x)} < u^{(m)}$, weil $\overline{T}(x) < u^{(m)}$. Um die zweite Ungleichung zu zeigen, betrachten wir zuerst

$$\begin{aligned} \frac{1}{2^{\ell(x)}} &= \frac{1}{2^{\lceil \log 1/P(x) \rceil + 1}} \\ &\leq \frac{1}{2^{\log 1/P(x) + 1}} \\ &= \frac{1}{2\frac{1}{P(x)}} = \frac{P(x)}{2}. \end{aligned} \tag{2}$$

Aus Lemma 3.1 folgt:

$$\overline{T}(x) = l^{(m)} + \frac{P(x)}{2}.$$

Dann bekommen wir mit Ungleichung (2)

$$\overline{T}(x) \geq l^{(m)} + \frac{1}{2^{\ell(x)}}.$$

Wenn wir also den Bruch $\overline{T}(x)$ hinter dem $\ell(x)$-ten Bit abschneiden, dann gilt wegen $\overline{T}(x) - \lfloor \overline{T}(x) \rfloor_{\ell(x)} \leq 2^{-\ell(x)}$:

$$\lfloor \overline{T}(x) \rfloor_{\ell(x)} \geq l^{(m)}$$

□

Jetzt zeigen wir, dass man den binären Code eindeutig decodieren kann. Diese Eigenschaft folgt direkt aus:

Satz 3.5 *Jeder Arithmetische Code ist Präfix-Code.*

Beweis: Es sei $a \in [0,1)$ und $0, b_1 b_2 \dots b_n$ die binäre Darstellung für a. Beobachten Sie zuerst, dass die folgende Eigenschaft gilt: Ein Bruch $b \in [0,1)$ hat eine binäre Darstellung mit Präfix $b_1 b_2 \dots b_n$ genau dann, wenn $b \in [a, a + \frac{1}{2^n})$. Alles, was man jetzt zum Beweis des Satzes noch zeigen muss, ist: Das ganze Intervall $\left[\lfloor \overline{T}(x) \rfloor_{\ell(x)}, \lfloor \overline{T}(x) \rfloor_{\ell(x)} + \frac{1}{2^{\ell(x)}} \right)$ liegt in $\left[l^{(m)}, u^{(m)} \right)$, wobei $l^{(m)}$ und $u^{(m)}$ die Grenzen für die Folge x sind.
Wir haben schon gezeigt, dass $\lfloor \overline{T}(x) \rfloor_{\ell(x)} \geq l^{(m)}$. Wir brauchen also nur noch die zweite Ungleichung zu beweisen. Wir haben (mit den Überlegungen des vorigen Beweises):

$$\begin{aligned} u^{(m)} - \lfloor \overline{T}(x) \rfloor_{\ell(x)} &\geq u^{(m)} - \overline{T}(x) \\ &= \frac{P(x)}{2} \geq \frac{1}{2^{\ell(x)}}. \end{aligned}$$

□

Jetzt schätzen wir ab, wie effizient diese Codierung ist. Man braucht, um eine Folge x zu codieren, $\ell(x) = \left\lceil \log \frac{1}{P(x)} \right\rceil + 1$ Bits. Es sei L_{A^m} die erwartete Länge für einen arithmetischen Code für eine Folge der Länge m. Dann gilt wegen $\ell(x) = \lfloor \log(P(x)) \rfloor + 1 \leq \log(P(x)) + 2$:

$$\begin{aligned} L_{A^m} &= \sum_{x \in \Sigma^m} P(x)\ell(x) \\ &\leq -\sum_{x \in \Sigma^m} P(x) \log P(x) + 2 \sum_{x \in \Sigma^m} P(x) \\ &= H(\Sigma^m) + 2. \end{aligned}$$

Weil L_{A^m} immer größer als die Entropie ist, bekommen wir

$$H(\Sigma^m) \leq L_{A^m} \leq H(\Sigma^m) + 2.$$

Für die durchschnittliche Länge L_A pro Symbol bekommen wir

$$\frac{H(\Sigma^m)}{m} \leq L_A \leq \frac{H(\Sigma^m)}{m} + \frac{2}{m}$$

und weil $H(\Sigma^m) = mH(\Sigma)$, gilt

$$H(\Sigma) \leq L_A \leq H(\Sigma) + \frac{2}{m}.$$

Wenn wir also die Länge der Folge vergrößern, dann kommen wir immer näher an die Entropie.

Algorithmus

Schwierigkeit: Rechengenauigkeit (denn die Länge der Intervalle konvergiert gegen Null).
Lösung: Skalierung der Intervalle.

Codiere das k-the Symbol a_{i_k};

```
/* am Anfang der Codierung:
    l := 0;  u := 1;  I := [l, u); bits_to_follow:=0 */
    u := l + |I| · F(i_k); l := l + |I| · F(i_{k-1}); I := [l, u);
    while(1) {
        if I ⊆ [0, 0.5) { Fall I
            output(0);
            while (bits_to_follow > 0)
```

```
            { output(1); bits_to_follows -- ; }
        Skalierung [0, 0.5) → [0, 1) : l =: 2l;  u =: 2u;
    }
    else if I ⊆ [0.5, 1) { Fall II
        output(1);
        while (bits_to_follow > 0)
            { output(0); bits_to_follows -- ; }
        Skalierung [0.5, 1) → [0, 1) : l := 2(l − 0.5);  u := 2(u − 0.5);
    }
    else if I ⊆ [0.25, 0.75) { Fall III
        bits_to_follows ++ ;
        Skalierung [0.25, 0.75) → [0, 1) :
              l := 2(l − 0.25);  u := 2(u − 0.25);
    }
    else break; Fall IV
}  /* while(1) */
```

Ende der Codierung;

```
bits_to_follow ++ ;
if (l < 0.25) { Fall V
    output(0);
    while (bits_to_follow > 0)
        { output(1); bits_to_follows -- ; };
}
else { Fall VI
    output(1);
    while (bits_to_follow > 0)
        { output(0); bits_to_follows -- ; };
}
```

Betrachten wir wieder das selbe Beispiel wie früher:

Beispiel 3.6 Wir wollen *abb* codieren bei den gegebenen Wahrscheinlichkeiten $P(a) = 0,7, P(b) = 0,2, P(c) = 0,1$.

Tabellarisch liefert der Algorithmus das Folgende:

k	Intervall	Fall	bits_to_follow	Ausgabe
1	$[0,0; 0,7]$	IV	0	-
2	$[0,49; 0,63]$	III	$0 \to 1$	-
2	$[0,48; 0,76]$	IV	1	-
3	$[0,676; 0,732]$	II	$1 \to 0$	10
3	$[0,352; 0,464]$	I	0	0
3	$[0,704; 0,928]$	II	$0 \to 1$	1
3	$[0,408; 0,856]$	VI	$1 \to 0$	10

4 Wörterbuch-Techniken

Die Idee für *Wörterbuch-Techniken* ist folgende: Konstruiere eine Liste der Muster, die im Text vorkommen, und codiere die Muster als Indizes der Liste. Diese Methode ist besonders nützlich, wenn sich eine kleine Anzahl Muster sehr häufig im Text wiederholt. Dann gibt es zwei Möglichkeiten: Wenn wir genügend Kenntnisse über den Text haben, dann benutzen wir ein statisches Verfahren (mit festem *Wörterbuch*, engl.: dictionary), sonst eine dynamische Methode, bei der das Wörterbuch im Laufe der (De-)codierung aufgebaut wird.

In abgewandelter Form werden wir Wörterbuchtechniken in Gestalt von Codebüchern, siehe Abschnitt 9.

4.1 Statische Verfahren

Die meisten *statischen Verfahren* sind nur für ganz besondere Fälle nützlich. Zum Beispiel für die Komprimierung von Dateien, in denen die Leistungen von Studenten gespeichert sind. Dort kommen Wörter wie „Name", „Matrikelnummer" und „Note" sehr oft vor.

Hier beschreiben wir eine Methode, die etwas allgemeiner angewandt werden kann, nämlich den *Digramm-Codier-Algorithmus*. Bei diesem Verfahren enthält das Wörterbuch alle einzelnen Buchstaben und dann möglichst viele Paare von Buchstaben, die wir *Digramme* nennen. Z. B. sind für ein Wörterbuch der Länge 256 die ersten 95 Einträge die druckbaren ASCII-Zeichen (ohne das Leerzeichen) und die restlichen 161 Einträge sind diejenigen Paare von Symbolen, die „erfahrungsgemäß" am häufigsten benutzt werden. Dann codiert man jedes Symbol bzw. jedes Symbol-Paar mit 8 Bits.

Ein wesentlicher Unterschied von Wörterbuch-Techniken zu früher erläuterten Verfahren wir schon hier deutlich: Während bislang immer nur einzelne Zeichen oder Blöcke fester Länge codiert wurden, werden jetzt direkt (also mit einem einzigen Codewort) zu codierende Textstücke unterschiedlicher Länge zugelassen, wobei meist „greedy"-artig versucht wird, ein möglichst langes Textstück direkt zu codieren. Etwas formaler bedeutet dies, dass ein zu codierender Text t zerlegt wird in $t = c_1 \dots c_n$, wobei die c_i direkt zu codierende Textstücke sind mit der Eigenschaft, dass c_i der längste Präfix von $c_i \dots c_n$ ist, der im Wörterbuch zu finden ist.

Im Beispiel des Digramm-Codier-Algorithmus bedeutet dies: findet sich ein Symbolpaar nicht im Wörterbuch, so wird das erste Zeichen des Paares direkt codiert und sodann versucht, das letzte Zeichen jenes Paares zusammen

mit dem nächsten Eingabezeichen direkt (als neues Symbolpaar) direkt zu codieren, usw.

4.2 Dynamische Verfahren

Alle wichtigen *dynamischen Verfahren* basieren auf zwei Algorithmen, die von Jacob Ziv und Abraham Lempel 1977 bzw. 1978 beschrieben wurden, nämlich LZ77 und LZ78. Hier beschreiben wir die grundlegenden Algorithmen und ihre Varianten:

- LZSS — Lempel-Ziv-Storer-Szymanski (gzip, ZIP und andere),
- LZW — Lempel-Ziv-Welch (GIF, TIFF),
- LZC — Lempel-Ziv (Unix Compress, im Skript LZ77 genannt) und
- LZMW — Lempel-Ziv-Miller-Wegman[4]

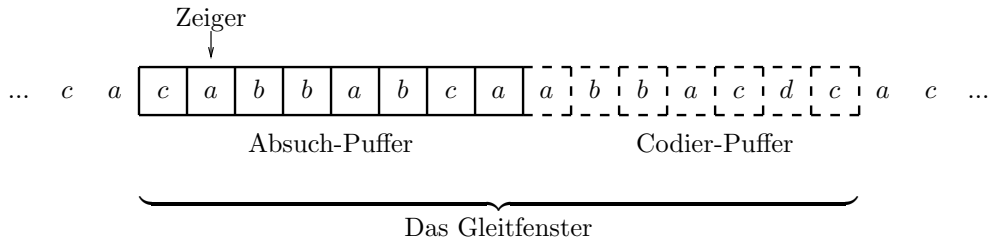


Abbildung 7: LZ77: Eine Momentaufnahme

LZ77, auch *Gleitfenster-Algorithmus* (engl.: sliding windows) genannt, bearbeitet den zu codierenden Text mit Hilfe eines Fensters, das über den Text hinweg verschoben wird. Das Fenster besteht aus zwei Teilen: den *Absuch-Puffer* und den *Codier-Puffer*. Der Absuch-Puffer enthält einen Suffix des Textes, der bis zu einem bestimmten Zeitpunkt verarbeitet wurde und die Zeichenfolge im Absuch-Puffer dient als Wörterbuch, um die Zeichen im Codier-Puffer zu codieren. Als Beispiel betrachte man Abbildung 7. In diesem Beispiel ist die Länge der Puffer 8 bzw. 7.

Um einen neuen Code zu konstruieren, benutzt LZ77 das folgende Verfahren:

1. Verschiebe den Zeiger auf das erste Symbol von rechts im Absuch-Puffer. Dann sei `Startsymbol, S :=` „das erste Symbol (von links) im Codier-Puffer" und `offset, L := 0`.

[4]Dies Verfahren wird nicht weiter im Skript erläutert, ebensowenig wie viele andere Varianten, siehe [16]. Im Internet gibt es auch gute Quellen insbesondere für Bildkompressionsverfahren und deren Vergleich, siehe z.B. die Internetseite 1 mit all ihren Verweisen.

2. Verschiebe den Zeiger im Absuch-Puffer so weit nach links, bis er das Zeichen findet, das gleich dem `Startsymbol` ist, oder die linke Grenze des Puffers erreicht hat. Wenn die Grenze des Puffers erreicht ist, dann gib als Code

   `(offset, L, K(S))`

   aus, wobei `K(S)` der Code des Zeichens `S` ist; sonst gehe zum nächsten Schritt.

3. Lies Zeichen für Zeichen gleichzeitig vom `Startsymbol` und Zeiger so lange, bis die Zeichen gleich sind. Wenn die Länge der durchgelesenen Zeichenfolgen größer als `L` ist, dann nimm diese Länge als neuen Wert für `L` und den Abstand des Zeigers zur rechten Grenze des Absuch-Puffers als neuen Wert für `offset`. Weiterhin nimm das `L+1`-ste Symbol des Codier-Puffers als `S` und beginne so wieder mit Schritt 2.

Beispiel 4.1 Wir verdeutlichen das Verfahren am Beispielfenster von Bild 7:

`Startsymbol`	`S`	`offset`	`L`	Zeiger	Ausgabe
a	a	0	0	0	—
a	b	1	1	1	—
a	b	1	1	2	—
a	b	1	1	3	—
a	b	4	2	4	—
a	b	4	2	5	—
a	b	4	2	6	—
a	c	7	4	7	—
a	c	7	4	8	$(7, 4, K(c))$

Die erste Zeile zeigt die Ausgangssituation. Danach beschreibt jede Zeile die Inhalte der Variablen des Verfahrens für jeden möglichen „Zeigerstand" *nach* der Abarbeitung der Schritte 2 und 3. Bei Zeigerstand 1 beispielsweise deutet der Zeiger auf das letzte Zeichen im Absuchpuffer. Dieses stimmt mit dem `Startsymbol` überein. Daher wird Schritt 3 durchgeführt und entdeckt, dass die Länge `L` der längsten Folge, die bei `Startsymbol` im Codier-Puffer beginnt und mit der beim Zeiger im Absuch-Puffer beginnenden Folge übereinstimmt, gleich 1 ist. Daher ist nun `L` gleich Eins und das erste auf diese Weise uncodierte Zeichen im Codier-Puffer ist b und wird in `S` gespeichert. Ferner wird in `offset` der (letzte „erfolgreiche") Zeigerstand vermerkt. Die Zeigerstände 2 und 3 in den beiden folgenden Zeilen melden keine „Treffer", weshalb die übrigen Variablen unverändert bleiben. Bei Zeigerstand 4 hingegen wird wieder ein a gefunden. Jetzt ist die längste übereinstimmende Zeichenfolge ab, und die Inhalte der Variablen werden entsprechend angepasst. In dieser Weise fortfahrend, wird $abba$ als längstes Teilstück bei Zeigerstand

7 gefunden und der entsprechende Code nach Abarbeitung des gesammten Absuch-Puffers ausgegeben.

Wenn die Erzeugung eines Codewortes fertig ist, dann verschiebt man das ganze Fenster um $L+1$ Symbole, um ein neues Codewort mit dem skizzierten Algorithmus zu konstruieren.

Beispiel 4.2 Wir betrachten, wie der Algorithmus den folgenden Text codiert:

abaabbabaacaacabb`eof`

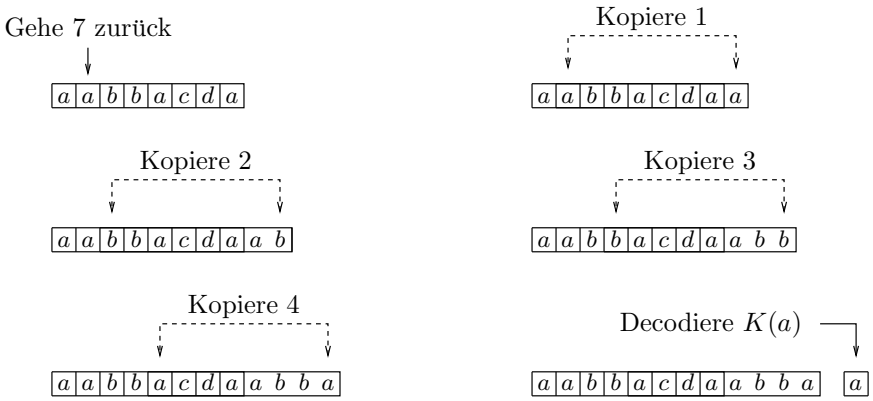
Das Zeichen `eof` steht dabei für das Textende (eigentlich Dateiende, engl.: end of file).

Inhalt der Puffer	*Generierter Code*
$\square \mid a\ b\ a\ a\ b \mid b\ a\ b\ a\ a\ c\ a\ a\ c\ a\ b\ b$	$(0, 0, K(a))$
$a \mid b\ a\ a\ b\ b \mid a\ b\ a\ a\ c\ a\ a\ c\ a\ b\ b$	$(0, 0, K(b))$
$a\ b \mid a\ a\ b\ b\ a \mid b\ a\ a\ c\ a\ a\ c\ a\ b\ b$	$(2, 1, K(a))$
$a\ b\ a\ a \mid b\ b\ a\ b\ a \mid a\ c\ a\ a\ c\ a\ b\ b$	$(3, 1, K(b))$
$a\ b\ a\ a\ b\ b \mid a\ b\ a\ a\ c \mid a\ a\ c\ a\ b\ b$	$(6, 4, K(c))$
$a\ b\ a\ a\ b \mid b\ a\ b\ a\ a\ c \mid a\ a\ c\ a\ b \mid b$	$(3, 4, K(b))$
$a\ b\ a\ a\ b\ b\ a\ b\ a\ a \mid c\ a\ a\ c\ a\ b \mid b$	$(1, 1, K(\texttt{eof}))$

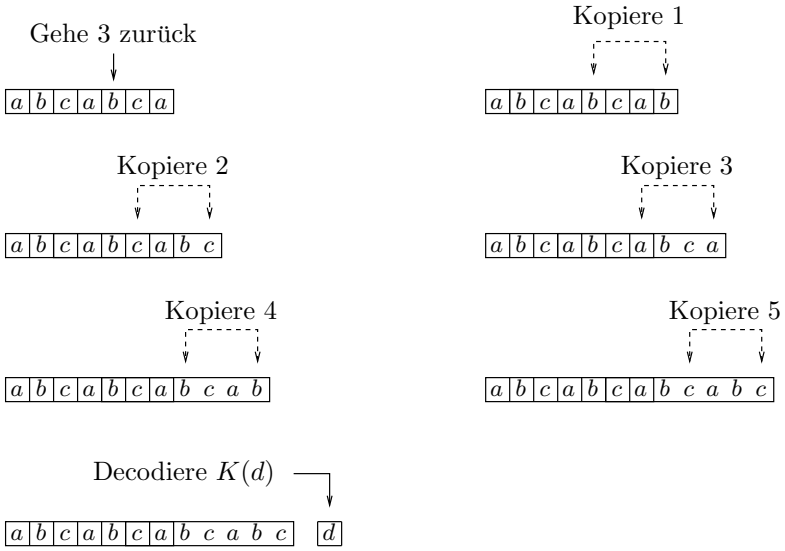
Beobachten Sie eine interessante Sache, die in Zeile 6 vorkommt: Die Zeichenfolge der Länge 4, die mit `offset` $= 3$ im Absuch-Puffer anfängt, endet im Codier-Puffer. Natürlich ist eine solche Situation sehr günstig, denn, je länger die Zeichenfolgen, desto kürzer der gesamte Code. Das Problem ist aber: Können wir solche überlappende Zeichenfolgen eindeutig decodieren? Wir zeigen unten, dass das Decodierungsverfahren für LZ77 auch in solchen Fällen korrekt funktioniert.

Das Decodierungsverfahren läuft entsprechend. Anstelle einer formalen Beschreibung zeigen wir zwei Beispiele, in denen die Länge der Absuch-Puffer 7 ist. Nehmen wir an, dass der Inhalt des Absuch-Puffers sowie der nächste Code die Eingaben des Verfahrens sind.

Beispiel 4.3 Der Inhalt des Absuch-Puffer ist: *aabbacda* und der Code: $(7, 4, K(a))$.



Beispiel 4.4 Der Inhalt des Absuch-Puffers ist: *abcabca* und der Code: $(3, 5, K(d))$ (die Decodierung für eine überlappende Zeichenfolge).



LZSS ist eine Variante von LZ77, die im Ausgabecode keinen Code für das nächste Zeichen angibt. Jetzt hat jeder Ausgabecode eins von zwei Formaten:

- $(0, \texttt{offset, Länge})$ oder
- $(1, \text{Zeichen})$.

Typischer Weise konstruiert LZSS den Code des ersten Formats, wenn die `Länge` größer als 2 ist, und dann benutzt der Algorithmus die Huffman-Codierung, um die Zeichenkette, bestehend aus ‘Ausgabecode-Symbolen’

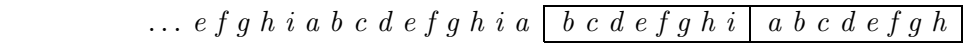
der beiden beschriebenen Formate, zu codieren. In der Praxis ist die Länge des Fensters gleich 32.

Beispiel 4.5 Wir betrachten zwei kleine Beispiele für LZSS:
Eingabe: *a a a a a a a*
Ausgabecode: $(1, a)$ $(0, 1, 6)$ $(1, \texttt{eof})$

Eingabe: *a a a b a b a b a*
Ausgabecode: $(1, a)$ $(1, a)$ $(1, a)$ $(1, b)$ $(0, 2, 5)$ $(1, \texttt{eof})$

Man beachte, dass der Ausgabecode im zweiten Beispiel *nicht* $(1, a), (0, 1, 2), (1, b) \ldots$ lautet, denn dann wäre die `Länge` ja nicht größer als 2.

LZ77 (und seine Varianten) bearbeitet besonders schlecht Eingaben, in denen sich Zeichenfolgen befinden, die sich im Eingabetext in größerem Abstand wiederholen, als die Länge des Fensters erfassen kann, so wie im folgenden Text:



Der Algorithmus kann die Zeichenfolgen, die sich im Text oft wiederholen, nicht komprimieren (wegen der Länge des Fensters) und er konstruiert für diesen Fall eine sehr uneffiziente Ausgabe. Die nächste Algorithmen werden Texte mit solchen Eigenschaften viel besser bearbeiten.

Wir betrachten zuerst das grundlegende **LZ78** Komprimierverfahren. Dieser Algorithmus speichert die Informationen über die Zeichenfolgen, die er im Eingabetext findet, in der Tabelle `Wörterbuch`. Am Anfang ist diese Tabelle leer, und dann speichert der Algorithmus im `Wörterbuch` sukzessiv neue Zeichenfolgen und gleichzeitig generiert er die Ausgabecodes für diese Folgen. Jeder Code hat das Format (i, k), wobei i der Index zum `Wörterbuch` ist und k der Code eines einzelnen Zeichens. Das heißt, dass jedes Paar (i, k) eine Zeichenfolge codiert, die eine Verkettung zweier Wörter ist: das erste Wort mit der Länge ≥ 0 und das zweite mit der Länge 1. Um den neuen Code zu generieren, findet der LZ78-Algorithmus im Eingabetext die längste Zeichenfolge w, die schon im `Wörterbuch` gespeichert wurde, und nimmt als i den Index für w. Wenn das Wort w leer ist, dann nehmen wir $i = 0$ an. Dann liest der Algorithmus das nächste Zeichen Z, fügt wZ dem `Wörterbuch` hinzu und gibt $(i, K(Z))$ aus. Für $K(Z)$ nehmen wir den Code für das Zeichen Z an so lange, bis der Algorithmus das erste Mal die Zeichenfolge wZ betrachtet, wobei w ein Leerwort ist. Dann ist $K(Z)$ gleich dem Index der Tabelle `Wörterbuch` für dieses Zeichen.

Beispiel 4.6 Das folgende Beispiel zeigt, wie der LZ78-Algorithmus mit der Eingabe *abababcabac* funktioniert.

Text	Wörterbuch Index	Wörterbuch Eintrag	Ausgang Code
a	1	a	$(0, K(a))$
b	2	b	$(0, K(b))$
a			
b	3	ab	$(1, 2)$
a			
b			
c	4	abc	$(3, K(c))$
a			
b			
a	5	aba	$(3, 1)$
c	6	c	$(0, K(c))$
eof			

Der Dekomprimieralgorithmus für die LZ78-Methode läuft ganz einfach: Er startet mit der leeren Tabelle `Wörterbuch` und dann rekonstruiert er sie.

Das **LZW** Komprimierverfahren, das Terry Welch 1984 entwickelt hat, ist eine Variante von LZ78. LZW gibt im Ausgabecode keinen Code für die Zeichen an, und als Codewörter benutzt der Algorithmus direkt die Indizes des Wörterbuches. Das Verfahren startet mit dem Wörterbuch, das die einzelnen Standardzeichen enthält. Nehmen wir an, dass wir 256 solche Zeichen betrachten werden und dass die Indizes der Tabelle `Wörterbuch` mit 0 anfangen. Der Codier-Algorithmus sieht folgendermaßen aus:

```
w = NIL
while (Z = readchar()) {
    if wZ in Wörterbuch
        w = wZ
    else {
        gib Code für w aus
        füge wZ ins Wörterbuch
        w = Z }
}
gib Code für w aus
```

Eine wichtige Eigenschaft ist, dass der Codier-Algorithmus einen Code für ein Wort `wZ` genau dann ausgibt, wenn er früher mindestens einmal das Wort im Text getroffen hat. Diese Eigenschaft spielt eine entscheidende Rolle bei der Decodierung.

Beispiel 4.7 Das folgende Beispiel zeigt, wie der Algorithmus mit der Eingabe

abababcabac

funktioniert —vergleichen Sie LZW mit LZ78 bei gleicher Eingabe, das haben wir früher betrachtet.

Text	`w`	`Z`	Neuer Eintrag im `Wörterbuch` Eintrag	Index	Ausgang Code
a		*a*			
b	*a*	*b*	*ab*	256	*a*
a	*b*	*a*	*ba*	257	*b*
b	*a*	*b*			
a	*ab*	*a*	*aba*	258	(256)
b	*a*	*b*			
c	*ab*	*c*	*abc*	259	(256)
a	*c*	*a*	*ca*	260	*c*
b	*a*	*b*			
a	*ab*	*a*			
c	*aba*	*c*	*abac*	261	(258)
`eof`	*c*				*c*

Das zweite Beispiel zeigt, wie der LZW einen Text mit vielen überlappenden Zeichenfolgen codiert.

Beispiel 4.8 LZW mit überlappenden Zeichenfolgen:

Text	`w`	`Z`	Neuer Eintrag im `Wörterbuch` Eintrag	Index	Ausgang Code
a		*a*			
a	*a*	*a*	*aa*	256	*a*
a	*a*	*a*			
a	*aa*	*a*	*aaa*	257	(256)
a	*a*	*a*			
a	*aa*	*a*			
a	*aaa*	*a*	*aaaa*	258	(257)
`eof`	*a*				*a*

Um den Code zu dekomprimieren, startet der Algorithmus mit der Tabelle `Wörterbuch`, die die einzelnen Standardzeichen enthält (wie beim Codier-Verfahren) und dann rekonstruiert er die Tabelle `Wörterbuch`. Der Decodier-Algorithmus sieht folgendermaßen aus:

```
v = Wörterbuch(readcode())
output v
while (C = readcode()) {
    if C in Wörterbuch
        w = Wörterbuch(C)
    else {
        K = v[0]  /* v[0] ist das erste Zeichen in v
    w = vK
    }
    output w
    Z = w[0]
    füge vZ ins Wörterbuch
    v = w
}
```

5 Weitere Verfahren

Im folgenden diskutieren wir drei weitere für die (verlustfreie) Kompression wichtige Verfahren als Ergänzung zum bisher Gesagten.

5.1 Lauflängencodierung

Sind in Texten (eher aber in Bildern, insbesondere Schwarz-Weiß-Bildern) lange Sequenzen gleicher Zeichen zu erwarten, so empfiehlt sich die *Lauflängencodierung*: Bei ihr wird neben dem Code für ein Zeichen (der evtl. per Huffman-Codierung erzeugt wurde) mitgeteilt, wie oft sich das Zeichen wiederholt. Bei Bildern sind oft ganze Bereiche schwarz oder weiß, was sich in langen Ketten von Nullen oder Einsen manifestiert. Bei dieser Anwendung muss man auch die Zeichen selbst nicht mehr (codiert) übertragen, da nur zwei verschiedene stets abwechselnd in Frage kommen. Nur noch die Lauflängen selbst müssen gesendet werden. Dieser Gedanke wird uns bei der Facsimile-Codierung (Abschnitt 6.2) wieder begegnen.

Eine Abwandlung dieses Gedankens findet Anwendung, wenn bekannt ist, dass in den vorliegenden zu übertragenden Daten lange Folgen von einer Sorte Zeichen vorkommen, gemischt mit anderen Zeichensorten, die typischer Weise nicht als lange Sequenzen vorkommen. Dann kann man je nach Zeichensorte Lauflängencodierung oder z. B. Huffmancodierung angwenden. Diese Idee wird uns insbesondere bei transformierten Datensätzen wieder begegnen.

5.2 Adaption durch Kontext: ppm

Cleary und Witten haben dieses bekannteste *kontext-basierte Verfahren* 1984 ersonnen. Wir werden im folgenden weniger die (obschon wichtigen) Feinheiten von *ppm-Algorithmen* (ppm steht für engl.:„Prediction with partial match") erläutern als vielmehr ihre Prinzipien darlegen, da ppm-Algorithmen zunehmend an Popularität gewinnen.

Vereinfacht sind ppm-Codierer Verfahren, die eine adaptive arithmetische Codierung nicht nur für einzelne Zeichen, sondern auch für Zeichen in ihrem *Kontext* (einer vorgegebenen Maximallänge) verwenden. Adaption bei arithmetischer Codierung bedeutet, dass die bisher beobachteten relativen Häufigkeiten als Grundlage für den Entwurf des *Wahrscheinlichkeitsmodells* genommen werden.

Bei der Codierung wird stets zunächst versucht, möglichst lange Kontexte anzunehmen. Dies bedeutet, dass zuerst ein Wahrscheinlichkeitsmodell

höchster Ordnung gewählt wird. (Die Ordnungsnummer des Modells ist dabei einfach die Länge des Kontextes.) Schlägt diese Annahme fehl (was dies bedeutet, wird an den folgenden Beispielen klar werden), geht der Codierer auf das Wahrscheinlichkeitsmodell nächstniederer Ordnung über. Diesen Übergang signalisiert er dem Empfänger durch Übermittlung eines Escape-Zeichens. Die verschiedenen Varianten von ppm unterscheiden sich im wesentlichen in der unterschiedlichen Behandlung der Zählung des Escapes. Wir erläutern im folgenden das einfachste Verfahren, ppma genannt. Hierbei wird in den Wahrscheinlichkeitsmodellen n-ter Ordnung (für $n \geq 0$) stets angenommen, dass das Escape mit Häufigkeit Eins gezählt wird. Um das Versagen des Modells bei „neuen" Zeichen zu verhindern, wird auch ein weiteres, nichtadaptives Wahrscheinlichkeitsmodell eingeführt, in dem sämtlichen Zeichen des Grundalphabets die selbe Wahrscheinlichkeit zugesprochen wird; dieses Modell erhält die Ordnungsnummer -1.

Betrachten wir die Codierung des folgenden Satzes über dem Alphabet $\{a, d, l, r, s, \_\}$:

$$lara\_lass\_das\_da$$

Als Kontextlänge wollen wir maximal Eins zulassen (was eine sehr kleine Zahl ist für die Praxis).

Das sogenannte Wahrscheinlichkeitsmodell (-1)-ter Ordnung ist nicht-adaptiv und ordnet der Einfachheit halber jedem der Zeichen die selbe Wahrscheinlichkeit zu, also in unserem Fall 1/6.

Das Wahrscheinlichkeitsmodell (oder besser Zählmodell) nullter Ordnung besteht nach dem Einlesen von $lara\_lass\_$ aus:

Buchstabe	Anzahl	kumulierte Anzahl	rel. Häufigkeit
a	3	3	3/11
l	2	5	2/11
r	1	6	1/11
s	2	8	2/11
_	2	10	2/11
ESC	1	11	1/11

Das Wahrscheinlichkeitsmodell (oder besser Zählmodell) erster Ordnung besteht nach dem Einlesen von $lara\_lass\_$ aus:

Kontext	Buchstabe	Anzahl	kumulierte Anzahl	rel. Häufigkeit
l	a	2	2	2/3
l	ESC	1	3	1/3
a	r	1	1	1/4
a	s	1	2	1/4
a	_	1	3	1/4
a	ESC	1	4	1/4
r	a	1	1	1/2
r	ESC	1	2	1/2
_	l	1	1	1/2
_	ESC	1	2	1/2
s	s	1	1	1/3
s	_	1	2	1/3
s	ESC	1	3	1/3

Wie man sieht, handelt es sich eigentlich um eine ganze Reihe von Wahrscheinlichkeitsmodellen, je eines für jeden bislang vorkommenden Kontext.

Als nächstes Zeichen hätten wir ein d zu verschlüsseln, und zwar im Kontext _. Bislang hatten wir diese Situation nicht vorgefunden, weshalb wir das Escape codieren. Ein Blick in die obige Tabelle lehrt, dass dies eine Übertragung des Intervalls $[1/2, 1)$ erfordert.[5] Gehen wir zum Kontext nullter Ordnung über, so bemerken wir, dass ein d bislang noch gar nicht beobachtet wurde, so dass wir wiederum ein Escape codieren. Hier bedeutet dies die Übertragung des Intervalls $[10/11, 1)$. Schließlich erfordert die Verwendung des Modells (-1). Ordnung die Übertragung des Intervalls $[1/6, 2/6)$. Natürlich müssen wir jetzt auch noch die Modelle nullter und erster Ordnung entsprechend anpassen.

Bei der Abarbeitung des folgenden Zeichens a versagt wiederum das Modell erster Ordnung, so dass ein Escape codiert werden muss. Das Modell nullter Ordnung gestattet jedoch die Codierung des a's, und zwar als Intervall $[0, 3/12)$. Man beachte, dass jetzt 12 Symbole (und nicht mehr 11 wie zuvor) insgesammt zu Grunde liegen.

Schließlich kommt bei der nachfolgenden Übertragung des s das Modell erster Ordnung zur Anwendung; es ist das Intervall $[1/4, 2/4)$ zu senden.

Es ist klar, das der Decodierer auf der Empfangsseite dieselben Statistiken führen muss und kann wie der Sender, so dass eine Entschlüsselung der gesendeten Daten leicht möglich ist.

Auf eine wichtige Feinheit und damit Varianten von ppm(a) wollen wir noch eingehen, nämlich das *Ausschlussprinzip*. Betrachten wir dazu nochmals im

[5] Hierzu verweisen wir auf den Abschnitt über arithmetische Codierung.

vorigen Beispiel die Übertragung des d's. Die Tatsache, dass eine Codierung mit dem Modell erster Ordnung gescheitert war, bedeutet doch, dass das zu übertragende Zeichen kein l sein kann. Entsprechend bedeute das Scheitern der Anwendung des Modells nullter Ordnung, dass kein a, l, r, s oder _ zu übetragen ist, so dass tatsächlich ein modifiziertes Modell (-1)ter Ordnung angewendet werden könnte, bei dem im konkreten Fall nur noch das triviale Gesamtintervall $[0, 1)$ zu codieren wäre. Beim a ist wiederum das Modell erster Ordnung gescheitert, was aber keine weiteren Ausschlüsse erlaubt, da jetzt das erste Mal der Kontext d beobachtet wurde.

5.3 Burrows-Wheeler-Transformation (BWT)

Die Idee des von Burrows und Wheeler 1994 vorgeschlagenen Kompressionsverfahrens besteht darin, durch (spezielles) Sortieren des zu codierenden Textes eine Folge zu erhalten, die viel mehr Struktur als der Urtext enthält und somit effizient (mit Hilfe irgendeines anderen Verfahrens) verschlüsselbar ist. Um weiterhin verlustfrei zu übertragen, muss neben dem sortierten Text noch Zusatzinformation gesendet werden. Im Gegensatz zu den anderen bislang vorgestellten Verfahren ist der Aufwand BWT-basierter Kompression (mindestens) quadratisch.

Konkret arbeitet der Transformationsalgorithmus wie folgt: Gegeben ein Text der Länge N, werden zunächst $N - 1$ Folgen durch zyklisches Vertauschen der Zeichen erzeugt. Diese insgesamt N Folgen je der Länge N werden nun lexikographisch sortiert.

Erinnerung: *lexikographische Ordnung* für Wörter gleicher Länge bedeutet, dass, ausgehend von einer linearen totalen Ordnung des Grundalphabets, $u = u_1 \dots u_N < v = v_1 \dots v_N$ gilt genau dann, wenn es ein $J \in \{1, \dots, N\}$ gibt, so dass $u_j = v_j$ für alle $j < J$ sowie $u_J < v_J$ gilt. Die so erzeugten Folgen werden als Zeilen einer $N \times N$-Matrix A angesehen. Übertragen wird nun die letzte Spalte L von A sowie der Index I der Zeile von A, die den ursprünglichen Text enthält.

Wie kann der Empfänger aus L und I den ursprünglichen Text D wieder herausfinden? Zunächst ist klar, dass $L[I]$ das letzte Zeichen von D enthält, also $D[N] = L[I]$ gilt. Außerdem kann man durch Sortieren von L leicht die Abfolge der Zeichen in der ersten Spalte F von A bekommen. Diese Informationen kann man dazu benutzen, um D vollständig „von hinten nach vorne" zu rekonstruieren. Dazu hilfreich ist eine Permutation T von $\{1, \dots, N\}$ mit der Eigenschaft $F[T[j]] = L[j]$ für alle $j \in \{1, \dots, N\}$. Hat man T gefunden, so leistet

$k \leftarrow I;$

`for` $j = 1$ `to` $N - 1$ `do` $k \leftarrow T[k]; D[N - j] \leftarrow L[k]$ `od`

die gewünschte Entschlüsselung. Dazu beobachte man, dass in der ursprünglichen Folge D $L[j]$ unmittelbarer Vorgänger von $F[j]$ ist, wenn man sich D zyklisch angeordnet denkt. Also ist auch $L[T[j]]$ unmittelbarer Vorgänger von $F[T[j]] = L[j]$. Daher buchstabiert sich D rückwärts als $L[I]$, $L[T[I]]$, $L[T[T[I]]]$, usw.

Im Übrigen liefert die Beobachtung, $L[j]$ ist unmittelbarer Vorgänger von $F[j]$, auch die Begründung dafür, dass die BWT eine Folge L mit langen Lauflängen einzelner Zeichen liefert, da in einer größeren Matrix A Zeilen mit gleichen Anfangssequenzen ja hintereinander stehen und quasi den Kontext für das (als letztes Zeichen in der Zeile aufgeführte) Vorgängerzeichen liefern, sodass gleiche Anfangssequenzen aufeinanderfolgender Matrixzeilen mit hoher Wahrscheinlichkeit hintereinander folgende gleiche Zeichen in L zeitigen. Daher ist eine Lauflängencodierung von L sinnvoll.

Wie lässt sich T aus L (sowie F) gewinnen? Wollen wir beispielsweise $T[1]$ bestimmen, so hilft uns die Gleichung $F[T[1]] = L[1]$ sicherlich, ein Intervall von Indizes auszumachen, die als Kandidaten von $T[1]$ in Frage kommen. Beachte, dass wir tatsächlich ein Intervall von Indizes erhalten, da A und damit F ja lexikographisch sortiert ist. Aus Sortierungsgründen ist es nun sinnvoll, in dem konkreten Fall den Index des Intervallanfangs als $T[1]$ anzusetzen. Ist im allgemeinen $L[j] = A$ das k-te Vorkommen von A in $L[1] \dots L[j]$, so wähle man das k-te Zeichen im Intervall $a = F[j_1], \dots, a = F[j_2]$ zur Bestimmung von $T[j]$.

Wir erläutern dies Verfahren wieder an einem Beispiel.

Beispiel 5.1 Wir wollen *lara_lass* codieren. Die sortierte 9×9-Matrix sieht wie folgt aus:

$$\begin{array}{ccccccccc}
a & r & a & _ & l & a & s & s & l \\
a & s & s & l & a & r & a & _ & l \\
a & _ & l & a & s & s & l & a & r \\
l & a & r & a & _ & l & a & s & s \\
l & a & s & s & l & a & r & a & _ \\
r & a & _ & l & a & s & s & l & a \\
s & l & a & r & a & _ & l & a & s \\
s & s & l & a & r & a & _ & l & a \\
_ & l & a & s & s & l & a & r & a
\end{array}$$

Daher wird die Folge

$$L = llrs\_asaa$$

und der Index $I = 4$ des ursprünglichen Textes übertragen. Wir berechnen jetzt (zur Decodierung) die Transformation T aus diesen Informationen.

Zuerst können wir $F = aaallrss\_$ durch Sortieren erzeugen. Dann bestimmen wir die Transformation „rückwärts”:

$$\begin{aligned}
L[1] = l = F[4] &\rightarrow T[1] = 4 \\
L[2] = l = F[5] &\rightarrow T[2] = 5 \\
L[3] = r = F[6] &\rightarrow T[3] = 6 \\
L[4] = s = F[7] &\rightarrow T[4] = 7 \\
L[5] = _ = F[9] &\rightarrow T[5] = 9 \\
L[6] = a = F[1] &\rightarrow T[6] = 1 \\
L[7] = s = F[8] &\rightarrow T[7] = 8 \\
L[8] = a = F[2] &\rightarrow T[8] = 2 \\
L[9] = a = F[3] &\rightarrow T[9] = 3
\end{aligned}$$

Dann können wir den ursprünglichen Text als

$$L[I] = L[4] = s, \quad L[T[4]] = L[7] = s, \quad L[T[7]] = L[8] = a, \ldots$$

uns von hinten kommend buchstabieren.

6 Verlustfreie Komprimierung von Bildern

Die große Effizienz der Arithmetischen Codierung für Texte folgt daraus, dass Zeichenfolgen (der selben Länge) ungleichmäßig verteilt sind. Auch die Wörterbuch-Techniken nutzen die Eigenschaft von Texten, dass sich viele Muster sehr häufig wiederholen. Um diese Methoden zu entwickeln, brauchten wir deshalb nicht nach neuen Modellen für Texte zu suchen.

Leider haben andere Datentypen, wie z. B. Sprache oder Bilder, keine so schönen Eigenschaften wie Texte. Bilder beispielsweise (oder genauer gesagt die Dateien mit den Bitmaps der Bilder) haben eine ganz andere Struktur als Textdateien. Ein Grund dafür ist einfach: Sprache (und insbesondere eine künstliche Sprache, wie z. B. eine Programmiersprache) wird mit Regeln generiert, die einen bestimmten Einfluss auf die Struktur der Textdatei haben. Im Falle von Bildern haben wir es mit ganz anderen Regeln zu tun. Deshalb brauchen wir, um gute Komprimierungsverfahren für Bilder zu entwickeln, zuerst ein gutes Modell, das den Eigenschaften der Bilder besser entspricht als ein Quellenalphabet mit Verteilung für Zeichen (bzw. Pixel im Bildfall).

In diesem Kapitel beschreiben wir das Markov-Modell für binäre Bilder und dann betrachten wir das Problem der verlustfreien Facsimile-Codierung und die fortschreitende Bildübertragung (engl.: progressive image transmission).

6.1 Bedingte Entropie und das Markov-Modell

In diesem Abschnitt diskutieren wir die *bedingte Entropie* –einen Begriff, den wir für verlustfreie Komprimierung von Bildern benutzen werden. Später zeigen wir, dass der Begriff auch für verlustbehaftete Komprimierung sehr nützlich ist.

Seien A und B zwei Ereignisse und $P(A|B)$ die bedingte Wahrscheinlichkeit, dass A eintritt unter der Bedingung, dass auch B eintritt (bzw. zeitlich interpretiert bereits eingetreten ist). Dann definieren wir den *bedingten Informationsgehalt* folgendermaßen:

$$i(A|B) = \log_2 \frac{1}{P(A|B)}.$$

Wie im Fall des Informationsgehaltes ist der Begriff intuitiv klar. Sei jetzt S eine bestimmte Informationsquelle und T ein Kontext, in dem die Nachrichten aus S vorkommen. Dann ist die bedingte Entropie $H(S|T)$ der "mittlere Informationsgehalt je Nachricht aus S mit dem Kontext T":

$$H(S|T) = \sum_{t \in T} P(t) \sum_{s \in S} P(s|t) \log_2 \frac{1}{P(s|t)}.$$

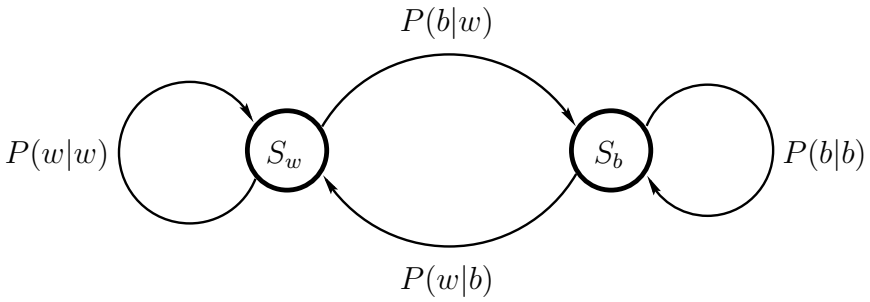


Abbildung 8: Markov-Modell für Schwarz-Weiß-Bilder

Wie früher werden wir meistens S als endliches Quellenalphabet Σ betrachten. Als Kontext T werden wir eine endliche Teilmenge von Σ^* annehmen. Es ist nicht schwer zu zeigen, dass $H(S|T) = H(S)$ gilt, wenn S unabhängig vom Kontext T ist; ansonsten haben wir: $H(S|T) < H(S)$. Die Kenntnis des Kontextes für S reduziert also die Entropie.

Wir stellen die zugehörigen bedingten Wahrscheinlichkeiten mit Hilfe des Markov-Modells dar. Um das Konzept zu verstehen, betrachten wir das folgende Beispiel. Nehmen wir an, dass das Bild nur zwei Sorten von Pixeln hat: weiß und schwarz. Das heißt, unser Quellenalphabet ist $\{w, b\}$. Als Kontext für ein Pixel x werden wir einfach das Pixel betrachten, das eine Position früher in der Zeile steht. Entsprechende Wahrscheinlichkeiten sind folgende:

$$
\begin{aligned}
P(w|w) &= \text{Wahrscheinlichkeit ein weißes Pixel zu erhalten,} \\
&\quad \text{wenn das vorherige Pixel weiß war;} \\
P(w|b) &= \text{Wahrscheinlichkeit ein weißes Pixel zu erhalten,} \\
&\quad \text{wenn das vorherige Pixel schwarz war;} \\
P(b|b) &= \text{Wahrscheinlichkeit ein schwarzes Pixel zu erhalten,} \\
&\quad \text{wenn das vorherige Pixel schwarz war;} \\
P(b|w) &= \text{Wahrscheinlichkeit ein schwarzes Pixel zu erhalten,} \\
&\quad \text{wenn das vorherige Pixel weiß war.}
\end{aligned}
$$

Diese Wahrscheinlichkeiten können wir als Markov-Modell gemäß Bild 8 darstellen. Hier entspricht S_w, bzw. S_b, dem Zustand, dass wir ein weißes, bzw. schwarzes Pixel lesen.

Betrachten wir jetzt ein ganz konkretes Beispiel:

Beispiel 6.1 Nehmen wir die folgenden Wahrscheinlichkeiten an:

$$\begin{aligned} P(w) = P(S_w) &= 0,85 \qquad & P(b) = P(S_b) &= 0,15 \\ P(w|w) &= 0,99 & P(b|w) &= 0,01 \\ P(w|b) &= 0,2 & P(b|b) &= 0,8. \end{aligned}$$

Dann ist die Entropie für das Quellenalphabet $\Sigma = \{w, b\}$ gleich

$$H(\Sigma) \;=\; -0,85 \times \log 0,85 \;-\; 0,15 \times \log 0,15 \;=\; 0,610.$$

Also ist für dieses Modell (wobei wir keine Korrelation zwischen Pixeln beachten) ist die (bestmögliche) mittlere Anzahl von Bits gleich $0,61$, um eine Nachricht aus Σ zu codieren. Wir zeigen, dass diese Zahl für das Markov-Modell viel besser ist. Die entsprechende bedingte Entropie berechnen wir folgendermaßen:

$$\begin{aligned} H(\Sigma|\Sigma) \;&=\; 0,85 \times (-0,99 \times \log 0,99 \;-\; 0,01 \times \log 0,01) \;+ \\ &\qquad 0,15 \times (-0,2 \times \log 0,2 \;-\; 0,8 \times \log 0,8) \\ &=\; 0,85 \times 0,081 + 0,15 \times 0,722 \;=\; 0,176. \end{aligned}$$

Für das Markov Modell ist die mittlere Anzahl der Bits gleich $0,176$, also um etwa 71% besser als früher (ohne Kontextberücksichtigung).

Die Codierung sieht folgendermaßen aus: Wir konstruieren separat eine Codierung für Σ im Kontext des weißen Pixels und eine Codierung für Σ im Kontext des schwarzen Pixels. Dann benutzen wir die erste Codierung, um die Pixel zu codieren, so lange wir im Zustand S_w bleiben; sonst nehmen die zweite. Das Decodierverfahren geht völlig entsprechend.

6.2 Facsimile Codierung (Faxen)

Die Aufgabe ist: Taste eine A4-Seite ab und komprimiere das Schwarz-Weiß-Bild, um es über eine Telefonleitung zu senden. Interessanter Weise waren Facsimile die ersten Anwendungen von Komprimierungsverfahren überhaupt. Diese Verfahren benutzen das Markov-Modell mit zwei Zuständen: S_w und S_b, wie wir es im vorherigen Abschnitt betrachtet haben. In diesem Abschnitt beschreiben wir zwei Methoden, die in der Praxis benutzt werden: das eindimensionale Schema **MH** (*modified Huffman*) und das zweidimensionale Schema **MR** (*modified READ*, wobei READ *relative element address designate* bedeutet).

Der MH-Algorithmus ist ganz einfach: Wir codieren mit Hilfe der Huffman-Codierung die *Lauflängen* (engl. *run length*), wobei ein *Lauf* die längste

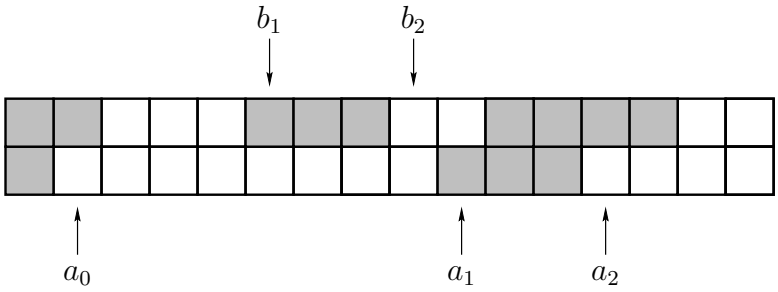


Abbildung 9: Pixelabhängigkeiten nach der MR-Methode

Reihenfolge des gleichen Pixels ist. Wir geben diese Länge für jede Zeile separat an und vermuten dabei, dass die erste Zahl den Lauf von weißen Pixeln codiert. Weil die Anzahl der Pixel je Zeile sehr groß ist (1728), können die Läufe auch so eine große Anzahl von verschiedenen Längen haben, d. h., das "Alphabet", welches wir nach dem Huffman-Verfahren zu codieren hätten, wäre sehr groß. Um die Huffman-Methode effizient zu nutzen, codieren wir jede Länge des Laufes r_l, der länger als 64 ist, folgendermaßen:

$$r_l = 64 \times m + t \qquad \text{for} \;\; m = 1, 2, \ldots, 27 \;\; \text{und} \;\; t = 0, 1, \ldots, 63$$

und stellen r_l als Paar $64 \times m$ und t dar. Wir nennen Codes für $64 \times m$ *Grobcodes* (engl.: make-up code) und Codes für t *Schlusscodes* (engl.: terminating code). Falls $r_l < 63$, benutzen wir nur den Schlusscode, um den Lauf zu codieren. Dann konstruieren wir den Huffman-Code für das Alphabet

$$\Sigma = \{64 \times 1,\ 64 \times 2,\ \ldots,\ 64 \times 27\} \cup \{0, 1, \ldots, 63\}$$

für die weißen Läufe und separat für das gleiche Alphabet Σ für die schwarzen Läufe. In dem *CCITT-Standard* (Consultative Committee on International Telephone and Telegraph, jetzt ITU-T: International Telecommunications Union) hat beispielsweise der schwarze Schlusscode für 2 die Länge 2 und für 63 die Länge 12. Für die weißen Schlusscodes sind diese Längen 4 bzw. 8. Es ist nämlich sehr wahrscheinlich, dass der schwarze Lauf die Länge 2 hat.

Der MR-Algorithmus nutzt noch mehr Korrelationen zwischen Pixeln als die MH-Methode. Das MR-Verfahren betrachtet nämlich die Abhängigkeiten in zwei Zeilen. Wir nennen das erste Pixel eines Laufes das *Übergangspixel*. Jetzt nehmen wir an, dass wir eine ganze Zeile codiert (bzw. decodiert) haben und dass wir nun die nächste Pixel-Zeile bearbeiten.

Wir definieren folgende Positionen für die Übergangspixel (als Beispiel, siehe Bild 9):

a_0: Das ist ein Pixel, das wir gerade betrachten. Wir nehmen an, dass jede Zeile mit einem weißen Lauf anfängt (wenn nicht, betrachten wir den weißen Lauf der Länge 0).

a_1: Das erste Übergangspixel rechts von Pixel a_0.

a_2: Das zweite Übergangspixel rechts von Pixel a_0.

b_1: Das erste Übergangspixel in der vorherigen Zeile, das eine andere Farbe als Pixel a_0 hat.

b_2: Das erste Übergangspixel rechts von Pixel b_1.

Das Codierungsverfahren kennt alle Positionen. Wenn wir decodieren, kennen wir nur b_1, b_2 und a_0, um das Ziel a_1 und a_2 zu finden.

Der Codier-/Decodier-Algorithmus betrachtet drei Fälle:

1. *Zwischen-Modus*: b_1 und b_2 liegen zwischen a_0 und a_1. Der Code ist 0001. Dann nimm Position $b_2 + 1$ als neuen Wert für a_0, suche neue Werte b_1 und b_2 und wiederhole den ganzen Prozess.

2. *Vertikal-Modus*: a_1 liegt vor b_2.

   2.1 Abstand zwischen a_1 und b_1 ≤ 3. Dann sind es folgende Codes:

Pos. a_1 bezüglich b_1	Code	Länge des Codes
b_1		
a_1	1	1
$b_1 a_1$	011	3
$b_1 \sqcup a_1$	000011	6
$b_1 \sqcup \sqcup a_1$	0000011	7
$a_1 b_1$	010	3
$a_1 \sqcup b_1$	000010	6
$a_1 \sqcup \sqcup b_1$	0000010	7

   2.2 Abstand zwischen a_1 und b_1 > 3. Dann sieht der Code folgendermaßen aus: erstens 001 und dann MH-Codes für zwei Läufe von a_0 nach $a_1 - 1$ und von a_1 nach $a_2 - 1$.

Das Problem der Fehlerfortpflanzung durch Codierungsabhängigkeiten zwischen den Zeilen wird durch häufiges, regelmäßiges Zwischenschreiben "neuer" MH-Codierungen angegangen.

6.3 Fortschreitende Bildübertragung

Betrachten wir die folgende Situation: Mit Hilfe des Browsers wollen wir 30 Bilder durchsehen. Wie es sehr häufig in der Praxis passiert, werden nicht alle Bilder für uns gleich wichtig sein: Nur manche werden für uns große Bedeutung haben und nur solche wollen wir auf dem Bildschirm haben. Nehmen wir an, dass jedes Bild schwarz-weiß ist und dass es die Größe 512×512 hat mit 8 Bits je Pixel. Wir wollen diese Bilder durch die Leitung mit der Geschwindigkeit 14.4 kbps übertragen und das heißt, dass die ganze Übertragung

$$30 \times (8 \times 512 \times 512) \ / \ 14\,400 \ = \ 62\,914\,560 \ / \ 14\,400 \ \approx \ 4\,369 \text{ s} \ \approx \ 73 \text{ min.}$$

dauert.
Jetzt modifizieren wir unser Verfahren so, dass wir statt der Bilder die Approximationen der Bilder übertragen. Die Approximationen haben eine viel kleinere Größe als die Bilder, aber trotzdem ist es immer möglich, ganz gut zu entscheiden, was das Bild zeigt. Deshalb können wir relativ schnell eine Übertragung für jene Bilder anhalten, die für uns uninteressant sind. Eine solche Übertragung (zuerst die Approximationen und auf diese Weise schließlich die verlustfreie Repräsentation der Bilder) heißt *fortschreitende Bildübertragung.*

Um eine Approximation zu bekommen, ist folgendes ein einfaches und schnelles Verfahren: Wir betrachten $b \times b$ Pixels des Bildes und dann stellen wir alle b^2 Pixel als ein Pixel dar. Das heißt, wir komprimieren mit dem Faktor b^2. Um das Bild zu übertragen, senden wir z.B. drei Approximationen (mit den Parametern $b = 8$, 4 und 2 wie unten) und die verlustfreie Codierung des Bildes nacheinander (selbstverständlich berücksichtigend, daß bereits Teile des Bildes übertragen wurden). Die entsprechenden Vergröberungen bzw. Verfeinerungen sind in Bild 10 dargestellt.

Jetzt können Sie mit Hilfe von Bild 11 vergleichen, welche Approximationen wir bekommen, wenn wir (für $b = 4$) als Repräsentanten für die $b \times b$ Pixel das im Quadrat links oben gelegene (siehe linkes Bild) oder den Medianwert aller b^2 Pixel (siehe rechtes Bild) nehmen. Das Original ist in Bild 12 dargestellt.

Wie effizient die fortschreitende Bildübertragung ist, können Sie in den Bildern 13 und 14 sehen, wo wir dieses Verfahren mit einer traditionellen Methode (verlustfreie Übertragung Zeile für Zeile) vergleichen. Die linken Approximationen haben die gleiche Größe (im Sinne der benötigten Bits zur Übertragung) wie die entsprechenden Teile des Originals auf der rechten Seite. Bild 13 ist für $b = 8$, Bild 14 zeigt $b = 4$ und in Bild 15 sehen Sie das Original-Bild.

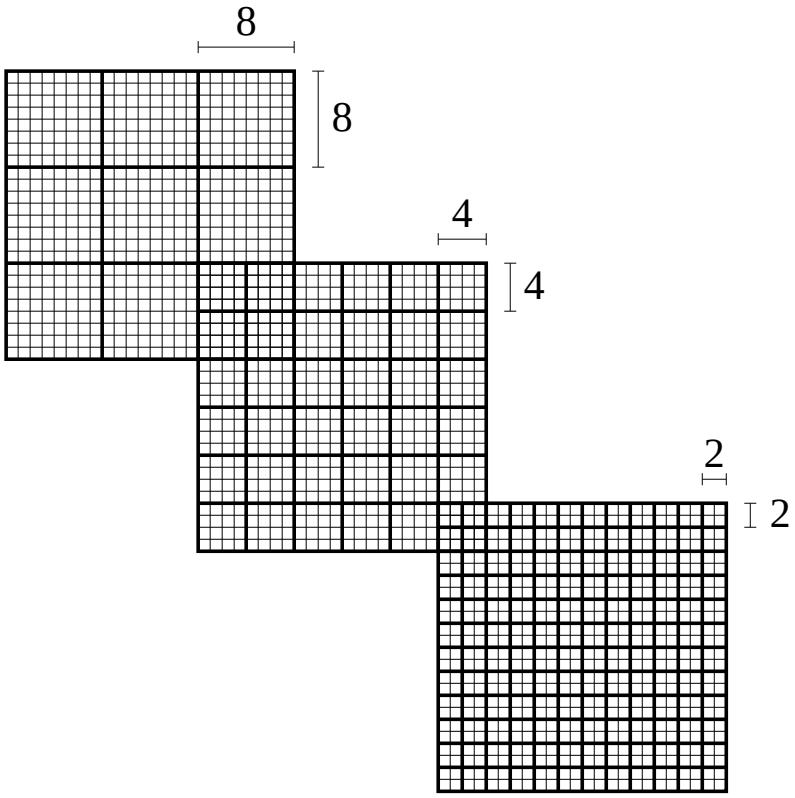


Abbildung 10: Fortschreitende Bildverfeinerung

Abbildung 11: Zwei “Vergröberungen” des Kameramanns

Abbildung 12: Der Kameramann im Original

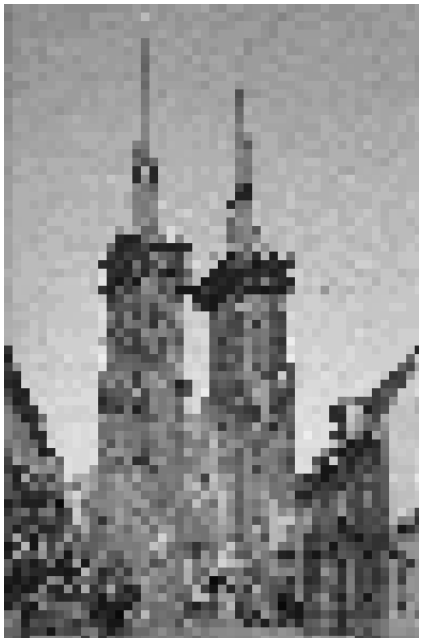


Abbildung 13: Ein Repräsentant für 8×8 Pixel

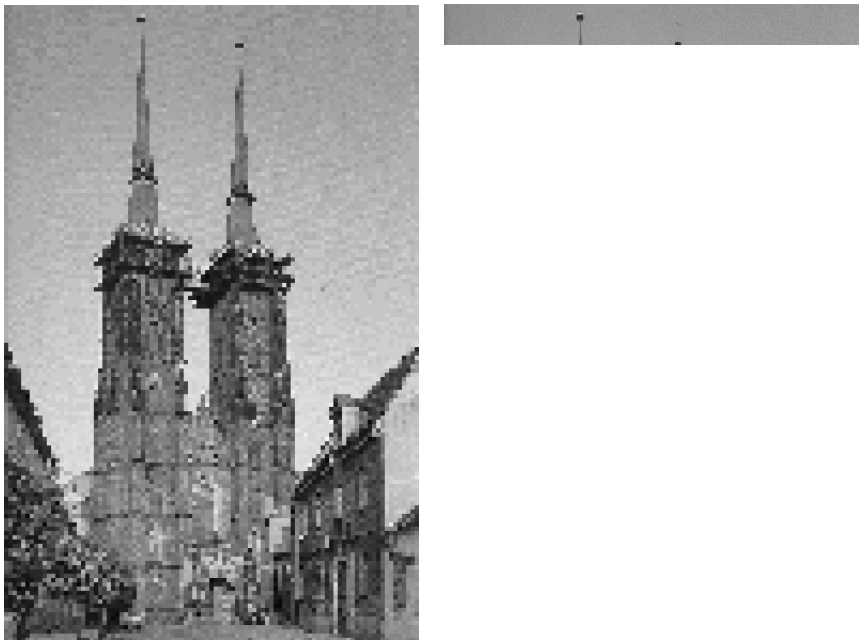


Abbildung 14: Ein Repräsentant für 4×4 Pixel

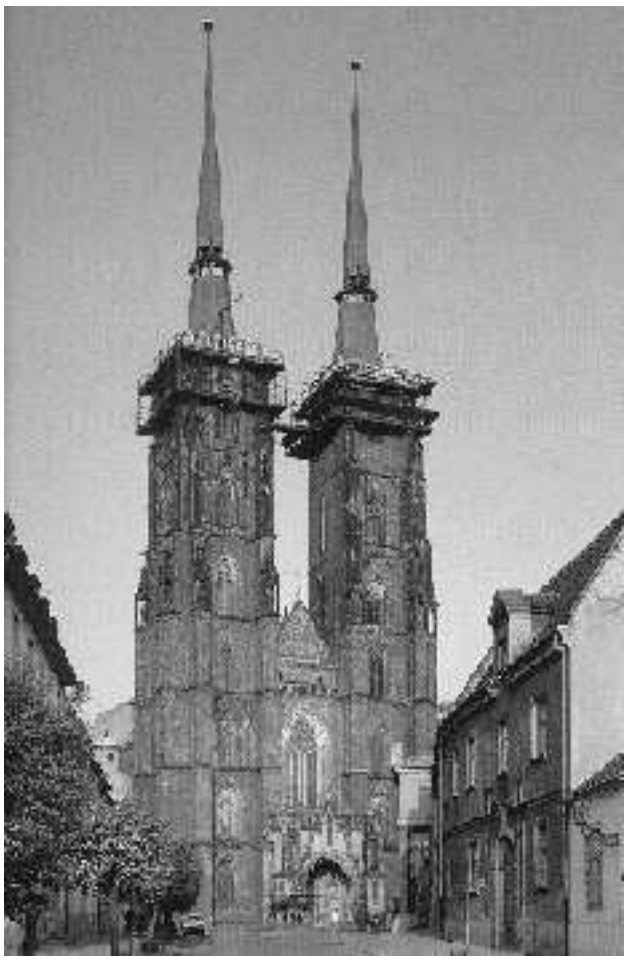


Abbildung 15: Der Dom im Original

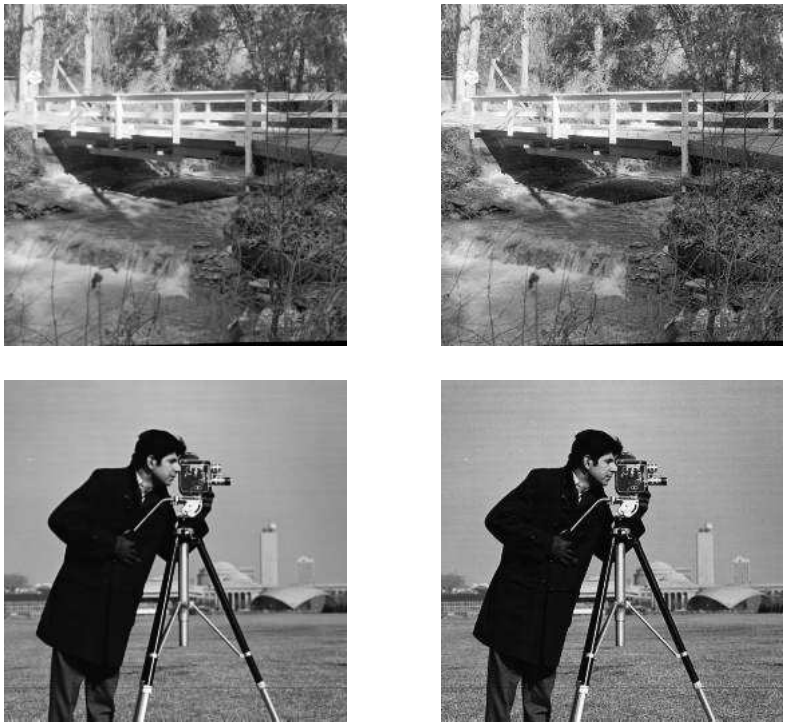


Abbildung 16: Zwei Beispiele zum verlustbehafteten JPEG-Verfahren

7 Verlustbehaftete Komprimierung: Grundlagen

Ein Verfahren A heißt verlustbehaftete Komprimierung, wenn man für die Eingabe $\mathcal{X}$ nicht von der Repräsentation $\mathcal{X}_c$, die A konstruiert, eine Ausgabe $\mathcal{Y}$ so rekonstruieren kann, dass $\mathcal{X} = \mathcal{Y}$. Solche Methoden benutzen wir, wenn es unmöglich ist, für $\mathcal{X}$ irgendeine „genaue" Repräsentation $\mathcal{X}_c$ zu bekommen: z. B., wenn $\mathcal{X}$ analog ist (Stimme) und wir die Repräsentation in digitaler Form haben wollen. Solche Methoden sind auch sehr nützlich, wenn wir einen (in der Praxis ganz kleinen) Unterschied zwischen $\mathcal{X}$ und $\mathcal{Y}$ tolerieren und den Kompressionsquotienten wesentlich kleiner haben wollen, als es die beste verlustfreie Methode bietet.

Beispiel 7.1 Betrachten wir als Beispiel die folgenden zwei 256×256 Schwarz-Weiß-Bilder, eine Brücke und eine Kamera zeigend, siehe Bild 16. Die linken Bilder sind die Originale, und die rechten Bilder zeigen die Rekonstruktionen, die wir mit Hilfe von verlustbehafteten JPEG-Verfahren bekommen. Unterschiede zwischen Original und „Fälschung" sind praktisch mit dem bloßen Auge nicht zu erkennen.

JPEG-Verfahren werden wir im Kapitel 13 genau diskutieren. Der Vorteil

des Verfahrens sehen Sie unten, wo wir die Größe für die Bit-Map und GIF Format (also für die genaue Repräsentationen von Bildern) mit der Größe der JPEG-Repräsentation vergleichen.

Bild	Bit Map	GIF	JPEG
Brücke	65 536	76 511	17 272
Kamera	65 536	55 484	10 889

In diesem Kapitel diskutieren wir die grundlegenden Begriffe für die verlustbehaftete Komprimierung. Es ist klar, dass wir jetzt nicht nur Kompressionsquotienten für die Algorithmen betrachten sollten, sondern auch ein Maß für den Unterschied zwischen $\mathcal{X}$ und $\mathcal{Y}$. Diesen Unterschied werden wir *Verzerrung* oder *Entstellung* (engl: distortion) nennen. Weiter unten wird dieser Begriff im Zuge der Diskussion von Wahrscheinlichkeitsmodellen noch technisch enger gefasst werden.

Sei $\mathcal{X} = \{x_n\}_{n=1}^N$ und $\mathcal{Y} = \{y_n\}_{n=1}^N$. Typische *Verzerrungsmaße* sind:

$$d(x_n, y_n) = (x_n - y_n)^2$$

das sogenannte *quadratische Fehlermaß* („Methode der kleinsten Quadrate") und

$$d(x_n, y_n) = |x_n - y_n|$$

das *Betragsfehlermaß*. Für diese Maße können wir den Unterschied zwischen $\mathcal{X}$ und $\mathcal{Y}$ folgendermaßen messen:

$$\sigma^2 = \frac{1}{N} \sum_{n=1}^{N} (x_n - y_n)^2$$

beziehungsweise

$$d_1 = \frac{1}{N} \sum_{n=1}^{N} |x_n - y_n| \ .$$

σ^2 heißt *mittlerer quadratischer Fehler* und d_1 *mittlerer Betragsfehler*. Für $\mathcal{X}$ und $\mathcal{Y}$ mit $\mathcal{X} = \mathcal{Y}$ haben wir $\sigma^2 = d_1 = 0$. Für unsere Beispiele betrachten Sie die Tabelle unten.

$\mathcal{X}$	$\mathcal{Y}$	σ^2	d_1
Brücke	JPEG-Repräsent. für Brücke	44,20	5,07
Kamera	JPEG-Repräsent. für Kamera	22,34	3,07
Brücke	Kamera	6 674,66	66,93

Als relatives Fehlermaß für verlustbehaftete Komprimierung $\mathcal{Y}$ von $\mathcal{X}$ benutzen wir das *Verhältnis von Signal zu Verzerrung* (engl. *signal-to-noise ratio*, kurz *SNR*), $SNR = \frac{\sigma_X^2}{\sigma^2}$, wobei σ_X^2 der mittlere quadratische Wert der Quellenausgabe (also des Signals) und σ^2 der mittlere quadratische Fehler ist. SNR wird häufig logarithmisch skaliert in Dezibel angegeben, also

$$SNR(dB) = 10 \log_{10} \frac{\sigma_X^2}{\sigma^2}.$$

Es sei jetzt X eine Zufallsvariable für das Eingabesymbol aus dem Quellenalphabet $\{x_0, x_1, \ldots, x_{N-1}\}$ und Y eine Zufallsvariable für das Ausgabesymbol aus $\{y_0, y_1, \ldots, y_{M-1}\}$. Wir nennen $\log[P(y_j|x_i)/P(y_j)]$ die *wechselseitige Information* (engl.: mutual information) und wir definieren die *durchschnittliche wechselseitige Information* folgendermaßen:

$$I(Y;X) \quad = \quad \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} P(x_i, y_j) \times \log[P(y_j|x_i)/P(y_j)].$$

Es ist einfach zu beweisen, dass die folgende schöne Eigenschaft gilt:

$$I(Y;X) = H(Y) - H(Y|X) \ ,$$

wobei $H(Y|X)$ die bedingte Entropie ist, die wir im Kapitel 6 definiert haben.[6] Dann definieren wir:

$$D = \sum_{i=0}^{N-1} P(x_i) \sum_{j=0}^{M-1} P(y_j|x_i) \times d(x_i, y_j).$$

Wir nennen D die *Verzerrung*. Beachte: D hängt ab von

1. der Art der Quelle (modelliert durch $P(x_i)$),
2. dem gewählten Kompressionsverfahren (die die Verteilungsfunktion $P(y_j|x_i)$ beeinflusst) und
3. dem Verzerrungsmaß d.

Sind Quelle und Verzerrungsmaß fixiert, können wir D als Funktion auffassen, die lediglich von der Verteilungsfunktion $P(y_j|x_i)$ abhängt.

Die Frage ist jetzt: Wie ist der Trade-off zwischen der Verzerrung und dem Kompressionsquotienten? Oder anders: Wie ist der bestmögliche Kompressionsquotient R für die Verzerrung, die kleiner oder gleich D ist? Wir wollen diese Abhängigkeit durch eine Funktion $R(D)$ beschreiben.

[6] Weitere Informationen finden Sie z.B. in [20].

Es sei $\Gamma(D)$ die Menge von Verteilungen für Y, so dass gilt:

$$\sum P(x_i) \sum P(y_j|x_i) \times d(x_i, y_j) \ \leq \ D.$$

Das heißt: für eine bestimmte Eingabe $P(x_0), P(x_1), \ldots, P(x_{N-1})$ sowie für ein gewisses D ist jede Ausgabe des Kompressionsverfahrens, das die Verteilung für Y aus $\Gamma(D)$ hat, mit der Verzerrung höchstens D behaftet. Shannon hat 1959 die folgende wichtige Eigenschaft bewiesen:

$$R(D) = \min_{P(\cdot|\cdot) \in \Gamma(D)} I(Y; X).$$

Der bestmögliche Kompressionsquotient wird also nach oben durch die wechselseitige Information der Zufallsvariablen für Ein- und Ausgabe beschränkt, sofern die Verteilungsfunktion dieser Zufallsvariablen in $\Gamma(D)$ liegt.

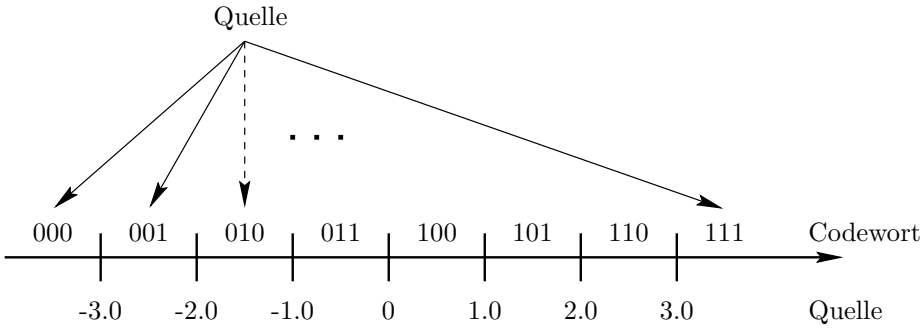


Abbildung 17: Eine 3-Bit-Quantisiervorschrift

8 Skalarquantisierung

8.1 Quantisierung

ist eine der einfachsten und allgemeinsten Ideen verlustbehafteter Komprimierung: Die Ausgabe einer Quelle muss durch eine endliche „kleine" Menge $\mathcal{C}$ von Codewörtern dargestellt werden, obwohl diese Ausgabe Werte aus einer sehr viel größeren, möglicherweise unendlichen Menge $\mathcal{A}$ annehmen kann. Die Quantisiervorschrift liefert also eine o. E. surjektive Abbildung $Q_C : \mathcal{A} \to \mathcal{C}$.

Da die Güte der Codierung (Kompressionsrate bzw. Verzerrung) sowohl von Q_C als auch von der Decodiervorschrift $Q_D : \mathcal{C} \to \mathcal{A}$ abhängt, begreifen wir unter einem *Quantisierer* $\mathbf{Q}$ das Paar (Q_C, Q_D). Das Verhalten von $\mathbf{Q}$ lässt sich durch seine Ein/Ausgabefunktion

$$\mathcal{A} \to \mathcal{A}, \quad x \mapsto Q_D(Q_C(x))$$

veranschaulichen, die wir der Einfachheit halber auch mit $\mathbf{Q}$ bezeichnen.

Bild 17 zeigt eine 3-Bit-Quantisiervorschrift (d.h., es werden drei Bits für die Codewörter benötigt, so dass wir $\mathcal{C} = \{0,1\}^3$ annehmen können), an der zwei Dinge noch deutlicher werden:

1. Codewörter bezeichnen Intervalle.[7]
2. Die Quantisiervorschrift $Q_C : \mathbb{R} \to \{0,1\}^3$ ist am Rand asymmetrisch, d.h., die zu codierenden Randintervalle sind meist größer als die inneren Intervalle.

[7] Auch wenn es theoretisch möglich wäre, andere Arten von Zahlenmengen als Intervalle zu codieren, ist dies unter dem Blickwinkel der Fehlerminimierung nicht sinnvoll.

Codewort	Repräsentant
000	-3,5
001	-2,5
010	-1,5
011	-0,5
100	0,5
101	1,5
110	2,5
111	3,5

Tabelle 4: Die Decodiervorschrift Q_D

t	$4\cos(2\pi t)$	A/D Ausgabe	D/A Ausgabe	Abweichung
0,05	3,804	111	3,5	0,304
0,10	3,236	111	3,5	-0,264
0,15	2,351	110	2,5	-0,149
0,20	1,236	101	1,5	-0,264

Tabelle 5: Die Abtastung einer Cosinuswelle

Da dieses Q_C (natürlicherweise) nicht injektiv ist, lässt sich der Originalwert der Quelle aus solch einem Repräsentanten nicht eindeutig rekonstruieren, m. a. W., Q_C ist verlustbehaftet. Etwas allgemeiner sind *A/D-Wandler* als Quantisiervorschrift Q_C auffassbar und entsprechend *D/A-Wandler* als Decodierer Q_D.

Die Decodiervorschrift Q_D wählt daher für jedes Intervall einen Repräsentanten aus, wie dies in Tabelle 4 für die 3-Bit-Quantisierung aufgelistet ist. Dort sind, von den Randintervallen abgesehen, die Repräsentanten die Mittelpunkte der codierten Intervalle. Kennt man die Quellenstatistik genauer, so ist es oft sinnvoll, andere Repräsentanten für die Intervalle zu wählen (s.u.). Für unseren 3-Bit-Quantisierer ist die Ein/Ausgabefunktion $\mathbf{Q}$ in Abb. 18 zu sehen.

Tabelle 5 zeigt die Wirkung dieser 3-Bit-Quantisierung bei Abtastung einer Cosinuswelle $4\cos(2\pi t)$ alle 50 ms. Dort ist auch die *Abweichung* des Repräsentanten vom ursprünglich zu codierenden Wert angegeben, also im Allgemeinen der Wert

$$|x - \mathbf{Q}(x)| = |x - Q_D(Q_C(x))|.$$

Die *Quantisieraufgabe* besteht darin, einen Quantisierer $\mathbf{Q}$ mit dem Ziel zu entwerfen,

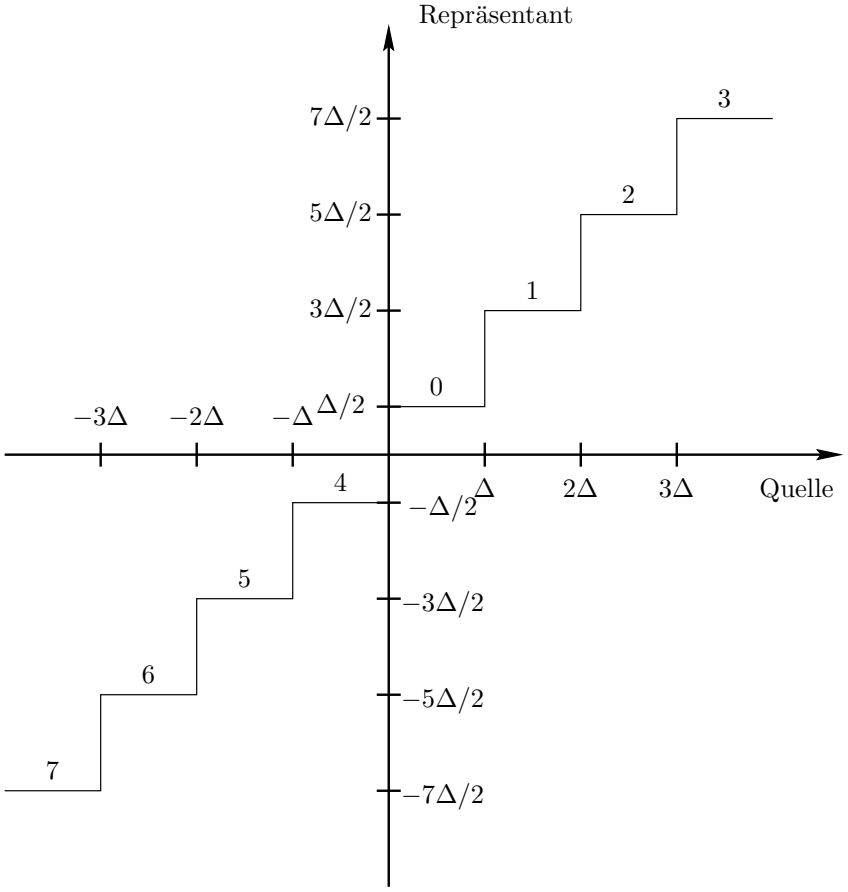


Abbildung 18: Die Ein/Ausgabefunktion eines 3-Bit-Quantisierers

- die *Rate* R (also die durchschnittliche Anzahl Bits, die für die Repräsentation eines Elementes aus $\mathcal{C}$ benötigt wird)[8] und
- den mittleren quadratischen Fehler σ_Q^2 (s. Kapitel 7)

zu minimieren. Da die Rate in unmittelbarem Zusammenhang zum Kompressionsquotienten steht, ist klar, dass diese beiden Forderungen einander widerstreitend sind. Geht man jedoch von Codewörtern fester gleicher Länge aus, so ist $R = \lceil \log_2 |\mathcal{C}| \rceil$ fixiert, so dass „nur" noch die Minimierung von σ_Q^2 als Aufgabe verbleibt.

Es werde nun die Quellenstatistik durch eine Zufallsgröße X mit Wahrscheinlichkeitsdichte f_X modelliert. Geschieht die Quantisierung mit Hilfe von M Intervallen, ist also $|\mathcal{C}| = M$ mit $\mathcal{C} = \{1, \dots M\}$, so haben wir die *Entscheidungsgrenzen* genannten Intervallendpunkte $b_0, \dots, b_M$ sowie die *Intervallrepräsentanten* $y_1, \dots, y_M$ anzugeben.[9] In unserer vorigen Notation ist daher für $1 \leq i \leq M$:

$$Q_C^{-1}(i) = (b_{i-1}, b_i],^{10} \quad Q_D(i) = y_i \quad \text{und somit}$$

$$\mathbf{Q}(x) = y_i \text{ genau dann, wenn } x \in (b_{i-1}, b_i].$$

Daraus ergibt sich

$$\begin{aligned} \sigma_Q^2 &= \int_{-\infty}^{+\infty} (x - \mathbf{Q}(x))^2 f_X(x) dx \\ &= \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f_X(x) dx \end{aligned} \tag{3}$$

Sind f_X und M vorgegeben und geht man von Codewörtern fester gleicher Länge aus, so besteht die Quantisieraufgabe darin, die Parameter $b_0, \dots, b_M$; $y_1, \dots, y_M$ so zu wählen, dass σ_Q^2 gemäß Gleichung 3 minimiert wird.

Geht man nicht von Codewörtern fester gleicher Länge aus und codiert die (Indizes der) Intervallrepräsentanten z. B. gemäß dem Huffman-Verfahren, so beeinflusst diese Codierung gemeinsam mit der Wahl der Entscheidungsgrenzen die Rate des Quantisierers, die so zu einer von der Wahrscheinlichkeit $P(y_i)$ des Auftretens von y_i abhängigen Zufallsvariablen wird.

Wegen $P(y_i) = \int_{b_{i-1}}^{b_i} f_X(x) dx$ gilt

$$R = \sum_{i=1}^{M} \ell_i P(y_i) = \sum_{i=1}^{M} \ell_i \int_{b_{i-1}}^{b_i} f_X(x) dx, \tag{4}$$

[8] Komprimiert man z. B. 8-Bit-Wörter durch Fortlassen des niederwertigsten Bits, so ist $R = |\mathcal{C}| = 7$.

[9] $b_0 = -\infty$ und $b_M = +\infty$ seien zugelassen.

[10] Zu welchen Intervallen die Intervallgrenzen selbst zählen, ist unerheblich.

wobei ℓ_i die Länge des y_i entsprechenden Codewortes ist.

So sind also die Aufgaben, optimale Entscheidungsgrenzen, Repräsentanten sowie Codierungen zu finden, miteinander verwoben. Im allgemeinen stellen sich daher zwei Arten von Optimierungsaufgaben:

1. Ist eine Verzerrungsschranke $D^* \geq \sigma_Q^2$ vorgegeben, so ist die Rate gemäß Gleichung 4 zu minimieren.
2. Ist eine Ratenschranke $R^* \geq R$ vorgegeben, so ist die Verzerrung nach Gleichung 3 zu minimieren.

Da die Entropie des Quantisierers durch

$$H = -\sum_{i=1}^{M} P(y_i) \log_2(P(y_i)) \tag{5}$$

gegeben ist, kann man auch noch eine dritte Quantisieraufgabe stellen:

3. Ist eine Entropieschranke $H^* \geq H$ vorgegeben, so ist die Verzerrung nach Gleichung 3 zu minimieren.

Diese Minimierungsaufgaben werden leicht nicht-trivial.

Im Folgenden kümmern wir uns —sofern nicht anders vermerkt— um die zweite der Minimierungsaufgaben, und wir gehen von einer Codierung mit fester gleicher Wortlänge aus. Diese zweite Aufgabe ist von der größten praktischen Bedeutung, da eine Ratenschranke oft durch die vorliegende Übertragungstechnik vorgegeben ist und die Übertragungssoftware auf diese Hardware hin zu optimieren ist.

8.2 Gleichquantisierer

Wir wollen jetzt den in der Praxis durchaus üblichen *Gleichquantisierer* untersuchen, bei dem alle Intervalle (mit Ausnahme evtl. der äußeren Intervalle) gleich groß sind und als Intervallrepräsentanten die Mittelpunkte dieser (inneren) Intervalle genommen werden, womit sowohl die Entscheidungsgrenzen als auch die Repräsentanten jeweils gleichabständig liegen; dieser Abstand wird auch *Schrittweite* genannt und mit Δ bezeichnet. Es geht jetzt um den Entwurf von verzerrungsminimierenden Gleichquantisierern bei vorgegebener Quellenverteilung und vorgegebenem Umfang M von $|\mathcal{C}|$.

Wir wollen jetzt einen Gleichquantisierer für eine im Intervall $[-X_\mu, X_\mu]$ gleichverteilte Quelle entwerfen mit Hilfe von M Intervallen, wobei M im folgenden o. E. gerade sei. Damit gilt:

$$\Delta = \frac{2X_\mu}{M}, \qquad f_X(x) = \begin{cases} \frac{1}{2X_\mu}, & x \in [-X_\mu, X_\mu] \\ 0, & \text{sonst} \end{cases}$$

$$\sigma_Q^2 = 2\sum_{i=1}^{M/2} \int_{(i-1)\Delta}^{i\Delta} \left(x - \frac{2i-1}{2}\Delta \right)^2 \frac{1}{2X_\mu} dx = \ldots = \frac{\Delta^2}{12}$$

Nun berechnen wir das Verhältnis von Signal zur Verzerrung. Da die Wahrscheinlichkeitsdichte der Quelle symmetrisch zum Ursprung ist, gilt $E[X] = 0$, d. h., der mittlere quadratische Wert der Quelle $E[X^2]$ ist gleich der Varianz, also ist $\sigma_X^2 = \frac{(2X_\mu)^2}{12}$. Daher ist:

$$\begin{aligned} SNR(dB) &= 10 \log_{10} \left(\frac{\sigma_X^2}{\sigma_Q^2} \right) \\ &= 10 \log_{10} \left(\frac{(2X_\mu)^2/12}{(\frac{2X_\mu}{M})^2/12} \right) \\ &= 6,02 \log_2 M dB \end{aligned}$$

Dies ist sozusagen der Präzisionsgewinn pro Bit, wie das am Sena-Bild 8.7 im Buch von Sayood gut zu sehen ist.

Die Annahme der Gleichverteilung ist tatsächlich meist falsch (Bilder haben meist Häufungen bei gewissen Grauwerten.), ebenso die der Unabhängigkeit aufeinander folgender Ereignisse (Bei Bildern sind benachbarte Pixel häufig gleich.). Die Methode der Vektorquantisierung (s. Abschnitt 9) geht auf den letzteren Punkt ein. Zunächst wollen wir uns mit dem Problem befassen, einen Gleichquantisierer für eine Quelle X mit beliebiger, bezüglich des Ursprungs symmetrischer Wahrscheinlichkeitsdichte f_X zu entwerfen. Für gerade M ist zu minimieren:

$$\begin{aligned} \sigma_Q^2 &= 2 \sum_{i-1}^{M/2-1} \int_{(i-1)\Delta}^{i\Delta} \left(x - \frac{2i-1}{2}\Delta \right)^2 f_X(x) dx \\ &+ 2 \int_{(M/2-1)\Delta}^{\infty} \left(x - \frac{M-1}{2}\Delta \right)^2 f_X(x) dx, \end{aligned}$$

was durch Nullsetzen der Ableitung $\frac{d\sigma_Q^2}{d\Delta}$ geschehen kann.

Auf diese Weise kann man die in Tabelle 6 aufgeführten optimalen Schrittweiten errechnen unter der Normierung $\sigma_X^2 = 1$ für drei Verteilungen. Die

Alph.-	Gleichverteilung		Gaußverteilung		Laplaceverteilung	
größe	Schrittweite	SNR	Schrittweite	SNR	Schrittweite	SNR
2	1,732	6,02	1,596	4,40	1,414	3,00
4	0,866	12,04	0,9957	9,24	1,0873	7,05
6	0,577	15,58	0,7334	12,18	0,8707	9,56
8	0,433	18,06	0,5860	14,27	0,7309	11,39
10	0,346	20,02	0,4908	15,90	0,6334	12,81
12	0,289	21,60	0,4238	17,25	0,5613	13,98
14	0,247	22,94	0,3739	18,37	0,5055	14,98
16	0,217	24,08	0,3352	19,36	0,4609	15,84
32	0,108	30,10	0,1881	24,56	0,2799	20,46

Tabelle 6: Optimale Schrittweiten für Gleichquantisierer

tatsächliche	angenomme Verteilung:		
Eingabeverteilung↓	Gleichverteilung	Gaußverteilung	Laplaceverteilung
Gleich-	18,06	15,56	13,29
Gauß-	12,40	14,27	13,37
Laplace-	8,80	10,79	11,39

Tabelle 7: Der Einfluss von Verteilungsannahmen auf SNR bei einem 3-Bit-Quantisierer

angegebene Schrittweite für die Gleichverteilung ergibt sich aus der Normierungsbedingung $\sigma_X^2 = (2X_\mu)^2/12 = 1$. Es nimmt nicht wunder, dass der Präzisionsgewinn pro Bit bei der Laplaceverteilung geringer ist als bei der Gaußverteilung und bei der Gaußverteilung wiederum geringer ist als bei der Gleichverteilung: schließlich „entspricht" die Gleichverteilung dem Gleichquantisierer, und die Laplace-Verteilung ist „am wenigsten gleichverteilt".

Tabelle 7 zeigt, dass es nicht unwichtig ist, welche Verteilung für die Quelle angenommen wurde. Eingerahmt sind die optimalen Werte (wie sie auch in Tabelle 6 in der Zeile von Alphabetgröße 8 dargestellt sind): sie finden sich naturgemäß auf der Hauptdiagonalen. Konkret bedeutet dies: Ist die Eingabe gleichverteilt, wir sind bei der Modellierung jedoch von einer Gaußverteilung ausgegangen, so verschlechtert sich der SNR-Wert gegenüber der richtigen Annahme einer Gleichverteilung. Noch schlechter liegen wir allerdings, wenn wir von einer Laplaceverteilung ausgehen; wie schon bemerkt ist sie noch „weniger gleichverteilt" als die Gaußverteilung. Allgemein gesprochen gilt: Ist (aufgrund einer falschen Verteilungsannahme) die Schritt-

weite kleiner als die der optimalen Annahme entsprechenden (so wie wir es in der ersten Zeile von Tabelle 7 sehen), so ist der Fehler größer, als wenn die Schrittweite überschätzt wird (beobachten Sie die letzte Zeile von Tabelle 7 zum Vergleich). Durch statistische Testverfahren, die außerhalb unseres Skriptums liegen, kann man diesem Fehler begegnen. Ähnlich kann man vorgehen, wenn die angenommene Varianz falsch geschätzt wurde. Eine elementare Einführung in statistische Verfahren finden Sie in [2].

8.3 Adaptive Quantisierung

Um den genannten Entwurfsfehlern (Annahme einer falschen Verteilung bzw. Varianz) grundsätzlich zu begegnen, betrachten wir jetzt die *adaptive Quantisierung*. Hierbei nehmen wir an, die Verteilung der Quelle ändere sich mit der Zeit (Dies ist sehr typisch bei Sprachsignalen zu beobachten.) oder aber ist uns schlicht unbekannt. Ändert sich lediglich der Mittelwert, so ist eine Form der Differentialcodierung zu wählen, die im nächsten Kapitel behandelt wird. Andernfalls empfiehlt sich eine Anpassung der Parameter des Quantisierers an die Quellenstatistik. Hier gibt es zwei Grundstrategien:

1. Voradaptierung und
2. Rückadaptierung.

8.3.1 Voradaptierung

ist ein typisches „off-line"-Verfahren. Die Ausgabe der Quelle wird in Blöcke unterteilt, deren Statistik separat analysiert wird; gemäß dieser Analyse werden die Quantisierparameter gesetzt. Wie jedes „off-line"-Verfahren bringt auch dieses eine Verzögerung der Übertragung durch den Puffer-Effekt mit sich; diese Verzögerung ist umso größer, je länger der Puffer ist. Andererseits erlaubt ein langer Puffer eine gute Verteilungsanalyse. Wie ist also der Puffer zu wählen? Da die Analyse auf den dem Decodierer nicht zur Verfügung stehenden Eingabedaten beruht, müssen die so ermittelteten Parameter als *Begleitinformationen* mit übertragen werden.

Wie (aus dem Buch von Sayood) Bild 8.15 im Vergleich zu Bild 8.7 unten rechts zeigt, kann diese Strategie durchaus Beachtliches leisten: Das in 8×8-Blöcke unterteilte Sena-Bild wurde wiederum unter Annahme der Gleichverteilung durch einen Gleichquantisierer komprimiert, nur dass jetzt als Begleitinformation je Block die Minimal- und Maximalwerte der Gleichverteilung mitgesendet wurden, was eine Übertragung von 3,25 Bits pro Pixel bedeutet anstelle von 3 Bits pro Pixel in Bild 8.7.

n	Δ_n	Eingabe	Code	Repräsent.	Abweich.	neue Schrittweite
0	0,5	0,1	0	0,25	0,15	$\Delta_1 = M_0\Delta_0$
1	0,4	-0,2	4	-0,2	0,0	$\Delta_2 = M_4\Delta_1$
2	0,32	0,2	0	0,16	0,04	$\Delta_3 = M_0\Delta_2$
3	0,256	0,1	0	0,128	0,028	$\Delta_4 = M_0\Delta_3$
4	0,2048	-0,3	5	-0,3072	-0,0072	$\Delta_5 = M_5\Delta_4$
5	0,1843	0,1	0	0,0922	-0,0078	$\Delta_6 = M_0\Delta_5$
6	0,1475	0,2	1	0,2212	0,0212	$\Delta_7 = M_1\Delta_6$
7	0,1328	0,5	3	0,4646	-0,0354	$\Delta_8 = M_3\Delta_7$
8	0,1594	0,9	3	0,5578	-0,3422	$\Delta_9 = M_3\Delta_8$
9	0,1913	1,5	3	0,6696	-0,8304	$\Delta_{10} = M_3\Delta_9$
10	0,2296	1,0	3	0,8036	0,1964	$\Delta_{11} = M_3\Delta_{10}$
11	0,2755	0,9	3	0,9643	0,0643	$\Delta_{12} = M_3\Delta_{11}$

Tabelle 8: Zur Arbeitsweise eines Jayant-Quantisierers

8.3.2 Rückadaptierung

Nur die in der Vergangenheit quantisierten Proben können zur Adaptierung verwendet werden. Will man die Übertragung von Begleitinformationen vermeiden, so besteht die Schwierigkeit darin, dass die unquantisierte Eingabe nur dem Codierer bekannt ist und so nicht zur Veränderung der Quantisiererparameter verwendet werden kann. Eine einfache Lösung für dies Problem wurde von N. S. Jayant gefunden. Grundgedanke hierbei ist: Fällt die letzte Eingabe in ein inneres Intervall, so schrumpft die Schrittweite, fällt sie in ein äußeres Intervall, so wird die Schrittweite größer. Trivialerweise steht die Information, in welches Intervall die Eingabe fiel, auch dem Decodierer zur Verfügung, so dass keine Begleitinformationen versendet werden müssen.

Genauer arbeitet der *Jayant-Quantisierer* wie folgt: Jedem der M Intervalle wird ein Faktor M_i, $1 \leq i \leq M$, zugeordnet. Die Faktoren der inneren Intervalle sind kleiner als Eins, die der äußeren größer als Eins. Bezeichnet Δ_n die Schrittweite zur Verarbeitung der n-ten Eingabe und falle diese ins Intervall $\ell(n)$, so wird die Schrittweite gemäß der Vorschrift

$$\Delta_{n+1} = M_{\ell(n)}\Delta_n$$

aktualisiert.

Tabelle 8 mag die Arbeitsweise eines Jayant-Quantisierers erläutern helfen. Ausgehend von dem Quantisierer aus Abb. 18 mit den dort angegebenen Intervallcodierungen sowie den Faktoren $M_0 = M_4 = 0,8$, $M_1 = M_5 = 0,9$, $M_2 = M_6 = 1$, $M_3 = M_7 = 1,2$ und der Anfangsschrittweite $\Delta_0 = 0,5$

erhält man die angegebene Schrittweiten- und Ausgabenfolge. Man berechne einmal den so erhaltenen quadratischen Fehler und vergleiche ihn mit einem nicht-adaptiven Quantisierer! Um aus numerischen Gründen zu vermeiden, dass Δ_n „zu nahe" an Null kommt bzw. „zu groß" wird, werden häufig Grenzwerte Δ_{min} bzw. Δ_{max} definiert.

Wie wählt man die Faktoren M_i? Als Faustregel kann gelten: Je mehr sich die Faktoren von Eins unterscheiden, desto rascher reagiert der Jayant-Quantisierer. Eine zu rasche Reaktion kann allerdings zu Instabilitäten führen. Geht man von einem stationären Eingabeprozess aus, bei dem schließlich die Wahrscheinlichkeiten, mit einer einzelnen Eingabe im i-ten Intervall zu landen, P_i beträgt, so sollten die P_i der Gleichung

$$\prod_{i=1}^{M} M_i^{P_i} = 1$$

genügen.

Abschließend noch zwei Warnungen zu adaptiven Verfahren:

- Kennt man die Quellenstatistik genau, so ist ein diesbeüglich optimierter Quantisierer einem adaptiven überlegen. Ganz allgemein ist es bei der Datenkompression essentiell, alle verfügbaren Informationen über die Quelle auszunutzen.
- Wählt man die Adaptionsparameter „ungeschickt" im Hinblick auf die Eingabedaten, so kann es –wie ja allgemein von Reglern bekannt– zu Schwingeffekten kommen.

8.4 Allgemeine Quantisierer

Ist die Quelle nicht gleichverteilt, so erscheint es sinnvoll, die Intervallaufteilung nicht gleichmäßig wie beim Gleichquantisierer zu gestalten. In diesem Abschnitt wollen wir solche *allgemeinen Quantisierer* betrachten.

Eine Idee ist es, bei bekannter Wahrscheinlichkeitsdichte f_X der Quelle direkt zu versuchen, Ausdruck (3) zu minimieren. Dies führt auf ein wechselseitig rekursives, nicht-lineares Gleichungssystem für die Repräsentanten und Entscheidungsgrenzen, deren Lösung —fußend auf Ideen der Variationsrechnung— unabhängig von einander S. P. Lloyd und J. Max sowie J. Lukaszewicz und H. Steinhaus in den Jahren 1950-1960 angaben. Wir werden diese Lösung im Folgenden vereinfachend *Lloyd-Verfahren* nennen. Da Varianten hiervon noch später von Interesse sein werden, geben wir das Verfahren in Tab. 9 an. *Zentroide* sind eine die Wahrscheinlichkeitsdichte berücksichtigende Verallgemeinerung des Mittelwertbegriffs, wie man sieht, wenn

1. Wähle Anfangsrepräsentanten $\{y_i^{(0)}\}_{i=1}^M$ und Abbruchschwellwert ε für die Verzerrungsdifferenz.
   Setze $D^{(-1)} = 0$, $k = 0$.

2. Berechne neue innere Entscheidungsgrenzen $b_1^{(k)}, \ldots, b_{M-1}^{(k)}$ (und damit die Intervalle) gemäß:

$$b_j^{(k)} = \frac{y_{j+1}^{(k)} + y_j^{(k)}}{2} .$$

3. Berechne die neue Verzerrung

$$D^{(k)} = \sum_{i=1}^{M} \int_{b_{i-1}^{(k)}}^{b_i^{(k)}} (x - y_i)^2 f_X(x) dx .$$

4. Falls $\left| D^{(k)} - D^{(k-1)} \right| < \varepsilon$, STOP.

5. Sonst: Setze $k = k+1$ und berechne neue Repräsentanten als Zentroide:

$$y_j^{(k)} = \frac{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} x f_X(x) dx}{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} f_X(x) dx} .$$

   Gehe zu Schritt 2.

Tabelle 9: Das Lloyd-Verfahren

man die Gleichverteilung als Dichte ansetzt; sie lassen sich als normierte Erwartungswerte begreifen. Evtl. ist es sinnvoll, andere als die äußeren Entscheidungsgrenzen als fest anzunehmen und dann den Algorithmus geeignet abzuwandeln.

Eine andere Idee ist es, einem Gleichquantisierer eine *Kompressor* c genannte Transformation vorzuschalten und eine *Expander* e genannte Transformation nachzuschalten. Ein Beispiel für solche sog. *zusammengesetzten Quantisierer* liefern der Kompressor

$$c(x) = \begin{cases} 2x, & \text{falls } x \in [-1,1] \\ \frac{2x}{3} + \frac{4}{3}, & \text{falls } x \in [1,+\infty) \\ \frac{2x}{3} - \frac{4}{3}, & \text{falls } x \in (-\infty,-1] \end{cases}$$

und der Expander $e = c^{-1}$. In diesem Fall ist die Schrittweite des Quantisierers Q_C im Intervall $[-1,1]$ kleiner als außerhalb dieses Intervalls, was z. B. bei Gauß- oder Laplace-verteilten Quellen erwünscht ist. Bei „Zwischenschalten" eines 3-Bit-Gleichquantisierers liefert der so definierte zusammengesetzte Quantisierer das in Abb. 19 gezeigte Verhalten **Q**. Die von Telefongesellschaften eingesetzten Quantisierer sind als zusammengesetzte definiert.

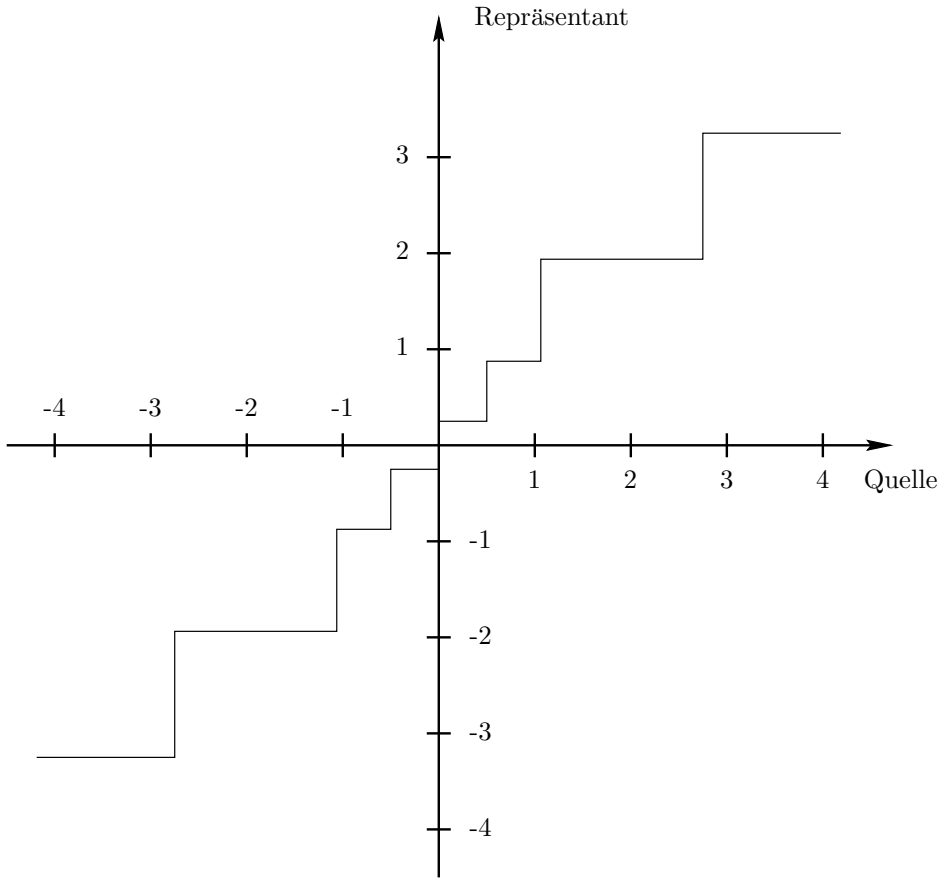


Abbildung 19: Die Ein/Ausgabefunktion eines zusammengesetzten 3-Bit-Quantisierers

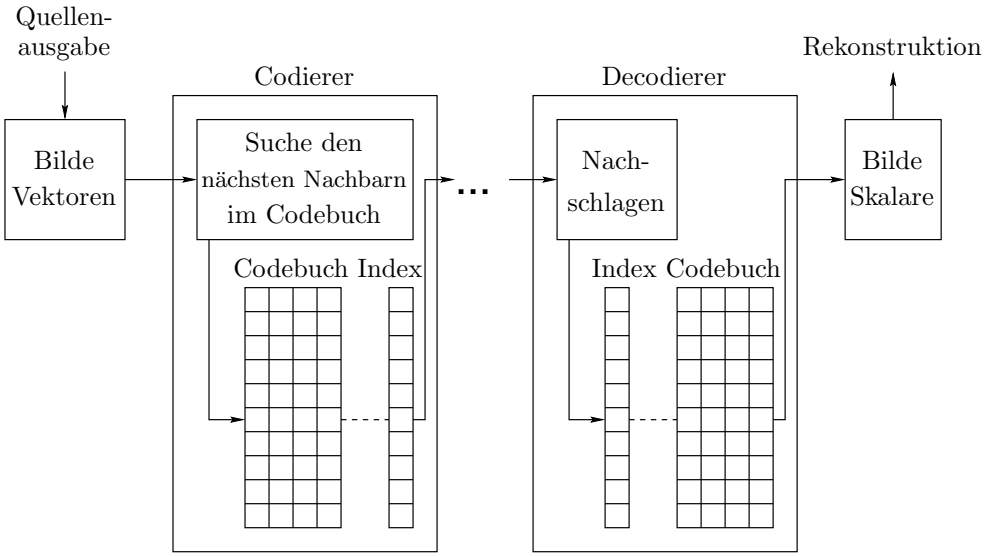


Abbildung 20: Die Arbeitsweise eines Vektorquantisierers

9 Vektorquantisierung

9.1 Skalare Quantisierung versus Vektorquantisierung

Bislang hatten wir versucht, einzelne Quellenausgaben (also skalare Werte) zu quantisieren. Man spricht hierbei auch von der *skalaren Quantisierung*. Tatsächlich bestehen jedoch häufig räumliche (Bsp.: Bilder) oder zeitliche (Bsp.: Sprache) Abhängigkeiten zwischen aufeinander folgenden Quellenausgaben. Ein wenig wurde dies bereits bei der adaptiven Quantisierung berücksichtigt.

Bei der jetzt zu betrachtenden *Vektorquantisierung* werden immer L aufeinander folgende Quellenausgaben zu einem L-dimensionalen Vektor zusammengefasst und gemeinsam quantisiert. Der Codierer enthält im wesentlichen eine *Codebuch* genannte indizierte Tabelle sog. *Codevektoren* und arbeitet für jeden Eingabevektor $\vec{i}$ wie folgt:

1. Suche Codevektor $\vec{v}$, der der Eingabe $\vec{i}$ am nächsten kommt.
2. Übertrage den so gefundenen Index von $\vec{v}$.

Der Decodierer enthält dasselbe Codebuch und kann über den Index leicht auf den $\vec{i}$ quantisierenden Codevektor $\vec{v}$ zugreifen.

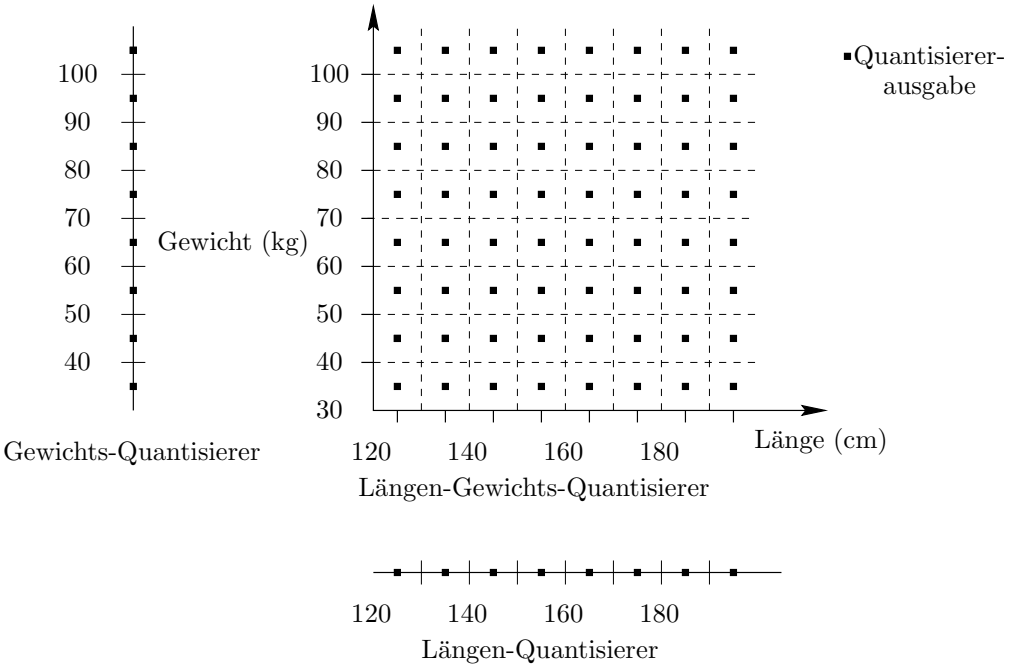


Abbildung 21: Länge und Gewicht von Personen

Das in Bild 20 nochmals veranschaulichte Schema eines Vektorquantisierers ist insofern asymmetrisch, als dass der Codierer einen erheblichen Rechenaufwand zu erledigen hat (Codebücher können sehr groß sein, und die Bestimmung des nächsten Nachbarn [s. Vorlesungen oder Bücher über „Algorithmische Geometrie"] kann so viel Zeit in Anspruch nehmen.), während die Decodierung geradezu trivial ist. Dieses allgemeine Schema der Vektorquantisierung ist daher insbesondere für Anwendungen geeignet, wo ein einmaliger Aufwand in der Codierung vielfach durch einfache Decodierung genutzt wird, z. B. im Multimediabereich. Aufgrund der sehr geringen Raten (d.h., hohen Kompressionsquotienten) ist Vektorquantisierung aber überhaupt sehr populär, insbesondere auch bei der Sprachübertragung.

Besteht ein Codebuch aus K L-dimensionalen Vektoren, so werden (bei Annahme fester Wortlänge der übertragenden Indizes) je Skalar lediglich $\lceil \log_2 K \rceil / L$ Bits benötigt, was sehr große Codebücher gestattet.

Veranschaulichen wir uns die Vektorquantisierung an zwei Beispielen.

Beispiel 9.1 Länge und Gewicht von Personen.

Selbst wenn –wie in Bild 21 dargestellt– Länge bzw. Gewicht[11] einer Stich-

[11] Bekanntermaßen ist das Kilogramm die Einheit der Masse und nicht des Gewichts;

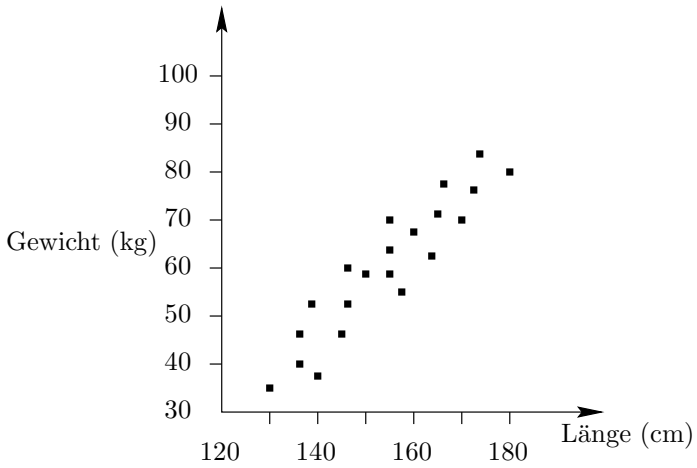


Abbildung 22: Eine Stichprobe von Länge und Gewicht von Personen

probe getrennt betrachtet jeweils gleichverteilt sein mögen in einem Intervall I_ℓ bzw. I_g, so darf man nicht die im Bild dargestellte Gleichverteilung im Quadrat $I_\ell \times I_g$ erwarten, denn Länge und Gewicht von Personen sind korreliert, so wie dies in Bild 22 angedeutet ist. Eine zweidimensionale Vektorquantisierung kann hier helfen.

Beispiel 9.2 Diskussion einer Laplace-verteilten Quelle.

Wie aus Tabelle 6 ersichtlich, besitzt ein optimaler 3-Bit-Gleichquantisierer für eine Laplace-verteilte Quelle eine Schrittweite von $\Delta = 0,7309$. Hier gilt aber: $P(x \in [0, \Delta)) = 0,3242$ und $P(x \in [3\Delta, \infty)) = 0,0225$. Im vorigen Beispiel 9.1 war die Wahl eines Vektorquantisierers durch die natürliche Korrelation der Eingabedaten begründet. Hier wollen wir –ganz im Gegensatz dazu– annehmen, aufeinanderfolgende Werte der Quelle seien unabhängig. Fassen wir dennoch die Eingabewerte zu Paaren zusammen, so entspricht dem (skalaren) Gleichquantisierer ein Vektorquantisierer, bei dem die „Einzugsbereiche" eines Vektors eine gitterartige Struktur liefern, s. Bild 23.

Die Wahrscheinlichkeit, dass ein Eingabepaar in das rechte obere „Quadrat" fällt, beträgt (bei Annahme der Unabhängigkeit) lediglich

$$P((x,y) \in [3\Delta, \infty) \times [3\Delta, \infty)) = P(x \in [3\Delta, \infty)) \cdot P(y \in [3\Delta, \infty)) = 0,0005\,.$$

Daher liegt die Idee nahe, den rechten oberen Codevektor durch den Nullpunkt zu ersetzen, wodurch sich die in Bild 24 angedeutete Gitterstruktur ergibt. Der SNR-Wert verbessert sich dadurch von $11,44dB$ auf $11,73dB$.

wir passen uns hier lediglich dem umgangssprachlichen Gebrauch an. Handelsübliche Personenwaagen messen im Übrigen tatsächlich das Gewicht und rechnen dies dann unter Berücksichtigung der Gravitation in Kilogramm um.

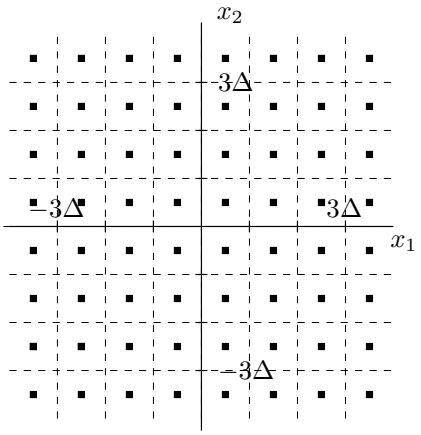


Abbildung 23: Die Gitterstruktur eines gepaarten 3-Bit-Gleichquantisierers

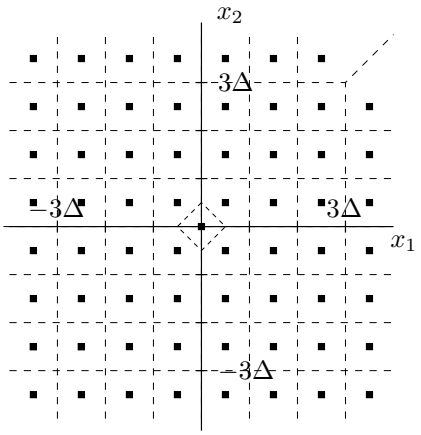


Abbildung 24: 64-elementiges Codebuch eines 2-dimensionalen Vektorquantisierers

1. Wähle Anfangsrepräsentanten $\{Y_i^{(0)}\}_{i=1}^M$ und Abbruchschwellwert ε für die Verzerrungsdifferenz.
   Setze $D^{(-1)} = 0$, $k = 0$.

2. Berechne neue Quantisier-Regionen mit Hilfe von:

$$V_j^{(k)} = \{X \mid \forall i \neq j : d(X, Y_j) < d(X, Y_i)\}\,.$$

3. Berechne die neue Verzerrung

$$D^{(k)} = \sum_{i=1}^{M} \int_{V_i^{(k)}} \left\| \vec{x} - Y_i^{(k)} \right\|^2 f_X(\vec{x}) d\vec{x}\,.$$

4. Falls $\frac{|D^{(k)} - D^{(k-1)}|}{D^{(k)}} < \varepsilon$, STOP.

5. Sonst: Setze $k = k+1$ und berechne neue Repräsentanten als Zentroide:

$$Y_j^{(k)} = \frac{\int_{V_j^{(k-1)}} \vec{x} f_X(\vec{x}) d\vec{x}}{\int_{V_j^{(k-1)}} f_X(\vec{x}) d\vec{x}}\,.$$

   Gehe zu Schritt 2.

Tabelle 10: Das Lindo-Buzo-Gray-Verfahren I

9.2 Entwurf guter Codebücher

Wie in obigem Beispiel 9.1 kann die innere Struktur der Eingabe zu einem optimierten Codebuch-Entwurf genutzt werden. Das Auffinden von Clustern muss bei großen Codebüchern automatisiert werden. Dazu dient eine von Lindo, Buzo und Gray ersonnene Modifikation des Lloydschen Algorithmus (vgl. Tab. 9), das *Lindo-Buzo-Gray-Verfahren* oder kurz *LBG-Verfahren*. An die Stelle der Entscheidungsgrenzen treten die *Quantisier-Regionen*, und die Verzerrung bzw. Zentroide sind entsprechend kompliziertere Integrale, s. Tab. 10.

Aufgrund der Kompliziertheit der auftretenden Integrale gibt es auch eine vereinfachte, sozusagen diskretisierte Version, bei der mit einer *Trainingsmenge* $\{X_n\}_{n=1}^N$ gearbeitet wird. Dieses Verfahren ist in Tab. 11 beschrieben. Es kommt ohne Annahmen über die Quellenstatistik aus.

Als Beispiel betrachten wir jetzt die in Tab. 12 dargestellte Trainingsmenge, wobei wir zunächst mit dem in Tab. 13 gegebenen Codebuch anfangen wollen. Diesem Codebuch entsprechen die in Bild 25 gezeigten Quantisier-Regionen. Kreise kennzeichnen die Zentren der neuen Quantisier-Regionen,

Es steht eine Trainingsmenge $\{X_n\}_{n=1}^N$ zur Verfügung.

1. Wähle Anfangsrepräsentanten $\{Y_i^{(0)}\}_{i=1}^M$ und Abbruchschwellwert ε für die Verzerrungsdifferenz.
   Setze $D^{(-1)} = 0$, $k = 0$.

2. Berechne neue Quantisier-Regionen mit Hilfe von:

$$V_j^{(k)} = \{X_n \mid \forall i \neq j : d(X_n, Y_j) < d(X_n, Y_i)\}\,.$$

   Sollte ein $V_j^{(k)}$ leer sein, so teile man statt dessen die größte Region in zwei „Hälften".

3. Berechne die neue Verzerrung

$$D^{(k)} = \sum_{i=1}^{M} \sum_{X_n \in V_i^{(k)}} \|X_n - Y_i\|^2\,.$$

4. Falls $\frac{\left|D^{(k)} - D^{(k-1)}\right|}{D^{(k)}} < \varepsilon$, STOP.

5. Sonst: Setze $k = k + 1$ und berechne neue Repräsentanten als Mittelwerte:

$$Y_j^{(k)} = \frac{\sum_{X_n \in V_j^{(k)}} X_n}{\left|\{X_n \in V_j^{(k)}\}\right|}\,.$$

   Gehe zu Schritt 2.

Tabelle 11: Das Lindo-Buzo-Gray-Verfahren II

Länge	Gewicht
180	90
180	88
163	60
148	60
155	57
160	75
150	55
163	81
140	46
143	44
175	86

Tabelle 12: Trainingsmenge für den Codebuchentwurf

Länge	Gewicht
113	25
113	59
188	59
200	90

Tabelle 13: Anfangscodebuch

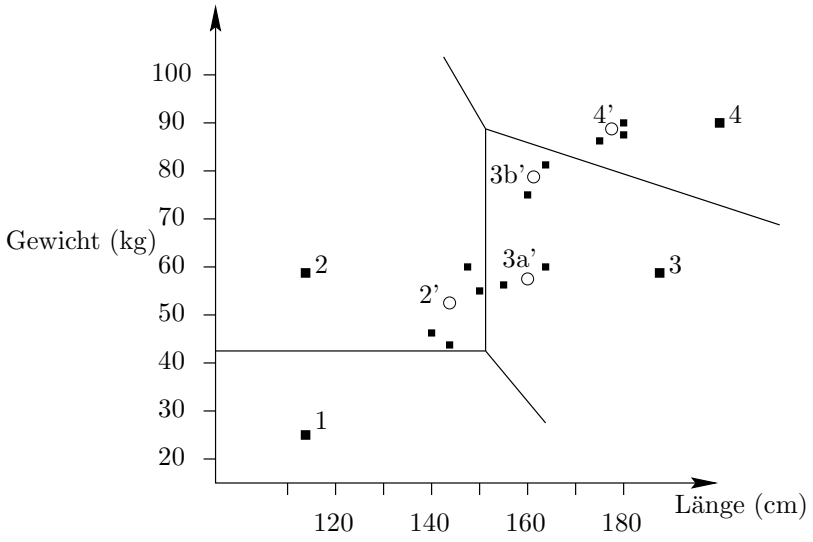


Abbildung 25: Anfängliche Quantisier-Regionen und erste Iteration

Entwirf K Bit großes Anfangscodebuch:

1. Falls $K = 0$: Nimm Mittelwert der Trainingsmenge $\{X_n\}_{n=1}^N$ als Codevektor.
2. Falls $K > 0$: Entwirf (rekursiv) $K - 1$ Bit großes Anfangscodebuch; Initialisiere K-Bit Codebuch durch 2^{K-1} so erhaltene Vektoren sowie um fixes δ gestörte Vektoren;
   Starte LBG-Algorithmus zur Ermittlung eines K-Bit-Anfangscodebuchs.

Tabelle 14: Das Lindo-Buzo-Gray-Verfahren III

die dann als deren Repräsentanten in der nächsten Iteration dienen. Beachten Sie, dass der Anfangspunkt 1 keinen nächsten Nachbarn in der Trainingsmenge gefunden hat. Die entsprechende Quantisier-Region bleibt daher leer, weshalb eine andere Region mit maximaler Trefferzahl, hier Region 3 (Region 2 wäre genauso wählbar) aufgespalten wird.

Leider kann die Güte des so erhaltenen Codebuchs von den (willkürlich gewählten) Anfangsrepräsentanten abhängen; das Verfahren konvergiert also nicht unabhängig von den Startwerten.

Wie lässt sich ein gutes *Anfangscodebuch* konstruieren? Wir diskutieren im Folgenden drei Ansätze:

1. Lindo, Buzo und Gray schlugen die in Tab. 14 angegebene *Aufspalttechnik* vor (der Abbruchschwellwert ε und die *Vektorstörung* δ sowie die Trainingsmenge sind als vorweg gegebene Parameter aufzufassen), vgl. auch Tab. 15 sowie die dazu gehörigen Regionen in Bild 26 für das Längen/Gewichtsbeispiel.
2. Eine quasi entgegengesetzte Idee zum Entwurf des Anfangscodebuchs hatte Equitz (*Equitz-Verfahren*): Ausgehend von einer Trainingsmenge werden solange paarweise nächstbenachbarte Cluster vereinigt, angefangen mit der Clustermenge $\{\{X_n\}\}_{n=1}^N$, bis der erwünschte Codebuchumfang erreicht ist; dabei ist der Abstand zwischen Clustern derjenige ihrer Mittelwerte.[12] Wie in Abb. 27 ersichtlich, kann man durch Aufschreiben der Größe der Punktmenge, für die ein Mittelwert steht (dies ist die Bedeutung der Zahl in Klammern) einfach erreichen, dass man stets ein gewichtetes Mittel betrachtet beim Bestimmen eines neuen Mittelwertes.

[12] Dieser Abstandsbegriff ist recht willkürlich; eine allgemeine und moderne Darstellung verschiedener agglomerativ arbeitender Clustering-Algorithmen finden Sie in [9]. In Büchern über Künstliche Intelligenz und Mustererkennung finden Sie weitere sog. *Clusteringverfahren*.

Codebuch	Länge	Gewicht
0 Bit	160	67
1 Bit Anfangs-	160	67
	170	72
1 Bit End-	150	54
	173	84

Tabelle 15: Die Aufspalttechnik am Beispiel

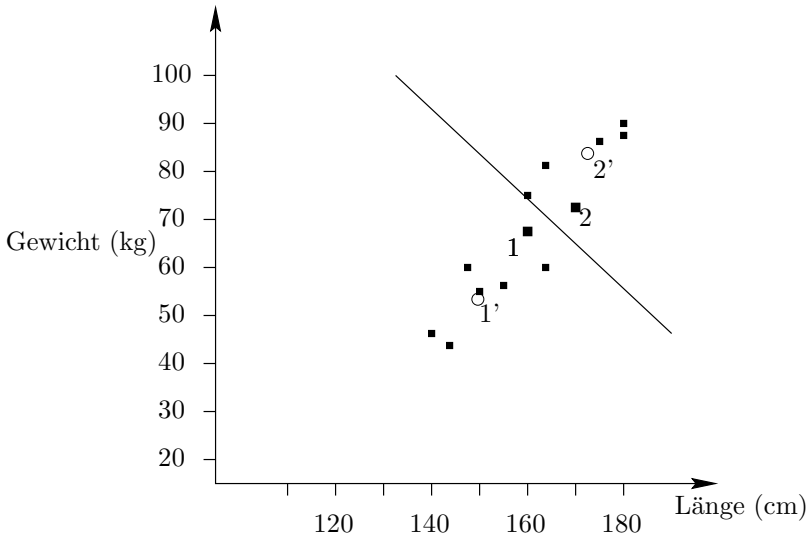


Abbildung 26: Ein-Bit Vektorquantisierer

3. Häufig am einfachsten und besten ist es, das Anfangscodebuch durch Würfeln als Teilmenge der Trainingsmenge zu wählen. Man kann dann den LBG-Algorithmus auf mehreren solchen Anfangscodebüchern laufen lassen und schließlich das beste Codebuch sich heraussuchen.

Zwei grundsätzliche Probleme birgt das LBG-Verfahren noch in sich:

1. Woher bekommt man „gute" Trainingsmengen für eine Anwendung?

   Bilder haben häufig unterschiedliche Charakteristika; ein auf ein Bild optimiertes Codebuch mag für ein anderes Bild nur bedingt brauchbar sein. Umgekehrt mindert ein als Begleitinformation übertragenes komplettes Codebuch natürlich die (bei Vektorquantisierern ansonsten hervorragende) Kompressionsrate, s. Tab. 16. Diesen Einfluss kann man gut an Tab. 17 beobachten, wo der Overhead für das einmalige (unkomprimierte) Übertragen des verwendeten Codebuchs zu sehen ist.

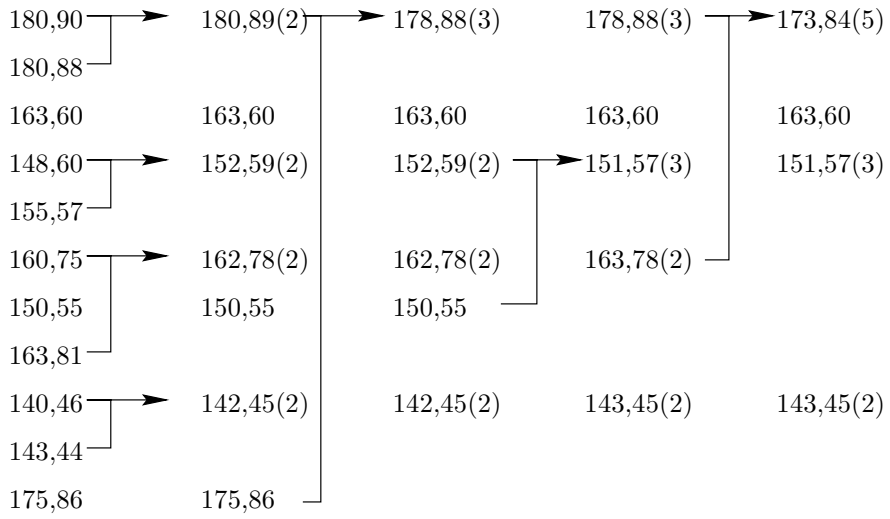


Abbildung 27: Der Ansatz von Equitz

Codebuchumfang K	$\log_2 K$	Bits pro Pixel	Kompressionsrate
16	4	0,25	32:1
64	6	0,375	21,33:1
256	8	0,50	16:1
1024	10	0,625	12,8:1

Tabelle 16: Verschiedene Kompresssionsmaße für ein Grauwertbild mit 8 Bit pro Pixel Auflösung

Codebuchumfang K	Overhead (Bits pro Pixel)
16	0,03125
64	0,125
256	0,50
1024	2,0

Tabelle 17: Overhead für die Codebuchübertragung

2. Das Auffinden des nächsten Codevektors kostet bei großen Codebüchern viel Zeit.

Das erste Problem kann gemildert werden, wenn man die Mittelwerte jedes Blocks („Vektors") skalarquantisiert überträgt und lediglich die Abweichungen vom Mittelwert trainiert. Bei diesem „hybriden" Verfahren kann man natürlich auch einen Jayant-Quantisierer einsetzen.

Dem zweiten Problem kann durch Wahl geeigneter Datenstrukturen begegnet werden, z. B. einer baumartigen Strukturierung der Codevektoren.

Beide Probleme lassen sich natürlich mit „regelmäßigen" Codebüchern umgehen, z. B. dem gitterartigen aus Bild 21. Allerdings hat sich eine hexagonale (wabenartige) Struktur für zweidimensionale Codebücher als besser erwiesen.

Folge $\{x_n\}$	6,2	9,7	13,2	5,9	8,0	7,4	4,2	1,8
Diff. $\{d_n\}$	6,2	3,5	3,5	-7,3	2,1	0,6	-3,2	-2,4
Quant. $\{\hat{d}_n\}$	6	4	4	-6	2	0	-4	-2
Rekonstr. $\{\hat{x}_n\}$	6	10	14	8	10	10	6	4
Abweichung	0,2	-0,3	-0,8	-2,1	-2	-2,6	-1,8	-2,2

Tabelle 18: Zur Fehlerfortpflanzung

10 Differentialcodierung

10.1 Motivation

Die Grundidee der *Differentialcodierung* ist es, die Korrelation aufeinander folgender Quellenausgaben dadurch auszunutzen, dass anstelle der Originalwerte die Differenzen zweier benachbarter Werte (codiert) übertragen werden, in der Hoffnung, dass dadurch der Wertebereich und die Varianz vermindert werden können. Verdeutlichen wir uns diesen Ansatz an einem einfachen Beispiel.

Beispiel 10.1 Wir betrachten eine Abtastung der Kosinusfunktion

$$f(t) = \cos(\frac{\pi}{6}t), \quad t \in [0, 6].$$

Die Werte $x_n = f(n/10)$, $n = 0, \ldots 60$ liegen zwischen -1 und 1. Hingegen fallen die Differenzen $d_n = x_n - x_{n-1}$ ins Intervall $[-0,2; 0,2]$.[13] Benutzt man einen 2-Bit-Gleichquantisierer, so führt dies (gemäß den Betrachtungen im Abschnitt 8.2) bei Übertragung der d_n zu einer Schrittweite von $2 \times 1/2^2 = 0,5$, und der betreffende Quantisierfehler liegt im Intervall $[-0,25; 0,25]$; hingegen nimmt man bei Übertragung der d_n die Schrittweite $0,1$, was den Quantisierfehler auf das Intervall $[-0,05; 0,05]$ beschränkt.

Bei Bildern führt der Übergang zu Pixeldifferenzen oft zu Histogrammen, die der Laplace-Verteilung ähneln. Dies berücksichtigend ist eine Quantisierung des „Differenzenbildes" mit fünf oder gar vier Bit genauso gut wie die des Originalbilds mit sieben Bit.

Stimmt diese „Bit-Analyse" wirklich? Bislang haben wir das Phänomen der *Fehlerfortpflanzung* bei verlustbehafteter Übertragung vernachlässigt, was an folgendem Beispiel deutlich wird:

[13] Eine Ausnahme bildet der erste Wert $d_0 = x_0$.

Beispiel 10.2 Wir benutzen einen Quantisierer mit sieben Ausgabewerten $\{-6,-4,-2,0,2,4,6\}$; Tabelle 18 enthält die Ausgangsfolge, die Differenzenfolge, die sich aus der Differenzenfolge ergebene Quantisierfolge, die dadurch gelieferte Rekonstruktion der Originalwerte und schließlich die Abweichung von selbigen. Wie man sieht, scheint die Abweichung tendenziell immer größer zu werden.

Wir wollen diese Fehlerfortpflanzung jetzt formaler untersuchen. Dazu sei die Ausgangsfolge mit $\{x_n\}$, die Differenzenfolge mit $\{d_n\}$, die quantisierte Differenzenfolge mit $\{\hat{d}_n\}$ und die rekonstruierte Folge mit $\{\hat{x}_n\}$ bezeichnet. Startwert sei x_0, und es gelte $\hat{x}_0 = x_0$. Dann gilt:

Lemma 10.3 $\hat{x}_n = x_n + \sum_{k=1}^{n} q_k$.

Beweis:

$$\begin{aligned} d_n &= x_n - x_{n-1}, \text{ falls } n > 0 \\ \hat{d}_n &= Q[d_n] = d_n + q_n, \ q_n \text{ ist der Quantisierfehler} \\ \hat{x}_n &= \hat{x}_{n-1} + \hat{d}_n, \text{ falls } n > 0 \\ &= x_n + \sum_{k=1}^{n} q_k, \end{aligned}$$

denn $\hat{x}_n - \hat{x}_{n-1} = \hat{d}_n = x_n - x_{n-1} + q_n$ liefert mit $\hat{x}_0 = x_0$ per Teleskopsumme:

$$\begin{aligned} &(\hat{x}_n - \hat{x}_{n-1}) + (\hat{x}_{n-1} - \hat{x}_{n-2}) + \ldots + (\hat{x}_1 - x_0) = \\ &\quad (x_n - x_{n-1}) + (x_{n-1} - x_{n-2}) + \ldots + (x_1 - x_0) + \sum_{k=1}^{n} q_k \,. \end{aligned}$$

□

Eine leichte Modifikation verhindert diese sich im Term $\sum_{k=1}^{n} q_k$ ausdrückende Fehlerfortpflanzung: setze nun $d_n = x_n - \hat{x}_{n-1}$; damit gilt:

$$\hat{x}_n = \hat{x}_{n-1} + \hat{d}_n = \underbrace{\hat{x}_{n-1} + d_n}_{x_n} + q_n \,.$$

Hierbei kann $\hat{x}_{n-1}$ als ein Schätzer für x_n interpretiert werden. Diese Idee wird im folgenden verallgemeinert.

10.2 Prädikative Differentialcodierung

Bild 28 zeigt das Schema eines Codierers im *prädikativen Differentialverfahren*. Dieses Schema wird englisch als „differential pulse code modulation",

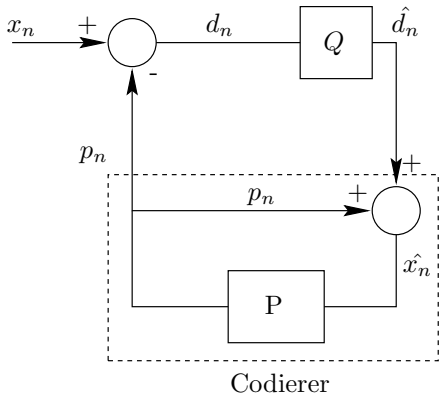


Abbildung 28: Der Codierer des prädikativen Differentialverfahrens

kurz *DPCM*, bezeichnet. Q bezeichnet dabei den zur Codierung der Differenzenfolge verwendeten Quantisierer und P den *Prädikator* oder *Vorhersager* genannten Schätzer für x_n. Während am Ende des vorigen Abschnitts $p_n = \hat{x}_{n-1}$ galt, kann man im allgemeinen

$$p_n = f(\hat{x}_{n-1}, \hat{x}_{n-2}, \ldots, \hat{x}_0)$$

ansetzen. Bemerkenswert ist, dass der Codierer den Decodierer (im gestrichelt eingekästelten Bereich in Bild 28) simuliert.

Wie ist die Vorhersagefunktion f zu wählen? Da für die Differenzenfolge $d_n = x_n - p_n$ gilt, beeinflusst die Schätzfolge p_n offenbar u. a. die Varianz σ_d^2 der Differenzfolge, die möglichst klein sein sollte, um eine gute (verlustbehaftete) Komprimierung durchführen zu können. Recht allgemein ist folgender Zusammenhang:

Satz 10.4 *Unter den Annahmen*

1. $p_n = f(\hat{x}_{n-1}, \hat{x}_{n-2}, \ldots, \hat{x}_0) \sim f(x_{n-1}, x_{n-2}, \ldots, x_0)$ *und*
2. $\{x_n\}$ *ist stationär*

kann man zeigen, dass σ_d^2 *durch den bedingten Erwartungswert*

$$E[x_n \mid x_{n-1}, \ldots, x_0]$$

minimiert wird.

Da dies immer noch zu kompliziert ist, nehmen wir weiter an, die Vorhersagefunktion sei linear, d. h.

$$p_n = \sum_{i=1}^{N} a_i \hat{x}_{n-i}\,;$$

N heißt auch *Ordnung* des Prädikators. Wegen der ersten Annahme des Satzes ist also

$$\sigma_d^2 = E[(x_n - \sum_{i=1}^{N} a_i x_{n-i})^2]$$

zu minimieren. Hierzu setzen wir die partiellen Ableitungen bezüglich der a_i gleich Null:

$$\begin{aligned}
\frac{\partial \sigma_d^2}{\partial a_1} &= -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right) x_{n-1}\right] = 0 \\
\frac{\partial \sigma_d^2}{\partial a_2} &= -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right) x_{n-2}\right] = 0 \\
\vdots &\quad \vdots \\
\frac{\partial \sigma_d^2}{\partial a_N} &= -2E\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right) x_{n-N}\right] = 0.
\end{aligned}$$

Unter Benutzung der *Autokorrelationsfunktion*

$$R_{xx}(k) = E[x_n x_{n+k}]$$

können wir also ansetzen:

$$\begin{aligned}
\sum_{i=1}^{N} a_i R_{xx}(i-1) &= R_{xx}(1) \\
\sum_{i=1}^{N} a_i R_{xx}(i-2) &= R_{xx}(2) \\
\vdots &\quad \vdots \\
\sum_{i=1}^{N} a_i R_{xx}(i-N) &= R_{xx}(N)
\end{aligned}$$

Dies lässt sich auch in Matrixform schreiben:

$$\mathbf{RA} = \mathbf{P}$$

wobei

$$\mathbf{R} = \begin{bmatrix}
R_{xx}(0) & R_{xx}(1) & R_{xx}(2) & \dots & R_{xx}(N-1) \\
R_{xx}(1) & R_{xx}(0) & R_{xx}(1) & \dots & R_{xx}(N-2) \\
R_{xx}(2) & R_{xx}(1) & R_{xx}(0) & \dots & R_{xx}(N-3) \\
\vdots & \vdots & \vdots & & \vdots \\
R_{xx}(N-1) & R_{xx}(N-2) & R_{xx}(N-3) & \dots & R_{xx}(0)
\end{bmatrix}$$

Quantisierer	Prädikatorordnung	SNR (dB)	SPER (dB)
2-Bit	keine	2,43	
	1	3,37	2,65
	2	8,35	5,9
	3	8,74	6,1
3-Bit	keine	3,65	
	1	3,87	2,74
	2	9,81	6,37
	3	10,16	6,71

Tabelle 19: Leistungswerte von DPCM-Systemen mit unterschiedlichen Quantisierern und Prädikatoren

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_N \end{bmatrix} \qquad \mathbf{P} = \begin{bmatrix} R_{xx}(1) \\ R_{xx}(2) \\ R_{xx}(3) \\ \vdots \\ R_{xx}(N) \end{bmatrix}$$

Mit den soeben eingeführten Notationen können wir daher notieren:

Korollar 10.5 *Unter den Annahmen von Satz 10.4 und für lineare Prädikatorfunktionen ergibt sich der Vektor der Prädikatorkoeffizienten durch:*

$$\mathbf{A} = \mathbf{R^{-1}P}$$

Natürlich können wir in der Praxis nicht annehmen, wir hätten bereits Kenntnis der genauen in Matrix $\mathbf{R}$ —der *Autokorrelationsmatrix*— zusammengefassten Autokorrelationswerte. Statt dessen verwendet man als Schätzer dieser Werte M Datenpunkte:

$$R_{xx}(k) = \frac{1}{M-k} \sum_{i=1}^{M-k} x_i x_{i+k} \,.$$

Für eine Spracheingabe wurden auf diese Weise die Prädikatorkoeffizienten ermittelt. Diese sind natürlich von der Ordnung des Prädikators abhängig. Tabelle 19 enthält das Verhältnis SNR von Signal zur Verzerrung sowie das *Verhältnis SPER von Signal zur Vorhersage*, d. i.,

$$SPER(dB) = \frac{\sum_{i=1}^{M} x_i^2}{\sum_{i=1}^{M} (x_i - p_i)^2}$$

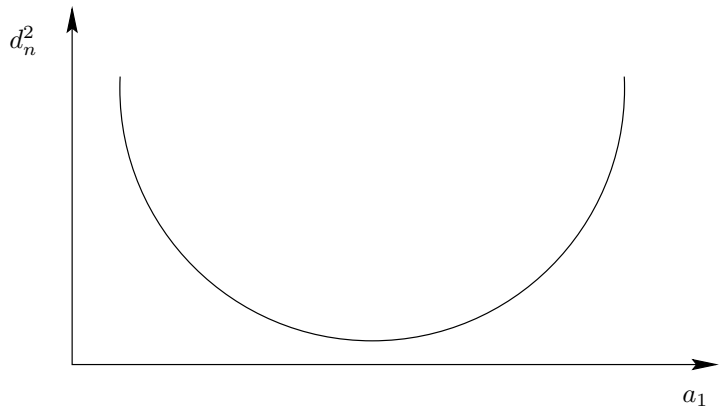


Abbildung 29: Fehlerfunktion in Abhängigkeit vom Prädikatorparameter

für dieses Sprachbeispiel, bei dem die Quelle (zu Vergleichszwecken) einmal mit einem 2-(bzw. 3-)Bit Quantisierer codiert wurde (daher keine Prädikatorordnung) und dann mit einem linearen Prädikator (der gemäß dem Ansatz von Kor. 10.5 erhalten wurde) der Ordnung $N = 1, 2, 3$.

Gerade bei Sprache ist die Stationaritätsannahme sicher falsch, was es nahelegt, adaptiv zu quantisieren.

10.3 Adaptive Differentialcodierung

Vergegenwärtigt man sich nochmals Abb. 28, so erkennt man, dass es grundsätzlich zwei Komponenten gibt, die adaptiv sein können, nämlich der Quantisierer Q und der Prädikator P. Die weiter erkennbare Rückkopplung lässt eine Voradaptierung des Quantisierers nicht sinnvoll erscheinen. Die Rückadaptierung von Q kann mit einem Jayant-Quantisierer geschehen.

Beim Entwurf der Prädikatorkoeffizienten wurde stets von der Stationarität der Quelle ausgegangen, was (z. B. bei Sprache) im Allgemeinen falsch ist. Daher erscheint eine Adaptierung der Prädikatorparameter sehr sinnvoll.

Bei DPCM mit *Voradaptierung des Prädikators* (DPCM-APF) wird die Eingabe in Blöcke unterteilt, der Schätzer der Autokorrelationskoeffizienten je Block berechnet und als Begleitinformation übertragen. Insbesondere bei Sprachübertragung kann DPCM-APF nicht tolerierbare Verzögerungen bewirken.

Wir beschäftigen uns daher jetzt mit der *Rückadaptierung des Prädikators* (DPCM-APB).

Der Fehler eines Prädikators erster Ordnung lässt sich durch

$$d_n^2 = (x_n - a_1\hat{x}_{n-1})^2$$

messen. Fassen wir d_n^2 als Funktion von a_1 auf, so ergibt sich eine Parabel wie in Bild 29. Ist der Wert $a_1^{(n)}$ weit vom Optimum entfernt, so ist eine raschere Adaptierung erwünscht als wenn fast das Optimum erreicht ist. Diesen Effekt erreichen wir durch Anwendung der Formel

$$\begin{aligned} a_1^{(n+1)} &= a_1^{(n)} - \alpha\frac{\partial d_n^2}{\partial a_1} \text{ mit} \\ \frac{\partial d_n^2}{\partial a_1} &= -2(x_n - a_1\hat{x}_{n-1})\hat{x}_{n-1} \\ &= -2d_n\hat{x}_{n-1}; \text{ also ist:} \\ a_1^{(n+1)} &= a_1^{(n)} + \alpha' d_n\hat{x}_{n-1} \end{aligned}$$

für eine die Adaptivität beeinflussende positive Konstante α bzw. α'. Da die nichtquantisierten Werte d_n nur dem Codierer zur Verfügung stehen, ersetzen wir in jener Formel d_n durch $\hat{d}_n$:

$$a_1^{(n+1)} = a_1^{(n)} + \alpha\hat{d}_n\hat{x}_{n-1}$$

Für einen Codierer höherer Ordnung $N > 1$ erhält man ganz entsprechend durch Betrachtung von

$$d_n^2 = \left(x_n - \sum_{i=1}^{N} a_i\hat{x}_{n-i}\right)^2$$

sowie der partiellen Ableitungen dieser Funktion nach den a_j:

Lemma 10.6 (Adaptionsregel für den j-ten Prädikatorkoeffizienten)

$$\begin{aligned} a_j^{(n+1)} &= a_j^{(n)} + \alpha\hat{d}_n\hat{x}_{n-j} \textit{ bzw. vektoriell} \\ \mathbf{A}^{(n+1)} &= \mathbf{A}^{(n)} + \alpha\hat{d}_n\hat{X}_{n-1} \textit{ mit} \\ \hat{X}_n &= \begin{bmatrix} \hat{x}_n \\ \hat{x}_{n-1} \\ \vdots \\ \hat{x}_{n-N+1} \end{bmatrix} \end{aligned}$$

10.4 Delta-Modulierung

Für die Sprachcodierung wird gerne der sog. Δ-*Modulator* verwendet, d. i. ein DPCM-Verfahren mit 1-Bit-Quantisierer. 1-Bit-Quantisierung bedeutet, dass der Codierer dem Decodierer quasi nur mitteilen darf, ob sich der aktuelle Eingabewert im Vergleich zum vorigen erhöht oder erniedrigt hat, und der Decodierer interpretiert diese Information, indem er $\Delta > 0$ zu seinem alten Wert hinzuaddiert bzw. davon subtrahiert. Um Fehler nicht allzu groß werden zu lassen, ist eine hohe Abtastrate erforderlich. Dennoch ist bei fixiertem 1-Bit-Quantisierer zu beobachten:

- Ist die Quelle fast konstant, so schwingt der Modulator.
- Bei steilem Signalanstieg oder -abfall kommt er nicht so schnell nach.

Diese nachteiligen Effekte lassen sich durch Wahl von Δ beeinflussen:

- Ist Δ klein, so beobachten wir schwaches Schwingen, aber schlechtes Nacheilen.
- Ist Δ groß, so sind auch die Schwingungen groß, aber das Nacheilverhalten ist gut.

Auch hier bietet sich eine *adaptive Delta-Modulierung* an; wir betrachten im folgenden die sog. *CFDM* (engl.: constant factor delta modulation). Δ_n bezeichne den vom Decodierer erhaltenen vorzeichenbehafteten Differenzterm im n-ten Schritt. Da die nachstehend angegebene Adaption mit den Faktoren $M > 0$ nicht das Vorzeichen von Δ_n beeinflusst (dieses ist schließlich die eigentlich übertragene Information), ist folgende Festlegung nicht zirkulär:

$$\begin{aligned} s_n &= \begin{cases} 1 & \text{wenn } \Delta_n > 0 \\ -1 & \text{wenn } \Delta_n < 0 \end{cases} \\ \Delta_n &= \begin{cases} M\Delta_{n-1} & \text{wenn } s_n = s_{n-1} \\ M^{-1}\Delta_{n-1} & \text{wenn } s_n \neq s_{n-1} \end{cases} \end{aligned}$$

CFDM lässt sich verbessern, indem nicht nur die letzte Ausgabe des Codierers in Betracht gezogen wird. Man mag sich klarmachen, was die folgenden Fälle „bedeuten":

$$\begin{aligned} s_n &\neq s_{n-1} = s_{n-2} \\ s_n &= s_{n-1} \neq s_{n-2} \\ s_n &= s_{n-1} = s_{n-2} \end{aligned}$$

Für die Sprachcodierung werden hierbei empfohlen:

$$\begin{array}{lllllll} s_n & \neq & s_{n-1} & = & s_{n-2} & \Rightarrow & M = 0,4 \\ s_n & \neq & s_{n-1} & \neq & s_{n-2} & \Rightarrow & M = 0,9 \\ s_n & = & s_{n-1} & \neq & s_{n-2} & \Rightarrow & M = 1,5 \\ s_n & = & s_{n-1} & = & s_{n-2} & \Rightarrow & M = 2,0 \end{array}$$

11 Teilbandcodierung

11.1 Einführung

Beobachten wir einmal folgende Beispielfolge $\{x_n\}$:

$$10 \quad 14 \quad 10 \quad 12 \quad 14 \quad 8 \quad 14 \quad 12 \quad 10 \quad 8 \quad 10 \quad 12$$

Bei Anwenden eines DPCM-Schemas müsste man die folgende Differenzenfolge codieren:

$$10 \quad 4 \quad -4 \quad 2 \quad 2 \quad -6 \quad 6 \quad -2 \quad -2 \quad -2 \quad 2 \quad 2$$

Abgesehen vom ersten Wert liegen alle Differenzen zwischen -6 und $+6$. Ein m-Bit Gleichquantisierer würde eine Schrittweite $\Delta = 12/2^m$ benutzen, was zu einem maximalen Quantisierfehler von $\Delta/2 = 6/2^m$ führte. Beachtet man, dass anstelle dieser Differenzen ohne Informationsverlust auch $z_n = \frac{x_n - x_{n-1}}{2}$ übertragen werden kann, so halbiert dies den Quantisierfehler.

Betrachten wir hingegen die Folge $y_n = \frac{x_n + x_{n-1}}{2}$ der Mittelwerte, so ergibt sich als Differenzenfolge der y_n:

$$10 \quad 2 \quad 0 \quad -1 \quad 2 \quad -2 \quad 0 \quad 2 \quad -2 \quad -2 \quad 0 \quad 2$$

Da diese Differenzen lediglich zwischen -2 und $+2$ liegen, ergibt sich ein maximaler Quantisierfehler für einen m-Bit Gleichquantisierer von $2/2^m$. Wie auch intuitiv zu erwarten, ist die Mittelwertfolge „glatter" als die Originalwertfolge.

Überträgt man die Codierungen der sich offenbar als angenehmer erweisenden Folgen y_n und z_n, so lässt sich hieraus durch Addition die eigentlich zu übertragende Folge x_n rekonstruieren. Beachtet man überdies, dass sich alle x_n-Werte aufgrund der Zusammenhänge

$$\begin{aligned} x_{2n-1} &= y_{2n} - z_{2n} \text{ und} \\ x_{2n} &= y_{2n} + z_{2n} \end{aligned}$$

wiedergewinnen lassen lediglich durch Übertragen der „halben" Folgen y_{2n} und z_{2n}, und auch die hierfür zu betrachtende Differenzenfolge $y_{2n} - y_{2n-2}$ verhält sich angenehm.

Die Idee, eine gegebene Folge durch sog. *Filter* in *(Teil-)Bänder* genannte Bestandteile zu zerlegen, die dann (evtl. mit jeweils angepassten unterschiedlichen Verfahren codiert) getrennt übertragen werden, ist Grundgedanke der *Teilbandcodierung*. Im Beispiel enthielt das Band der Mittelwertsfolge „langfristige Informationen", also *niederfrequente* Anteile, und das Band der Differenzenfolge *hochfrequente* Anteile. *Niederfrequenzbänder* lassen sich oft gut

mit DPCM-Technik komprimieren. Ein Filter, der vornehmlich niederfrequente Anteile „durchlässt", wird *Tiefpass(filter)* (engl.: low pass) genannt. Entsprechend heißt ein Filter, den fast ausschließlich hochfrequente Anteile „passieren", *Hochpass(filter)* (engl.: high pass). Anstelle von Tief- und Hochpassfiltern spricht man auch von *Glättungs- und Differenzierfiltern*.

11.2 Frequenzfilter

In Verallgemeinerung des obigen Beispiels wollen wir jetzt Filter betrachten, die eine *Eingabefolge* $\{x_n\}$ in eine *Ausgabefolge* $\{y_n\}$ vermöge

$$y_n = \sum_{i=0}^{N-1} a_i x_{n-i} + \sum_{i=1}^{M} b_i y_{n-i}$$

überführen. Auf die spezielle *Impuls-Folge*

$$x_n = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}$$

„antwortet" ein Filter mit der *Impulsantwort* $\{h_n\}$. Sind alle b_i Null, so handelt es sich um einen Filter mit *endlicher Impulsantwort* (FIR: engl.: finite impulse response) mit höchstens N Nicht-Null-Werten $h_0, \ldots, h_{N-1}$, andernfalls um einen mit *unendlicher Impulsantwort* (IIR: engl.: infinite impulse response).[14]

Beispielsweise errechnet man als Impulsantwort für den durch die Nicht-Null-Werte $a_0 = 1,25$ und $a_1 = 0,5$ spezifizierten Filter:

$$\begin{aligned} h_0 &= a_0 x_0 + a_1 x_{-1} = 1,25 \\ h_1 &= a_0 x_1 + a_1 x_0 = 0,5 \\ h_n &= 0 \text{ sonst} \end{aligned}$$

Für den durch die Nicht-Null-Werte $a_0 = 1$ und $b_1 = 0,9$ spezifizierten Filter bekommt man als Impulsantwort:

$$\begin{aligned} h_0 &= a_0 x_0 + b_1 h_{-1} = 1(1) + 0,9(0) = 1 \\ h_1 &= a_0 x_1 + b_1 h_0 = 1(0) + 0,9(1) = 0,9 \\ h_2 &= a_0 x_2 + b_1 h_1 = 1(0) + 0,9(0,9) = 0,81 \\ &\vdots \quad\quad \vdots \\ h_n &= (0.9)^n \end{aligned}$$

[14] Wir beschäftigen uns hier nur mit eindimensionalen Signalen und führen den zweidimensionalen Fall hierauf zurück. Der allgemeine mehrdimensionale Fall ist z. B. in [13] dargestellt.

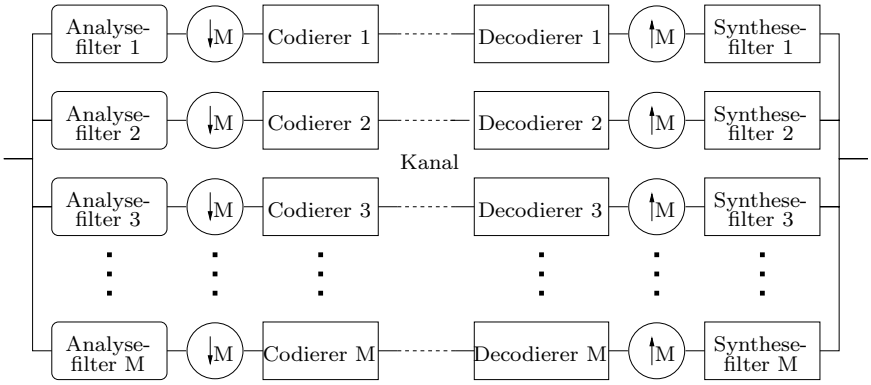


Abbildung 30: Blockdiagramm zur Teilbandcodierung

Ähnlich erhält man als Impulsantwort h_n bzw. h'_n des „Mittelwertfilters y_n" bzw. des „Differenzfilters z_n" aus dem Beispiel im vorigen Abschnitt:

$$h_n = \begin{cases} 0,5 & n = 0 \\ 0,5 & n = 1 \\ 0 & \text{sonst} \end{cases} \qquad h'_n = \begin{cases} 0,5 & n = 0 \\ -0,5 & n = 1 \\ 0 & \text{sonst} \end{cases}$$

Allgemein legt die Impulsantwort die Filterfunktion in folgender Weise fest:

$$y_n \quad = \quad \sum_{k \geq 0} h_k x_{n-k}$$

Idealisiert kann man sich ein Teilbandcodierverfahren wie in Bild 30 vorstellen: Auf der Codiererseite wird der einkommende Signalstrom zunächst durch eine Filterbank in M (sich evtl. überlappende) Teilbänder zerlegt; jeder gefilterte Signalstrom wird dann „dezimiert", d.h., nur jedes M-te Datum wird dem eigentlichen Codierer weiter geleitet. In dem Beispiel des vorigen Abschnitts hatten wir gesehen, das dies für die Rekonstruktion hinreichend ist, und jene Überlegung lässt sich verallgemeinern. Anstelle von *Dezimierung* spricht man auch von *Unterabtastung*. Um unterschiedliche Charakteristika verschiedener Bänder desselben Signals auszunutzen, werden die für die Einzelbänder benutzten Codierverfahren unterschiedlich sein; typischerweise sind niederfrequente Bänder DPCM-quantisiert und höherfrequente skalar- oder vektorquantisiert. Im Decodierer müssen umgekehrt die rekonstruierten Signale der verschiedenen Bänder wieder zusammengesetzt werden. Die Synthesefilter sind natürlich im Allgemeinen von den Analysefiltern verschieden.

Die Tab. 20 bis 22 zeigen die Arbeitsweise des „Tiefpassfilters" (also des Mittelwertfilters) und des „Hochpassfilters" (des Differenzenfilters) aus dem

10	14	10	12	14	8	14	12
10	12	8	12	10	6	10	12
12	10	8	6	8	10	12	14
8	6	4	6	4	6	8	10
14	12	10	8	6	4	6	8
12	8	12	10	6	6	6	6
12	10	6	6	6	6	6	6
6	6	6	6	6	6	6	6

Tabelle 20: Ein Beispiel„bild"

Dezimierte Tiefpassausgabe				Dezimierte Hochpassausgabe			
5	12	13	11	5	-2	1	3
5	10	11	8	5	-2	1	2
6	9	7	11	6	-1	1	1
4	5	5	7	4	-1	-1	1
7	11	7	5	7	-1	-1	1
6	10	8	6	6	-2	-2	0
6	8	6	6	6	-2	0	0
3	6	6	6	3	0	0	0

Tabelle 21: Ein Beispielbild — gefilterte und dezimierte Zeilen

LL-Bereich				HL-Bereich			
2,5	6	6,5	5,5	2,5	-1	0,5	1,5
5,5	9,5	9	9,5	5,5	-1,5	1	1,5
5,5	8	6	6	5,5	-1	-1	1
6	9	7	6	6	0	0	0
LH-Bereich				HH-Bereich			
2,5	6	6,5	5,5	2,5	-1	0,5	1,5
0,5	-0,5	-2	1,5	0,5	0,5	0	-0,5
1,5	3	1	-1	1,5	0	0	0
0	1	-1	0	0	0	1	0

Tabelle 22: Ein Beispielbild — 4 gefilterte und dezimierte Teilbilder

Beispiel im vorigen Abschnitt, so wie diese üblicherweise im Falle von Bildern eingesetzt werden: zunächst wird zeilenweise, dann spaltenweise gefiltert. Links und oben wird dabei eine Spalte bzw. Zeile mit Nullen angesetzt. Dagegen bleiben die rechte Spalte und die unterste Zeile unbeachtet. Leider sind in diesem Mini-Beispiel die Entstellungen an den Rändern noch überdeutlich. Davon einmal abgesehen, soll auch hier schon klar werden, dass die Hauptinformation des Bildes im niederfrequenten (oben links dargestellten) Teilbild vorhanden ist und „fast keine" Information im hochfrequenten Teilbild (unten rechts) in Tab. 22. „Gute" Filter konzentrieren möglichst viel Energie im niederfrequenten Teilbild. Um die verschiedenartigen Dezimierungskaskaden zu verdeutlichen, werden, wie in Tabelle 22, die Abkürzungen LL, HL, LH und HH verwendet, wobei L hier für engl. „low" und H für engl. „high" steht.

11.3 Filterbänke

Filterbänke sind oft kaskadenartig aus Paaren von Tief- und Hochpassfiltern aufgebaut. Besonders beliebt sind *Spiegelfilter* (engl.: quadrature mirror filters QMF); ist $\{h_n\}$ die Impulsantwort des Tiefpassfilters, so ist $\{(-1)^n h_{N-1-n}\}$ die Impulsantwort des Hochpassfilters. Wie oben nachgerechnet, sind die Filter aus dem einführenden Beispiel im vorigen Abschnitt von dieser Art. Die beliebten Filter von Johnston und Smith-Barnwell sind ebenfalls von dieser Gestalt, so dass zu ihrer Angabe die Impulsantwort des Tiefpassfilters genügt.

Natürlich beeinflusst die Wahl der Filterbank auch die Art und Weise der Quantisierung der einzelnen Teilbänder. Nehmen wir an, R Bits sollen pro Datum höchstens übertragen werden, und die Eingabesignalvarianz des k-ten Quantisierers ($1 \leq k \leq M$) sei $\sigma_{y_k}^2$, so arbeitet folgendes Verfahren gut für die Verteilung der zur Verfügung stehenden R Bits auf die Teilbänder:

1. Berechne $S_k = \sigma_{y_k}^2$ und setze $R_k = 0$ für $1 \leq k \leq M$.
2. Wähle ein k', so dass $S_{k'}$ maximal für $1 \leq k' \leq M$.
3. Inkrementiere $R_{k'}$; dividiere $S_{k'}$ durch zwei.
4. Dekrementiere R. Falls $R > 0$, gehe zu Schritt 2.

Nach Durchlauf des Verfahrens sollten R_k Bits für das k-te Band genommen werden.

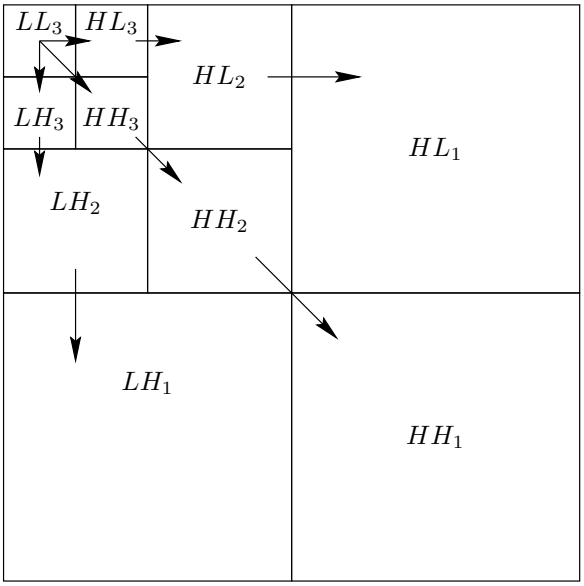


Abbildung 31: Abkömmlingsbaum à la Shapiro

11.4 Shapiros EZW-Algorithmus

Eine andere, adaptive Idee zur Bitverteilung bei der Bildkompression mit Teilbandverfahren ist im *EZW-Algorithmus*[15] von Shapiro enthalten [18]. Wie oben erläutert, wird ein Bild zeilen- und spaltenweise mit einem Tief- bzw. Hochpassfilter in vier Bandbereiche zerlegt; für den Low-Low-Bereich, in dem sich ja die Energie bei geeigneter Filterwahl konzentrieren sollte, kann man nun diese Zerlegung wiederholen, was bei dreimaliger Wiederholung zu der Aufteilung des „gefilterten und dezimierten" Bildes führt, die in Abb. 32 wiedergegeben ist.

Wir erläutern den Algorithmus am besten anhand eines Beispiels. Es seien die in Abb. 33 gegebenen gefilterten und dezimierten Werte vorgegeben. Der betragsmäßig größte Wert ist 63. Als initialen Schwellwert T_0 wählen wir daher dessen gerundete Hälfte, d. i. $T_0 = 32$. Im ersten Hauptdurchgang wird nun das Bild, wie in 32 durch den Pfeil angegeben, abgetastet. Für Teilbilder wird rekursiv dieselbe Abtastreihenfolge gewählt. In den Hauptdurchgängen wird ein Strom von Symbolen aus { POS, NEG, IZ, ZTR } erzeugt; diese Zeichen sind natürlich mit je zwei Bit codierbar. Dabei bedeutet POS, dass der abgetastete Wert größer als der aktuelle Schwellwert T_k ist, NEG, dass er negativ und kleiner als $-T_k$ ist und IZ bzw. ZTR, dass er im Intervall

[15] EZW steht für engl.: „Embedded Zerotree Wavelet"; im Ggs. zum Namen ist das Grundprinzip des Vorgehens nicht auf Wavelets beschränkt.

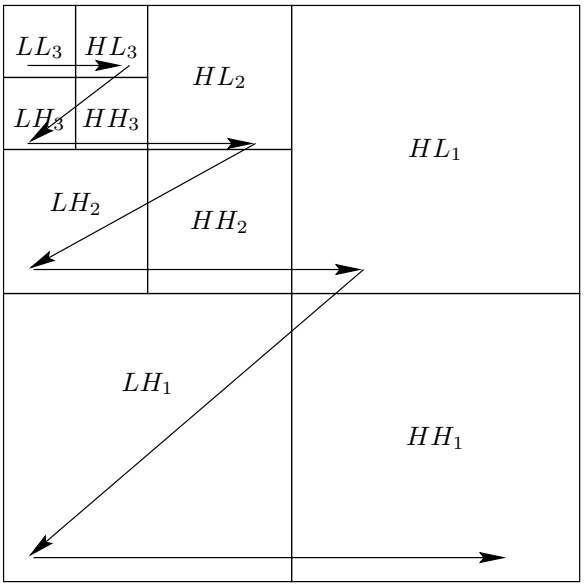


Abbildung 32: Abtasten von Filterkoeffizienten à la Shapiro

63	-34	49	10	7	13	-12	7
-31	23	14	-13	3	4	6	-1
15	14	3	-12	5	-7	3	9
-9	-7	-14	8	4	-2	3	2
-5	9	-1	47	4	6	-2	2
3	0	-3	2	3	-2	0	4
2	-3	6	-4	3	6	3	6
5	11	5	6	0	3	-4	4

Abbildung 33: Ein dreimal wiederholt gefiltertes und dezimiertes „Bild”

$[-T_k, T_k]$ liegt. Werte, für die POS oder NEG erzeugt wird, heißen auch *signifikant*, die übrigen *insignifikant*. ZTR (engl.: zerotree) bedeutet überdies, dass alle „Abkömmlinge" im Sinne des in Bild 31 mit Pfeilen dargestellten Baumes (der im übrigen wieder rekursiv fortgesetzt zu denken ist, s. u. Beispiel) im Intervall $[-T_k, T_k]$ liegen und so einen *Nullbaum* bilden; dahingegen bezeichnet IZ eine *vereinzelte Null* (engl.: isolated zero).

Der erste Hauptdurchgang ist in Tabelle 23 aufgelistet. Als ersten Wert erhalten wir so 63. Dieser ist größer als der Schwellwert 32, weshalb Symbol POS generiert wird. Da -34 kleiner als -32 ist, wird als nächstes NEG erzeugt. -31 ist im Intervall $[-32, 32]$, im darunterhängenden Teilbaum liegt aber noch ein „signifikanter Wert", nämlich 47, so dass eine vereinzelte Null vorliegt und IZ erzeugt wird. Dahingegen ist sowohl der Eintrag im HH_3-Band wie alle seine Abkömmlinge (in den Bändern HH_2 und HH_1) insignifikant, weshalb wir bei (3) ein ZTR ausgeben. Insbesondere bedeutet dies, dass wir keine weiteren Symbole mehr für Einträge aus dem HH_2- und HH_1-Band erzeugen werden in diesem Hauptdurchgang.

Jetzt wird das Teilband HL_2 abgetastet. Bei (4) ist zu bemerken, dass die Abkömmlinge des Eintrags 10 eben $\{-12, 7, 6, 1\}$ sind, was durch rekursive Forsetzung des in Bild 31 dargestellten Abkömmlingsbaums zu sehen ist, und diese sämtlich dem Betrage nach unter dem Schwellwert liegen, weshalb ein Nullbaum diagnostiziert wird. Bei (5) (innerhalb der Analyse des LH_2-Bands) merken wir an, dass zwar 14 selbst unterhalb des Schwellwerts liegt, das Abkömmlingsteilband $\{-1, 47, -3, 2\}$ jedoch mit 47 einen signifikanten Wert enthält, weshalb IZ auszugeben ist. Beachte bei (6), dass das HH_2-Band bereits berücksichtigt wurde und daher gleich zur Codierung des HL_1-Bands geschritten wird. Da die HL_1 und LH_1-Bänder keine Abkömmlinge besitzen, können wir die ZTR und IZ-Symbole zu einem einzigen Z-Symbol verschmelzen und so noch kompakter codieren.

Tabelle 24 enthält die Ergebnisse der sich anschließenden ersten Verfeinerung, bei der für die vier signifikanten Werte (aus dem ersten Hauptdurchgang) angegeben wird, ob sie (dem Betrage nach) im Intervall $[32, 48)$ oder in $[48, 64)$ liegen.

Beim nächsten Hauptdurchgang werden nur die bislang für insignifikant gehaltenen Werte berücksichtigt; die bislang signifikanten Werte werden hingegen in diesem Durchgang als Nullen (im Sinne der Nullbaumkonstruktion) angesehen. Als Schwellwert nehmen wir $T_1 = T_0/2 = 16$. Im zweiten Verfeinerungsdurchgang hingegen werden alle aus den bisherigen (beiden) Hauptdurchgängen gewonnenen Werte verfeinert, was also zu einer Intervalleinteilung $[16, 24)$, $[24, 32)$, $[32, 40)$, $[40, 48)$, $[48, 56)$ und $[56, 64)$ führt.

Insbesondere wird so hoffentlich die Bedeutung des Wortes „embedded" deutlich: Jeder hochauflösendere Code enthält sämtliche niederauflösenden

Kommentar	Teilband	Wert	Symbol	rekonstruierter Wert
(1)	LL_3	63	POS	48
	HL_3	-34	NEG	-48
(2)	LH_3	-31	IZ	0
(3)	HH_3	23	ZTR	0
	HL_2	49	POS	48
(4)	HL_2	10	ZTR	0
	HL_2	14	ZTR	0
	HL_2	-13	ZTR	0
	LH_2	15	ZTR	0
(5)	LH_2	14	IZ	0
	LH_2	-9	ZTR	0
	LH_2	-7	ZTR	0
(6)	HL_1	7	Z	0
	HL_1	13	Z	0
	HL_1	3	Z	0
	HL_1	4	Z	0
	LH_1	-1	Z	0
(7)	LH_1	47	POS	48
	LH_1	-3	Z	0
	LH_1	-2	Z	0

Tabelle 23: Ergebnisse des ersten Hauptdurchgangs

Koeffizienten-größe	Symbol	Rekonstruktions-größe
63	1	56
34	0	40
49	1	56
47	0	40

Tabelle 24: Die erste Verfeinerung

Codes eines Bildes als Präfix; m. a. W., EZW eignet sich sehr gut zur fortschreitenden Bildübertragung.

11.5 Filter aus Transformationen

Wie kann man *systematisch* ein Eingabesignal in Frequenzbereiche zerlegen? Eine Möglichkeit ist allgemein bekannt: die *Fourier-Entwicklung bzw. -Transformation* . Jede reelle (oder komplexe) Funktion f mit Periode T lässt sich darstellen in der Form:

$$\begin{aligned}
f(t) &= \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos\left(n\frac{2\pi}{T}t\right) + \sum_{n=1}^{\infty} b_n \sin\left(n\frac{2\pi}{T}t\right) \text{ mit} \\
a_n &= \frac{1}{T}\int_0^T f(t)\cos\left(n\frac{2\pi}{T}t\right)dt, \\
b_n &= \frac{1}{T}\int_0^T f(t)\sin\left(n\frac{2\pi}{T}t\right)dt; \qquad \text{alternativ:} \\
f(t) &= \sum_{-\infty}^{\infty} c_n e^{in\frac{2\pi}{T}t} \text{ mit} \\
c_n &= \frac{1}{T}\int_0^T f(t)e^{-in\frac{2\pi}{T}t}dt
\end{aligned}$$

Hierbei ist, wie üblich, $i = \sqrt{-1}$. So erhält man nicht nur eine „Sinusentwicklung" von periodischen Funktionen, sondern auch eine „Sinus-Approximation", indem man die Reihenentwicklung nach endlich vielen Gliedern abbricht[16] Diese ist wichtig, da —elektrophysikalisch bedingt— „physikalische Filter" keine „reinen" Bandpassfilter sind. Wir können diese immer nur durch Schwingungsfunktionen annähern.

Eine lediglich auf einem Intervall $[0, T]$ definierte Funktion lässt sich leicht „periodisch fortsetzen".[17] Da wir zumeist mit „diskreten Funktionen", sprich Folgen $\{x_n\}$ der Länge N beschäftigen, ist für uns die *diskrete Fourier-Transformation* wichtig:

$$\begin{aligned}
c_k &= \frac{1}{N}\sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \\
x_n &= \sum_{k=1}^{N-1} c_k e^{\frac{i2\pi kn}{N}}
\end{aligned}$$

Im Kapitel 13 werden wir hierauf zurückkommen.

[16] Für sog. quadratisch integrierbare Funktionen ist Konvergenz garantiert.

[17] Betrachtet man den Fall „$T \to \infty$", so erhält man die Fouriertransformierte von f.

Ein flexibleres und modernes Instrument zur Signalzerlegung liefern *Wavelets*. Bei ihnen steht sowohl der Zeit- als auch der Frequenzbereich parametrisiert zur Verfügung. Für Informationen aus dem Internet zu Wavelets verweisen wir auf die Adressen 6 und 7.

Sehr beliebt sind die sog. *Haar-Wavelets*. Aus der einfachen *Urfunktion* (engl.: mother wavelet)

$$\psi_{0,0}(x) = \begin{cases} 1 & 0 \le x < \frac{1}{2} \\ -1 & \frac{1}{2} \le x < 1 \\ 0 & \text{sonst} \end{cases} \quad \text{wird durch}$$

$$\begin{aligned} \psi_{j,k}(x) &= \psi_{0,0}(2^j x - k) \\ &= \begin{cases} 1 & k2^{-j} \le x < (k+\frac{1}{2})2^{-j} \\ -1 & (k+\frac{1}{2})2^{-j} \le x < (k+1)2^{-j} \\ 0 & \text{sonst} \end{cases} \end{aligned}$$

eine Schar von Funktionen. j kann als *Streckparameter* (*Frequenzparameter*) und k als *Verschiebeparameter* (*Zeitparameter*) gedeutet werden.

Um eine Idee von der Arbeitsweise von Wavelets zu erhalten, betrachten wir exemplarisch nun als Urfunktion

$$\phi_{0,0}(x) = \begin{cases} 1 & 0 \le x < 1 \\ 0 & \text{sonst} \end{cases}$$

sowie die erzeugte Schar

$$\phi_{j,k}(x) = \phi_{0,0}(2^j x - k)\,.$$

Angenommen, wir wollten eine Funktion $f : [0, N] \to \mathbb{R}_+$ approximieren. In erster Näherung setzen wir

$$\begin{aligned} \phi_f^0(t) &= \sum_{k=0}^{N-1} c_{0,k}\phi_{0,k} \quad \text{mit} \\ c_{0,k} &= \int_k^{k+1} f(t)[\phi_{0,k}(t)]dt \end{aligned}$$

Durch Wahl eines höheren Frequenzparameters können wir die Genauigkeit der Approximation steigern:

$$\begin{aligned} \phi_f^j(t) &= \sum_{k=0}^{2^j N-1} c_{j,k}\phi_{j,k} \quad \text{mit} \\ c_{j,k} &= 2^j \int_{k/2^j}^{(k+1)/2^j} f(t)dt\,. \end{aligned}$$

Offensichtlich gilt

$$c_{j-1,k} \quad = \quad 1/2(c_{j,2k} + c_{j,2k+1})\,. \tag{6}$$

Unser Ziel ist es, Funktionen in Bestandteile (mit evtl. unterschiedlichen Charakteristika) zu zerlegen. Wenn nun ϕ_f^1 die Funktion f genügend approximiert, kann man ϕ_f^0 als niederfrequenten Anteil betrachten und muss dann die Differenz $\phi_f^1 - \phi_f^0$ diskutieren. Es gilt wegen Gleichung (6):

$$\begin{aligned}
&\phi_f^1(t) - \phi_f^0(t) \\
&\quad = \quad \begin{cases} c_{0,k} - c_{1,2k} = -1/2c_{1,2k} + 1/2c_{1,2k+1} & k \le t < k + 1/2 \\ c_{0,k} - c_{1,2k+1} = 1/2c_{1,2k} - 1/2c_{1,2k+1} & k + 1/2 \le t < k + 1 \end{cases}
\end{aligned}$$

M. a. W., die Differenz lässt sich leicht mit Haar-Wavelets ausdrücken, nämlich:

$$\phi_f^1(t) - \phi_f^0(t) = \underbrace{(-c_{1,2k} + c_{1,2k+1})}_{b_{0,k}} \psi_{0,k}(t)\,.$$

Die $2N$-Punkte Folge $c_{1,k}$ kann so in zwei N-Punkt-Folgen $c_{0,k}$ und $b_{0,k}$ zerlegt werden; die zweite Folge kann als Faktoren von Wavelets interpretiert werden. Allgemein lässt sich jede Funktionenschar $\phi_{j,k}$, die gewissen Skalier-, Darstellungs- und Integrabilitätsbedingungen genügt, dafür benutzen, eine zugehörige Wavelet-Familie zu definieren.

Aus der Darstellungsbeziehung

$$\phi_{0,0}(t) = \sum h_n \phi_{1,n}(t)$$

gewinnt man die Impulsantwort h_n des Glättungsfilters, denn $\phi_{0,0}$ kann als stetige Form des Impulses gesehen werden, und die Darstellung

$$\psi_{0,0}(t) = \sum (-1)^{N-n-1} h_n \phi_{1,n}(t)$$

des zugehörigen Ur-Wavelets liefert die Impulsantwort des Differenzierfilters. Beliebt sind insbesondere die so zu erhaltenen *Daubechies- und Coiflet-Filter*. Wavelets liefern Spiegelfilter.

Wichtig ist schließlich, dass man die Aufspaltung des Ursignals in Glätte- und Differenzanteil weitertreiben kann, indem der Glätteanteil rekursiv weiter aufgespalten wird. Die Differenzanteile m-ter Stufe lassen sich dann durch die Wavelets $\psi_{m,k}$ darstellen. Die so mögliche rekursive *Multiresolutionsanalyse* ist einer der wesentlichen Vorteile Wavelet-basierter Zeit/Frequenz-Analyse gegenüber dem (älteren) Fourier-Ansatz. Diese Zusammenhänge sind auch sehr schön abstrakt im siebenten und achten Kapitel von [8] dargestellt. Populärwissenschaftliche Artikel sind [14, 21].

12 Fraktale Codierung

Der mathematische Grundgedanke dieser Codierungstechnik ist sehr einfach: Statt ein „Datum" zu übertragen, übertrage man die Parameter einer kontrahierenden Abbildung f, deren Fixpunkt eben dieses Datum ist. Der Decodierer kann nun durch ein einfaches Iterationsverfahren (berechne $y_i = f(y_{i-1})$ ausgehend von einem beliebigen „Punkt" y_0 solange, bis eine gewünschte Genauigkeit erreicht wird) oder schneller noch durch randomisierte Techniken das zu übertragende Datum rekonstruieren. Nach dem Banachschen Fixpunktsatz funktioniert diese Idee prinzipiell für jeden vollständigen metrischen Raum, zum Beispiel auf dem $\mathbb{R}^2$ mit dem Euklidischen Abstand als Metrik ρ_E.[18] *Iterierte Funktionensysteme* (IFS) gestatten die Beschreibung von Bildern als Fix„punkte".

Es sei (X, ρ) ein metrischer Raum. Die *r-Umgebung* von $A \subseteq X$ ist gegeben durch:

$$B_r(A) := \{y \in X | (\exists x \in A)(\rho(x, y) < r)\} (= \bigcup_{x \in A} B_r(x)).$$

Bezeichne $\mathcal{K}(X, \rho)$ die Menge aller nicht leeren aber kompakten Teilmengen von (X, ρ). Es seien $A, B \in \mathcal{K}(X, \rho)$. Die Abbildung

$$\rho_H(A, B) := \inf\{r : A \subseteq B_r(B) \text{ und } B \subseteq B_r(A)\}$$

ist eine Metrik auf $(\mathcal{K}(X, \rho), \rho_H)$. Sie heißt *Hausdorff-Metrik* oder *Hausdorff-Abstand* oder. Ein *IFS* F (auf X), ist gegeben durch eine endliche Liste $F = (F(1), \ldots, F(n))$ von kontrahierenden Ähnlichkeitsabbildungen. Zugeordnet ist die *Liste der Ähnlichkeitsfaktoren* $(r_{F(1)}, \ldots, r_{F(n)})$. Jedes IFS $F = (F(1), \ldots, F(n))$ auf (X, ρ) lässt sich als Abbildung $\mathcal{K}(X, \rho) \to \mathcal{K}(X, \rho)$ auffassen durch

$$F(A) := \bigcup_{i=1}^{n} F(i)(A).$$

F ist so eine Kontraktion auf $(\mathcal{K}(X, \rho), \rho_H)$ mit einem Kontraktionsfaktor $r_F := \max_{i=1}^{n} r_{F(i)} < 1$. Nach dem *Banachschen Fixpunktsatz* bestimmt F so eindeutig einen Fix„punkt" $A_F \in \mathcal{K}(X, \rho)$, falls (X, ρ) (und damit $(\mathcal{K}(X, \rho), \rho_H)$) vollständig ist.

Da sich der Abstand des Fixpunktes x_f einer kontrahierenden Abbildung f (mit Kontraktionsfaktor r_f) auf einem vollständigen metrischen Raum (X, ρ) von einem beliebigen Punkt $x \in X$ durch $(\rho(x, x_f) \leq \frac{1}{1-r_f} \cdot \rho(x, f(x)))$ abschätzen lässt, ist die n-te Iterierte $f^n(x)$ höchstens $\frac{r_f^n}{1-r_f} \cdot \rho(x, f(x))$ vom

[18] Näheres zu metrischen Räumen findet man beispielsweise in [7]. Der Zusammenhang zu iterierten Funktionensystemen ist in [4] oder deutschsprachig in [6] dargestellt.

Fixpunkt entfernt, was[19] eine recht einfache Abschätzung des ansonsten nicht so einfach zu bestimmenden Hausdorff-Abstandes liefert und überdies ein schnelles Abbruchkriterium bei der Decodierung „fraktalen Codes" liefert.

Als Beispiel betrachten wir die Koch-Kurve, als Raum nehmen wir $X = [0,1]\times[0,1]$ mit der Euklidischen Metrik und als IFS $F = (F(1), F(2), F(3), F(4))$, wobei die einzelnen Abbildungen wie folgt definiert sind.

$$
\begin{aligned}
F(1)(x_1,x_2) &= \frac{1}{3}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \\
F(2)(x_1,x_2) &= \frac{1}{3}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} \frac{2}{3} \\ 0 \end{pmatrix}, \\
F(3)(x_1,x_2) &= \frac{1}{3}\begin{pmatrix} \cos(60°) & -\sin(60°) \\ \sin(60°) & \cos(60°) \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} \frac{1}{3} \\ 0 \end{pmatrix}, \\
F(4)(x_1,x_2) &= \frac{1}{3}\begin{pmatrix} \cos(-60°) & -\sin(-60°) \\ \sin(-60°) & \cos(-60°) \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{3}\sin(60°) \end{pmatrix}.
\end{aligned}
$$

Der Deutlichkeit halber wollen wir in Abb. 34 mit der kompakten Menge $\{(x,0)|0 \le x \le 1\}$ starten. Beachte, dass der Hausdorff-Abstand der fünften Iteration von der Koch-Kurve sich mit $\frac{1}{3^5(1-1/3)}\frac{1}{3} \approx 0,002$ abschätzen lässt.

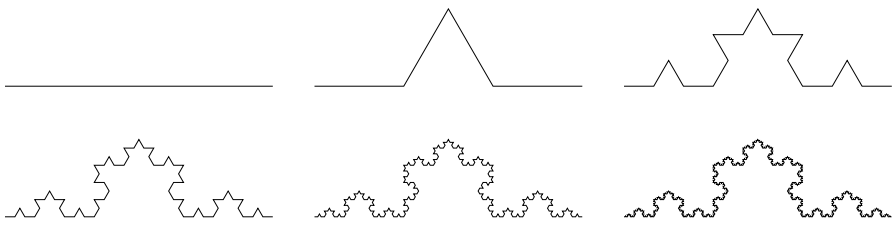


Abbildung 34: Die Koch-Kurve ist überall stetig, nirgends differenzierbar.

Wie auch anhand der Koch-Kurve deutlich wird, haben IFS-Fraktale stets eine *selbstähnliche Struktur* oder *Selbstähnlichkeit*, die zwar ansatzweise – wie B. Mandelbrot in [12] herausgearbeitet hat– sich auch „in der Natur" und damit auch bei natürlichen Bildern anzutreffen ist, sicher aber nicht in der durch IFS bedingten „Reinkultur". Andererseits wird die wesentliche Idee eines auf „Selbstähnlichkeit" abhebenden Kompressionsverfahrens klar: Die Übertragung der viermal je 6 Zahlen, die jeweils die affinen Abbildungen der Koch-Kurve spezifieren, ist wesentlich effizienter als die Übertragung beispielsweise des PS-Files der letzten Approximation in Bild 34. Will man Bilder wie angedeutet fraktal codieren, so ergibt sich das Problem, wie man

[19] In der Literatur über Fraktale heißt dieser Zusammenhang auch *Collage-Theorem*.

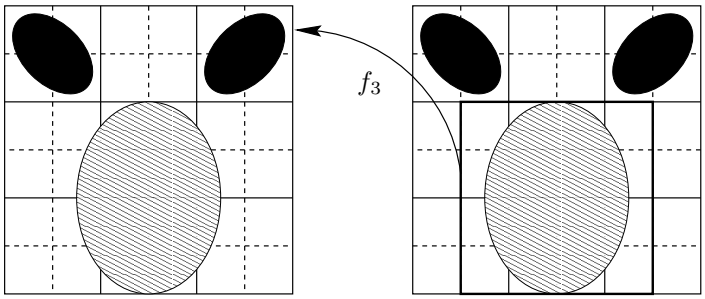


Abbildung 35: Bereichsblöcke und Urbildblöcke

denn zu einem gegebenen Bild die Parameter einer es spezifizierenden Schar von affinen Abbildungen generieren kann.

Eine erste praktikable Lösung dieser Probleme stammt von A. Jacquin. Hierbei wird zunächst das Bild in sog. *Bereichsblöcke* R_k unterteilt. Für jeden Bereichsblock wird im Bild selbst ein *Urbildblock* D_k (und eine vermittelnde kontrahierende Abbildung f_k mit $R_k = f_k(D_k)$) gesucht. Die Urbildblöcke erhält man, indem man ein $K \times K$-Fenster mit Schrittweite $K/2$ oder $K/4$ über das Bild gleiten lässt und das „günstigste" Teilbild D_k auswählt, s. Bild 35. Da nicht immer die Existenz eines optimalen Urbilds gewährleistet ist, handelt man sich hierbei einen systematischen Fehler ein, der dieses Verfahren verlustbehaftet macht. Tatsächlich wird der Unterschied von R_k zu $f_k(D_k)$ nicht per Hausdorff-Abstand, sondern durch eines der früher diskutierten Verzerrungsmaße bestimmt. Man kann zeigen, dass mit den (partiell definierten) Funktionen f_k die zugehörige mengenwertige Funktion $F(A) = \bigcup_k f_k(A)$ kontrahierend ist und somit die für IFS gemachten Überlegungen auch hier gelten.

Bislang haben wir uns lediglich um Schwarz-Weiß-Bilder in unseren Überlegungen gekümmert. Tatsächlich gilt das Gesagte im Wesentlichen auch für Grauwert- bzw. Farbbilder; hier kommt dann noch eine Anpassung der Farbinformation hinzu, so wie die Abbildung f_3 in Bild 35 nicht nur ein Verschieben und Drehen, sondern auch eine Grauwertanpassung beinhaltet. Nach Jacquin wird f_k durch $f_k = m_k \circ g_k$ zerlegt, wobei g_k eine Verschiebung und Skalierung bewirkt und m_k eine Transformation der Masse. Ist –genauer– $T_k = g_k(D_k)$ und bezeichnet t_{ij}, $i, j \in \{0, \ldots, M-1\}$, das (i,j)-te Pixel in T_k, so ist $m_k(t_{ij}) = \alpha_k t_{I_k(i,j)} + \Delta_k$. $\alpha_k t + \Delta_k$ bezeichnet dabei eine Grauwertanpassung. I_k ist eine Isometrie, die man als Permutation der Pixel innerhalb von T_k auffassen kann. Konkret betrachtet man die folgenden Isometrien:

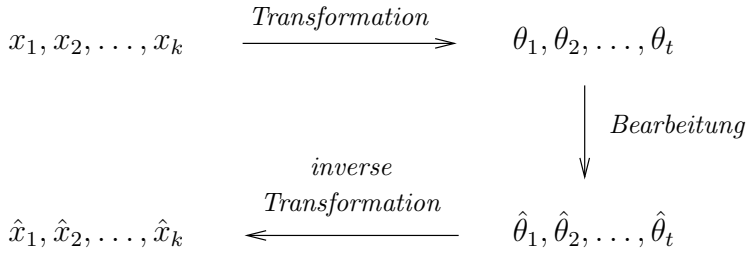
- vier Drehungen um 0, 90, 180 und 270 Grad und

- vier Spiegelungen an der vertikalen und horizontalen Spiegelachse sowie an den beiden Diagonalen.

Zur Darstellung von R_k benötigt man so g_k, α_k, Δ_k und I_k. In der Praxis schränkt man die Wahl der α_k-Werte noch drastisch ein, und häufig betrachtet man lediglich die Identität für I_k. Dennoch bleibt die Bestimmung der Parameter ein erhebliches Rechenproblem. Interessierte können sich zu diesem Thema recht aktuell unter der Web-Adresse 8, der Homepage der „Fraktalgruppe" von Dietmar Saupe, informieren. Dort gibt es u.a. Arbeiten, die zum einen die Schwierigkeit optimaler Blockzuordnung betreffen (Nachweis der NP-Härte), aber auch verschiedendste heuristische Lösungsansätze, in der unterschiedliche auch hier besprochene Methoden wie Vektorquantisierung, Fouriertransformation und Adaption in fraktale Komprimierungsschemata eingebaut werden. Weitere Informationen finden Sie unter den Web-Adressen 9 und 10.

13 Transformcodierung und JPEG-Standard

Das allgemeine Konzept der *Transformation* oder *Umformung* ist sehr gut in der Mathematik und auch in anderen Gebieten bekannt. Das Schema für eine Transformation sieht allgemein folgendermaßen aus: Seien $x_1, x_2, \ldots, x_k$ Größen, wie z.B. Zahlen, Vektoren, Funktionen etc. Dann ändern wir diese Objekte mit Hilfe einer Transformation T und bekommen $\theta_1, \theta_2, \ldots, \theta_t$, die möglicherweise ganz anders als x_i aussehen, die man aber einfach weiterverarbeiten kann. Schematisch kann man eine Transformation wie folgt darstellen:



Unser Ziel ist es, Transformationen zu finden, sodass man $\theta_1, \theta_2, \ldots, \theta_t$ effizienter (verlustbehaftet) komprimieren kann als die Folge $x_1, x_2, \ldots, x_k$ direkt. Dann ist $\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_t$ eine Rekonstruktion des Komprimierungsverfahrens der Folge $\theta_1, \theta_2, \ldots, \theta_t$, und $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_k$ ist eine Rekonstruktion der Eingabe $x_1, x_2, \ldots, x_k$. Viele der bislang vorgestellten Verfahren lassen sich auch als Transformationscodierungen begreifen. Eine konkrete verlustfreie Transformation haben wir auch schon explizit als solche kennen gelernt: die BWT in Abschnitt 5.3.

Dennoch wollen wir das Grundprinzip anhand eines einfachen Beispiels erläutern, nämlich dem schon früher verwendeten, bei dem die Daten aus einer Folge von Paaren (Länge,Gewicht) von Personen bestehen. Ein möglicher Datensatz ist in Tabelle 25 aufgeführt. Betrachtet man Länge und Gewicht als (x, y)-Koordinaten, so sieht man in Bild 36, dass sich die Daten längs einer Geraden der Art $y = 0,5x$ häufen. Drehen wir den Datensatz also durch $\theta = \mathbf{A}\mathbf{x}$, wobei

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

der zweidimensionale Eingabevektor ist und

$$\mathbf{A} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}$$

die Drehmatrix sowie

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

Größe	Gewicht
163	85
188	94
150	75
175	85
140	65
200	102
170	80
125	55
100	40
125	77
173	74
155	70
190	82
160	60

Tabelle 25: Der Eingangs-Datensatz

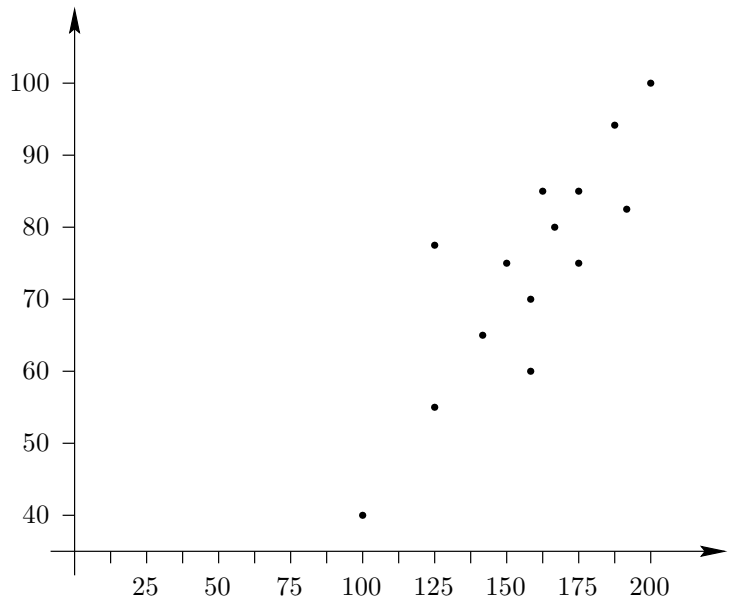


Abbildung 36: Graphische Darstellung der Eingangsdaten

x-Wert	y-Wert
184	3
210	0
168	0
195	-2
154	-4
225	2
188	-4
136	-7
107	-9
146	13
188	-11
170	-7
207	-12
170	-18

Abbildung 37: Die transformierten Werte

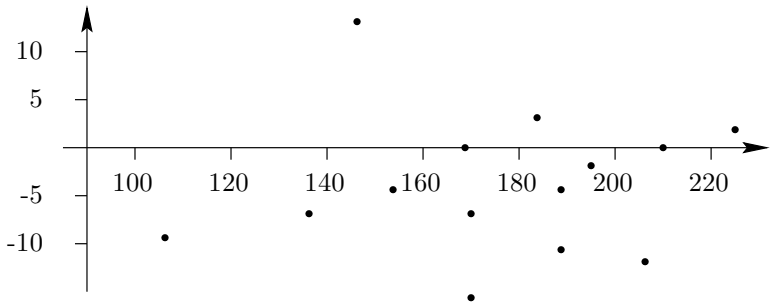


Abbildung 38: Die transformierten Werte

der Vektor der Transformierten; speziell ist hier $\phi = \arctan(0,5) \approx 26,565^o$, d. h.

$$\mathbf{A} = \begin{bmatrix} 0,89442719 & 0,4472136 \\ -0,4472136 & 0,89442719 \end{bmatrix}$$

Auf diese Weise erhält man als Tabelle der transformierten Werte die Werte aus 37, was graphisch der in Bild 38 dargestellten Situation entspricht.

Wir beobachten bei den transformierten Werten, dass sich die „Energie" in der ersten Komponente „ballt", was intuitiv eine (verlustbehaftete) Komprimierung um den Faktor zwei gestattet. Täten wir dies tatsächlich und wendeten auf die „komprimierte Folge" die inverse Transformation, gegeben

Größe	Gewicht
165	82
188	94
150	75
174	87
138	69
201	101
168	84
122	61
96	48
131	65
168	84
152	76
185	93
152	76

Tabelle 26: Die rücktransformierten Werte

durch

$$\mathbf{A}^{-1} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$

an, so erhielten wir in etwa die Werte aus Tabelle 26.

In diesem Fall gilt:

$$\sum_{i=0}^{N-1} (x_i - \hat{x}_i)^2 = \sum_{i=0}^{N-1} (\theta_i - \hat{\theta}_i)^2$$

wobei

$$\hat{\theta}_i = \begin{cases} \theta_i & i = 0, 2, 4, \ldots \\ 0 & \text{sonst} \end{cases}$$

und $\hat{x}_i$ der x_i entsprechende rekonstruierte Wert ist. Diese Eigenschaft gilt allgemein für Transformationsmatrizen mit $A^{-1} = A^T$. Letztere Eigenschaft nennt man auch *Orthonormalität*.

13.1 Einfache Transformationen

Zuerst betrachten wir Transformationen für eindimensionale Folgen, wie sie zum Beispiel (digitalisierte) Sprache und allgemeiner Audio-Folgen liefern. Es sei $p_0, p_1, p_2, \ldots$ die Eingabefolge. Wir werden folgende *lineare Transformationen* betrachten: Für jeden Block

$$x_0 = p_\ell,\ x_1 = p_{\ell+1}, \ldots, x_{N-1} = p_{\ell+N-1}$$

der Länge N, mit $\ell = 0, N, 2N, \ldots$, transformieren wir $x_0, x_1, \ldots, x_{N-1}$ folgendermaßen:

$$\theta_n = \sum_{i=0}^{N-1} x_i a_{n,i},$$

wobei die Transformation vollständig durch die Konstanten $a_{n,i}$ definiert wird. Die Länge N hängt von praktischen Anwendungen ab. Es ist klar, dass mit größerem N auch die Transformation komplexer wird. Seien nun x und θ die Vektoren. Dann können wir eine lineare Transformation in Matrixform darstellen: $\theta = Ax$. Die inverse Transformation definiert eine Matrix B, so dass $x = B\theta$ gilt. Die Matrix B ist die Inverse zu A, das heißt $AB = BA = I$. Transformationscodierung ist eine der populärsten Methoden für die Komprimierung von Bildern. Deshalb werden wir bis Ende dieses Kapitels meistens den zweidimensionalen Fall der linearen Transformation betrachten. Wir definieren solche Transformationen für einen gegebenen $N \times N$ Block

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,N-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,N-1} \\ \vdots & & & \vdots \\ x_{N-1,0} & x_{N-1,1} & \cdots & x_{N-1,N-1} \end{bmatrix}$$

folgendermaßen:

$$\Theta = AXA^T$$

und die inverse Transformation: $X = B\Theta B^T$. Alle Transformationen, die wir hier betrachten werden, sind orthonormal, das heißt, dass die inverse Matrix B (für den reellen Bereich) einfach die transponierte Matrix A^T ist:

$$B = A^{-1} = A^T.$$

Daher ist es einfach, die inverse Transformation zu bekommen:

$$X = A^T \Theta A$$

Die Zeilen der Transformationsmatrix nennen wir *Basisvektoren* und die Elemente nach der Transformation nennen wir oft *Transformationskoeffizienten*.

Man beachte, dass wir auf diese Art und Weise nur sog. *separable Transformationen* für den zweidimensionalen Fall erhalten; denn die allgemeine Transformationsformel lautet

$$\Theta_{k,\ell} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} X_{i,j} a_{i,j,k,\ell},$$

(dies wäre ein Tensorprodukt) und wir betrachten als Spezialisierung:

$$\Theta_{k,\ell} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} X_{i,j} a_{k,i} a_{\ell,j}.$$

Die Wirksamkeit einer Transformation hängt davon ab,

- wie groß die *Dekorrelation*, also die Reduktion der Korrelation, der Eingabepixel (oder Signale) ist und
- wieviel *Energiebündelung* (engl.: energy compaction) die Transformation ergibt.

Die Korrelation in einer Eingabefolge ist die Quelle der Redundanz, die in einem effizienten Komprimierungsverfahren entfernt werden soll.

Eine wichtige Eigenschaft der orthonormalen Transformationen ist, dass solche Transformationen die Energie erhalten:

$$\sum x_{i,j}^2 = \sum \theta_{i,j}^2.$$

Die Transformationen sollen nur die Energie „nach links oben" in der Matrix verschieben, also dort bündeln.

Beispiel 13.1 Betrachten wir die folgende Transformationsmatrix:

$$A = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

A ist orthonormal, weil

$$\frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^T = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^T \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Die erste Zeile der Matrix $(1/\sqrt{2}, 1/\sqrt{2})$ enspricht einem Tiefpass und die zweite Zeile $(1/\sqrt{2}, -1/\sqrt{2})$ enspricht einem Hochpass. (Beide Begriffe haben wir weiter unten genauer besprochen.) Betrachten wir eine Folge, in der beide Elemente gleich sind. Nach der Transformation soll dann das zweite Element Null sein, d. h., die Energie ist vollständig in der ersten Komponente gebündelt. Es sei (α, α) eine solche Eingabefolge. Es gilt:

$$\theta = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \alpha \end{bmatrix} = \begin{bmatrix} \sqrt{2}\alpha \\ 0 \end{bmatrix}$$

Wir diskutieren jetzt den zweidimensionalen Fall.

Beispiel 13.2 Wir nehmen die selbe Transformationsmatrix wie im vorigen Beispiel. Für die Eingabematrix $\begin{bmatrix} \alpha & \alpha \\ \alpha & \alpha \end{bmatrix}$ erhalten wir die folgende Transformierte:

$$\Theta = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} \alpha & \alpha \\ \alpha & \alpha \end{bmatrix}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}^T = \begin{bmatrix} 2\alpha & 0 \\ 0 & 0 \end{bmatrix}$$

Eine interessante Eigenschaft ist, dass nach der Transformation der oberen Eingaben sich die ganze Energie im linken oberen Eck der Matrix konzentriert.

Aus historischen Gründen heißt der $\theta_{0,0}$-Koeffizient (im linken oberen Eck der Matrix der Transformierten Werte) *DC-Koeffizient* oder *Niederfrequenz-Koeffizient* und die anderen *AC-Koeffizienten* oder *Hochfrequenz-Koeffizienten.* DC steht für *direct current*, zu deutsch Gleichstrom, und AC steht für *alternating current* (Wechselstrom). Einen Grund für diese Namensgebung sieht man, wenn man eine inverse Transformation für A betrachtet:

$$X = A\Theta A^T = \frac{1}{2}\begin{bmatrix} \theta_{0,0}+\theta_{0,1}+\theta_{1,0}+\theta_{1,1} & \theta_{0,0}-\theta_{0,1}+\theta_{1,0}-\theta_{1,1} \\ \theta_{0,0}+\theta_{0,1}-\theta_{1,0}-\theta_{1,1} & \theta_{0,0}-\theta_{0,1}-\theta_{1,0}+\theta_{1,1} \end{bmatrix}$$

Das heißt:

$$X = \theta_{0,0}\alpha_{0,0} + \theta_{0,1}\alpha_{0,1} + \theta_{1,0}\alpha_{1,0} + \theta_{1,1}\alpha_{1,1},$$

wobei

$$\alpha_{0,0} = \tfrac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \alpha_{0,1} = \tfrac{1}{2}\begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$\alpha_{1,0} = \tfrac{1}{2}\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix} \quad \alpha_{1,1} = \tfrac{1}{2}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Beobachten Sie, dass alle Elemente der $\alpha_{0,0}$ Matrix gleich sind. Daher kommt die DC-Benennung.

13.2 Spezielle Transformationen für Bildverarbeitung

Wir stellen die wichtigsten Transformationen vor, die man für die Bildverarbeitung benutzt. Weitere Einzelheiten zu diesem Thema finden Sie z. B. in [3].

13.2.1 Karhunen-Loéve-Transformation KLT

KLT ist nachweislich optimal hinsichtlich zweier Kriterien:

- Sie dekorreliert die Daten am besten und
- sie bündelt damit am besten die Information in den niederfrequenten Bereich.

Die entsprechenden mathematischen Messgrößen η_C und η_E werden wir weiter unten einführen.

Die wichtigsten Anfangsschritte der Transformation sind:

1. Berechne Kovarianzmatrix $COV(X)$ der Daten.
2. Bestimme die Eigenvektoren.

Das alles gilt jeweils für jedes Bild oder für jede Bildgruppe, da die Basisvektoren nur für die jeweilige Datenmenge chrakteristisch sind. Zusätzlich müssen noch die Basisvektoren zusammen mit den Kompressionseingaben als Begleitinformationen verschickt werden. Deshalb ist diese Methode in der Praxis nicht nützlich, stellt aber doch eine wichtige Messlatte für die Güte von Kompressionsverfahren dar.

Wir geben jetzt die wichtigsten Begriffe für eine Bildtransformation, und dann zeigen wir, wie die Karhunen-Loéve-Transformation aussieht.

Wir erinnern zunächst an die Begriffe *Mittelwert* $\bar{x}$ und *Varianz* σ^2 für das ganze Bild ($N \times M$ Pixel):

$$\bar{x} = \frac{1}{N \times M} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} x_{i,j}$$

$$\sigma^2 = \frac{1}{N \times M} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (x_{i,j} - \bar{x})^2.$$

Jetzt betrachten wir –als Vorbereitung für den zweidimensionalen Fall– die Korrelation im Eindimensionalen.

Für eine (eindimensionale) Folge x ist die Varianz:

$$\sigma^2 = \sigma_{xx}^2 = E[(x - \bar{x})^2].$$

Für zwei Folgen x und y definieren wir als ihre *Kovarianz*

$$\sigma_{xy}^2 = E[(x - \bar{x})(y - \bar{y})].$$

Hier und im Folgenden werden wir als „Erwartungswert" stets den Mittelwert ansetzen, was ja auch ein üblicher Schätzer ist. Ähnlich „lässig" verfahren wir mit anderen Begriffen aus Wahrscheinlichkeitstheorie und Statistik.

Ist $\sigma_{xy}^2 = 0$, so sind die beiden Folgen *unkorreliert*. Jetzt betrachten wir als Beispiel drei Folgen x, y und z. Dann erhalten wir die folgende *Kovarianzmatrix*:

$$COV(X) = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 & \sigma_{xz}^2 \\ \sigma_{yx}^2 & \sigma_{yy}^2 & \sigma_{yz}^2 \\ \sigma_{zx}^2 & \sigma_{zy}^2 & \sigma_{zz}^2 \end{bmatrix}$$

Für k Folgen ist $COV(X)$ analog eine $k \times k$-Matrix.

Jetzt definieren wir für die Folge

$$x_1, x_2, \ldots, x_M$$

als *k-te Kovarianz*

$$\sigma_{1,k+1}^2 = \sigma_{k+1,1}^2 = \frac{1}{M-k} \sum_{i=1}^{M-k} (x_i - \bar{x})(x_{i+k} - \bar{x}),$$

d. i. die Kovarianz für die Folgen $x_1, x_2, \ldots, x_{M-k}$ und $x_{k+1}, x_{k+2}, \ldots, x_M$, wobei wir als Mittelwert für beide Folgen (vereinfachend) $\bar{x}$ nehmen. Vereinfachend setzen wir weiter:

$$\sigma_{1,k+1}^2 = \sigma_{k+1,1}^2 = \frac{1}{M} \sum_{i=1}^{M-k} (x_i - \bar{x})(x_{i+k} - \bar{x}).$$

Für $k \leq M$ ist das immer eine gute Abschätzung. Dann bekommen wir:

$$\sigma_{11}^2 = \sigma_{22}^2 = \ldots \sigma_{kk}^2 = \sigma^2$$

$$\sigma_{12}^2 = \sigma_{23}^2 = \ldots \sigma_{k-1,k}^2 = \sigma_1^2$$

usw., wobei wir $\sigma_{\ell,k}^2$ ähnlich definieren wie $\sigma_{1,k+1}^2$ nur für die Folgen, die mit x_ℓ bzw. $x_{\ell+k}$ starten. Dann ist die *Kovarianzmatrix einer Folge*:

$$COV(X) = \begin{bmatrix} \sigma^2 & \sigma_1^2 & \sigma_2^2 & \ldots & \sigma_{k-1}^2 \\ \sigma_1^2 & \sigma^2 & \sigma_1^2 & \ldots & \sigma_{k-2}^2 \\ \vdots & \vdots & \vdots & & \vdots \\ \sigma_{k-1}^2 & \sigma_{k-2}^2 & \sigma_{k-3}^2 & \ldots & \sigma^2 \end{bmatrix}$$

Wir normalisieren die Matrix:

$$COV(X) = \sigma^2 \begin{bmatrix} 1 & \rho_1 & \rho_2 & \dots & \rho_{k-1} \\ \rho_1 & 1 & \rho_1 & \dots & \rho_{k-2} \\ \vdots & \vdots & \vdots & & \vdots \\ \rho_{k-1} & \rho_{k-2} & \rho_{k-3} & \dots & 1 \end{bmatrix},$$

wobei $\rho_k = \sigma_k^2/\sigma^2$ der *k-te Korrelationskoeffizient* ist. Zur Bildverarbeitung passt der folgende Fall erfahrungsgemäß sehr gut:[20]

$$\rho_k = \rho^k,$$

mit $\rho = \rho_1$. Wir nennen ρ *Zwischen-Element-Korrelation*. Dann definieren wir die *Korrelationsmatrix*

$$COR(X) = \begin{bmatrix} 1 & \rho & \rho^2 & \dots & \rho^{k-1} \\ \rho & 1 & \rho & \dots & \rho^{k-2} \\ \vdots & \vdots & \vdots & & \vdots \\ \rho^{k-1} & \rho^{k-2} & \rho^{k-3} & \dots & 1 \end{bmatrix}$$

Man vergleiche die gemachten Ausführungen auch mit denen in Abschnitt 10.2 zum Thema Autokorrelation.

Für den zweidimensionalen Fall verallgemeinern sich diese Begriffe mit Hilfe von vertikalen und horizontalen Folgen. Die Verallgemeinerung ist nicht trivial und wir werden sie hier nicht vertiefen.

Daher betrachten wir jetzt nur (noch) den eindimensionalen Fall.

Es seien $COV(X) = (X_{i,j})$ und $COV(Y) = (Y_{i,j})$ die Kovarianz-Matrizen für eine eindimensionale Folge und für ihre Transformation. Dann setzen wir:

$$\Sigma X = \sum_{\substack{1 \le i,j \le N \\ i \ne j}} |X_{i,j}|$$

und

$$\Sigma Y = \sum_{\substack{1 \le i,j \le N \\ i \ne j}} |Y_{i,j}|$$

[20] Genau genommen wird dabei mit der Annahme gearbeitet, die Eingabefolge sei ein stationärer Markov-Prozess k-ter Ordnung. In der Praxis bewährt sich diese an und für sich falsche Annahme.

und definieren damit die *Dekorrelationswirkung* η_C folgendermaßen:

$$\eta_C = 1 - \frac{\Sigma Y}{\Sigma X}.$$

Eine andere Größe, die die Qualität einer Transformation beschreibt, ist der *Energie-Bündelungs-Koeffizient* in den ersten M von N diagonalen Komponenten:

$$\eta_E = \frac{\sum_{j=1,k=j}^{M} Y_{j,k}}{\sum_{j=1,k=j}^{N} Y_{j,k}}.$$

Die Zeilen der Karhunen-Loéve-Transformation sind die Eigenvektoren der *Autokorrelationsmatrix*. Für jedes $i = 0, 1, \ldots, N-1$ lösen wir folgende Gleichung

$$\tan N\omega_i = \frac{-(1-\rho^2)\sin\omega_i}{(1+\rho^2)\cos\omega_i - 2\rho},$$

wobei ρ die Zwischen-Element-Korrelation ist, und dann benutzen wir die Lösungen, um die Eigenvektoren zu berechnen. Dafür setzen wir:

$$\lambda_i = \frac{1-\rho^2}{1+\rho^2-2\rho\cos\omega_i}.$$

Basisvektoren für $0 \le i, j \le N-1$ sind dann:

$$a_{i,j} = \Big(\frac{2}{N+\lambda_i}\Big)^{1/2} \sin\Big[\omega_i\Big(j - \frac{N-1}{2}\Big) + \frac{(i+1)\pi}{2}\Big]$$

In Bild 39 sehen Sie die Werte der Basisvektoren für $N = 8$ und $\rho = 0,91$.

Intuitiv stellt die Autokorrelationsmatrix gerade die Abhängigkeiten zwischen aufeinander folgenden Gliedern einer Folge dar. Da die Eigenvektoren (als neue Basisvektoren) immer die „Richtungen" angeben, in die Folge „strebt", leistet KLT die gewünschte (optimale) Energiebündelung.

13.2.2 Diskrete Fouriertransformation

Diese ist die einzige komplexzahlige Transformation, die wir behandeln werden. Wir hatten sie schon kurz in Abschnitt 11.5 betrachtet. Die (komplexe) Transformationsmatrix lautet:

$$a_{\ell,k} = \sqrt{1/N}\Big(\cos\frac{2\pi\ell k}{N} - i\sin\frac{2\pi\ell k}{N}\Big).$$

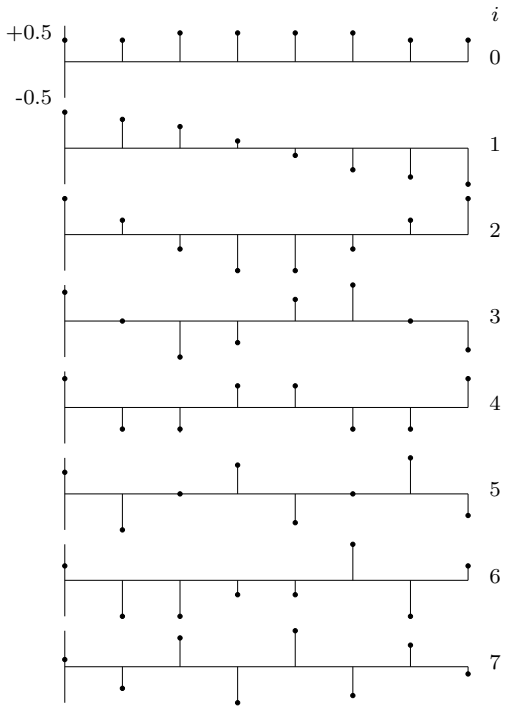


Abbildung 39: Die acht Basisvektoren der KLT für $\rho = 0,91$

Die Basisvektoren für $N = 8$ sehen folgendermaßen aus: Die reelle Basis:

$$\begin{bmatrix} 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 \\ 0,35 & 0,25 & 0,00 & -0,25 & -0,35 & -0,25 & 0,00 & 0,25 \\ 0,35 & 0,00 & -0,35 & 0,00 & 0,35 & 0,00 & -0,35 & 0,00 \\ 0,35 & -0,25 & 0,00 & 0,25 & -0,35 & 0,25 & 0,00 & -0,25 \\ 0,35 & -0,35 & 0,35 & -0,35 & 0,35 & -0,35 & 0,35 & -0,35 \\ 0,35 & -0,25 & 0,00 & 0,25 & -0,35 & 0,25 & 0,00 & -0,25 \\ 0,35 & 0,00 & -0,35 & 0,00 & 0,35 & 0,00 & -0,35 & 0,00 \\ 0,35 & 0,25 & 0,00 & -0,25 & -0,35 & -0,25 & 0,00 & 0,25 \end{bmatrix}$$

und die imaginäre Basis

$$\begin{bmatrix} 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 \\ 0,00 & 0,25 & 0,35 & 0,25 & 0,00 & -0,25 & -0,35 & -0,25 \\ 0,00 & 0,35 & 0,00 & -0,35 & 0,00 & 0,35 & 0,00 & -0,35 \\ 0,00 & 0,25 & -0,35 & 0,25 & 0,00 & -0,25 & 0,35 & -0,25 \\ 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 & 0,00 \\ 0,00 & -0,25 & 0,35 & -0,25 & 0,00 & 0,25 & -0,35 & 0,25 \\ 0,00 & -0,35 & 0,00 & 0,35 & 0,00 & -0,35 & 0,00 & 0,35 \\ 0,00 & -0,25 & -0,35 & -0,25 & 0,00 & 0,25 & 0,35 & 0,25 \end{bmatrix}$$

Es scheint so zu sein, dass man den imaginären Anteil nicht einfach vernachlässigen kann, so wie dies in der Praxis wohl manchmal passiert. Das Problem der „Datenaufblähung" durch Einführung eines Imaginärteils umgeht die diskrete Cosinustransformation:

13.2.3 DCT — Diskrete Cosinus-Transformation

Die (reelle) Transformationsmatrix lautet hier:

$$a_{i,j} = \begin{cases} \sqrt{1/N} & i = 0,\ j = 0, 1, \ldots, N-1 \\ \sqrt{2/N} \cos \frac{(2j+1)i\pi}{2N} & i = 1, \ldots, N-1,\ j = 0, 1, \ldots, N-1. \end{cases}$$

Für den 8×8-Fall ergeben sich als Basisvektoren:

$$A = \begin{bmatrix} 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 & 0,35 \\ 0,49 & 0,42 & 0,28 & 0,10 & -0,10 & -0,28 & -0,42 & -0,49 \\ 0,46 & 0,19 & -0,19 & -0,46 & -0,46 & -0,19 & 0,19 & 0,46 \\ 0,42 & -0,10 & -0,49 & -0,28 & 0,28 & 0,49 & 0,10 & -0,42 \\ 0,35 & -0,35 & -0,35 & 0,35 & 0,35 & -0,35 & -0,35 & 0,35 \\ 0,28 & -0,49 & 0,10 & 0,42 & -0,42 & -0,10 & 0,49 & -0,28 \\ 0,19 & -0,46 & 0,46 & -0,19 & -0,19 & 0,46 & -0,46 & 0,19 \\ 0,10 & -0,28 & 0,42 & -0,49 & 0,49 & -0,42 & 0,28 & -0,10 \end{bmatrix}$$

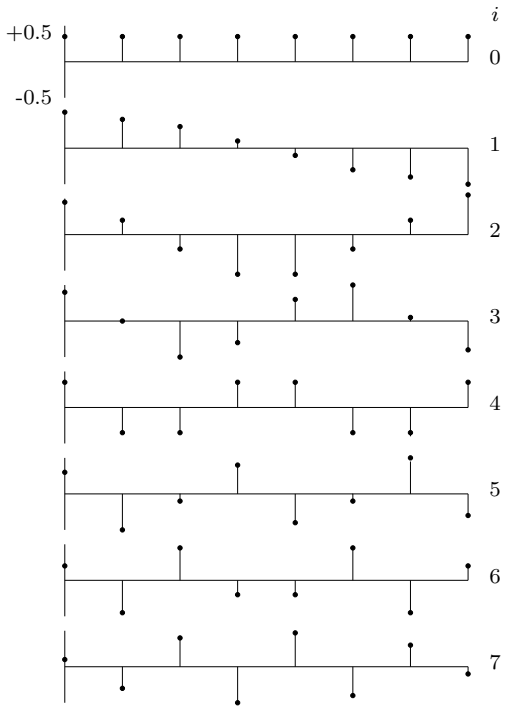


Abbildung 40: Die acht Basisvektoren der DCT

Ihre graphische Darstellung entnehmen Sie bitte Bild 40.

Vergleichen Sie die Werte der Basisvektoren der diskreten Cosinus-Transformation mit den Werten für KLT in Bild 39. Sie werden bemerken, dass sich die Werte von DCT und KLT kaum unterscheiden, was die gute Qualität der DCT erklärt.

Beispiel 13.3 Wir betrachten jetzt drei Beispiele zur DCT-Transformation:

1. Zwei aneinanderstoßende Flächen

$$X = \begin{bmatrix} 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \\ 10 & 10 & 10 & 10 & 128 & 128 & 128 & 128 \end{bmatrix}$$

besitzen folgende Cosinus-Transformierte:

$$\Theta = \begin{bmatrix} 552,0 & -427,7 & 0,0 & 150,2 & 0,0 & -100,4 & 0,0 & 85,1 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \\ 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 & 0,0 \end{bmatrix}$$

2. Ein Block des Bildes „Brücke" (entnommen aus Bild 16)

$$X = \begin{bmatrix} 94 & 96 & 113 & 135 & 196 & 116 & 110 & 106 \\ 91 & 125 & 120 & 153 & 192 & 135 & 108 & 124 \\ 87 & 119 & 144 & 122 & 190 & 131 & 115 & 132 \\ 88 & 93 & 141 & 100 & 168 & 149 & 128 & 122 \\ 95 & 93 & 139 & 169 & 172 & 147 & 154 & 135 \\ 87 & 100 & 124 & 173 & 180 & 132 & 173 & 178 \\ 104 & 85 & 112 & 120 & 167 & 158 & 132 & 166 \\ 112 & 83 & 124 & 123 & 154 & 152 & 148 & 165 \end{bmatrix}$$

wird transformiert in:

$$\Theta = \begin{bmatrix} 1049,9 & -125,7 & -121,2 & 4,7 & 50,1 & -40,0 & 2,4 & 55,2 \\ -30,4 & 59,0 & -48,7 & 6,0 & -0,2 & -35,8 & -22,7 & 13,3 \\ -17,2 & 5,7 & 12,8 & 16,4 & 24,6 & -11,0 & 0,6 & -5,4 \\ 15,7 & -14,8 & -17,5 & -9,4 & 25,7 & 26,1 & -30,5 & -10,9 \\ -19,9 & 5,6 & -0,6 & 14,8 & -13,1 & 22,2 & 14,5 & 12,3 \\ -30,6 & -3,6 & 13,5 & 17,2 & -8,3 & -18,8 & 20,3 & 19,7 \\ 11,1 & -6,8 & 2,1 & -19,6 & 4,3 & 5,6 & -16,0 & 13,4 \\ 0,5 & 9,5 & -7,4 & 2,8 & 4,0 & 1,0 & 5,5 & -0,2 \end{bmatrix}$$

3. Ein einzelner Punkt

$$X = \begin{bmatrix} 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 128 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 & 10 & 10 \end{bmatrix}$$

ergibt:

$$\Theta = \begin{bmatrix} 94,8 & 4,1 & -19,3 & -11,6 & 14,8 & 17,3 & -8,0 & -20,5 \\ 4,1 & 1,1 & -5,3 & -3,2 & 4,1 & 4,8 & -2,2 & -5,6 \\ -19,3 & -5,3 & 25,2 & 15,1 & -19,3 & -22,7 & 10,4 & 26,7 \\ -11,6 & -3,2 & 15,1 & 9,1 & -11,6 & -13,6 & 6,3 & 16,1 \\ 14,8 & 4,1 & -19,3 & -11,6 & 14,8 & 17,3 & -8,0 & -20,5 \\ 17,3 & 4,8 & -22,7 & -13,6 & 17,3 & 20,4 & -9,4 & -24,1 \\ -8,0 & -2,2 & 10,4 & 6,3 & -8,0 & -9,4 & 4,3 & 11,1 \\ -20,5 & -5,6 & 26,7 & 16,1 & -20,5 & -24,1 & 11,1 & 28,4 \end{bmatrix}$$

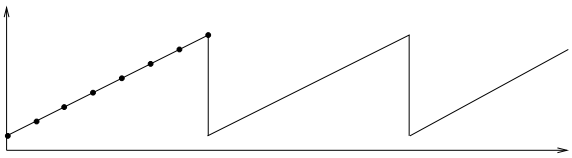
Diskrete Cosinus-Transformation und diskrete Fourier-Transformation hängen eng zusammen: Spiegelt man die N-Punkt-Folge einer DFT am rechten Rand, so erhält man die „zugehörige" $2N$-Punkt-Folge der DCT.

Bemerkung 13.4 *Die wichtigste Unterschiede zwischen DCT und DFT sind folgende:*

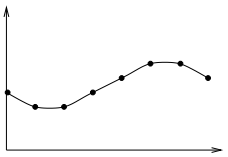
1. *DFT generiert komplexe Zahlen, während DCT nur reelle Zahlen ausgibt.*
2. *DFT nimmt an, dass die Funktion, die aus $x_0, x_1, \ldots, x_{N-1}$ rekonstruiert wird, periodisch ist (mit Periode N). So betrachtet DFT die Folge*

$$8, 16, 24, 32, 40, 48, 56, 64$$

als die Werte der folgenden periodischen Funktion:



Die inverse Transformation für die „bearbeiteten Werte" des Vektores Ax (z. B. nach Quantisierung) gibt daher die Werte der folgenden Funktion:



DCT hat solche Eigenschaft nicht. Wir betrachten DCT als die ersten N reellen Werte der DFT für die Folge:

$$x_0, x_1, \ldots, x_{N-1}, x_{N-1}, \ldots, x_1, x_0$$

und deshalb entfernen wir diese Nichtstetigkeiten.

3. *Die Komplexität der DCT ist höher als die der DFT.*

Bemerkung 13.5 *In der Praxis benutzt Software zur Berechnung der DCT Festpunktarithmetik. Der „Weltrekord" für die schnellste Ausführung der DCT liegt bei 11 Multiplikationen und 29 Additionen [C. Loeffler, A. Ligtenberg and G. Moschytz, Practical Fast 1-D DCT Algorithms with 11 Multiplications, Proc. Int'l. Conf. on Acoustics, Speech, and Signal Processing 1989 (ICASSP '89), pp. 988-991].*

13.3 Bit-Verteilung

Wie aus den Beispielen ersichtlich, ist der Informationsgehalt der Einträge der transformierten Matrix unterschiedlich; ähnlich wie bei der zuvor betrachteten Teilbandkompression massiert sich die Energie im linken oberen Eck, und das ist ja auch eines der Kriterien für eine gute Transformation. Das bedeutet wiederum, dass es sinnvoll ist, bei der quantisierten Übertragung dieser transformierten Koeffizienten unterschiedlich viele Bits zu verwenden. Diese Beobachtung führt zur Anwendung des Bitverteilungsalgorithmus aus Unterabschnitt 11.3 und wird in diesem Kontext *Zonenabtastung* (engl.: zonal sampling) genannt. Tabelle 27 zeigt solch eine typische Bitverteilung.

8	7	5	3	1	1	0	0
7	5	3	2	1	0	0	0
4	3	2	1	1	0	0	0
3	3	2	1	1	0	0	0
2	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabelle 27: Typische Bitverteilung für eine 8×8-Transformation

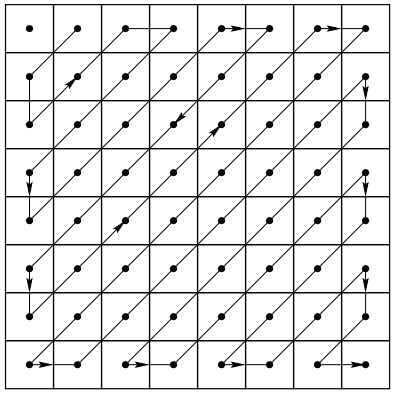


Abbildung 41: *Zickzack-Abtastung* eines 8×8-Musters

Chen und Pratt haben ein anderes Bitverteilungsverfahren vorgeschlagen, bei dem die zu übertragenden Werte des transformierten Bildes zunächt quantisiert werden (mit einem Quantisierer, der u. a. Null als Repräsentanten eines um den Nullpunkt symmetrischen Intervalls hat) und dann in der in Bild 41 angegebenen Weise zickzackartig abgetastet werden. Man kann annehmen, dass die sich so ergebene abgetastete Zahlenfolge ab einem gewissen Index konstant Null ist (dieser Annahme liegt ja offenkundig die in Tab. 27 angegebene fixe Bitverteilung zugrunde); daher erscheint es sinnvoll, eine solche konstante Nullfolge durch ein Sonderzeichen, hier EOB (*Blockende*, engl.: end of block) genannt, dem Empfänger anzuzeigen.

Der Vorteil dieses auch in JPEG verwendeten Schemas liegt darin, dass evtl. noch vorhandene „hochfrequente" Informationen nicht automatisch vernachlässigt werden.[21]

13.4 JPEG

Wir haben jetzt das nötige Rüstzeug zusammen, um den von der JPEG vorgeschlagenen Standard (im Folgenden auch JPEG genannt) verstehen zu können.

Die wichtigsten Bestandteile von JPEG (schematisch dargestellt in Bild 42) sind folgende:

- diskrete Cosinus-Transformation (DCT),
- Quantisierung,
- Differentialcodierung auf DC-Komponenten,
- Zickzack-Abtastung der AC-Komponenten,
- Lauflängencodierung (RLE) auf AC-Komponenten sowie
- Huffman-Codierung oder arithmetische Codierung.

Schauen wir uns die einzelnen Schritte des Schemas 42 genauer an:

1. Der erste Schritt ist fakultativ: Eine Umwandlung von RGB nach YIQ.

   RGB und YIQ sind dreidimensionale Räume, die die *Farben* und die *Helligkeit* beschreiben.[22] RGB hat ein passendes Format für die Hardware, wohingegen YIQ eher der menschlichen Wahrnehmung entspricht. I stellt einen

[21] Einer ähnlichen Idee waren wir ja schon in Shapiros EZW in Kapitel 11 begegnet.
[22] Mehr zu RBG und anderen Farbformaten finden Sie auf der WWW-Seite 1.

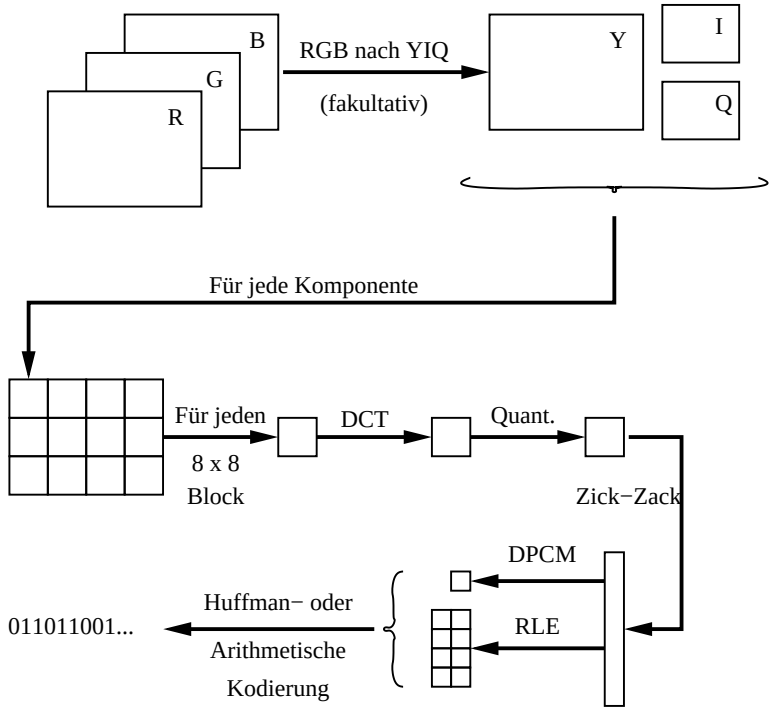


Abbildung 42: Das Schema von JPEG

124	125	122	120	122	119	117	118
121	121	120	119	119	120	120	118
126	124	123	122	121	121	120	120
124	124	125	125	126	125	124	124
127	127	128	129	130	128	127	125
143	142	143	142	140	139	139	139
150	148	152	152	152	152	150	151
156	159	158	155	158	158	157	156

Tabelle 28: Ein 8×8-Block aus dem Sena-Bild...

39,88	6,56	-2,24	1,22	-0,37	-1,08	0,79	1,13
-102,43	4,56	2,26	1,12	0,35	-0,63	-1,05	-0,48
37,77	1,31	1,77	0,25	-1,50	-2,21	-0,10	0,23
-5,67	2,24	-1,32	-0,81	1,41	0,22	-0,13	0,17
-3,37	-0,74	-1,75	0,77	-0,62	-2,65	-1,30	0,76
5,98	-0,31	-0,45	-0,77	1,99	-0,26	1,46	0,00
3,97	5,52	2,39	-0,55	-0,051	-0,84	-0,52	-0,13
-3,43	0,51	-1,07	0,87	0,96	0,09	0,33	0,01

Tabelle 29: liefert diese DCT-Koeffizienten nach JPEG.

Farbausgleich zwischen Orange und Cyan dar. Die Q-Komponente beschreibt einen Ausgleich zwischen Grün und Magenta. I und Q zusammen stellen die Farbe jedes Pixels dar. Die Y-Komponente beschreibt die Helligkeit des Pixels.

Fr eine Umwandlung von RGB nach YIQ muss man entscheiden, wieviele Bits den einzelnen Komponenten zukommen. Im allgemeinen Fall gewähren wir viermal soviel Bits für die Y- wie für die I- oder die Q-Komponente; denn die Augen sind empfindlicher für Änderungen der Helligkeit als für Änderungen der Farbe. Weitere Informationen zu Farbformaten finden Sie in Abschnitt 14.1.2.

2. Teile das Bild in Pixelblöcke der Größe 8×8.

3. Führe DCT für jeden Block aus.

   (Dabei wird zuvörderst jeder Pixelwert von 0 bis $2^P - 1$ durch Subtraktion von 2^{P-1} in einen um den Nullpunkt symmetrischen Bereich abgebildet.) Man beobachte das Beispiel eines Blocks aus dem Sena-Bild in den Tabellen 28 und 29.

   Warum benutzt JPEG DCT und nicht DFT?

   Zur Beantwortung dieser Frage diskutieren wir die Punkte von Bemerkung 13.4.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Tabelle 30: Die Standard-Quantisiermatrix von JPEG für Helligkeit

2	1	0	0	0	0	0	0
-9	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabelle 31: Die Matrix $\ell_{i,j}$ der Quantisierlabels des Sena-Blocks...

Natürlich ist Punkt (a) kein Problem und die entscheidenen Unterschiede sind (b) und (c). Das JPEG-Kommittee hat DCT gewählt wegen (b) und trotz (c). Man kann empirisch zeigen, dass die Pixel eines Bildes nicht einer periodischen Funktion entsprechen und daher der Grundannahme der DFT widersprechen.

4. Quantisierung

   Der Quantisierfehler ist die Hauptquelle der Verzerrung in JPEG. Das Verfahren benutzt eine 8×8 Quantisiermatrix Q und das heißt, dass JPEG für jeden 8×8-Block X die Werte

   $$\ell_{i,j} = \lfloor \Theta_{i,j}/Q_{i,j} + 0,5 \rfloor$$

   berechnet, wobei Θ die Matrix X nach der diskreten Cosinus-Transformation ist. Die übliche JPEG-Quantisiermatrix Q können Sie Tabelle 30 entnehmen.

   Im Beispiel in Tabelle 29 ist $\Theta_{0,0} = 39,88$, d.h.

   $$\ell_{0,0} = \left\lfloor \frac{39,88}{16} + 0,5 \right\rfloor = \lfloor 2,9925 \rfloor = 2.$$

   Der zugehörige Repräsentant ist daher $32 = 16 * 2$, und der Quantisierfehler wäre hier $7,88$. Die weiteren Daten sind in Tabelle 31 abgelegt; die Matrix der zugehörigen Repräsentanten ist in Tabelle 32 aufgeführt.

32	11	0	0	0	0	0	0
-108	0	0	0	0	0	0	0
42	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabelle 32: ...und die zugehörigen Repräsentanten

JPEG definiert zwei Default-Quantisiermatrizen: eine für die Helligkeit (diese sehen Sie in Tab. 30), das heißt die Y-Komponente, und eine für die Farben, das heißt für die I- und Q-Komponenten.

Es ist möglich, in JPEG eine eigene Quantisiermatrix definieren. Dann fügt man diese Matrix in den Kopf des codierten Bildes.

Unten sehen Sie die Werte nach der Quantisierung für den Block „Brücke" (vgl. Beispiel 13.3) und die Default-Matrix Q:

$$\begin{array}{rrrrrrrr} 66 & -11 & -12 & 0 & 2 & -1 & 0 & 1 \\ -3 & 5 & -3 & 0 & 0 & -1 & 0 & 0 \\ -1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

5. DPCM auf DC-Komponenten

   DC-Komponenten sind groß und haben unterschiedliche Werte. Man kann aber sehen, dass viele DC-Komponenten ähnliche Werte wie ihre Vorgänger haben. Deshalb wählt man anstelle der direkten Codierung der DC-Komponenten ein DPCM-Verfahren: JPEG codiert die Differenzen zwischen Nachbarkomponenten. Diese Differenzen haben die Tendenz, klein zu sein.

   Die Differenzen komprimiert man mit Hilfe des Huffman-Verfahrens oder der arithmetischen Codierung.

6. Zickzack-Abtastung der AC-Komponenten

   Hierdurch wird der zweidimensionale Block in einen eindimensionalen Vektor umgewandelt, so dass diese Konversion die Energie des Blockes im Anfang des Vektores bündelt.

7. Lauflängencodierung auf AC-Komponenten

123	122	122	121	120	120	119	119
121	121	121	120	119	118	118	118
121	121	120	119	119	118	117	117
124	124	123	122	122	121	120	120
130	130	129	129	128	128	128	127
141	141	140	140	139	138	138	137
152	152	151	151	150	149	149	148
159	159	158	157	157	156	155	155

Tabelle 33: Rekonstruktion des Sena-Blocks

Man kann sehen, dass viele AC-Komponenten Null sind. JPEG codiert diese Werte als Paare

$$(skip, value),$$

wobei *skip* die Anzahl der Nullen und *value* die nächste Nicht-Null-Komponente ist. $(0, 0)$ steht für das Ende. Dann komprimiert JPEG diese Folgen mit Hilfe des Huffman-Verfahrens oder der arithmetischen Codierung.

Weitere Einzelheiten des JPEG-Verfahrens finden Sie z.B. in [16, 17].

Die rekonstruierten Werte für den Sena-Block finden Sie in Tabelle 33.

Wer Lust bekommen hat, etwas mit JPEG herumzuexperimentieren, der sei auf die WWW-Referenz 5 verwiesen.

Es sei abschließend erwähnt, dass es auch eine JPEG-Variante gibt, die die fortschreitende Bildübertragung unterstützt, s. [19].

13.5 Ein Vergleich von Verfahren zur Bildkompression

„Ein Bild sagt mehr als tausend Worte" beschreibt treffend den Wert visueller Information für den Menschen. Leider umfasst ein hochaufgelöstes Digitalbild auch weit mehr Datenbits als nämliche tausend Worte, was die Notwendigkeit des Einsatzes von Kompressionsverfahren unterstreicht. Manche Anwendungen, z. B. im medizinischen oder militärischen Bereich erfordern eine getreue Wiedergabe des (digitalisierten) Originalbildes, da verlustbehaftete Kompressionsverfahren dort nicht tolerierbare Störungen mit sich bringen („Ist jener Fleck dort ein kleines Geschwür oder ein *Artefakt*, also eine Auswirkung der Fehlerbehaftung des Kompressionsverfahrens?") In WWW-Anwendungen sind typischerweise gewisse Verzerrungen tolerierbar (z. B. in elektronischen Zeitungen); als Kompromiss bietet sich die fortschreitende Bildübertragung an.

Prädikative Methoden (wie Differentialcodierung oder verlustfreier JPEG-Standard), bei denen im wesentlichen codierte Differenzen des aktuellen Pixels vom „erwarteten Pixel" (das aus der Kenntnis benachbarter Pixel errechnet wird) übertragen werden, sind im wesentlichen in Anwendungen im Einsatz, wo nur geringe Störungen toleriert werden können und dafür geringe Kompressionsfaktoren annehmbar sind.

Für mittlere Kompressionsraten sind Umformungs- und Teilbandcodierungen sehr geeignet. Umformungsbasierte Kompressionsverfahren wie JPEG arbeiten auf Datenblöcken und zeigen daher bei hohen Kompressionsraten Artefakte an den Blockrändern (*Blockeffekt*, engl.:blocking effect), welche Multiresolutionsansätze nicht aufweisen. Besonders deutlich erscheint der Blockeffekt in „ruhigen" Bereichen. Andererseits neigen Teilbandcodierungen dazu, aufgrund der nicht „perfekten" Filter *Welleneffekte* (engl.: ringing effect) an Rändern aufzuweisen. Durch geeignete Abstimmung der Filter aufeinander lässt sich dieser Effekt jedoch minimieren. Durch Techniken wie bei EZW lässt sich eine fortschreitende Bildübertragung erreichen.

Fraktale Kompressionsverfahren zeigen gute Leistungen bei relativ hohen Kompressionsraten (70 bis 80), die Bilder erscheinen dann aber oft „verschmiert" im Vergleich zum Original. Des weiteren erfordern sie einen hohen Rechenaufwand auf Seiten des Codierers und sind daher eher für *asymmetrische Anwendungen* wie im Multimediabereich geeignet, bei denen typischer Weise der Sender mehr Rechenkapazität besitzt als der Empfänger. Ähnlich einzuordnen sind dieser Tage aktuelle (in dieser Darstellung nur in Abschnitt 14.1.4 kurz besprochene) Kompressionsverfahren „der zweiten Generation", bei denen der Codierer zunächst versucht, gewisse Strukturen innerhalb des Bildes zu finden und dann diese Information zu übertragen. Auch diese Verfahren zeigen bei hohen Kompressionsraten sehr gute Bildqualitäten.

Wer hierzu vergleichende Bilder studieren will, sei auf [5, 19] verwiesen.

14 Zwei weitere Anwendungen

In diesem Abschnitt wollen wir zwei Anwendungsgebiete der Datenkompression behandeln, die bislang vernachlässigt wurden: die Kompression von Bewegtbildern am Beispiel des MPEG-Standards und —sehr knapp— die Kompression von Audio-Daten.

14.1 Allgemeines zu Bewegtbildern und deren Kompression

Bewegtbilder kann man im wesentlichen wie *Standbilder* komprimieren, nur dass eben noch die Zeit als weitere Dimension erscheint. Durch prädikative Ansätze wie bei der Differentialcodierung kann man ausnutzen, dass aufeinanderfolgende Bilder i. d. R. einander „ähnlich" sind. Natürlich wird man auch weiterhin die Ähnlichkeiten innerhalb einzelner Bilder ausnutzen wollen; dabei ist zu beachten, dass dem menschlichen Auge Störungen in „ruhigen Zonen" bei Bewegtbildern schlimmer als bei Standbildern erscheinen, während umgekehrt Kantenstörungen bei Standbildern stärker als bei Bewegtbildern wahrgenommen werden.

Ferner hat man bei Bewegtbildern häufig sehr unterschiedliche Anforderungen seitens der Anwendungen vorzuliegen: Bei *Ein-Weg-Kommunikation* — der heute wohl immer noch verbreitetsten Form der Bewegtbildnutzung— sind asymmetrische Verfahren denkbar, während bei der *Zwei-Weg-Kommunikation* (Bildtelefon, Videokonferenz,...) schnelle *symmetrische Algorithmen* gefordert sind.

14.1.1 Bewegungskompensation

Wesentlich bei allen Kompressionsverfahren für Bewegtbilder ist die *Bewegungskompensation*, um die zeitliche Abhängigkeit aufeinander folgender Bilder auszunutzen: Wie man in Abb. 43, 44 sieht, erscheint eine pixelweise Differenz von Bildern keine gute Dekorrelation zu liefern, denn Objekte (wie der Kopf und das Dreieck im Beispiel) bewegen sich unterschiedlich durch Szenen, und diese Bewegung muss erkannt und modelliert werden.

Algorithmisch wird dies Problem durch die sog. *blockbasierte* Bewegungskompensation, s. Abb. 45 angegangen. Das zu codierende Bild wird in $M \times M$ Pixelblöcke unterteilt. Im „früheren" Bild wird ein „möglichst ähnlicher Block" —Differenzen zwischen Blöcken werden z.B. durch MSE gemessen— gesucht und, falls der Fehler einen vorgegebenen Schwellwert unterschreitet, dann wird eine *Fehlerkorrektur* nebst dem *Bewegungsvektor* übertragen; bei

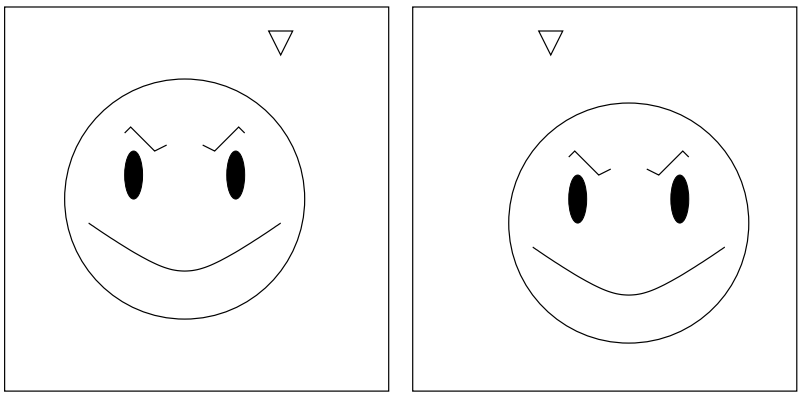


Abbildung 43: Zwei aufeinanderfolgende Bilder

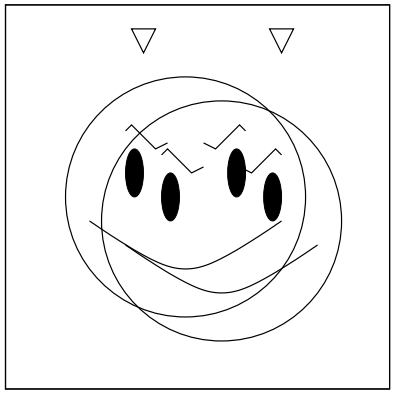


Abbildung 44: überlagert

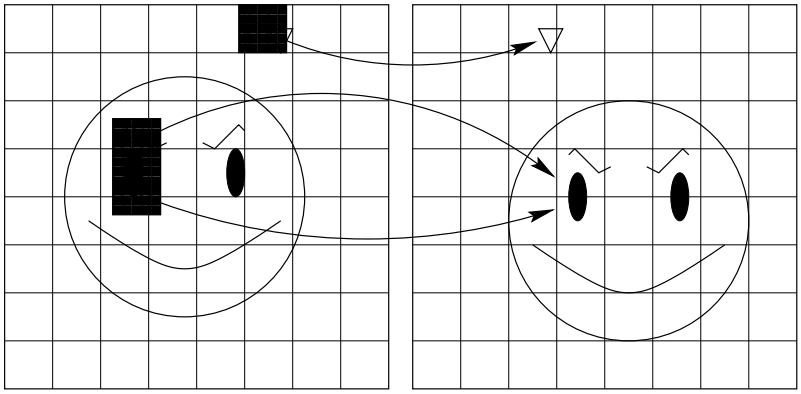


Abbildung 45: Blockbasierte Bewegungskompensation

Überschreiten des Schwellwertes wird der Block direkt codiert. Der Suchaufwand zum Auffinden „ähnlicher Blöcke" ist erheblich, wenn auch kleiner als bei fraktaler Kompression (vielleicht vergleichbar mit Vektorquantisierung). Man beachte, dass nicht nur die Blöcke im „früheren Bild" untersucht und verglichen werden, sondern auch feinere Verschiebungen, wie in Abb. 45 ersichtlich.

14.1.2 Farbbildformate

Bekanntermaßen lässt sich ein Farbbild aus den drei Grundfarben Rot, Grün und Blau zusammensetzen (*RGB-Format*). Dieses Format ist von Farbfernsehern bzw. Computermonitoren sowie Scannern her bekannt. Ebenso benutzt das menschliche Auge diese drei Farbkomponenten zur Farbanalyse. Farbdrucker arbeiten hingegen mit den Komplementärfarben Cyan (ein helles Blau, komplementär zu Rot), Magenta (ein Purpurton, komplementär zu Grün) und Gelb (komplementär zu Blau) (*CMY-Format*), wobei häufig noch Schwarz hinzugenommen wird, das sich nur theoretisch aus den CMY-Komponenten ergibt (*CMYK-Format*). Während im Computermonitor ein Farbton additiv aus den RGB-Grundfarben gemischt wird –so ergibt Rot plus Grün ein Gelb–, entsteht der Cyan-Eindruck beim Drucker durch Herausfiltern des Rot-Anteils aus dem Farblichtspektrum, so dass beispielsweise ein Überlagern von Rot und Grün eine Art Dunkelbraun liefert.[23]

Historisch gab es bei der Einführung des Farbfernsehens ein weiteres Problem: Viele Leute besaßen bereits Schwarz-Weiß-Fernseher, und um diese weiterverwenden zu können, wurde das RGB-Signal in modifizierter Form übertragen:

- als *YUV-Signal* in Europa, also insbesondere bei PAL,
- als *YIQ-Signal* in den Vereinigten Staaten, und
- neuerdings in der Digitalfernsehnorm (CCIR-601) gemäß der Formeln:

$$\begin{aligned} Y &= 0,229R + 0,587G + 0,114B \\ U' &= B - Y \\ V' &= R - Y \end{aligned}$$

Y beschreibt dabei die *Luminanz* (also der *Helligkeit*) des Bildes (die selbe Formel wird auch bei YUV und YIQ verwendet), und nur das Luminanzsignal wird von einem Schwarz-Weiß-Gerät umgesetzt. Für die menschliche

[23] Abgesehen davon ist es nicht klar, was „genau" z. B. Rot ist, weshalb letztlich der Ausdruck eines eingescannten Farbbildes durchaus anders als das Original erscheinen kann, s. [11].

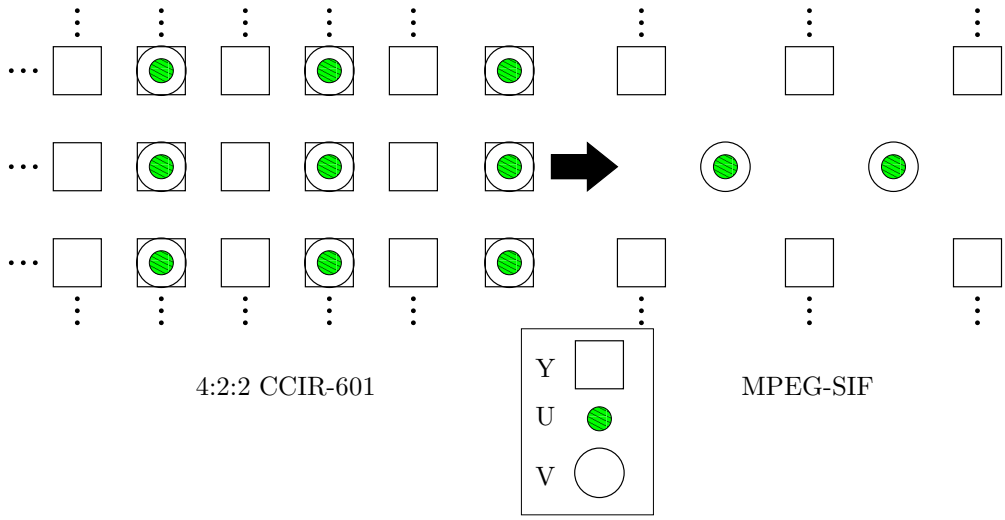


Abbildung 46: Zwei digitale Bildformate

Wahrnehmung ist das Y-Signal bedeutender als die U'- und V'-Signale, die auch *Chrominanzsignale* oder *Farbsignale* heißen. Typische weitere Chrominanzsignale sind einerseits U (Rot-Cyan Balance) und V (Gelb-Blau Balance) sowie andererseits I (Cyan-Orange-Balance) und Q (Magenty-Grün Balance). Durch Hinzunahme des Luminanzsignals Y enstehen die YIQ und YUV Farbsysteme. Die etwas komplizierteren (linearen) Umrechnungsformeln findet man auf der Internetseite 11 gut erklärt. Der Einfachheit halber werden wir das erläuterte $YU'V'$-Signal fürs Digitalfernsehen im Folgenden auch als YUV-Signal ansprechen.

Obschon es in manchen Details unterschiedliche Farbfernsehformate gibt, hat man sich auf ein einheitliches Format fürs *Digitalfernsehen* geeinigt (CCIR 601); eigentlich ist es eine ganze Schar von Formaten. Grundfrequenz ist stets $3,375$ MHz. Beliebtestes Format ist 4:2:2, d. h., das Luminanzsignal wird mit $4 * 3,375 = 13,5$ MHz und die Chrominanzsignale mit je $2 * 3,375 = 6,75$ MHz übertragen. Anvisiert wird eine Übertragung von 30 Bildern je Sekunde zu 720×480 Bildpunkten. Wie in Abb. 46 erkennbar, wird für jeden Bildpunkt ein Luminanzsignal und für jeden zweiten noch zwei Chrominanzsignale erzeugt. Diese werden (normalisiert und quantisiert) mit je einem Byte übertragen, so dass $60 \times 720 \times 480$ Bytes, also ca. 2 MB, je Sekunde zu übertragen sind. Um weichere Bildübergänge zu erzielen, wird jedes Bild in zwei Durchgängen übertragen: zuerst die geradzahlig und dann die ungeradzahlig numerierten Zeilen.

Das in Abb. 46 dargestellte *MPEG-SIF-Format* (für MPEG-1 Standard) viertelt die zu übertragende Datenmenge in der angedeuteten Weise. Für

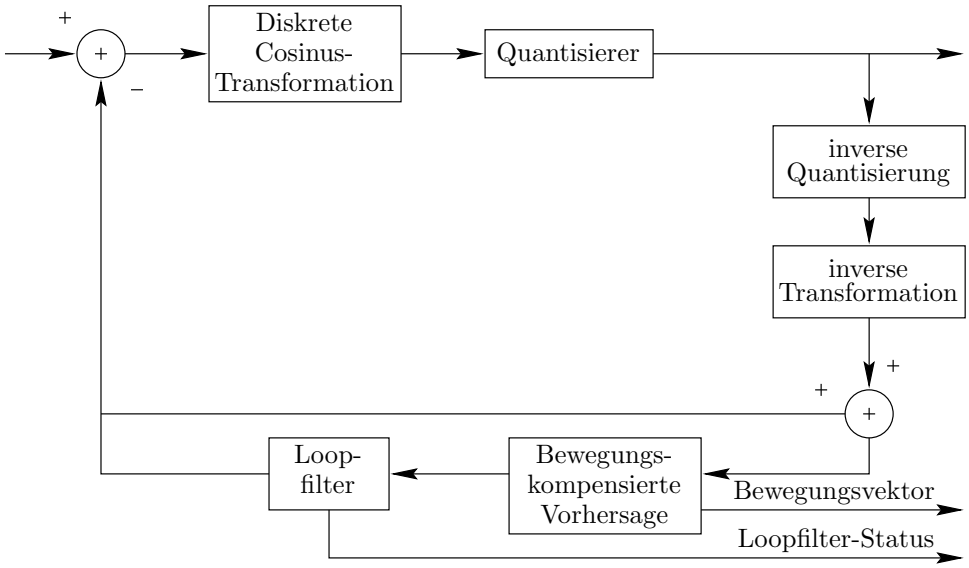


Abbildung 47: Der ITU-T H.261 Codierer

Videokonferenzen haben sich noch gröbere Auflösungen (ähnlich aufgebaut wie MPEG-SIF) als Standards durchgesetzt (wieder 30 Bilder je Sekunde):

CIF	Luminanz 288×352	Chrominanz 144×176
QCIF	Luminanz 144×176	Chrominanz 72×88

14.1.3 Videokonferenz

Wir erklären im Folgenden den auf dem [Q]CIF-Bildformat beruhenden *ITU-T H.261* Standard für Videokonferenzen; im Übrigen ist der MPEG-1 Standard sehr ähnlich (s.u.). Das Grundschema der Komprimierung ist in Abb. 47 erläutert. Die Dekomprimierung ist als Teil-Algorithmus im Komprimierer enthalten, da dieser prädikativ arbeitet.

Angesichts der Zielanwendung sollten sowohl Codierung als auch Decodierung sehr schnell gehen (Echtzeit); das gewählte Bildformat gewährleistet an und für sich schon eine relativ geringe Bitrate.

Am rechenintensivsten erscheint prinzipiell die schon oben diskutierte Bewegungskompensation. Wie begegnet der ITU-T H.261 Standard diesem Problem?

Der Luminanz- bzw. die Chrominanzanteile eines Bildes werden jeweils in 8×8 große Blöcke unterteilt. Ein *Macroblock* besteht aus 4 *Luminanzblöcken*

110	218	116	112
108	210	110	114
110	218	210	112
112	108	110	116

Tabelle 34: Ein 4×4 Beispielblock...

110	165	140	112
108	159	135	114
110	188	187	112
112	109	111	116

Tabelle 35: ... wird zuerst horizontal...

und je einem (entsprechenden) *Chrominanzblock*. Zu jedem Macroblock wird der günstigste „Partner" im früheren Bild gesucht, wobei nur die Luminanzblöcke zur Differenzbildung herangezogen werden. Der Bewegungsvektor der entsprechenden Chrominanzblöcke wird durch Halbierung des für die Luminanz berechneten gewonnen. Der Suchumfang wird auf ± 15 Pixel in vertikaler und horizontaler Richtung für jeden Macroblock eingeschränkt.

Eine Besonderheit des Standards ist in den Tab. 34, 35, 36 dargestellt: der *Loopfilter*. Er ist ein Tiefpassfilter, der für „innere Blockpunkte" (im Beispiel sind dies bei horizontaler Anwendung die beiden mittleren Spalten und bei vertikaler die beiden mittleren Zeilen) den „neuen" Pixelwert durch die Formel

$$x_{neu} = x/2 + x'/4 + x''/4$$

berechnet, wobei x' bzw. x'' die Pixelwerte der beiden Nachbarn sind. So ergibt sich der Wert in der ersten Zeile, zweite Spalte in Tab. 35 durch $165 = 110/4 + 218/2 + 116/4$ (ganzzahlige Division!), und der Wert in der zweiten Zeile, erste Spalte in Tab. 36 durch $108 = 110/4 + 108/2 + 110/4$ (beachte, dass bei „normaler Division" sich hier 109 ergäbe). Wie in Abb. 47 zu sehen, ist solch ein Loopfilter in die Macroblocksuchschleife eingebaut;

110	165	140	112
108	167	148	113
110	161	154	113
112	109	111	116

Tabelle 36: ... und dann vertikal gefiltert.

genauer gesagt: passt ein solchermaßen tiefpassgefilterter Luminanzanteil des Macrovergleichsblocks „am besten" bei der „Partnersuche", so wird dies dem Decodierer mit einem Extra-Bit (*Loopfilter-Status*) mitgeteilt.

Wie bei JPEG ist die im ITU-T H.261 Standard vorgesehene Transformation die DCT, angewendet jeweils auf 8×8-Blöcke. Da nun aber teilweise auf Pixeln [sog. „intra mode"] und auf Pixeldifferenzen [sog. „inter mode"] (Bewegungskompensation!) gearbeitet wird, haben die Koeffizienten sehr unterschiedliche Charakteristika, was die einheitliche Quantisierung erschwert. H.261 begegnet dieser Schwierigkeit, indem zwischen zweiunddreißig verschiedenen Gleichquantisierern für die AC-Koeffizienten von Macroblock zu Macroblock hin- und hergeschaltet werden kann. Da die Übertragung der quantisierten DCT-Koeffizienten wie bei JPEG nach Zickzackabtastung geschieht, ist bei den AC-Koeffizienten Null stets auch ein Repräsentant im Quantisierer. Es ist zu beachten, dass große Schrittweiten zwar evtl. eine schlechte Güte, andererseits aber einen ziemlich großen „Schwanz" von Nullen mit sich bringen, die Quantisiererwahl also auch die Übertragungsrate beeinflusst. Um der Notwendigkeit zu begegnen, für jeden Macroblock einen eigenen Quantisierschlüssel zu übertragen, werden 3×11 Macroblöcke zu einer *Blockgruppe* (GOB $\widehat{=}$ engl.: group of blocks) organisiert, und für jede GOB wird ein Default-Quantisierer im Kopf übertragen.

Bei Videokonferenzen steht den Teilnehmern nur eine beschränkte Übertragungsrate zur Verfügung. Andererseits ist —im wesentlichen bedingt durch unterschiedlich „günstige" zu codierende Lauflängen und Nullschwänze nach der Quantisierung der zickzackabgetasteten DCT-Koeffizienten— je Block eine unterschiedliche Anzahl Bits zu übertragen. Deshalb werden die Daten vor dem Versenden zunächst kurz zwischengepuffert (dies ist nicht in Abb. 47 dargestellt). Da diese Pufferung unerwünschte Übertragungsverzögerungen mit sich bringt, ist der Puffer nur recht klein. Droht er überzulaufen, so wird eine entsprechende Nachricht an den „Quantisierer" gesendet, die Rate zu senken, sei es durch Wahl eines Quantisierers mit größerer Schrittweite oder (im schlimmsten Fall) durch Auslassen eines Blocks bei der Übertragung.

14.1.4 MPEG

Die „Moving Pictures Expert Group" (MPEG) hat eine Reihe von Vorschlägen zur Bewegtbildcodierung gemacht, die sich vor allem hinsichtlich ihrer Funktionalität unterscheiden. So soll MPEG-1 für Übertragungsraten um 1,5 Mbps gute Ergebnisse liefern, MPEG-2 für 10 Mbps und MPEG-3 für 40 Mbps. Da MPEG-2 auch schon die für MPEG-3 gemachten Forderungen erfüllte, werden wir im Folgenden MPEG-1, MPEG-2 und (kurz) MPEG-4 diskutieren.

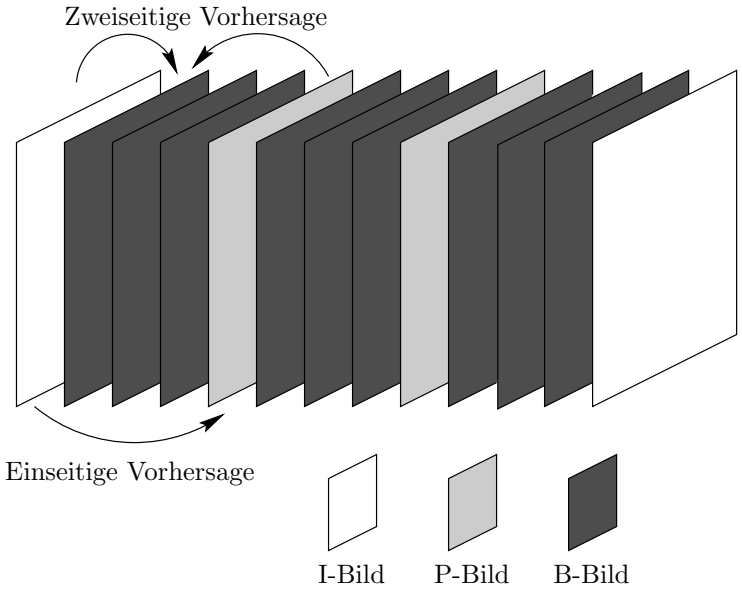


Abbildung 48: Eine mögliche Bildgruppe

MPEG-1 Das Grundschema von Abb. 47 ist auch hier gültig. Wesentlicher Unterschied zu H.261 ist die primäre Ausrichtung der MPEG-Standards auf Multimedia-Anwendungen, was insbesondere bedeutet, dass man eine Art „wahlfreien Zugriff" auf die Bildfolge gewährleisten muss; schon beim unidirektionalen (digitalen) Fernsehen will der Zuschauer ja evtl. sich später in einen laufenden Film einschalten, was aber unmöglich ist, wenn über die Bewegungskompensation immer auf beliebig weit zurückliegend gesendetete Information zugegriffen wird. Daher werden nicht zeitprädikativ codierte Bilder periodisch übertragen (sog. *I-Bilder*). Daneben gibt es sog. *P-Bilder* (P steht hier für „vorhersagecodiert", engl.: „prediction coded"), die im Wesentlichen dem H.261-Standard entsprechen, nur dass —aus gleich verständlich werdenden Gründen— P-Bilder (die mit den I-Bildern zusammen auch *Ankerbilder* genannt werden) nicht unbedingt auf das letzte Bild der zu sendenden Bildfolge, sondern beispielsweise auf das fünfletzte zurückgreifen; dies ist natürlich auch bei der Bewegungskompensation zu berücksichtigen (Es wird empfohlen, den Suchraum für Macroblöcke in Abhängigkeit vom „Abstand" gesendeter Ankerbilder anzupassen). Um eine gute Komprimierung zu gewährleisten, gibt es in MPEG zusätzliche *B-Bilder* (engl.: „bidirectionally prediction coded"), die Korrelationen sowohl zum in der zu sendenden Bildfolge vorausgehenden Ankerbild als auch zum in der zu sendenden Bildfolge später kommenden Ankerbildes ausnutzen, was bei Szenenwechseln sinnvoll ist. Eine *Bildgruppe* (GOP $\widehat{=}$ engl.: „group of pictures") enthält mindestens ein I-Bild und ist die kleinste Bildfolge, in die sich ein Benutzer

I	B	B	P	B	B	P	B	B	P	B	B	I
1	2	3	4	5	6	7	8	9	10	11	12	13

Tabelle 37: Anzeigeordnung einer GOP

I	P	B	B	P	B	B	P	B	B	I	B	B
1	2	3	4	5	6	7	8	9	10	11	12	13

Tabelle 38: Übertragungsordnung der selben GOP

zuschalten kann, s. Abb. 48.

Die GOP-Struktur von MPEG-1 ist auch vorteilhaft gegen eine mögliche Fehlerfortpflanzung; aus ähnlichen Gründen wirkt sich die rückgekoppelte Ratensteuerung nur auf die Übertragung der „unwichtigen" B-Bilder aus.

Wegen der Abhängigkeiten zwischen den verschiedenen Bildern unterscheidet man zwischen der *Anzeigeordnung* (engl.: display order), also der Reihenfolge, in der die Bilder anzuzeigen sind, und der *Übertragungsordnung* (engl.: „bitstream order"), der Reihenfolge also, in der die Bilder übertragen werden. Wie im Beispiel Tab. 37, 38 zu erkennen ist, werden zunächst die Ankerbilder übertragen und dann die sich unmittelbar darauf beziehenden B-Bilder.

MPEG-1 liefert eine Qualität, die mit VHS-Videos vergleichbar ist, wenn nicht zuviel Bewegung darin enthalten ist.

MPEG-2 ist eigentlich eine Sammlung von Algorithmen(empfehlungen) bzw. Spezifikationen, jeweils angepasst auf definierte sog. *Profile* (Anforderungen) und *Levels*. Es gibt deren fünf Profile; in der Reihenfolge aufsteigender Anforderungen heißen sie: *einfach* (engl.: simple), *Haupt-* (engl.: main), *SNR-skalierbar*, *räumlich skalierbar* (engl.: spatially scalable) und *hoch*. Höhere Profile können jeweils niedere decodieren. Die vier Levels sind: *niedrig*, *Haupt-*, *hoch 1440* und *hoch*. Sie definieren die Bildauflösung (352×240, 720×480, 1440×1152 und 1920×1080) und damit die zu übertragende Datenmenge. Übrigens benutzt digitales HDTV MPEG-2 mit Hauptprofil und hohem Level.

Die einfachen und Haupt-Profile entsprechen im Wesentlichen dem zu MPEG-1 Gesagten; das einfache Profil verwendet keine B-Bilder. Die drei übrigen Profile benutzen mehr als einen Bitstrom zur Codierung. Der Grundbitstrom ist eine niederratige Codierung der Bildfolge; die anderen Bitströme dienen

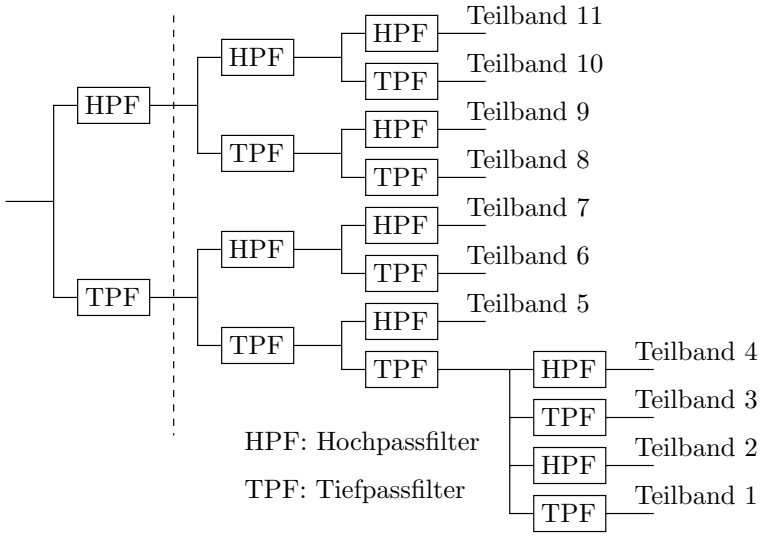


Abbildung 49: Filteranalyse nach Karlsson/Vetterli; die vertikale gestrichelte Linie trennt Zeit- und Raumfilter

zur Bildqualitätserhöhung. Bei Netzwerkübertragung braucht bei Überlast nur der Grundbitstrom übertragen zu werden, um so die Bitrate zu senken. Daher werden diese Profile auch „skalierbar" genannt. Ideen ähnlich der von der fortschreitenden Bildübertragung kommen zur Anwendung, nur dass die Filter sowohl zeitlich als auch räumlich arbeiten, s. Abb. 49.

MPEG-2 gestattet —im Gegensatz zu MPEG-1— aufwändigere Bewegungskompensation, indem

- auch Informationen des schon übertragenen Teils des aktuellen Bildes benutzt werden und
- unterschiedliche Bewegungsrichtungen von 16×8-Sub-Macroblöcken berücksichtigt werden (können).

MPEG-4 ist eine Antwort auf (im wesentlichen auf die Zukunft gerichtete) Schlagwörter wie „interaktives Fernsehen", „multiperspektives Fernsehen", „Video on demand", etc.

Nach Rautenberg [15] lassen sich die Funktionalitäten wie in Tab. 39 darstellen. Es folgen kurze Erläuterungen zu den Funktionalitäten:

1. Ein Bild wird aus verschiedenen, priorisierten *audiovisuellen Objekten* (AVOs) zusammen gesetzt gedacht, die jeweils mit unterschiedlicher

Funktionalitäten	Kategorien
1. Inhaltsbasierte Skalierbarkeit 2. Inhaltsbasierte Interaktivität und Editieren von Datenströmen 3. Werkzeuge für den inhaltsbasierten Zugriff auf Multimedia-Daten 4. Hybride Kodierung von Daten mit natürlichem und synthetischem Ursprung	I. Inhaltsbasierte Interaktivität
5. Codierung mehrfacher, nebenläufiger, zeitlich verzahnter Datenströme 6. Verbesserte Kodierungsleitung	II. Kompression
7. Robustheit in fehlerträchtigen Umgebungen 8. Verbessertes Zeitverhalten bei wahlfreiem Zugriff	III. Universeller Zugriff

Tabelle 39: Funktionalitäten bei MPEG-4

Genauigkeit übertragen werden können. Natürlich ist es schwierig, solche AVOs automatisch zu identifizieren; dem Hersteller von MPEG-4-Filmen ist es aber freigestellt, diese (aus seinem Wissen heraus) zu definieren. Die Übertragungseigenschaften (Rate, SNR) verschiedener AVOs können dabei unterschiedlich sein.

2. Diese Funktionalität (die z. B. zu einem Hobby „Editieren von Filmen" führen könnte) soll dadurch gewährleistet werden, dass Bewegungsinformationen von AVOs durch (in einer Erweiterung von C++ geschriebenen) *Kompositionsskripte* übertragen werden.

3. Werkzeuge können sein: Indizes, Hyperlinks, ...

4. Hybride Codierung soll auch *Multiperspektivität* unterstützen. So könnte sich ein Zuschauer in der Zukunft eine Torraumszene eines Fußballspiels wahlweise aus einer Torwart- oder einer Tribünenperspektive betrachten.

5. Codierungen mehrfacher Datenströme können beispielsweise mehrere Tonspuren (Sprachen) derselben Szene oder mehrere Bildperspektiven (3D-Fernsehen) betreffen; die dabei erzeugte Redundanz sollte durch intelligente Komprimierung kompensiert werden.

8. Z. B. soll der „schnelle Suchlauf rückwärts" für ein selektiertes AVO möglich sein.

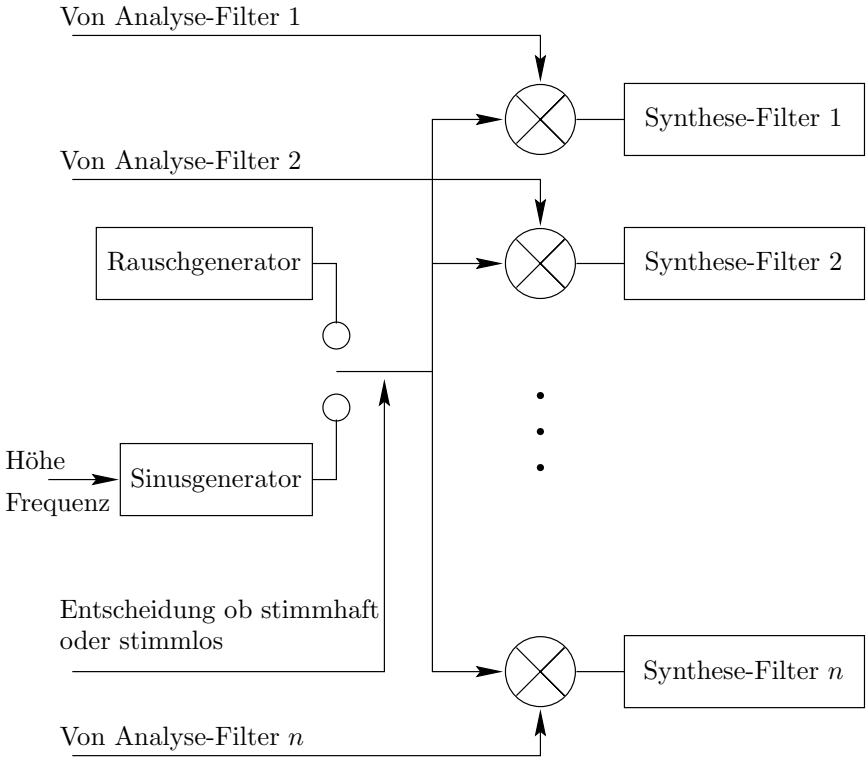


Abbildung 50: Schema eines Sprachdecodierers

14.2 Audio-Daten

Natürlich sind bei Filmen beispielsweise nicht nur Bild- sondern auch Toninformationen zu übertragen. Für eine gute Kompression von solchen Audio-Daten (insbesondere Sprache) ist zu berücksichtigen:

1. eingeschränkter hörbarer Frequenzbereich;
2. besondere Empfindlichkeit für Tonhöhen;
3. Lautstärkewahrnehmung ist mit Tonhöhenwahrnehmung korreliert;
4. in europäischen Sprachen gibt es deutlich voneinander trennbare „stimmhafte" (sinusartige) und „stimmlose" (rauschartige) Laute; ein Decodierer für solche Sprachen kann daher wie in Bild 50 dargestellt arbeiten.

Wie kann der Codierer eine Entscheidung darüber treffen, ob ein Laut stimmhaft oder stimmlos ist? Eine Möglichkeit ist die Filteranalyse; denn Rauschsignale sind hochfrequenter als sinusartige.

Der Tonhöhenempfindlichkeit wird beispielsweise im auch bei MPEG verwendeten CELP („code excited linear prediction" von Atal und Schroeder) durch eine spezielle adaptive Codebuchtechnik für die Erregungssignalformen Rechnung getragen.

15 Interessante Adressen aus dem Internet

Hier finden Sie eine Liste aller Webadressen, auf die wir an verschiedenen Stellen im Fließtext verwiesen haben.

1. www.j-a-b.net
2. www.internz.com/compression-pointers.html
3. www-2.cs.cmu.edu/afs/cs.cmu.edu/project/pscico-guyb/realworld/www/compress.html
4. www.cs.waikato.ac.nz/~singlis/ratios.html
5. www.cs.sfu.ca/undergrad/CourseMaterials/CMPT479/material/misc/interactive_jpeg/Ijpeg.html
6. www.wavelet.org
7. www.iro.umontreal.ca/~pigeon
8. www.informatik.uni-leipzig.de/cgip/projects/fic.html
9. www.iterated.com
10. inls.ucsd.edu/y/Fractals/
11. astronomy.swin.edu.au/ pbourke/colour/convert/

Literatur

[1] T. C. Bell, J. G. Cleary und I. H. Witten. *Text compression,* Prentice Hall, 1990.

[2] K. Bosch. *Elementare Einführung in die angewandte Statistik,* Vieweg, 1987.

[3] R. J. Clarke. *Transform Coding of Images,* Academic Press, 1985 u. 1990.

[4] G. A. Edgar. *Measure, Topology, and Fractal Geometry,* Springer, 1990.

[5] O. Egger, P. Fleury, T. Ebrahimi, M. Kunt. *High-Performance Compression of Visual Information—A Tutorial Review— Part I: Still Pictures,* Seiten 974–1011 in: Proc. IEEE, Band 87, Nr. 6, Juni 1999. **Hinweis:** Diese Zeitschrift enthält meist empfehlenswerte Übersichtsartikel zu verschiedendsten Gebieten der Informatik.

[6] H. Fernau. *Iterierte Funktionen, Sprachen und Fraktale,* BI-Verlag, 1994.

[7] H. Herrlich. *Einführung in die Topologie,* Heldermann , 1986.

[8] G. Kaiser. *A Friendly Guide to Wavelets,* Birkhäuser, 1994.

[9] S. D. Kamvar, D. Klein, Ch. D. Manning. *Interpreting and Extending Classical Agglomarative Clustering Algorithms Using a Model-Based Approach*, Seiten 283–290 in: Proc. ICML (International Conference on Machine Learning), Morgan Kaufmann, 2002.

[10] M. Li, P. Vitányi. *An introduction to Kolmogorov complexity and its applications,* 2. Auflage, Springer, 1997.

[11] J. Loviscach, W. Fastenrath. *Villa Kunterbunt; Farbkorrektur mit Color-Management-Systemen,* Seiten 180 ff. in: c't, Magazin für Computertechnik, Nr. 10, Oktober 1996.

[12] B. B. Mandelbrot. *Die fraktale Geometrie der Natur,* Birkhäuser, 1991.

[13] J.-R. Ohm. *Digitale Bildcodierung,* Springer, 1995.

[14] S. Pigeon. *Image Compression with Wavelets,* in: Dr. Dobb's Journal, August 1999.

[15] M. Rautenberg. *MPEG-4: Ein neuer Standard für mehr Funktionalität in Multimedia-Anwendungen,* Seiten 82–87 in: Informatik-Spektrum, Band 22, Heft 2, April 1999.

[16] D. Salomon. *Data Compression; The Complete Reference,* Springer, 1998.

[17] K. Sayood. *Introduction to Data Compression,* Morgan Kaufmann Publishers, Inc., 1996. Neue erweiterte Auflage: 2000.

[18] J. M. Shapiro. *Embedded Image Coding Using Zerotrees of Wavelet Coefficients,* Seiten 3445–3462 in: IEEE Transactions on Signal Processing, Band 51, Nr. 12, Dezember 1993. **Hinweis:** Diese Zeitschrift enthält sehr häufig Artikel zum Thema Kompressionsverfahren.

[19] U. Simon, M. Berndtgen. *Handlich bunt; Kompressionstechniken für Bild- und Videodateien im Vergleich,* Seiten 222 ff. in: c't, Magazin für Computertechnik, Nr. 11, November 1996.

[20] F. Topsøe. *Informationstheorie,* Teubner, 1974.

[21] M. Zeller. *Flinkes Wellenspiel; Signalverarbeitung mit Wavelets,* Seiten 258 ff. in: c't, Magazin für Computertechnik, Nr. 11, November 1994.

Index