Analyse und Vergleich kontextbasierter Bildkompressionsverfahren

Diplomarbeit

Karsten Scheibler

Betreuer: Dr. Jörg Ritter

Martin-Luther-Universität Halle-Wittenberg
Fachbereich Mathematik und Informatik
Institut für Informatik

22. März 2006
Revision 2

Ich versichere, dass ich die vorliegende Diplomarbeit selbstständig und unter ausschliesslicher Verwendung der angegebenen Literatur angefertigt habe.

Karsten Scheibler

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den vergangenen Jahren hat die Digitalfotografie ihren Siegeszug angetreten und den bisher verwendeten Film immer mehr verdrängt. Ein ähnlicher Wandel vollzieht sich auch bei den Videokameras. Es gibt kaum mehr Modelle, die Videos analog aufzeichnen, stattdessen dominieren digitale Formate. Auch bei der Fernsehübertragung ist ein Ende von analogen Übertragungen abzusehen. Hier wird DVB in den Varianten DVB-C, DVB-S und DVB-T eingesetzt um Signale via Kabel, Satellit oder terrestrisch digital zu übertragen. In den meisten Anwendungsfällen ist eine Kompression der Bildinhalte erforderlich, weil einerseits immer nur eine begrenzte Bandbreite für die Übertragung zur Verfügung steht und andererseits die Kapazität zum Speichern der Bildinhalte limitiert ist. Häufig wird dabei auch eine verlustbehaftete Datenreduktion in Kauf genommen, um eine höhere Kompression zu erreichen.

Speziell in Bezug auf das Internet wächst zwar die nutzbare Bandbreite stetig, genauso wie auch die verfügbaren Speichermedien immer grösser werden, in mindestens dem gleichen Maße wachsen aber auch die Datenmengen der Inhalte. Man denke hierbei nur an die Einführung von HDTV und die steigenden Megapixel-Auflösungen der Digitalkameras. Bilddatenkompression ist also weiterhin unerlässlich.

In dieser Arbeit soll ausschliesslich die Kompression von Einzelbildern betrachtet werden. Gängige Verfahren hierfür sind beispielsweise JPEG oder JPEG2000. Beide Verfahren transformieren das Bild vor der eigentlichen Kompression in eine andere Darstellungsform. Die Grundidee dabei ist, das Bild in der neuen Darstellungsform mit möglichst wenigen Koeffizienten beschreiben zu können und so eine Datenreduktion zu erzielen. Im Falle der verlustbehafteten Kompression erfolgt zusätzlich noch eine Auswahl der Koeffizienten, um nur die “wichtigsten” zu berücksichtigen.

Während JPEG das Bild in 8×8 Blöcke zerlegt und auf jeden Block einzeln die Diskrete Kosinus-Transformation anwendet, nutzt JPEG2000 die Wavelet-Transformation und wendet diese auf das gesamte Bild an. Dadurch werden insbesondere die, von JPEG bekannten, “Klötzchen”-Artefakte vermieden. Dies ist sicherlich einer der Gründe weshalb bei der Kompression von Einzelbildern die Wavelet-Transformation bevorzugt zum Einsatz kommt. Deshalb soll auch in der vorliegenden Arbeit ausschliesslich diese Transformation verwendet werden.

Zur Strukturierung der Arbeit: Nach einer kompakten Vorstellung der Arithmetischen Kodierung und der Diskreten Wavelet-Transformation in den Kapiteln zwei und drei, werden in Kapitel vier bereits existierende Verfahren zur Komprimierung wavelettransformierter Bilder vorgestellt. Darunter auch EBCOT – das Verfahren von JPEG2000. EBCOT und insbesondere der ebenfalls vorgestellte SPIHT-Algorithmus werden häufig als Vergleichsmaßstab für neue Verfahren genutzt. Deshalb wird auch der in Kapitel fünf schrittweise entwickelte eigene Algorithmus YAWICA abschliessend diesen beiden Verfahren gegenübergestellt. Im Verlauf der Untersuchungen werden die Eigenschaften wavelettransformierter Bilder und die Ideen aus den bisher existierenden Verfahren aufgegriffen und näher betrachtet. Dadurch werden auch Rückschlüsse auf SPIHT und EBCOT möglich, die verstehen helfen warum diese Verfahren genau so und nicht anders aufgebaut sind.

Mein Dank gilt Prof. Dr. Paul Molitor und Dr. Jörg Ritter für die interessante Aufgabenstellung und die Betreuung während der Erstellung dieser Diplomarbeit.

Kapitel 2

Arithmetische Kodierung

Die Arithmetische Kodierung [1] nutzt statistische Eigenschaften der Eingabedaten aus, um eine Datenreduktion zu erzielen. Eine solche Kodierung wird als *Entropiekodierung* bezeichnet. Geht man beispielsweise davon aus, dass die Eingabedaten aus einer Folge von Bytes bestehen, so kann jedes Byte genau einen von 256 möglichen Werten annehmen – das Alphabet der Eingabedaten umfasst also 256 Symbole. Kommen einige dieser Symbole sehr häufig vor, andere hingegen nur selten, so ist es vorteilhaft für die besonders häufig auftretenden Symbole eine kurze Bitfolge und für nur selten auftretende Symbole eine lange Bitfolge als Kodierung zu wählen, weil sich dadurch die Eingabedaten in Summe mit weniger Bits darstellen lassen.

Die Symbole der Eingabedaten, die zuvor immer mit 8 Bit kodiert wurden, werden jetzt also mit einer variablen Bitanzahl kodiert. Wobei sich die Anzahl der verwendeten Bits pro Symbol nach der Auftrittswahrscheinlichkeit des einzelnen Symbols richtet (je häufiger es auftritt, desto weniger Bits). Die Arithmetische Kodierung erlaubt es aber nicht nur pro Symbol Codes mit einer ganzzahligen Menge von Bits zu verwenden, sondern auch Codes mit einer nichtganzzahligen Bitanzahl zu nutzen. Dies wird beispielsweise dann nötig, wenn das Eingabealphabet nur zwei Symbole umfasst.

2.1 Die Idee

Anstatt für jedes Symbol ein alleinstehendes Codewort zu erzeugen, generiert die Arithmetische Kodierung für die gesamten Eingabedaten ein einziges langes Codewort. Dieses Codewort entspricht dabei den Nachkommastellen einer Zahl aus dem Intervall $[0,0\ ;\ 1,0)$. Deshalb wird es auch rein rechnerisch möglich nur Bruchteile eines Bits für ein Symbol zu verwenden.

Um die Eingabedaten arithmetisch zu kodieren, werden sie symbolweise betrachtet. Ausgangspunkt für das erste Symbol ist das Intervall $[0,0\ ;\ 1,0)$. Jedem möglichem Symbol des Eingabealphabets wird genau ein Teilintervall aus $[0,0\ ;\ 1,0)$ zugeordnet, all diese Teilintervalle müssen dabei paarweise disjunkt und deren Länge proportional zu der Auftrittswahrscheinlichkeit des jeweiligen Symbols sein. Für das erste Symbol der Eingabedaten wird nun das zugehörige Teilintervall ausgewählt und bildet den Ausgangspunkt für die

Kodierung des zweiten Symbols. Es erfolgt also eine fortwährende Intervallschachtelung, die als Ergebnis wiederum ein Intervall liefert.

Nun ist es aber wenig zweckmässig dieses Intervall dem Decoder zu übermitteln, es reicht hingegen vollkommen aus eine Zahl aus diesem Intervall zu übertragen. Die Dekodierung erfolgt nämlich in ähnlicher Weise wie die Kodierung und nutzt ebenfalls eine Intervallschachtelung, um aus der übertragenen Zahl pro Iteration ein Symbol zu extrahieren. Um das erste Symbol zu erhalten, unterteilt der Decoder auf die gleiche Weise wie der Encoder das Intervall $[0,0\ ;\ 1,0)$ in Teilintervalle und prüft nun in welchem Teilintervall die empfangene Zahl liegt. Ist dieses ermittelt steht damit auch das erste Symbol fest, das gefundene Intervall wird nun wiederum unterteilt und die Prüfung wird erneut durchgeführt, um das nächste Symbol zu erhalten. Dieser Prozess wiederholt sich solange bis die gewünschte Anzahl Symbole extrahiert wurde. Dem Decoder ist dabei die Symbolanzahl mitzuteilen. Entweder explizit als Seiteninformation, oder durch ein zusätzliches, dem Eingabealphabet hinzuzufügendes, Symbol, welches bei seinem Auftreten dem Decoder signalisiert, dass das Datenende erreicht wurde. Diesem Symbol wird sinnvollerweise eine sehr geringe Auftrittswahrscheinlichkeit zugeordnet, weil es immer nur genau einmal auftritt.

Um einige später benötigte Variablen einzuführen, soll das eben beschriebene Vorgehen nochmals in etwas formalerer Form dargestellt werden. Sei n die Länge der Eingabedaten und m die Grösse des Eingabealphabets, ferner bezeichne $D_i, i = 1 \ldots n$ das i-te Symbol der Eingabedaten, $S_{i,D_i}, i = 1 \ldots n, j = 1 \ldots m$ das zu diesem Symbol zugehörige Teilintervall und $p_j, j = 1 \ldots m$ die Auftrittswahrscheinlichkeiten aller Symbole des Eingabealphabets, so muss also gelten:

1. $S_{0,D_0} = [0,0\ ;\ 1,0)$
2. $S_{i,D_i} \subsetneq S_{i-1,D_{i-1}}$
3. $\forall i\ \forall j \neq k\ (S_{i,j} \cap S_{i,k} = \oslash)$, d.h. für beliebiges aber festes $i = 1 \ldots n$ sind alle Teilintervalle $S_{i,j}$ für alle $j = 1 \ldots m$ paarweise disjunkt
4. $\forall i, j(e_{i,j} - s_{i,j} \sim p_j)$ mit $S_{i,j} = [s_{i,j}\ ;\ e_{i,j})$, d.h. die Teilintervallängen sind proportional zu den Auftrittswahrscheinlichkeiten der Symbole des Eingabealphabets

2.2 Ein Beispiel

Angenommen das Alphabet der Eingabedaten würde die 4 Symbole `a`, `b`, `c` und `d` umfassen, jedes Symbol könnte also mit 2 Bit kodiert werden. Für zehn Symbole würden somit 20 Bit benötigt. Unter Berücksichtigung der Auftrittswahrscheinlichkeiten der einzelnen Symbole

$$p_a = 0,1 \quad p_b = 0,6 \quad p_c = 0,2 \quad p_d = 0,1$$

soll nun beispielhaft an der Symbolfolge `bbcbabdcbb` Schritt für Schritt gezeigt werden, wie bei der Arithmetischen Kodierung vorgegangen wird.

1. Wie bereits erwähnt ist das Intervall $[0,0\ ;\ 1,0)$ der Ausgangspunkt, das erste zu kodierende Symbol ist `b`. Jedem im Eingabealphabet vorkommenden Symbol wird nun genau ein Teilintervall in $[0,0\ ;\ 1,0)$ zugeordnet, wobei die Intervalllänge den Auftrittswahrscheinlichkeiten $p_a \ldots p_d$ entspricht. Die Reihenfolge ist willkürlich gewählt, genauso gut kann auch `b` als unterstes Teilintervall genommen werden, wichtig ist nur, dass der Decoder später exakt die gleiche Zuordnung macht.

   $$\texttt{a} \rightarrow [0,0\ ;\ 0,1) \quad \texttt{b} \rightarrow [0,1\ ;\ 0,7) \quad \texttt{c} \rightarrow [0,7\ ;\ 0,9) \quad \texttt{d} \rightarrow [0,9\ ;\ 1,0)$$

   Es wird nun das zu `b` gehörige Teilintervall $[0,1\ ;\ 0,7)$ als Ausgangspunkt für die weiteren Intervallschachtelungen ausgewählt.

2. Ausgangspunkt ist nun das Intervall $[0,1\ ;\ 0,7)$, zu kodieren ist wiederum das Symbol `b`. Entsprechend den Wahrscheinlichkeiten $p_a \ldots p_d$ ergeben sich nun folgende Teilintervalle.

   $$\texttt{a} \rightarrow [0,10\ ;\ 0,16) \quad \texttt{b} \rightarrow [0,16\ ;\ 0,52) \quad \texttt{c} \rightarrow [0,52\ ;\ 0,64) \quad \texttt{d} \rightarrow [0,64\ ;\ 0,70)$$

   Entsprechend dem zu kodierenden Symbol ist nun das neue Intervall $[0,16\ ;\ 0,52)$.

3. Ausgangspunkt ist jetzt das Intervall $[0,16\ ;\ 0,52)$, zu kodieren ist das Symbol `c`. Wiederum erfolgt die Einteilung in Teilintervalle gemäss $p_a \ldots p_d$, letztendlich ist aber nur das Teilintervall von `c` interessant und jenes ist $[0,412\ ;\ 0,484)$. Bei genauerer Betrachtung wird also folgende Berechnung ausgeführt.

   $$\begin{aligned}
   c_0 &= 0 \\
   c_j &= \sum_{k=1}^{j} p_k \\
   s_{0,D_0} &= 0 \\
   e_{0,D_0} &= 1 \\
   l_{i-1,D_{i-1}} &= e_{i-1,D_{i-1}} - s_{i-1,D_{i-1}} \\
   s_{i,D_i} &= s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i-1} \\
   e_{i,D_i} &= s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i}
   \end{aligned}$$

   Die neuen Intervallgrenzen werden also mit Hilfe der kumulierten Wahrscheinlichkeiten c_j berechnet. Es lässt sich sogar etwas Rechenarbeit sparen, wenn man mit der unteren Intervallgrenze und der Intervallänge arbeitet, anstatt mit beiden Inter-

vallgrenzen.

$$\begin{aligned}
s_{0,D_0} &= 0 \\
l_{0,D_0} &= 1 \\
s_{i,D_i} &= s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i-1} \\
l_{i,D_i} &= e_{i,D_i} - s_{i,D_i} \\
&= (s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i}) - (s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i-1}) \\
&= l_{i-1,D_{i-1}} \cdot c_{D_i} - l_{i-1,D_{i-1}} \cdot c_{D_i-1} \\
&= l_{i-1,D_{i-1}} \cdot (c_{D_i} - c_{D_i-1}) \\
&= l_{i-1,D_{i-1}} \cdot p_{D_i}
\end{aligned}$$

4. ... 10. Es sind immer wieder dieselben Schritte durchzuführen, deshalb ist die Auswahl der Folgeintervalle nur noch in Kurzform aufgelistet:
   $\texttt{b} \rightarrow [0,4192\ ;\ 0,4624)$
   $\texttt{a} \rightarrow [0,4192\ ;\ 0,42352)$
   $\texttt{b} \rightarrow [0,419632\ ;\ 0,422224)$
   $\texttt{d} \rightarrow [0,4219648\ ;\ 0,422224)$
   $\texttt{c} \rightarrow [0,42214624\ ;\ 0,42219808)$
   $\texttt{b} \rightarrow [0,422151424\ ;\ 0,422182528)$
   $\texttt{b} \rightarrow [0,4221545344\ ;\ 0,4221731968)$

Für die Symbolfolge wurde mit den gegebenen Auftrittswahrscheinlichkeiten $p_a \ldots p_d$ durch die Arithmetische Kodierung das Intervall $[0,4221545344\ ;\ 0,4221731968)$ ermittelt. Eine Zahl aus diesem Intervall reicht nun aus, um die Symbolfolge zu kodieren, die Binärzahl $0,0110110000010011_{bin}$ entspricht dezimal $0,4221649169921875$ und erfüllt damit diese Bedingung. Letztendlich kann von der Binärzahl die Null vor dem Komma gestrichen werden, da die Arithmetische Kodierung bedingt durch das Startintervall von $[0,0\ ;\ 1,0)$ immer eine Zahl kleiner Eins liefert. Zu übertragen wären also nur die 16 Bits nach dem Komma. Gegenüber den 20 Bits bei unkomprimierter Übertragung ist das eine Reduktion der Bitmenge auf 80%.

Die Dekodierung verläuft nun in ähnlicher Weise. Dem Decoder wurde die Ausgabe des Encoders übermittelt. Zusätzlich muss der Decoder die Wahrscheinlichkeiten $p_a \ldots p_d$ kennen und die Einteilung in Teilintervalle auf exakt dieselbe Weise durchführen wie der Encoder. Ausserdem muss dem Decoder mitgeteilt werden wieviel Symbole zu extrahieren sind. Hier soll einfach davon ausgegangen werden, dass diese Anzahl als Seiteninformation zur Verfügung gestellt wurde und der Decoder damit weiss, dass zehn Symbole zu dekodieren sind.

1. Auch hier ist der Ausgangspunkt das Intervall $[0,0\ ;\ 1,0)$, genauso wie der Encoder führt der Decoder die Einteilung in Teilintervalle durch.

$$\texttt{a} \rightarrow [0,0\ ;\ 0,1) \quad \texttt{b} \rightarrow [0,1\ ;\ 0,7) \quad \texttt{c} \rightarrow [0,7\ ;\ 0,9) \quad \texttt{d} \rightarrow [0,9\ ;\ 1,0)$$

Es wird nun geprüft in welchem Teilintervall die empfangene Zahl

$$0,0110110000010011_{bin} = 0,4221649169921875$$

liegt. Es zeigt sich, dass das erste Symbol ein `b` sein muss, da die Zahl in dem zu `b` zugehörendem Teilintervall liegt.

2. Der Decoder kennt nun das erste Symbol und kann damit die weitere Unterteilung des Intervalls durchführen, um auf die gleiche Weise das zweite Symbol zu ermitteln.

   $\texttt{a} \rightarrow [0,10\ ;\ 0,16) \quad \texttt{b} \rightarrow [0,16\ ;\ 0,52) \quad \texttt{c} \rightarrow [0,52\ ;\ 0,64) \quad \texttt{d} \rightarrow [0,64\ ;\ 0,70)$

   Die Zahl liegt wiederum im Intervall von `b`.

3. ... 10. Die Vorgehensweise bleibt die gleiche. Im Gegensatz zum Encoder kann der Decoder aber nicht so einfach das verwendete Intervall ermitteln, er muss vielmehr für jedes Teilintervall testen ob die übermittelte Zahl darin liegt oder nicht, erst wenn er dieses Intervall gefunden hat, kann er sich mit dem nächsten Symbol befassen. Der Encoder hat es hier einfacher, da er das Symbol kennt, braucht er einfach nur die neuen Intervallgrenzen berechnen.

2.3 Adaptivität

In dem vorangegangenen Beispiel wurde von vorgegebenen Wahrscheinlichkeiten ausgegangen, die über die gesamten Eingabedaten gelten. In der Praxis macht es aber selten Sinn mit festen Wahrscheinlichkeiten zu arbeiten, stattdessen wäre es sinnvoller, wenn sich der Arithmetische Kodierer an die Auftrittswahrscheinlichkeiten der einzelnen Symbole von selbst anpasst. Zumal es dadurch auch möglich wird, sich – in gewissen Grenzen – auch an variierende Auftrittshäufigkeiten anzupassen. Dabei muss aber darauf geachtet werden, dass En- und Decoder immer dieselben Anpassungen der Wahrscheinlichkeiten vornehmen.

Die Idee ist nun, zu Beginn von Gleichverteilung auszugehen und nach einer bestimmten Anzahl von übertragenen Symbolen, die zugrundeliegenden Wahrscheinlichkeiten gemäss den bisher verarbeiteten Daten anzupassen. Dies erfordert aber auch eine gewisse Mindestmenge an Daten, damit sich eine sinnvolle Verteilungsstatistik aufbauen lässt.

2.4 Implementierung

2.4.1 Arithmetischer Kodierer für m-äre Eingabealphabete

In der Implementierung soll Integerarithmetik eingesetzt werden, deshalb müssen zunächst die Auftrittswahrscheinlichkeiten entsprechend umgewandelt werden. Denkbar wäre beispielsweise eine Annäherung der p_j durch rationale Zahlen mit dem Nenner 8192:

$$p_j \approx \frac{\hat{p}_j}{8192} \quad \text{mit} \quad 0 < \hat{p}_j < 8192,\ \hat{p}_j \in \mathbb{N} \quad \text{und} \quad \sum_{j=1}^{m} \frac{\hat{p}_j}{8192} \leq \frac{8192}{8192} = 1$$

Mit dem Nenner 8192 ist bewusst eine Zweierpotenz ($2^{13} = 8192$) ausgewählt worden, da sich so eine Division einfach durch Shifts realisieren lässt. Für die im Beispiel verwendeten Werte für $p_a \dots p_d$ sind folgende Werte eine sinnvolle Approximation:

$$p_1 = p_a = 0,1 \approx \frac{819}{8192} \qquad p_2 = p_b = 0,6 \approx \frac{4915}{8192}$$

$$p_3 = p_c = 0,2 \approx \frac{1639}{8192} \qquad p_4 = p_d = 0,1 \approx \frac{819}{8192}$$

die kumulierten Wahrscheinlichkeiten sind somit

$$c_0 = \frac{0}{8192} \quad c_1 = \frac{819}{8192} \quad c_2 = \frac{5734}{8192} \quad c_3 = \frac{7373}{8192} \quad c_4 = \frac{8192}{8192} \quad \text{mit} \quad c_j = \frac{\hat{c}_j}{8192}$$

Wie im Beispiel zu sehen war, ist es weniger rechenintensiv mit der unteren Intervallgrenze (`base`) und der Intervalllänge (`length`) zu arbeiten. Geht man davon aus, dass `base` und `length` jeweils 32 Bit breit sind und beide die Bitstellen nach dem Komma repräsentieren, so ergibt sich für die im obigen Beispiel genannte Rechenvorschrift folgende Umsetzung (in C-Notation):

1. $s_{0,D_0} = 0$
   $\rightsquigarrow$ `base = 0x00000000`

2. $l_{0,D_0} = 1$
   $\rightsquigarrow$`length = 0xffffffff`
   Genaugenommen, müsste `length` auf `0x100000000` gesetzt werden, da diese Zahl aber nicht mehr mit 32 Bit dargestellt werden kann, wird die nächstkleinere Zahl genommen. Die dadurch entstehende Ungenauigkeit ist zu vernachlässigen.

3. $s_{i,D_i} = s_{i-1,D_{i-1}} + l_{i-1,D_{i-1}} \cdot c_{D_i-1}$
   $\rightsquigarrow$`base += (length >> 13) *` $\hat{c}_{D_i-1}$

4. $l_{i,D_i} = l_{i-1,D_{i-1}} \cdot p_{D_i}$
   $\rightsquigarrow$`length = (length >> 13) *` $\hat{p}_{D_i}$

Um ein Symbol zu kodieren und `base` sowie `length` entsprechend anzupassen, müssen also folgende Rechenschritte durchgeführt werden:
`length >>= 13`
`base +=` `length *` $\hat{c}_{D_i-1}$
`length *=` $\hat{p}_{D_i}$

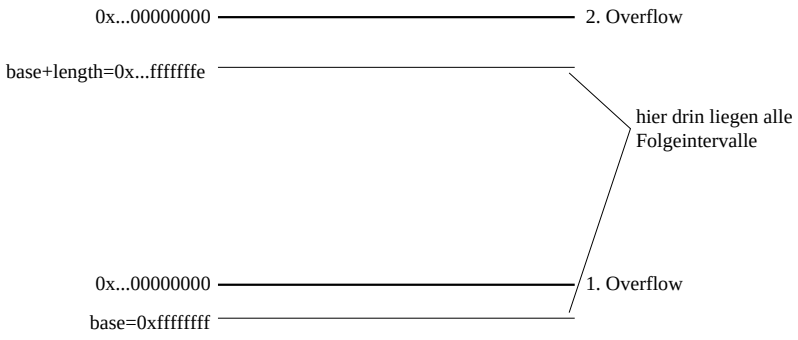
Tabelle 2.1 zeigt wie sich die Werte für `base` und `length` entwickeln, wenn die zehn Symbole des Beispiels kodiert werden. Es ist ersichtlich, dass die Zahlen in `length` mit zunehmendem i immer kleiner werden, gemäss der Rechenvorschrift muss dies auch so sein, bei der begrenzten Rechengenauigkeit von 32 Bit wird das jedoch zum Problem, denn bereits nach einigen weiteren Symbolen würde `length >> 13` Null werden und das ist nicht zulässig.

i	D_i	`length >> 13`	`base += ...`	`length *= ...`
1	b	0x0007ffff	0x1997fccd	0x9997eccd
2	b	0x0004ccbf	0x28f303da	0x5c26f70d
3	c	0x0002e137	0x697379c4	0x126fe721
4	b	0x0000937f	0x6b4b5911	0x0b0fcf4d
5	a	0x0000587e	0x6b4b5911	0x011b1b1a
6	b	0x000008d8	0x6b67a419	0x00a9cb08
7	d	0x0000054e	0x6c006b8f	0x0010f88a
8	c	0x00000087	0x6c0c3b59	0x00036051
9	b	0x0000001b	0x6c0c91ba	0x00020661
10	b	0x00000010	0x6c0cc4ea	0x00013330

Tabelle 2.1: Die Werte für `base` und `length`, wenn die zehn Symbole des Beispiels kodiert werden.

Bisher werden in `base` und `length` immer die ersten 32 Bit nach dem Komma gespeichert, die Idee ist nun, sobald `length` kleiner wird als `0x80000000` eine Renormalisierung durchzuführen, d.h. beide Variablen werden solange nach links geschoben bis das oberste Bit in `length` wieder gesetzt ist. Die Bits die dabei aus `base` rausgeshiftet werden, müssen gespeichert werden, da sie Teil des Codewortes sind, das die Arithmetische Kodierung am Ende liefert.

Beide Variablen speichern jetzt also durch die Renormalisierung immer die letzten 32 Bits des Codewortes. Es ist aber nun ein Sonderfall zu berücksichtigen, der ohne Renormalisierung nicht auftrat, der Wert in `base` kann jetzt einen Overflow generieren, der auf bereits rausgeshiftete Bits weiter propagiert werden muss. Es stellt sich nun die Frage, ob aus `base` rausgeshiftete Bits mehrfach von Overflows betroffen werden können, oder ob dies für jedes dieser Bits maximal einmal gilt.



Eine der Grundvoraussetzungen für das Funktionieren der Arithmetischen Kodierung ist, dass das Intervall für das i-te Symbol vollständig in dem Intervall des $i-1$-ten Symbols enthalten ist: $S_{i,D_i} \subsetneq S_{i-1,D_{i-1}}$. Damit Bits die bereits von einem vorhergehenden Overflow verändert wurden durch einen späteren Overflow nochmals berührt werden, müsste diese Voraussetzung verletzt werden. Die obige Grafik soll dies zu veranschaulichen. Die

grösstmögliche Intervalllänge ist `0xffffffff`, selbst wenn `base` auch diesen Wert enthält kann damit nur einmal ein Overflow generiert werden. Alle Folgeintervalle können niemals die jetzt festgesetzte obere Intervallgrenze überschreiten, um einen zweiten Overflow zu erzeugen, der Bits betreffen würde, die bereits von dem ersten Overflow berührt wurden.

Diese Tatsache ausnutzend, kann man den Prozess des Rausshiftens aus `base` noch etwas optimieren. Sobald eine 0 geschrieben wird zählt ein Counter die darauffolgenden Einsen, wird eine weitere 0 rausgeshiftet wird die Bitfolge unverändert geschrieben $01\ldots1$, wobei die Anzahl der Bits in $1\ldots1$ dem Zählerstand des Counters entspricht. Passiert hingegen ein Overflow wird stattdessen die Bitfolge $10\ldots0$ geschrieben.

Die Implementierung des Decoders arbeitet prinzipiell gleich, nur dass jetzt in `base` von rechts bei jeder Renormalisierung von `length` die Bits des gespeicherten Codeworts nachgeschoben werden. Ausserdem muss das Intervall mit einigen Tests ermittelt werden, durch geschickte Anordnung der Intervalle, den Einsatz von binärer Suche oder die Verwendung von Lookuptabellen, lässt sich dieser Vorgang jedoch beschleunigen [2]. Die Berechnung von `length` ist die gleiche wie im Encoder, von `base` wird jetzt aber – im Gegensatz zum Encoder – die gefundene untere Intervallgrenze subtrahiert.

2.4.2 Arithmetischer Kodierer für binäre Eingabealphabete

Ein binärer Arithmetischer Kodierer braucht nur die Auftrittswahrscheinlichkeit eines Symbols zu berücksichtigen, da sich aus ihr automatisch jene für das zweite Symbol ergibt: $p_2 = 1-p_1$. Zusätzlich werden auch keine kumulierten Wahrscheinlichkeiten mehr benötigt, weil der einzig benötigte Wert c_1 gleich einer der beiden Wahrscheinlichkeiten p_1 oder p_2 ist. Da das Eingabealphabet nur zwei Symbole umfasst, braucht der Decoder nur einen Vergleich durchzuführen, um das gespeicherte Symbol zu ermitteln. Eine aufwendige Suche entfällt damit also.

Die in dieser Arbeit verwendete Implementierung führt die Renormalisierung von `length` erst durch wenn `length` kleiner als `0x01000000` ist, somit können immer ganze Bytes aus `base` rausgeshiftet werden. Ausserdem kommt nur die adaptive Version des binären Arithmetischen Kodierers zum Einsatz. Zur Ermittlung der Auftrittswahrscheinlichkeit der Symbole dienen zwei Counter

1. `bitcount`, zählt alle bisher verarbeiteten Symbole

2. `bitcount0`, zählt wie häufig das erste Symbol vorkam

In regelmässigen Abständen wird daraus die bei der Kodierung verwendete Auftrittswahrscheinlichkeit berechnet. Sobald `bitcount` den Wert 8191 überschreitet werden beide Counter halbiert. Zum einen ist die Auftrittswahrscheinlichkeit selbst nur als rationale Zahl mit Nenner 8192 gespeichert, so dass es keinen Sinn macht grössere Counterwerte zuzulassen, weil die zusätzliche Genauigkeit nicht genutzt werden kann und zum zweiten wird dadurch eine gewisse Lokalität erreicht, weit zurückliegende Eingabesymbole haben einen geringen Einfluss auf die Auftrittswahrscheinlichkeit als erst vor kurzem bearbeitete.

2.5 Optimalität

Der Informationsgehalt eines Symbols s_i hängt von seiner Auftrittswahrscheinlichkeit ab und ergibt sich wie folgt [1]:

$$I(s_i) = \log_2 \frac{1}{p_i} = -\log_2 p_i \qquad .$$

Geht man von einer Datenquelle aus, deren ausgegebene Symbole voneinander vollkommen unabhängig sind und mit einer festen Wahrscheinlichkeit auftreten, so stellt die *Entropie*

$$H = -\sum_{i=1}^{m} p_i \cdot \log_2 p_i$$

eine untere Schranke für die durchschnittlich benötigte Bitmenge für ein Symbol dieser Datenquelle dar. Ist $m = 2$ – das Eingabealphabet also binär – vereinfacht sich die Gleichung zu

$$H = -p_1 \cdot \log_2 p_1 - (1 - p_1) \cdot \log_2(1 - p_1) \qquad .$$

Ein von der Arithmetischen Kodierung generiertes Codewort muss bekanntlich in dem Intervall $[s_{n,D_n} \ ; \ s_{n,D_n} + l_{n,D_n})$ liegen. Dazu ist eine Mindestbitmenge erforderlich, um die nötige Genauigkeit zu erreichen. Die Schrittweite des LSB darf nämlich nicht grösser sein als die Intervalllänge l_{n,D_n}, denn sonst kann nicht mehr sichergestellt werden, dass ein Codewort in dem Intervall erzeugt werden kann. Somit werden mindestens $\lceil -\log_2 l_{n,D_n} \rceil$ Bits für die Darstellung des Codewortes benötigt.

Beinhalte nun O die, für den Decoder, nötigen Seiteninformationen (wie Symbolwahrscheinlichkeiten und Anzahl der Symbole) sowie die Anzahl der eventuell nötigen Füllbits, um das Codewort in einer Bytefolge abspeichern zu können. Dann ergibt sich die durchschnittliche Bitmenge M, die zur Kodierung eines Symbols von einem Arithmetischen Kodierer benötigt wird, wie folgt:

$$M \leq \frac{O + \lceil -\log_2 l_{n,D_n} \rceil}{n} \leq \frac{O + 1 - \log_2 l_{n,D_n}}{n} = \frac{\overline{O} - \log_2 l_{n,D_n}}{n} \qquad .$$

Aus $l_{i,D_i} = l_{i-1,D_{i-1}} \cdot p_{D_i}$ ergibt sich

$$\begin{aligned} l_{n,D_n} &= l_{0,D_0} \cdot p_{D_1} \cdot p_{D_2} \cdots p_{D_n} = \prod_{i=1}^{n} p_{D_i} \\ \rightsquigarrow \quad M &\leq \frac{\overline{O} - \log_2 \prod_{i=1}^{n} p_{D_i}}{n} = \frac{\overline{O} - \sum_{i=1}^{n} \log_2 p_{D_i}}{n} \qquad . \end{aligned}$$

Sei nun $E[\cdot]$ der Operator des Erwartungswertes.

$$\begin{aligned}
\overline{M} \;=\; E[M] \;&\leq\; \frac{\overline{O} - \sum_{i=1}^{n} E[\log_2 p_{D_i}]}{n} = \frac{\overline{O} - \sum_{i=1}^{n}\sum_{j=1}^{m} p_j \cdot \log_2 p_j}{n} \\
&\leq\; H + \frac{\overline{O}}{n} \\
\rightsquigarrow \quad H \;\leq\; \overline{M} \;&\leq\; H + \frac{\overline{O}}{n} \\
\rightsquigarrow \quad \lim_{n\to\infty} \overline{M} \;&=\; H
\end{aligned}$$

Das heisst die Arithmetische Kodierung nähert sich mit zunehmender Länge der Eingabedaten immer mehr der Entropie an und kann deshalb als optimal bezeichnet werden – unter der genannten Voraussetzung, dass alle Symbole vollkommen unabhängig voneinander sind.

2.6 Kontexte

Da sehr häufig keine vollkommene Unabhängigkeit zwischen den Symbolen der Eingabedaten vorliegt, ist es sinnvoll auch dies mit zu berücksichtigen. Statt nur einer Tabelle mit Auftrittswahrscheinlichkeiten könnte man nun mehrere verwenden und je nach Bedarf zwischen ihnen hin- und herschalten. Will man beispielsweise die Bitfolge `010101010101` $\ldots_{bin}$ mit einem adaptiven binären Arithmetischen Kodierer komprimieren, so wird man damit wenig Erfolg haben, da die Auftrittswahrscheinlichkeiten beider Symbole bei 50% liegen. Verwendet man hingegen zwei Tabellen mit Auftrittswahrscheinlichkeiten und wählt immer dann die erste Tabelle, wenn das letzte übertragene Symbol eine Null war, so wird sich ein wesentlich besseres Kompressionsergebnis erzielen lassen. Nach einer gewissen Anpassungszeit werden sich in dieser Konstellation folgende Werte eingestellt haben.

	letztes Symbol war `0`	letztes Symbol war `1`
p_1	$\approx 0\%$	$\approx 100\%$
p_2	$\approx 100\%$	$\approx 0\%$

Das Beispiel ist zugegebenermassen sehr konstruiert, es verdeutlicht aber worum es geht. Durch die Berücksichtigung von Kontextinformationen – zumeist ist das eine Auswahl von vorangegangenen Symbolen – kann mit höherer Treffsicherheit vorhergesagt werden, welches das nächste Symbol sein könnte und dies führt letztlich zu einer besseren Kompression der Eingabedaten. Entscheidend dabei ist die Auswahl der Kontextinformationen, nimmt man beispielsweise ein Bild so ist es sicherlich vorteilhafter die Pixel der direkten Nachbarschaft in den Kontext einfliessen zu lassen, anstatt weiter entfernte Pixel zu nutzen.

Kapitel 3

Diskrete Wavelet-Transformation

Eine Transformation wird häufig dann eingesetzt, wenn ein Signal in eine andere, für die spätere Weiterverarbeitung besser geeignete, Darstellungsform überführt werden soll. Speziell bei der verlustbehafteten Bilddatenkompression ist eine Darstellungsform wünschenswert, die es ermöglicht die wesentlichen Merkmale eines Bildes mit nur wenigen Koeffizienten zu beschreiben, um damit eine platzsparende Speicherung zu erreichen. Die Wavelet-Transformation, ist eine solche Transformationen. Wobei in konkreten Anwendungen häufig nicht die Wavelet-Transformation selbst, sondern die Diskrete Wavelet-Transformation zum Einsatz kommt, weil es sich bei den Eingabedaten ebenfalls um diskrete Werte handelt. Der Name Wavelet bedeutet soviel wie "kleine Welle" und rührt vom Aussehen der Basisfunktionen her.

In der weiteren Betrachtung sollen nur Graustufenbilder berücksichtigt werden, da Farbbilder bekanntermassen aus drei Komponenten (zum Beispiel RGB[1] oder YUV[2]) bestehen und sich die später vorgestellten Kompressionsverfahren dort prinzipiell genauso anwenden lassen, nämlich getrennt für jede der drei Komponenten.

3.1 Ein Beispiel

In der Literatur [3, 4, 6] wird die Wavelet-Transformation auf verschiedene Arten hergeleitet. Hintergrund dessen ist die historische Entwicklung in unterschiedlichen Teildisziplinen. Eine verständliche Einführung in die Thematik ist beispielsweise in [5] oder [6] zu finden.

Ein Beispiel soll zunächst als Einstieg dienen. In fotorealistischen Bildern haben benachbarte Pixel häufig ähnliche Werte. Es wäre deshalb denkbar in jeder Bildzeile von jeweils einem Paar aufeinanderfolgender Pixel einerseits den Durchschnitt und andererseits die Differenz zu bilden. Viele dieser Differenzwerte werden somit nahe Null liegen, weil die Unterschiede zwischen den Pixeln gering sind. Bei einer verlustbehafteten Kompression könnten jetzt alle Differenzen, deren Betrag kleiner als ein gewisser Schwellwert ist,

[1] Rot, Grün, Blau

[2] Y ist der Luminanzwert und U, V sind die Chrominanzwerte

Abbildung 3.1: Links: Originalbild, Mitte: Durchschnitt bzw. Differenz Rechts: das mittlere Bild zurücktransformiert. Dabei wurden alle Differenzen aus dem Intervall $(-8, 8)$ auf Null gesetzt. Somit sind 99.479 der 131.072 Werte jetzt Null, dennoch sind keine grösseren Artefakte im rechten Bild erkennbar.

auf Null gesetzt werden, so dass nur noch Werte oberhalb des Schwellwertes zu berücksichtigen und zusammen mit allen Durchschnittswerten abzuspeichern wären. Die eben beschriebene Berechnung würde also wie folgt aussehen:

$$\begin{aligned} y_i &= \frac{1}{2}(x_{2i} + x_{2i+1}) \\ y_{n+i} &= x_{2i} - x_{2i+1} \end{aligned}$$

bzw.

$$\begin{aligned} x_{2i} &= y_i + \frac{1}{2} y_{n+i} \\ x_{2i+1} &= y_i - \frac{1}{2} y_{n+i} \end{aligned}$$

für die Rückrichtung. Wobei davon ausgegangen wird, dass eine Bildzeile eine geradzahlige Anzahl von Pixeln enthält ($2n$ mit $n \in \mathbb{N}$), die einzelnen Pixel mit $x_i, i = 0 \ldots 2n-1$ bezeichnet sind und die zu berechnenden Durchschnitts- bzw. Differenzwerte in y_j bzw. $y_{n+j}, j = 0 \ldots n-1$ abgelegt werden.

Abbildung 3.1 zeigt die Auswirkungen der Berechnung an einem Beispielbild[3]. Dabei ist das mittlere Bild in gewisser Weise eine gefilterte Version des Originalbildes, der linke Teil ist vergleichbar mit dem Ergebnis einer Tiefpassfilterung, während der rechte Teil dem Resultat einer Hochpassfilterung entspricht. Die gezeigte Berechnung entspricht genau der einmaligen Anwendung der Haar-Transformation und ist die einfachste Wavelet-Transformation.

Üblicherweise wird die Wavelet-Transformation aber nicht nur einmal angewendet sondern mehrfach. Zudem bietet es sich an, sie nicht nur auf die Zeilen sondern auch auf die

[3]Dieses Bild ist in Publikationen zum Thema Bilddatenkompression sehr häufig zu finden, die nicht ganz uninteressante Hintergrundgeschichte dazu lässt sich unter http://www.lenna.org/ nachlesen

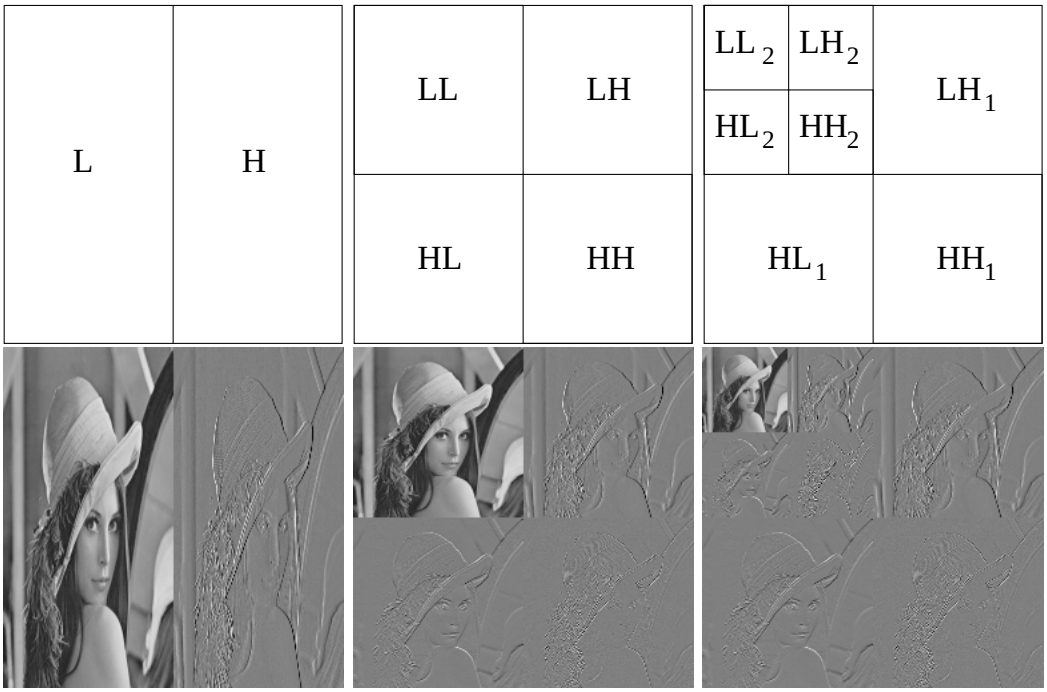


Abbildung 3.2: Mehrfache Anwendung der Wavelet-Transformation sowohl in x- als auch in y-Richtung.

Spalten des Bildes anzuwenden, so wie in Abbildung 3.2 dargestellt. Dies ist möglich weil die Wavelet-Transformation *separierbar* ist, d.h. eine mehrdimensionale Transformation durch nacheinander ausgeführte eindimensionale Transformationen realisiert werden kann. Je nachdem ob mit den Zeilen oder Spalten zuerst begonnen wird, variieren in der Literatur auch die Bezeichnungen der Subbänder, letztlich sind dabei aber nur die Bezeichnungen LH und HL vertauscht.

Das LL-Subband ist eine verkleinerte Version des Originalbildes, man spricht in diesem Zusammenhang deshalb auch von einer *Multiresolution-Darstellung*. Bei geeigneter Speicherung kann man so beispielsweise zunächst eine verkleinerte Version des Bildes betrachten, bevor man das gesamte Bild dekomprimiert – für sehr grosse Bilder ist dies sicherlich wünschenswert. Die in Abbildung 3.2 dargestellte Zerlegung wird als *Octaveband-Zerlegung* bezeichnet, weil jede weitere Transformationsstufe immer nur das LL-Subband berücksichtigt. Bei der *Spacl-Zerlegung* werden zusätzlich auch die übrigen Subbänder weiter transformiert. In dieser Arbeit wird eine solche Zerlegungsform jedoch nicht weiter betrachtet.

Bei der verlustbehafteten Komprimierung wird zumeist nicht nur ein Schwellwert eingesetzt, sondern auch eine Einteilung der verbliebenen Werte in Intervalle. Damit lassen sich nahe beieinander liegende Werte zwar nicht mehr unterscheiden, jedoch führt dies zu einer zusätzlichen Datenreduktion. Eine solche Einteilung wird auch als *Quantisierung* bezeichnet.

3.2 Allgemeine diskrete Transformation

Wenn es um Wavelet-Transformationen geht, treten auch häufig die Begriffe orthogonal, orthonormal und biorthogonal auf. Die folgenden Erläuterungen orientieren sich an [3] und sollen zumindest einen kurzen Überblick geben. Eine allgemeine eindimensionale diskrete Signaltransformation lässt sich wie folgt ausdrücken:

$$y_j = \sum_{i=0}^{n-1} x_i \cdot a_{i,j} \qquad j = 0 \ldots n-1$$

Dabei wird ein zeitdiskretes Originalsignal x_i mit Hilfe des Transformationskerns $a_{i,j}$ in ein transformiertes Signal y_j überführt. Für die Rücktransformation wird entsprechend

$$x_i = \sum_{j=0}^{n-1} y_j \cdot b_{j,i} \qquad i = 0 \ldots n-1$$

durchgeführt, wobei $b_{j,i}$ den Kern der Rücktransformation bildet. Werden das Originalsignal und das transformierte Signal als Spaltenvektoren

$$\underline{x} = (x_0, x_1, \ldots, x_{n-1})^T \qquad \underline{y} = (y_0, y_1, \ldots, y_{n-1})^T$$

sowie die Transformationskerne als Matrizen

$$A = a_{i,j} \qquad B = b_{j,i}$$

betrachtet, so lassen sich Hin- und Rücktransformation in Matrixschreibweise ausdrücken. Wobei die Rücktransformationsmatrix die Inverse der Hintransformationsmatrix sein muss, um die perfekte Rekonstruktion des Originalsignals zu gewährleisten

$$\underline{y} = A \cdot \underline{x} \qquad \underline{x} = B \cdot \underline{y} \qquad B = A^{-1} \qquad .$$

Die Spaltenelemente der Rücktransformationsmatrix B können zu Spaltenvektoren $\underline{b}_j$ zusammengefasst werden. Die Rücktransformation lässt sich somit wie folgt ausdrücken

$$\underline{x} = B \cdot \underline{y} = \sum_{j=0}^{n-1} y_j \cdot \underline{b}_j \qquad .$$

Das Originalsignal $\underline{x}$ entsteht also aus einer Überlagerung der, mit den Transformationskoeffizienten y_i gewichteten, Vektoren $\underline{b}_j$. Diese werden deshalb auch als Basisvektoren bezeichnet.

3.2.1 Orthogonale Transformation

Die Basisvektoren einer Transformation bilden ein Orthogonalsystem, wenn sie paarweise senkrecht aufeinander stehen, d.h. es muss gelten $\forall k, l$ mit $k, l = 0 \ldots n-1$

$$\langle \underline{b}_k, \underline{b}_l \rangle = \sum_{j=0}^{n-1} b_{k,j} \cdot b_{l,j} = \begin{cases} 0 & \text{wenn } k \neq l \\ c & \text{wenn } k = l, \text{ mit } c > 0 \end{cases}$$

und

$$\sum_{i=0}^{n-1} a_{i,k} \cdot a_{i,l} = \begin{cases} 0 & \text{wenn } k \neq l \\ c & \text{wenn } k = l, \text{ mit } c > 0 \end{cases} \quad .$$

Hat darüber hinaus die Konstante c den Wert Eins, so ist das Basissystem nicht nur orthogonal, sondern auch *orthonormal*. Eine Transformationsmatrix ist orthonormal, wenn die inverse Matrix mit der Transponierten übereinstimmt

$$A^{-1} = A^T \quad \text{d.h. wenn gilt} \quad AA^T = A^T A = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix} \quad .$$

Ist die Matrix der Hintransformation bekannt, so kann für orthonormale Systeme die Rücktransformationsmatrix recht einfach mit

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

ermittelt werden. Die Basisvektoren sind also nicht nur Spaltenvektoren in B, sondern auch Zeilenvektoren in A.

3.2.2 Biorthogonale Transformation

Erfüllen die Matrizen A und B einer Transformation, das Kriterium der perfekten Rekonstruktion und ist B selbst unter Berücksichtigung einer Skalierungsmatrix nicht die transponierte Matrix von A, so wird eine solche Transformation als biorthogonal bezeichnet. Die Basisvektoren beider Matrizen stehen paarweise senkrecht aufeinander

$$\forall k, l \text{ mit } k, l = 0 \dots n-1 : \sum_{i=0}^{n-1} a_{i,k} \cdot b_{l,i} = \begin{cases} 0 & \text{wenn } k \neq l \\ c & \text{wenn } k = l, \text{ mit } c > 0 \end{cases} \quad ,$$

müssen aber innerhalb einer Matrix keine speziellen Bedingungen erfüllen.

3.3 Wavelet-, Skalierungsfunktion

Alle Basisfunktionen der Differenzwerte des vorangegangenen Beispiels sind Waveletfunktionen, die von einem Mutter-Wavelet

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}} \cdot \psi\left(\frac{t-b}{a}\right) \qquad a, b \in \mathbb{R}, a > 0$$

abgeleitet werden können. Durch Stauchen oder Dehnen (Veränderung des Parameters a) und Verschieben (Veränderung des Parameters b) können damit alle Wavelets eines

Transformationskerns konstruiert werden. Weil in der geplanten Anwendung nur diskrete Eingabedaten vorliegen, sind auch diskrete Werte für a und b erforderlich:

$$\begin{aligned} a &= a_0^{-j} \\ b &= k \cdot b_0 \cdot a_0^{-j} \\ \rightsquigarrow \quad \psi_{j,k}(n) &= \sqrt{a_0^j} \cdot \psi(n \cdot a_0^j - k \cdot b_0) \qquad j,k \in \mathbb{Z} \quad . \end{aligned}$$

Dabei bestimmt der Parameter j die Auflösungsstufe. Üblicherweise wird $a_0 = 2$ gewählt, man spricht dann von *dyadischen* Wavelets, da sich die Frequenzen der Basisfunktionen verdoppeln. Was die Waveletfunktion ψ für die Differenzwerte ist, ist die Skalierungsfunktion ϕ für die Durchschnittswerte

$$\phi_{j,k}(n) = \sqrt{a_0^j} \cdot \phi(n \cdot a_0^j - k \cdot b_0) \qquad j,k \in \mathbb{Z} \quad .$$

Die Wavelet-Transformation stellt das Bindeglied zwischen den Signaltransformationen und den Filterbänken dar, deshalb ist es genauso möglich die Wavelet-Transformation nur als Filterbankstruktur zu interpretieren und auf diese Weise die Waveletkoeffizienten zu berechnen. Die verwendeten Wavelets werden in dieser Arbeit als gegeben betrachtet. Die konkrete Durchführung der Wavelet-Transformation wird im folgenden Abschnitt erläutert.

3.4 Das Lifting-Scheme

Wim Sweldens stellte 1995 mit dem Lifting-Scheme eine elegante Methode zur Realisierung von Wavelet-Transformationen vor [7]. Zum einen lässt sich dadurch die Zahl der nötigen Rechenoperationen bei der Transformation verringern und zum zweiten lassen sich mit Hilfe des Lifting-Scheme auch Wavelets konstruieren [8], sogenannte *second generation wavelets*.

Ein weiteres Beispiel soll dies verdeutlichen. Wie bei der zuvor vorgestellten Haar-Transformation, sollen auch diesmal alle Pixel einer Bildzeile in $x_i, i = 0 \ldots 2n-1$ vorliegen. Zunächst werden die x_i in zwei disjunkte Mengen aufgeteilt, die eine Menge enthält alle Pixel mit geradzahligem, die andere alle Pixel mit ungeradzahligem Index (diese Aufteilung wird als Split S bezeichnet). Unter der Annahme, dass benachbarte Pixel ähnliche Werte haben liegt es nahe mit Hilfe einer der beiden Mengen und einer Vorhersagefunktion (Predictor P) die Elemente der anderen Menge zu berechnen. Da der Predictor nicht exakt sein muss verbleiben Differenzwerte, die für eine korrekte Rekonstruktion der anderen Menge nötig sind. Ein einfacher Predictor für einen Pixel mit ungeradzahligem Index ist der Durchschnitt seiner beiden Nachbarpixel mit geradzahligem Index, die verbleibende Differenz errechnet sich also wie folgt

$$d_i = x_{2i+1} - \frac{1}{2}(x_{2i} + x_{2i+2}) \qquad i = 0 \ldots n-1 \quad .$$

Da für $i = n-1$ kein Wert x_{2i+2} existiert, gilt folgende Sonderbehandlung: $x_{2i+2} = x_{2i}$. Nach dem Berechnen der Differenzen sind die Pixelpaare des Originalsignals (x_{2i}, x_{2i+1})

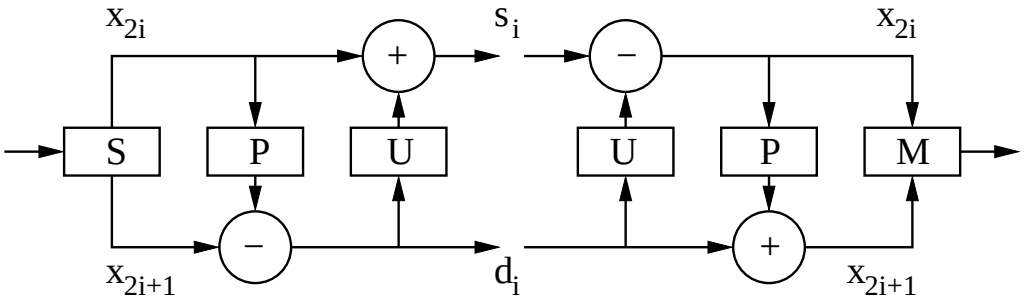


Abbildung 3.3: Das Lifting-Scheme: S = Split, P = Predict, U = Update, M = Merge

mit $i = 0 \ldots n-1$ transformiert worden in (x_{2i}, d_i), also Paare bestehend aus einem Pixel mit geradzahligem Index und einem Differenzwert. Sollen die x_{2i} nun den Signalmittelwert des Originalsignal beibehalten, soll also gelten

$$\sum_{i=0}^{n-1} x_{2i} = \frac{1}{2} \sum_{i=0}^{n-1} (x_{2i} + x_{2i+1}) \qquad ,$$

so wird ein weiterer Schritt nötig, der die x_{2i} anpasst (bezeichnet als Update U). Als Resultat erhält man die s_i-Werte

$$s_i = x_{2i} + \frac{1}{4}(d_{i-1} + d_i) \qquad i = 0 \ldots n-1 \qquad ,$$

wobei für $i = 0$ wieder eine Sonderbehandlung gilt: $d_{i-1} = d_i$. Nach beiden Schritten wurde das Originalsignal also umgewandelt in (s_i, d_i). Das allgemeine Prinzip des Lifting-Schemes für die Hin- als auch die Rücktransformation ist in Abbildung 3.3 verdeutlicht. Es ist zu beachten, dass bei der Hintranstransformation mit aufsteigendem i $(i = 0 \ldots n-1)$ gearbeitet wird, während bei der Rücktransformation $i = n-1 \ldots 0$ verwendet wird. Die Rücktransformation sieht damit wie folgt aus

$$\begin{aligned} x_{2i} &= s_i - \frac{1}{4}(d_{i-1} + d_i) \\ x_{2i+1} &= d_i + \frac{1}{2}(x_{2i} + x_{2i+2}) \qquad i = n-1 \ldots 0 \qquad . \end{aligned}$$

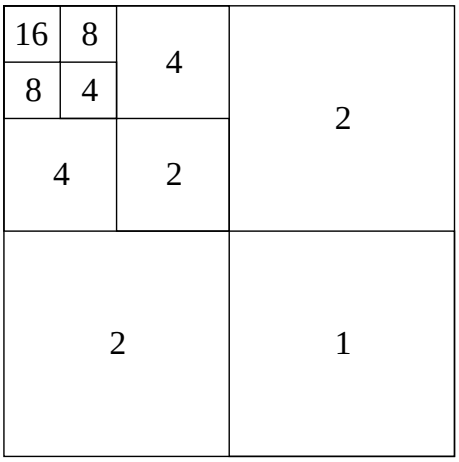


Abbildung 3.4: Zusätzliche Wichtungsfaktoren für das 5/3- und Haar-Wavelet, sowie die S+P-Transformation.

Schreibt man die Berechnung der Hintransformation etwas um

$$
\begin{aligned}
d_i &= x_{2i+1} - \frac{1}{2}(x_{2i} + x_{2i+2}) \\
&= -\frac{1}{2}x_{2i} + 1x_{2i+1} - \frac{1}{2}x_{2i+2} \\
s_i &= x_{2i} + \frac{1}{4}(d_{i-1} + d_i) \\
&= x_{2i} + \frac{1}{4}\left(\left[x_{2i-1} - \frac{1}{2}(x_{2i-2} + x_{2i})\right] + \left[x_{2i+1} - \frac{1}{2}(x_{2i} + x_{2i+2})\right]\right) \\
&= 1x_{2i} + \frac{1}{4}x_{2i-1} + \frac{1}{4}x_{2i+1} + \frac{1}{4}\left(-\frac{1}{2}(x_{2i-2} + x_{2i}) - \frac{1}{2}(x_{2i} + x_{2i+2})\right) \\
&= 1x_{2i} + \frac{1}{4}x_{2i-1} + \frac{1}{4}x_{2i+1} - \frac{1}{8}x_{2i-2} - \frac{1}{8}x_{2i} - \frac{1}{8}x_{2i} - \frac{1}{8}x_{2i+2} \\
&= -\frac{1}{8}x_{2i-2} + \frac{1}{4}x_{2i-1} + \frac{3}{4}x_{2i} + \frac{1}{4}x_{2i+1} - \frac{1}{8}x_{2i+2}
\end{aligned}
$$

so fällt auf, dass die verwendeten Faktoren denen des 5/3-Wavelets entsprechen. Alle s_i-Werte bilden dabei das L-Subband, alle d_i-Werte das H-Subband. Üblicherweise wird zusätzlich das L-Subband mit $\sqrt{2}$ und das H-Subband mit $\frac{1}{\sqrt{2}}$ gewichtet. Bei der zweidimensionalen Anwendung ergeben sich dadurch Wichtungsfaktoren, die – bis auf das HH_1-Subband – ganzzahlige Vielfache von 2 sind. Um alle Subbänder mit ganzzahligen Wichtungsfaktoren zu versehen, erfolgt nochmals eine Multiplikation mit 2, so dass man Faktoren wie in Abbildung 3.4 erhält. Dadurch lassen sich die Wichtungen als einfache Shifts realisieren.

Eine weitere Eigenschaft des Lifting-Scheme ist die Möglichkeit die Transformation *in-place* durchzuführen, d.h. es ist kein zusätzlicher Speicher nötig, um Zwischenergebnisse

abzuspeichern. Dazu werden die d_i-Werte in den Speicherstellen der x_{2i+1} abgelegt und die s_i-Werte in denen der x_{2i}. Die Berechnung läuft immer so ab, das überschriebene Werte nicht nochmals benötigt werden.

Das zuvor vorgestellte 5/3-Wavelet kommt in einer leicht abgewandelten Version zum Einsatz, um die Berechnung mit ganzen Zahlen zu ermöglichen. Folgend sind die Berechnungsschritte der Hintransformation für weitere Wavelets aufgeführt. Dabei ist gegebenenfalls an Randpositionen eine Sonderbehandlung ähnlich der des 5/3-Wavelets durchzuführen, um den Zugriff auf Pixel ausserhalb der Bildzeile zu unterbinden.

Haar-Wavelet (ganzzahlig, verlustfrei)

$$\begin{aligned} d_i^{(0)} &= x_{2i+1} \\ s_i^{(0)} &= x_{2i} \\ d_i &= d_i^{(0)} - s_i^{(0)} \\ s_i &= s_i^{(0)} + \mathit{ROUND}\left(\frac{1}{2}\ ,\ d_i\ ,\ 1\right) \end{aligned}$$

$$\text{mit} \quad \mathit{ROUND}(r_0, r_1, r_2) = \begin{cases} \lfloor r_0(r_1 - r_2) \rfloor & \text{wenn} \quad r_1 < 0 \\ \lfloor r_0(r_1 + r_2) \rfloor & \text{wenn} \quad r_1 \geq 0 \end{cases}$$

5/3-Wavelet (ganzzahlig, verlustfrei)

$$\begin{aligned} d_i^{(0)} &= x_{2i+1} \\ s_i^{(0)} &= x_{2i} \\ d_i &= d_i^{(0)} - \mathit{ROUND}\left(\frac{1}{2}\ ,\ s_i^{(0)} + s_{i+1}^{(0)}\ ,\ 1\right) \\ s_i &= s_i^{(0)} + \mathit{ROUND}\left(\frac{1}{4}\ ,\ d_{i-1} + d_i\ ,\ 2\right) \end{aligned}$$

9/7-Wavelet (verlustbehaftet)

$$\begin{aligned} d_i^{(0)} &= x_{2i+1} \\ s_i^{(0)} &= x_{2i} \\ d_i^{(1)} &= d_i^{(0)} + \alpha(s_i^{(0)} + s_{i+1}^{(0)}) \\ s_i^{(1)} &= s_i^{(0)} + \beta(d_{i-1}^{(1)} + d_i^{(1)}) \\ d_i^{(2)} &= d_i^{(1)} + \gamma(s_i^{(1)} + s_{i+1}^{(1)}) \\ s_i^{(2)} &= s_i^{(1)} + \delta(d_{i-1}^{(2)} + d_i^{(2)}) \\ d_i &= d_i^{(2)}/\zeta \\ s_i &= \zeta s_i^{(2)} \end{aligned}$$

mit den Konstanten

$$\begin{aligned}
\alpha &\approx -1,586134342 \\
\beta &\approx -0,05298011854 \\
\gamma &\approx 0,8829110762 \\
\delta &\approx 0,4435068522 \\
\zeta &\approx 1,149604398
\end{aligned}$$

Alle übrigen Transformationen verwenden die Wichtungsfaktoren aus Abbildung 3.4. Das 9/7-Wavelet bringt mit ζ bereits seinen eigenen Wichtungsfaktor mit. In der später vorgestellten Implementierung von YAWICA kann man zwar auch mit dem 9/7-Wavelet verlustfrei komprimieren, jedoch ist dies einfach ein Resultat der begrenzten Genauigkeit der Eingabedaten. Als Eingabedaten werden 8 Bit Graustufenbilder verwendet, bei der Rücktransformation wird die Rechenungenauigkeit des 9/7-Wavelets ab einem bestimmten Punkt so klein, dass nach der Rundung auf 8 Bit-Werte eine exakte Rekonstruktion des Originalbildes möglich wird. Dafür werden aber wesentlich mehr Bits benötigt als beispielsweise mit dem 5/3-Wavelet, deshalb wird das 9/7-Wavelet hier nur für verlustbehaftete Bilddatenkompression eingesetzt.

S+P-Transformation (ganzzahlig, verlustfrei)

$$\begin{aligned}
d_i^{(0)} &= x_{2i+1} \\
s_i^{(0)} &= x_{2i} \\
d_i^{(1)} &= d_i^{(0)} - s_i^{(0)} \\
s_i^{(1)} &= s_i^{(0)} + \mathit{ROUND}\left(\frac{1}{2}\ ,\ d_i\ ,\ 1\right) \\
d_i &= d_i^{(1)} + \mathit{ROUND}\left(\frac{1}{8}\ ,\ 2s_{i-1}^{(1)} + s_i^{(1)} + 3s_{i+1}^{(1)} + 2d_{i+1}^{(1)}\ ,\ 4\right) \\
s_i &= s_i^{(1)}
\end{aligned}$$

Auch die S+P-Transformation aus [9] lässt sich so umformen, dass sie im Rahmen des Lifting-Scheme berechnet werden kann. Dabei wird das Haar-Wavelet durch einen weiteren Predict-Schritt erweitert.

Kapitel 4

Bisherige Ansätze

Wie bereits festgestellt, besteht der Sinn der Wavelet-Transformation darin möglichst viele Koeffizienten nahe Null zu erzeugen. Es gibt nun mehrere Möglichkeiten diese Koeffizienten zu kodieren. Zum einen könnte – noch bevor mit der eigentlichen Kodierung begonnen wird – ein Qualitätsmass festgelegt werden, woraus in einem weiteren Schritt die Quantisierung abgeleitet wird. Alle Koeffizienten die nach dieser Quantisierung ungleich Null sind müssten dann gespeichert werden. Zum anderen könnte man auch ein progressives Übertragungsschema wählen, d.h. mit jedem zusätzlich übertragenem Bit wird die Qualität des rekonstruierten Bildes gesteigert. Dies lässt sich beispielsweise durch eine schrittweise Quantisierung der Koeffizienten erreichen. Die Koeffizienten werden dabei stückchenweise übertragen anstatt einmal quantisiert und komplett übermittelt zu werden. Gerade diese schrittweise Quantisierung ist für wavelet-tranformierte Bilder interessant, die Koeffizienten werden dabei üblicherweise in Bitschichten übertragen und erzeugen damit einen sogenannten *eingebetteten Datenstrom*.

4.1 Das Prinzip des Bitplanecoders

Das hier beschriebene Vorgehen wird von vielen Algorithmen genutzt, die wavelettransformierte Bilder komprimieren und einen eingebetteten Datenstrom erzeugen [10, 11, 16, 17]. Zunächst werden die Koeffizienten in die Betrag/Vorzeichen-Darstellung umgewandelt und der vom Betrag her grösste Koeffizient v wird ermittelt. Mit $B = \lfloor \log_2 v \rfloor$ erhält man dann das höchste Bit ungleich Null in v, damit ist die oberste Bitplane B bekannt.

Die Bits der Koeffizienten werden nun Bitplane für Bitplane übertragen, d.h. es wird zunächst das i-te Bit aller Koeffizienten übermittelt bevor überhaupt das $i-1$-te Bit betrachtet wird. Die i-ten Bits aller Koeffizienten bilden dabei die i-te Bitplane. Es wird mit der Bitplane $i = B$ begonnen und nach jedem Durchlauf i um eins dekrementiert. Handelt es sich bei den betrachteten Koeffizienten um ganze Zahlen wird bei $i = 0$ gestoppt, da nun dem Decoder alle relevenaten Informationen für eine exakte Rekonstruktion des Originalbildes zur Verfügung stehen.

Es ist sinnvoll das Vorzeichen erst nach dem ersten 1-Bit eines Koeffizienten zu übermitteln. Denn es würde dem Decoder keine Vorteile bringen, das Vorzeichen bereits früher zu kennen, da er dann noch keine Informationen über den Betrag des Koeffizienten besitzen würde. Durch diese Sonderstellung des ersten 1-Bits, kann man drei Typen von Bits unterscheiden:

1. *Significance-Bits*, das sind alle führenden 0-Bits, sowie das erste 1-Bit des Koeffizienten. Wird das erste 1-Bit übertragen, so wird ein Koeffizient signifikant.

2. *Sign-Bit*, das Vorzeichen des Koeffizienten.

3. *Magnitude-Bits*, alle übrigen Bits des Koeffizienten.

Wenn alle Bits eines Typs innerhalb einer Bitplane oder im gesamten Bild gemeint sind, so werden auch die Begriffe *Significance-Map*, *Sign-Map* und *Magnitude-Map* verwendet.

Anhand eines Beispiels soll das Vorgehen verdeutlicht werden. Angenommen ein transformiertes Bild hat acht Bitplanes – nummeriert von 7 bis 0 – und es ist der Koeffizient -42 zu kodieren, in Betrag/Vorzeichendarstellung sieht der Koeffizient dann wie folgt aus:

Vorzeichen	Betrag in Bitplanes							
	7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1	0

Beim Durchlaufen der Bitplanes wird dieser Koeffizient dann so erfasst:

Bitplane	Typ	Bit
7	Significance	0
6	Significance	0
5	Significance, Sign	1, 1
4	Magnitude	0
3	Magnitude	1
2	Magnitude	0
1	Magnitude	1
0	Magnitude	0

Eine einfache Realisierung eines Bitplanecoders könnte nun einen adaptiven binären Arithmetischen Kodierer einsetzen, der für die Significance-, Sign- und Magnitude-Map jeweils einen Kontext nutzt. Genau das macht Codec 1, der später in Abschnitt 5.2.1 vorgestellt wird.

Mit jeder weiteren übertragenen Bitplane, kann der Decoder den Koeffizient genauer approximieren, bis schliesslich der exakte Wert feststeht. Es findet eine implizite Quantisierung statt, mit jeder zusätzlichen Bitplane wird die Länge des Quantisierungsintervalls halbiert. Das Intervall um die Null ist dabei immer doppelt so breit wie die übrigen Intervalle. Diese Art der Quantisierung wird auch als *embedded deadzone quantizer* bezeichnet. Zur Minimierung des mittleren Fehlers wird auf der Decoderseite immer die Intervallmitte verwendet. Tabelle 4.1 zeigt am Beispiel der -42, wie sich die vom Decoder rekonstrierten

übertragen bis Bitplane	Vorzeichen	Bitplane 7	6	5	4	3	2	1	0	Wert
7		0	0	0	0	0	0	0	0	0
6		0	0	0	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	-48
4	1	0	0	1	0	1	0	0	0	-40
3	1	0	0	1	0	1	1	0	0	-44
2	1	0	0	1	0	1	0	1	0	-42
1	1	0	0	1	0	1	0	1	1	-43
0	1	0	0	1	0	1	0	1	0	-42

Tabelle 4.1: Am Beispiel der -42 wird gezeigt, wie sich der rekonstruierte Wert auf Decoderseite entwickelt.

Werte über die Bitplanes hinweg entwickeln. Alle unterstrichenen Bits hat der Decoder in der betreffenden Bitplane bereits empfangen.

Wie bereits erwähnt wird das Bild Bitplane für Bitplane durchlaufen, wobei mit der obersten Bitplane B begonnen wird. Geht man nun davon aus, dass die Wavelet-Transformation sehr viele Koeffizienten nahe Null erzeugt hat, so müssen für diese Koeffizienten viele Significance-Bits kodiert werden, weil die ersten 1-Bits vieler Koeffizienten erst in den unteren Bitplanes auftreten. Daraus lässt sich schlussfolgern, dass es deutlich mehr Significance-Bits als Sign- oder Magnitude-Bits geben muss. Das Bild `lena.pgm` verfügt beispielsweise nach einer 9 stufigen 5/3-Wavelet-Transformation über 3.360.945 Significance-, 228.334 Sign- und 309.071 Magnitude-Bits (eine genauere Aufschlüsselung für jede Bitplane erfolgt in Tabelle 5.2). Deshalb ist es nicht verwunderlich, dass der Komprimierung der Significance-Map das Hauptaugenmerk gilt.

Die vorgestellten Verfahren unterscheiden sich hauptsächlich in der Art und Weise, wie sie die Significance-Map komprimieren. Im wesentlichen sind es vier Eigenschaften wavelet-transformierter Bilder die dabei ausgenutzt werden:

1. In den oberen Bitplanes sind nur wenige Koeffizienten signifikant.

2. Koeffizienten, die in ihrer näheren Umgebung bereits signifikante Koeffizienten haben, besitzen eine höhere Wahrscheinlichkeit ebenfalls signifikant zu werden.

3. Die verschiedenen Transformationsstufen eines Subbandes haben eine gewisse Ähnlichkeit (wie beispielsweise in Abbildung 3.2 zu erkennen ist).

4. Häufig werden Koeffizienten in Transformationsstufe l erst dann signifikant, wenn der, in der nächsthöheren Transformationsstufe $l+1$ an vergleichbarer Position liegende, Koeffizient bereits signifikant ist. Dadurch ergibt sich eine gewisse Pyramidenstruktur. Hauptgrund dafür ist die unterschiedliche Wichtung der Koeffizienten in den Transformationsstufen.

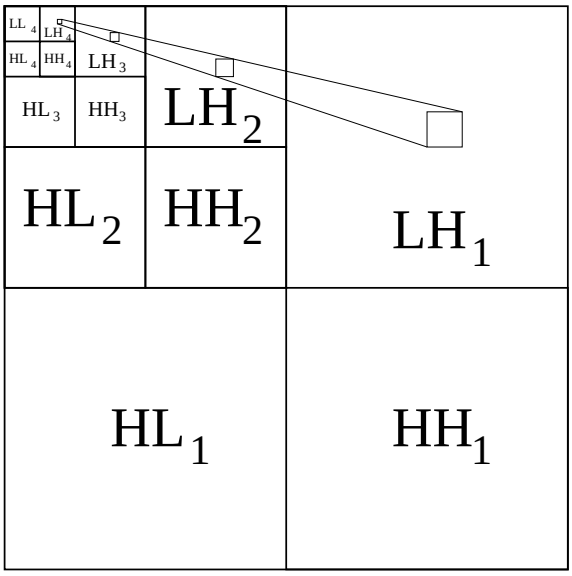


Abbildung 4.1: Insignifikante Koeffizienten, die an vergleichbaren Positionen in verschiedenen Transformationsstufen desselben Subbands liegen bilden einen Zerotree.

4.2 EZW, SPIHT

EZW [10] (Embedded Zerotree Wavelet) und SPIHT [11] (Set Partitioning In Hierarchical Trees) machen sich den Punkt 4 der vorangegangenen Aufzählung zunutze. Bedingt durch die Anordnung der Subbänder lässt sich die Position des Koeffizienten in der nächsthöheren Transformationsstufe recht einfach berechnen. Angenommen ein Koeffizient in Transformationsstufe l hat die Position (x, y), der an vergleichbarer Position liegende Koeffizient in $l + 1$ – häufig auch als Elternkoeffizient bezeichnet – hat dann die Position $(\frac{x}{2}, \frac{y}{2})$. Es gibt somit immer 4 Kindkoeffizienten, die zu einem Elternkoeffizienten gehören. In Abbildung 4.1 ist dies verdeutlicht, der Koeffizient aus LH_4 hat 4 "Kinder" in LH_3, 16 "Enkel" in LH_2 und 64 "Urenkel" in LH_1. Ist nun der Koeffizient in LH_4 insignifikant, so ist es aufgrund der pyramidenförmigen Verteilung der Signifikanzen wahrscheinlich, dass auch alle Kinder, Enkel und Urenkel insignifikant sind. Eine solche baumartige Struktur wird als *Zerotree* bezeichnet. Durch geschickte Kodierung lassen sich so grosse Bereiche insignifikanter Koeffizienten mit wenigen Bits repräsentieren.

SPIHT ist eine Weiterentwicklung von EZW und arbeitet bei der Kodierung etwas effizienter. Das Verfahren nutzt dazu 3 Listen:

1. LIS - *List of Insignificant Sets*, enthält die Koordinaten der Elternkoeffizienten aller Zerotrees. SPIHT unterscheidet dabei zwei Typen:

   A) die Elternkoeffizienten gehören nicht zum Zerotree

   B) sowohl die Eltern- aus auch die Kindkoeffizienten sind nicht Bestandteil des Zerotrees

Der Ablauf des Algorithmus stellt dabei sicher, dass die Koeffizienten, die nicht zum Zerotree gehören bereits in LIP oder LSP enthalten sind.

2. LIP - *List of Insignificant Pixels*, enthält die Koordinaten insignifikanter Koeffizienten.

3. LSP - *List of Significant Pixels*, enthält die Koordinaten bereits signifikanter Koeffizienten, also die Positionen, für die nur noch Magnitude-Bits übertragen werden.

Zu Beginn sind die Listen wie folgt initialisiert:

- LIP enthält die Koordinaten von $\mathrm{LL}_{l_{max}}$, $\mathrm{LH}_{l_{max}}$, $\mathrm{HL}_{l_{max}}$ und $\mathrm{HH}_{l_{max}}$, wobei l_{max} die höchste Transformationsstufe des Bildes bezeichnet.
- LIS enthält die Koordinaten der Elternknoten der Zerotrees, also alle Koordinaten von $\mathrm{LH}_{l_{max}}$, $\mathrm{HL}_{l_{max}}$ und $\mathrm{HH}_{l_{max}}$ ($\mathrm{LL}_{l_{max}}$ hat keine Kindkoeffizienten).
- LSP ist leer.

In jeder Bitplane werden alle 3 Listen genau einmal durchlaufen.

1. Für alle Koeffizienten in LIP wird das Significance-Bit übertragen. Wird der Koeffizient dabei signifikant, so wird er in LIP gelöscht, an LSP angehängt und zusätzlich wird noch das Sign-Bit übermittelt.

2. Für alle Zerotrees wird geprüft ob sie in dieser Bitplane signifikante Koeffizienten enthalten, abhängig vom Typ des Zerotrees werden unterschiedliche Aktionen ausgeführt:

   A) falls ja, so werden die Significance-Bits der Kindkoeffizienten übertragen und jene zu LIP bzw. LSP hinzugefügt, der bisherige Eintrag in LIS für den Zerotree wird gelöscht und es wird ein Eintrag vom Typ B an LIS angehängt, dabei werden die gleichen Koordinaten verwendet wie beim bisherigen Eintrag

   B) falls ja, so wird der Eintrag in LIS gelöscht und es werden 4 Einträge vom Typ A an LIS angehängt, wobei diesmal die Koordinaten der Kindkoeffizienten verwendet werden

   Die Tatsache, dass SPIHT zwei verschiedene Typen von Zerotrees einsetzt, lässt sich abermals mit der pyramidförmigen Verteilung der Signifikanzen begründen. Da es unwahrscheinlich ist, dass auch die Enkelkoeffizienten in derselben Bitplane signifikant werden, wie die Kindkoeffizienten, ist es effizienter alle 4 Zerotrees der Kinder auf einmal zu testen (ein LIS-Eintrag vom Typ B) anstatt gleich vier Typ A Einträge überprüfen zu müssen.

3. Für alle bereits signifikanten Koeffizienten in LSP werden die Magnitude-Bits übertragen, dabei werden jedoch nur Koeffizienten berücksichtigt, die nicht in dieser Bitplane signifikant geworden sind.

SPIHT nutzt implizit aber auch andere, der zuvor aufgezählten, Eigenschaften wavelettransformierter Bilder. So haben Koeffizienten aus LIP mit hoher Wahrscheichlichkeit bereits signifikante Koeffizienten in ihrer näheren Umgebung. Sind nämlich in einem Zerotree vom Typ A Koeffizienten signifikant, so sind dies mit hoher Wahrscheinlichkeit einige der Kindkoeffizienten, die verbliebenen insignifikanten Kindkoeffizienten werden an LIP angehängt und in der nächsten Bitplane erneut getestet. Man kann davon ausgehen, dass etwa jeder dritte Koeffizient beim Durchlaufen der LIP signifikant wird. Auch die Reihenfolge in der die Listen abgearbeitet werden ist nicht willkürlich, sie ist nämlich so gewählt, dass Listen, die den PSNR im Verhältnis zu den benötigten Bits am schnellsten steigern können, zuerst bearbeitet werden. Durch den Einsatz von Arithmetischer Kodierung zur weiteren Komprimierung des SPIHT-Datenstroms lassen sich die PSNR-Werte zusätzlich steigern, jedoch etwas zu Lasten der Geschwindigkeit. Durch eine Vorberechnung der Signifikantwerdung von Teilbäumen kann man hingegen die Zerotree-Tests und damit den gesamten Algorithmus beschleunigen.

Der entscheidende Ansatz bei SPIHT ist also die geschickte Gruppierung von insignifikanten Koeffizienten zu Zerotrees, um jene mit möglichst wenig Bits kodieren zu können. Zerotrees sind dabei aber nur eine Möglichkeit der Gruppierung. In [15] wird ein anderer Ansatz vorgestellt, der eine anschliessende Arithmetische Kodierung unnötig werden lässt. Weitere Detailverbesserungen an SPIHT, welche den bisherigen Zerotree-Ansatz beibehalten sind in [13] aufgeführt. In [14] wird schliesslich eine Umsetzung von SPIHT ohne Listen erläutert. Dabei wird einmal durch die Bitplane gelaufen und bei jedem Koeffizient entschieden zu welcher Liste er gehören würde, um die nötigen Abarbeitungschritte durchzuführen. Die getrennte Abarbeitung der Listen hatte es jedoch erlaubt all jene Koeffizienten zuerst zu bearbeiten, die den PSNR am schnellsten zu steigern vermögen – dieser Vorteil geht nun infolge der geänderten Reihenfolge verloren. Die benötigten Bits pro Bitplane bleiben jedoch exakt gleich, da nur eine Umordnung innerhalb der Bitplane erfolgt.

4.3 ECECOW

Während SPIHT auch ohne Arithmetische Kodierung sehr gut funktioniert, ist dies bei ECECOW undenkbar. ECECOW [16] (Embedded Conditional Entropy Coding Of Wavelet Coefficients) macht aktiv Gebrauch von der adaptiven Arithmetischen Kodierung und verwendet dabei etliche Kontexte. Zur Kontextbildung werden all jene Bits der umgebenden Koeffizienten herangezogen, die sowohl En- als auch Decoder bekannt sind. Die Auswahl dieser Koeffizienten erfolgt dabei in Abhängigkeit des Subbands. Angenommen es wird gerade Bitplane i durchlaufen, dann werden folgende benachbarte Koeffizienten berücksichtigt:

$$
\begin{aligned}
S_{LH} &= \{N_{B..i}, W_{B..i}, NW_{B..i}, NE_{B..i}, NN_{B..i}, S_{B..i+1}, P_{B..i}, PN_{B..i}, PS_{B..i}\} \\
S_{HL} &= \{N_{B..i}, W_{B..i}, NW_{B..i}, NE_{B..i}, WW_{B..i}, E_{B..i+1}, P_{B..i}, PW_{B..i}, PE_{B..i}\} \\
S_{HH} &= \{N_{B..i}, W_{B..i}, NW_{B..i}, NE_{B..i}, S_{B..i+1}, E_{B..i+1}, P_{B..i}, c_{HL}, c_{LH}\}
\end{aligned}
$$

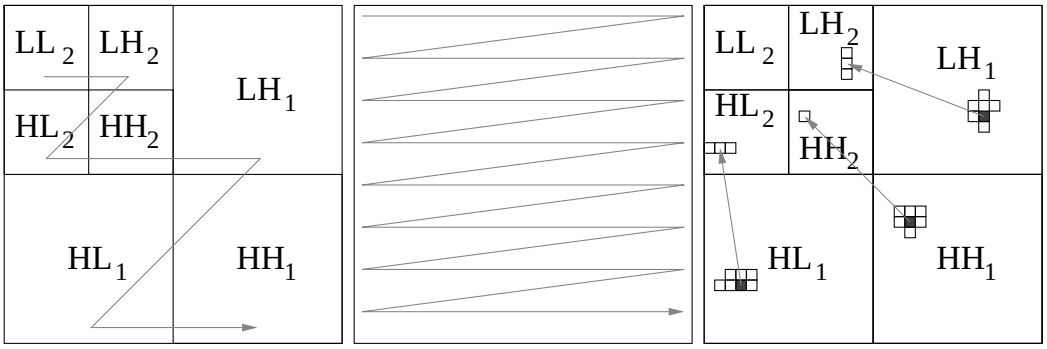


Abbildung 4.2: Im linken und mittleren Bild wird die Reihenfolge dargestellt, in der die Koeffizienten bearbeitet werden. In jeder Transformationsstufe wird zunächst das LH-Subband komplett durchlaufen, bevor das HL- und HH-Subband betrachtet wird, erst dann wird in die nächsttiefere Transformationsstufe gewechselt. Das mittlere Bild veranschaulicht, wie jeder Subbandblock Zeile für Zeile von links nach rechts durchlaufen wird. Das rechte Bild zeigt die Koeffizienten, die bei der Kontextbildung berücksichtigt werden.

c_{HL} und c_{LH} sind dabei die "Geschwister" aus den anderen Subbändern der gleichen Transformationsstufe. N, S, W und E bezeichnen die Ausrichtung (North, West, ...), mit einem vorangestellten P ist der entsprechende Elternkoeffizient gemeint. Der tiefgestellte Index gibt die verfügbaren Bits jedes Koeffizienten an. Je nachdem in welcher Reihenfolge (Scanorder) die Koeffizienten durchlaufen werden kann bereits das i-te Bit verfügbar sein oder nicht. Abbildung 4.2 veranschaulicht dies, darüber hinaus wird dort auch die Auswahl der umgebenden Koeffizienten gezeigt.

Laut [16] werden nun über einer Testbildmenge die Parameter $\alpha_{LH,j}$, $\alpha_{HL,j}$ und $\alpha_{HH,j}$ sowie die Werte für die Quantisierer Q_{LH}, Q_{HL} und Q_{HH} ermittelt. Dabei müssen die α-Werte so bestimmt werden, dass die zugehörigen Δ-Werte

$$\Delta_{LH} = \sum_{z_j \in S_{LH}} \alpha_{LH,j} z_j \qquad \Delta_{HL} = \sum_{z_j \in S_{HL}} \alpha_{HL,j} z_j \qquad \Delta_{HH} = \sum_{z_j \in S_{HH}} \alpha_{HH,j} z_j$$

über alle Koeffizienten, die quadratische Abweichung zum betrachteten Koeffizienten minimieren. Die Quantisierer dienen nun dazu die Δ-Werte optimal in Kontexte einzuteilen. Zusätzlich wird ein Bitvektor $T = t_4 t_3 t_2 t_1 t_0$ mit

$$\begin{aligned} t_0 &= Z_2(N_{B..i}\ ,\ c_{B..i+1}) \\ t_1 &= Z_2(W_{B..i}\ ,\ c_{B..i+1}) \\ t_2 &= Z_2(S_{B..i+1}\ ,\ c_{B..i+1}) \\ t_3 &= Z_2(E_{B..i+1}\ ,\ c_{B..i+1}) \end{aligned}$$

$$t_4 = \begin{cases} Z_2(P_{B..i} + PN_{B..i} + PS_{B..i}\ ,\ 6c_{B..i+1}) & \text{wenn in LH-Subband} \\ Z_2(P_{B..i} + PW_{B..i} + PE_{B..i}\ ,\ 6c_{B..i+1}) & \text{wenn in HL-Subband} \\ 0 & \text{sonst} \end{cases}$$

$$\text{mit} \quad Z_2(x_0, x_1) = \begin{cases} 0 & \text{wenn} \quad x_0 > x_1 \\ 1 & \text{sonst} \end{cases}$$

bei der Kontextbildung berücksichtigt. Das i-te Bit des Koeffizienten c wird also in dem Kontext $Q_\theta(\Delta_\theta) \cdot T$, $\theta \in \{LH, HL, HH\}$ kodiert. Die Erläuterungen in [16] lassen darauf schliessen, dass Significance- und Magnitude-Bits mit denselben Kontexten kodiert werden. Für Sign-Bits gibt es hingegen separate Kontexte, die aus den Vorzeichen der benachbarten Koeffizienten gebildet werden.

Es bleiben aber etliche Fragen offen. In [16] werden zwar PSNR-Werte für `lena.pgm` und `barbara.pgm` veröffentlicht, jedoch sind weder die verwendeten α-Werte, noch die genutzten Quantisierer Q angegeben. Sollten diese über einer Testbildmenge ermittelt worden sein, so ist auch nicht klar welche Bilder darin enthalten waren. Ausserdem wurde die Zahl der verwendeten Kontexte an keiner Stelle explizit genannt, so dass man auch hier nur mutmassen kann. Etwas überraschend ist ebenfalls, dass ECECOW nur genau einmal durch die Bitplane läuft, so wie etwa die SPIHT Implementierung ohne Listen [14], schon alleine durch eine Umordnung der betrachteten Koeffizienten sollten sich Verbesserungen erzielen lassen, jedoch müssten eventuelle Rückwirkungen auf die Kontexte dabei berücksichtigt werden.

4.4 EBCOT

Die bisher vorgestellten Verfahren produzieren einen Datenstrom der *SNR-scalable* ist, d.h. mit jedem zusätzlichen Bit wird die Bildqualität insgesamt verbessert. EBCOT [17, 18] (Embedded Block Coding with Optimized Truncation) macht es zusätzlich möglich, dass ein komprimiertes Bild auch *Resolution-scalable* sein kann, d.h. es kann zunächst eine Auflösung komplett in der vorgegebenen Qualität dargestellt werden, bevor auch nur ein Bit für eine grössere Auflösung verwendet wird. Die Wavelet-Transformation macht dies durch ihre Multiresolution-Darstellung erst möglich. Diese Flexibilität von EBCOT war sicherlich auch ein Grund für die Entscheidung dieses Verfahren für den JPEG2000-Standard auszuwählen.

Grundsätzlich wäre es auch möglich beispielsweise den SPIHT-Algorithmus so zu verändern, dass auch er einen Datenstrom erzeugt, der Resolution-scalable ist, die Besonderheit an EBCOT ist jedoch, dass ein einmal erzeugter Datenstrom ohne erneute Komprimierung des Ausgangsbildes sowohl SNR-scalable als auch resolution-scalable sein kann, es ist nur eine Umordnung des bestehenden Datenstromes nötig.

Um dies zu ermöglichen, teilt EBCOT jedes Subband des transformierten Bildes in Blöcke ein – üblicherweise mit der Grösse 32×32 bzw. 64×64. Jeder dieser Blöcke wird vollkommen unabhängig von den übrigen Blöcken komprimiert. Die Anordnung der Datenströme der komprimierten Blöcke entscheidet nun darüber, welche Eigenschaften der Gesamtdatenstrom besitzt. Durch die Blockaufteilung lässt sich beispielsweise auch eine

Region of Interest realisieren, da man sich einfach nur auf die Blöcke zu beschränken braucht, die in dem interessierenden Bereich des Bildes liegen. Die reine Komprimierung der einzelnen Blöcke und die Erstellung eines Datenstromes kann man also durchaus getrennt betrachten.

4.4.1 Komprimierung der Blöcke

Die Kodierung der einzelnen Blöcke ist in gewisser Weise ähnlich zu ECECOW, denn sie ist ebenfalls ohne adaptiven Arithmetischen Kodierer nicht denkbar. Auch hier werden die umgebenden Koeffizienten genutzt, um daraus einen Kontext zu bilden, in dem der gerade betrachtete Koeffizient kodiert wird. Aufgrund der Grösse der Blöcke ist insbesondere darauf zu achten, dass nur eine überschaubare Anzahl von Kontexten zum Einsatz kommt, denn sonst kann der Overhead zusätzlicher Kontexte die möglichen Einsparungen übersteigen. Im Gegensatz zu ECECOW wird eine Bitplane aber nicht nur einmal durchlaufen, sondern mehrfach in sogenannten *fractional bitplane scans*. Die Durchgänge sind im einzelnen:

1. *Significance Propagation Pass* – hier werden nur Koeffizienten betrachtet, die bereits signifikante Koeffizienten in ihrer Umgebung haben, in gewisser Weise kann man es mit der LIP von SPIHT vergleichen. Beim SPIHT-Algorithmus ergab sich dies aber implizit durch das Verfahren selbst, bei EBCOT werden hingegen die benachbarten Koeffizienten wirklich überprüft.

2. *Magnitude Refinement Pass* – die Magnitude-Bits bereits signifikanter Koeffizienten werden hier übermittelt (analog zu LSP beim SPIHT).

3. *Cleanup Pass* – alle Koeffizienten, die keine signifikante Nachbarschaft haben, werden in diesem Durchgang berücksichtigt.

Auch in Bezug auf die Scanorder verhält sich EBCOT anders als ECECOW, statt zeilenweise von links nach rechts zu laufen, werden die Zeilen immer in 4er Bündeln Spalte für Spalte durchlaufen, Abbildung 4.3 veranschaulicht dies. Beispielhaft sind ebenfalls all jene benachbarten Koeffizienten dargestellt, die bei der Kontextbildung berücksichtigt werden. Die im Significance Propagation Pass verwendeten Kontexte sind in Tabelle 4.2 aufgeführt. Dabei ergeben sich die Werte für h, v, d, $\hat{h}$ und $\hat{v}$ wie folgt:

$$\begin{aligned} h &= W + E \\ v &= N + S \\ d &= NW + NE + SW + SE \\ N, S, W, E, \ldots &= \begin{cases} 0 & \text{wenn benachbarter Koeffizient nicht signifikant} \\ 1 & \text{wenn benachbarter Koeffizient signifikant} \end{cases} \end{aligned}$$

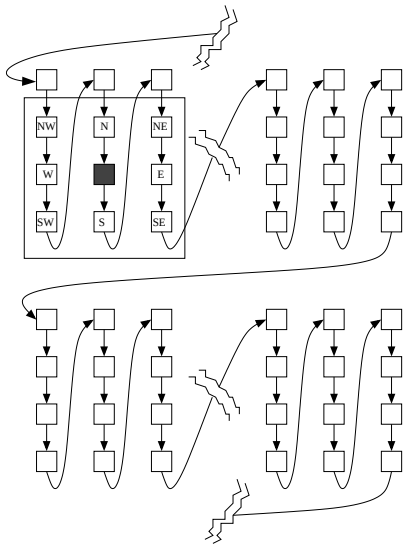


Abbildung 4.3: EBCOT nutzt eine abweichende Scanorder, anstatt Zeile für Zeile von links nach rechts zu laufen, werden immer 4 Koeffizienten einer Spalte durchlaufen. Bei der Kodierung eines Koeffizienten werden jeweils die horizontal, vertikal und diagonal direkt benachbarten Koeffizienten genutzt, um den Kontext auszuwählen.

$$
\begin{aligned}
\hat{h} &= \hat{W} + \hat{E} \\
\hat{v} &= \hat{N} + \hat{S} \\
\hat{N}, \hat{S}, \hat{W}, \hat{E} &= \begin{cases} -1 & \text{wenn benachbarter Koeffizient negativ} \\ 0 & \text{wenn benachbarter Koeffizient nicht signifikant} \\ 1 & \text{wenn benachbarter Koeffizient positiv} \end{cases}
\end{aligned}
$$

Um die vollkommene Unabhängigkeit der Blöcke sicherzustellen, werden benachbarte Koeffizienten, die nicht innerhalb des Blockes liegen als insignifikant angenommen. Bei der Kontextbildung für die Sign-Bits wird Symmetrie ausgenutzt, um die Anzahl der Kontexte zu reduzieren. Deshalb muss bei der Ermittlung des zu kodierenden Vorzeichens mit dem Wert s aus Tabelle 4.2 multipliziert werden. Im Magnitude Refinement Pass werden drei Kontexte verwendet:

$$
\# = \begin{cases} 0 & \text{wenn Koeffizient in letzter Bitplane signifikant geworden} \\ & \text{und benachbarte Koeffizienten nicht signifikant} \\ 1 & \text{wenn Koeffizient in letzter Bitplane signifikant geworden} \\ & \text{und benachbarte Koeffizienten signifikant} \\ 2 & \text{sonst} \end{cases}
$$

Im Cleanup Pass kommt eine RLE-ähnliche Kodierung zum Einsatz, um die Signifikanz der verbleibenden Koeffizienten zu kodieren.

Significance-Bits								
	LL, LH			HL			HH	
#	v	h	d	h	v	d	d	$h+v$
0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	1
2	0	0	>1	0	0	>1	0	>1
3	0	1	x	0	1	x	1	0
4	0	2	x	0	2	x	1	1
5	1	0	0	1	0	0	1	>1
6	1	0	>0	1	0	>0	2	0
7	1	>0	x	1	>0	x	2	>0
8	2	x	x	2	x	x	>2	x

Sign-Bits			
#	$\hat{h}$	$\hat{v}$	s
4	1	1	1
3	1	0	1
2	1	-1	1
1	0	1	-1
0	0	0	1
1	0	-1	1
2	-1	1	-1
3	-1	0	-1
4	-1	-1	-1

Tabelle 4.2: Kontexte im Significance Propagation Pass (in [17, 18] werden die Subbänder anders bezeichnet: LH ist dort HL und umgekehrt, die hier verwendeten Bezeichnungen orientieren sich an Kapitel 3).

Ohne weitere Massnahmen, könnte jetzt zwar jeder Block unabhängig komprimiert werden, es liesse sich daraus aber nur sehr schwer ein sinnvoller Gesamtdatenstrom zusammensetzen, weil man bisher einen Block nur ganz oder gar nicht nutzen kann. Hier kommen die *Truncation Points* ins Spiel, sie definieren Schnittpunkte im Datenstrom eines Blockes, die später dazu genutzt werden können den kodierten Block aufzuteilen. Dadurch lässt sich wesentlich einfacher ein Gesamtdatenstrom erzeugen.

Ein Truncation Point beinhaltet dabei immer zwei Werte, zum einen die Zahl der benötigten Bits (Length L) und zum zweiten ein Maß für die Bildqualität (Distortion D – üblicherweise ist dies der MSE). Das Ende eines jeden Durchlaufes durch die Bitplane scheint für einen Truncation Point wie prädestiniert. Jedoch werden später nur jene Truncation Points berücksichtigt, die auf der konvexen Hülle der L-D-Kurve liegen, also genau diejenigen die den MSE im Verhältnis zur verwendeten Bitmenge am meisten reduzieren.

4.4.2 Erzeugung des Gesamtdatenstroms

Mit Hilfe der zuvor komprimierten Blöcke und den zugehörigen Truncation Points – also den (L, D)-Paaren – lässt sich nun ein Datenstrom für das Gesamtbild erzeugen. Soll dieser Resolution-scalable sein, so werden zuerst all jene Blöcke genommen, die für die gewünschte Auflösung nötig sind, erst wenn sie komplett im erzeugten Datenstrom enthalten sind, werden die übrigen Blöcke betrachtet.

Soll der erzeugte Datenstrom SNR-scalable sein – so wie dies normalerweise bei SPIHT und ECECOW der Fall ist – werden aus all den (L, D)-Paaren aller Blöcke immer jeweils die heraus gesucht, die den MSE im Verhältnis zu den benötigten Bits am schnellsten reduzieren. Was zuvor also auf Blockebene zur Bestimmung der konvexen Hülle gemacht wurde, erfolgt nun für das ganze Bild. Jedoch kann es so zu sehr vielen kleinen Blockteilen kommen, die alle mit einem Header versehen werden müssen. Um diesen Overhead zu

reduzieren werden Quality-Layer eingeführt. Innerhalb eines Quality-Layers werden alle Teile eines Blockes wieder zusammengefasst. Das führt zwar dazu, dass innerhalb des Layers keine perfekte SNR-Scalability mehr gewährleistet ist, reduziert aber den Overhead.

Je mehr Quality-Layer ein Bild hat, desto mehr Header für die Blockteile werden nötig, was letztendlich zu schlechteren PSNR-Werten führt. Übliche Werte für Quality-Layer sind beispielsweise 0,5 bpp oder 1,0 bpp. Werden mehrere Quality-Layer verwendet so werden sie standardmässig in logarithmischen Abständen gesetzt, beispielsweise bei 5 Layern: 0,125 bpp; 0,25 bpp; 0,5 bpp; 1,0 bpp und 2,0 bpp.

Ergänzend lässt sich noch sagen, dass in gewissen Grenzen ein Datenstrom sogar beides sein kann, Resolution-scalable und SNR-scalable. Solange nämlich die Blöcke für die gewünschte Auflösung in den Datenstrom geschrieben werden, können diese wiederum so angeordnet werden, dass sie SNR-scalable sind.

Kapitel 5

Eigene Ansätze

In den vorigen Kapiteln wurde die Wavelet-Transformation erläutert und es wurde das Prinzip des Bitplanecoders vorgestellt. Ebenso wurden die Grundideen einiger bereits existierender Verfahren zur effizienten Bitplane-Kodierung genannt. Als Ausgangspunkt der nun folgenden eigenen Untersuchungen soll eine sehr einfache Umsetzung eines Bitplanecoders dienen, der in diesem Kapitel immer weiter ausgebaut wird und schliesslich zu dem YAWICA-Algorithmus führt. Um sinnvolle Vergleiche mit den bereits existierenden Verfahren durchführen zu können, soll jedoch zunächst geklärt werden, wie Vergleichsmessungen durchgeführt werden.

5.1 Messung

Allgemein geht es bei der Datenkompression immer darum mit möglichst wenig Bits zur Darstellung der Eingabedaten auszukommen. Speziell bei der Bilddatenkompression ist aber zusätzlich neben der verlustlosen auch die verlustbehaftete Wiederherstellung der Eingabedaten von Interesse, wenn sich dadurch zusätzlich Bits sparen lassen. Um also die Leistungsfähigkeit eines Bildkompressionsverfahrens zu beurteilen, könnte man die Frage stellen,

(a) welche Bildqualität bei einer vorgegebenen Bitanzahl erreicht wurde.

(b) wieviel Bits benötigt wurden, um eine vorgegebene Bildqualität zu erreichen.

Auf den ersten Blick scheinen beide Fragen gleichwertig zu sein, sich auf eine Fragestellung festzulegen sollte also ausreichend sein. Jedoch wird sich zeigen, dass es unter bestimmten Umständen durchaus sinnvoll ist beide Fragestellungen zu berücksichtigen. Dabei wird als Maß für die Bildqualität häufig der, in dB gemessene, PSNR-Wert[1] herangezogen. So auch hier:

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=0}^{N-1} (x_i - \hat{x_i})^2$$

[1] PSNR steht für Peak Signal to Noise Ratio und MSE für Mean Square Error.

$$\text{PSNR} = 10 \cdot \log_{10}\left(\frac{x_{max}^2}{\text{MSE}}\right) \quad .$$

N entspricht hierbei der Anzahl der x-Werte. Die x_i sind die Originalwerte, $\hat{x}_i$ bezeichnet die rekonstruierten Gegenstücke und x_{max} gibt den grösstmöglichen Wert an, den ein x_i annehmen kann. Bei den hier verwendeten 8 Bit Graustufenbildern ist $x_{max} = 255$.

Die benötigten Bits werden aber nicht direkt als absolute Zahl sondern immer in Relation zur Auflösung eines Bildes in *Bits per Pixel* angegeben. Bei einem 8 Bit Graustufenbild mit einer Auflösung von 512×512 entsprechen 0,5 bpp somit 131.072 Bits und damit einem Sechszehntel der ursprünglichen Grösse. Zudem bietet dies die Möglichkeit einfach einen Durchschnittswert über mehrere Bilder berechnen zu können.

Beide Fragestellungen parallel zu betrachten dient im wesentlichen dazu, den Unterschied zwischen Codec 4 und Codec 5 aufzuzeigen. Codec 5 nutzt eine, gegenüber Codec 4, geänderte Übertragungsreihenfolge. Am Ende einer Bitplane erzielen beide die gleiche Datenreduktion, jedoch nicht innerhalb einer Bitplane. Deshalb wird bei Fragestellung (b) als vorgegebene Qualität auch kein fester PSNR-Wert genommen sondern immer eine feste Bitplane. Diese Bitplane ist in Bezug auf die oberste, im wavelet-transformierten Bild verwendete, Bitplane B gewählt. Die Notation $B-5$ bedeutet also, dass im Bild alle sechs Bitplanes von B bis einschliesslich $B-5$ betrachtet werden.

Für die Messungen werden 11 Testbilder verwendet, die auch auf der beiliegenden CD enthalten sind (siehe Anhang 7). Die Bilder haben jeweils unterschiedliche Charakteristiken und eignen sich deshalb für einen repräsentativen Vergleich. Ausserdem wird häufig auf sie zurückgegriffen, wenn in Veröffentlichungen PSNR-Werte genannt werden, somit bilden sie eine Art Quasi-Benchmark. Die Ergebnisse für die einzelnen Codecs werden immer über alle Testbilder berechnet. In Situationen in denen es jedoch wenig sinnvoll ist, den Durchschnitt über mehrere Bilder zu bilden, wird auf `lena.pgm` Bezug genommen.

Zusätzlich zu den bisher genannten Maßen PSNR und bpp, kommen auch noch *Bits per Significance* und *Seconds per Million Pixels* zum Einsatz. Mit ersterem wird angegeben wieviel Bits benötigt werden, um einen signifikanten Koeffizienten zu kodieren – entscheidend ist hier also das Verhältnis von Significance-Bits die Null sind zu denen die Eins sind. Das zweite Maß wird für die Geschwindigkeitsmessung verwendet. Mit dem Kehrwert erhält man ein einfaches Maß für den Durchsatz.

5.2 Einfache Umsetzung eines Bitplanecoders

5.2.1 Codec 1

Das in Abschnitt 4.1 vorgestellte Prinzip eines Bitplanecoders unterteilt die Kodierung eines Koeffizienten in drei Phasen. Zuerst werden die Significance-Bits kodiert. Wurde Signifikanz festgestellt wird gleich anschliessend das Vorzeichen kodiert und in den Folgebitplanes die Magnitude-Bits. In allen drei Phasen treten ausschliesslich binäre Ereignisse auf (insignifikant/signifikant, Vorzeichen +/-, Magnitude 0 oder 1), deshalb kann ein binärer Arithmetischer Kodierer eingesetzt werden, genauer gesagt ein adaptiver binärer Arithmetischer Kodierer mit drei Kontexten – jeweils einer für die Significance-, Sign- und Magnitude-Map. Die Resultate sind in Tabelle 5.1 aufgelistet.

	S+P	5/3	9/7
0,10 bpp	25,41 dB	25,24 dB	25,69 dB
0,25 bpp	28,56 dB	28,22 dB	28,95 dB
0,50 bpp	31,45 dB	31,15 dB	31,99 dB
0,75 bpp	33,50 dB	33,29 dB	34,12 dB
1,00 bpp	35,05 dB	34,84 dB	35,84 dB
2,00 bpp	39,88 dB	39,85 dB	41,22 dB
B bis $B-5$	0,02 bpp	0,03 bpp	0,02 bpp
B bis $B-6$	0,06 bpp	0,08 bpp	0,06 bpp
B bis $B-7$	0,16 bpp	0,18 bpp	0,16 bpp
B bis $B-8$	0,37 bpp	0,36 bpp	0,35 bpp
B bis $B-9$	0,72 bpp	0,67 bpp	0,67 bpp
B bis $B-10$	1,30 bpp	1,14 bpp	1,18 bpp
verlustfrei	4,85 bpp	4,86 bpp	-

Tabelle 5.1: Durchschnittswerte der mit Codec 1 komprimierten Testbilder.

Bitplane	Significance-Bits	Sign-Bits	Magnitude-Bits
14	262.144 Bits	1 Bits	0 Bits
13	262.143 Bits	10 Bits	1 Bits
12	262.133 Bits	36 Bits	11 Bits
11	262.097 Bits	112 Bits	47 Bits
10	261.985 Bits	216 Bits	159 Bits
9	261.769 Bits	582 Bits	374 Bits
8	261.187 Bits	1.267 Bits	954 Bits
7	259.920 Bits	2.634 Bits	2.212 Bits
6	257.285 Bits	4.967 Bits	4.811 Bits
5	252.316 Bits	9.817 Bits	9.636 Bits
4	242.472 Bits	20.751 Bits	18.904 Bits
3	221.552 Bits	46.958 Bits	37.520 Bits
2	173.593 Bits	70.781 Bits	76.263 Bits
1	98.414 Bits	56.398 Bits	114.578 Bits
0	21.935 Bits	13.804 Bits	43.601 Bits
$\sum$	3.360.945 Bits	228.334 Bits	309.071 Bits
Codec 1	642.217 Bits	228.317 Bits	296.371 Bits

Tabelle 5.2: Die Anzahl unkomprimierter Bits pro Bitplane in `lena.pgm` (5/3 transformiert) bei Verwendung von Codec 1. Die letzten beiden Zeilen enthalten die aufsummierten unkomprimierten Bits, sowie die, nach Einsatz der adaptiven Arithmetischen Kodierung, von Codec 1 benötigten Bits.

	Significance-Bits	Sign-Bits	Magnitude-Bits
0	3.132.611 Bits	113.511 Bits	189.395 Bits
1	228.334 Bits	114.823 Bits	119.676 Bits
$\sum$	3.360.945 Bits	228.334 Bits	309.071 Bits
p	6,8%	50,3%	38,7%
$H(p)$	0,3584	0,9999	0,9628
$H(p) \cdot \sum$	1.204.563 Bits	228.311 Bits	297.574 Bits
Codec 1	642.217 Bits	228.317 Bits	296.371 Bits

Tabelle 5.3: Die Anzahl der Bits pro Kontext die ein statischer Arithmetischer Kodierer ($H(p) \cdot \sum$) benötigen würde verglichen mit der Bitanzahl, die der adaptive Arithmetische Kodierer in Codec 1 benötigt am Beispiel von `lena.pgm` (5/3 transformiert).

Eine genauere Betrachtung der drei Kontexte soll nun am Beispiel von `lena.pgm` erfolgen. In Tabelle 5.2 ist die Anzahl der unkomprimierten Bits pro Bitplane und Kontext aufgeführt, die von dem Arithmetischen Kodierer verarbeitet werden. Es fällt zunächst einmal auf, dass für die Kodierung der Signifikanz die meisten Bits aufgewendet werden müssen. Zugleich lassen sich aber auch genau hier die höchsten Kompressionsraten erzielen. Während die Significance-Map auf etwa ein Fünftel der unkomprimierten Grösse reduziert werden konnte, konnten die Sign-Map nahezu gar nicht und die Magnitude-Map nur um 4% komprimiert werden.

Wie gut oder schlecht ein Bitstrom komprimiert werden kann hängt von der Verteilung der Nullen und Einsen ab. Nimmt man die Summenwerte und geht von einem statischen binären Arithmetischen Kodierer aus, so hat nur die Auftrittswahrscheinlichkeit der 1 Einfluss auf das Kompressionsergebnis[2]. Wiederum am Beispiel von `lena.pgm` sehen die Zahlen aus wie in Tabelle 5.3 dargestellt, wobei $H(p)$ gemäss Kapitel 2 wie folgt definiert ist:

$$H(p) = -(1-p) \cdot \log_2(1-p) - p \cdot \log_2 p$$

Es fällt deutlich auf, dass die in Codec 1 verwendete adaptive Arithmetische Kodierung gerade etwas mehr als die Hälfte der Bits benötigt, die eine statische Arithmetische Kodierung für die Significance-Map benötigen würde. Dies ist aber nicht verwunderlich, da die 1-Wahrscheinlichkeit in der Significance-Map stark von der Bitplane des transformierten Bildes abhängt. Dies lässt sich auch gut in Tabelle 5.2 erkennen. Die Spalte Significance-Bits enthält die Grösse der Significance-Map und der Anteil der Einsen darin ist genau gleich der Anzahl der kodierten Vorzeichen, da für jeden signifikant gewordenen Koeffizienten auch gleich das Vorzeichen mit kodiert wird. Während in Bitplane 14 der Einsanteil ($= p$) $\frac{1}{262.144} \approx 0,0004\%$ beträgt, liegt er in Bitplane 0 bei $\frac{13.804}{21.935} \approx 63\%$. Allgemein kann man also sagen: je tiefer die Bitplane desto wahrscheinlicher ist die Signifikanz

[2]Genausogut hätte man sich auch auf die Auftrittswahrscheinlichkeit der 0 festlegen können, da sich bei einem binären Arithmetischen Kodierer die Auftrittswahrscheinlichkeit des jeweils anderen Symbols aus der des gewählten Symbols errechnen lässt.

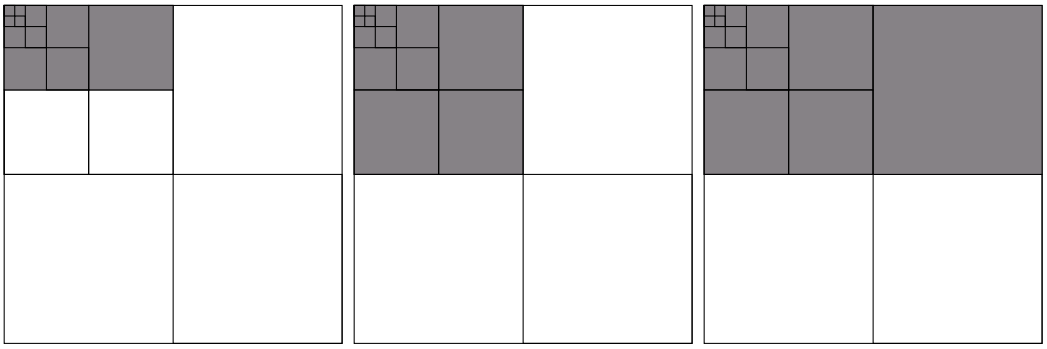


Abbildung 5.1: In `lena.pgm` (5/3 transformiert) sind in Bitplane 9 alle Koeffizienten in HL_2, HH_2, LH_1, HL_1 und HH_1 insignifikant, in Bitplane 8 hingegen nur in LH_1, HL_1 und HH_1. Bitplane 7 enthält zusätzlich noch signifikante Koeffizienten in LH_1

eines bisher insignifikanten Koeffizienten. Ein adaptiver Arithmetischer Kodierer passt sich kontinuierlich an die Auftrittswahrscheinlichkeit der 1 an und benötigt deshalb am Ende sogar weniger Bits als ein statischer Arithmetischer Kodierer, dies setzt aber immer eine gewisse Bitmenge zur Adaption voraus.

Aber selbst wenn man die Signifikanzwahrscheinlichkeit pro Bitplane berücksichtigen würde, wären 693.016 Bits nötig, um die Significance-Map zu kodieren. Dies legt nahe, dass auch innerhalb ein- und derselben Bitplane die Signifikanzen nicht gleichverteilt sind. Der Grund ist in der pyramidähnlichen Struktur des transformierten Bildes und in der Scanorder zu suchen. In den oberen Bitplanes enthalten etliche Blöcke der unteren Transformationsstufen keine signifikanten Koeffizienten. Als Verbesserung könnte man deshalb die Insignifikanz solcher Blöcke separat kodieren und sich somit etliche Bits in der Significance-Map sparen. Genau das macht Codec 2, der nun vorgestellt werden soll.

5.2.2 Codec 2

Abbildung 5.1 stellt die betrachteten Blöcke bei `lena.pgm` in Bitplane 9, 8 und 7 dar. In Bitplane 8 werden alle Blöcke ausser die der untersten Transformationstufe berücksichtigt. In Bitplane 7 werden zusätzlich noch Koeffizienten in LH_1 signifikant und erst ab Bitplane 5 enthalten alle Blöcke signifikante Koeffizienten. Somit werden vor allem in den oberen Bitplanes sehr viele Koeffizienten gleich von vornherein von der Kodierung ausgeschlossen. Tabelle 5.4 zeigt, dass die Grösse der unkomprimierten Significance-Map dadurch halbiert werden konnte. Dennoch ist die komprimierte Significance-Map nur geringfügig kleiner geworden: von 642.217 auf 639.699 Bits. Die adaptive Arithmetische Kodierung hatte also die grossen Bereiche mit insignifikanten Koeffizienten bereits effizient kodiert, deshalb fallen auch die Verbesserungen gegenüber Codec 1 nur moderat aus (vgl. Tabelle 5.2 und Tabelle 5.4).

Die Kodierung der Insignifikanz der Blöcke geschieht dabei wie folgt: zu Beginn jeder Bitplane wird für jeden Block, der bisher keine signifikanten Koeffizienten enthielt, geprüft ob er in dieser Bitplane Koeffizienten enthält die signifikant werden, das Ergebnis dieser

Bitplane	Significance-Bits	Sign-Bits	Magnitude-Bits
14	256 Bits	1 Bits	0 Bits
13	255 Bits	10 Bits	1 Bits
12	501 Bits	36 Bits	11 Bits
11	2.001 Bits	112 Bits	47 Bits
10	3.937 Bits	216 Bits	159 Bits
9	32.393 Bits	582 Bits	374 Bits
8	64.579 Bits	1.267 Bits	954 Bits
7	128.848 Bits	2.634 Bits	2.212 Bits
6	191.749 Bits	4.967 Bits	4.811 Bits
5	252.316 Bits	9.817 Bits	9.636 Bits
4	242.472 Bits	20.751 Bits	18.904 Bits
3	221.552 Bits	46.958 Bits	37.520 Bits
2	173.593 Bits	70.781 Bits	76.263 Bits
1	98.414 Bits	56.398 Bits	114.578 Bits
0	21.935 Bits	13.804 Bits	43.601 Bits
$\sum$	1.434.801 Bits	228.334 Bits	309.071 Bits
Codec 2	639.699 Bits	228.317 Bits	296.371 Bits

Tabelle 5.4: Die Anzahl unkomprimierter Bits pro Bitplane in `lena.pgm` (5/3 transformiert) bei Verwendung von Codec 2. Die letzten beiden Zeilen enthalten die aufsummierten unkomprimierten Bits, sowie die nach Einsatz der Arithmetischen Kodierung von Codec 2 benötigten Bits. Durch die separate Kodierung der Blockinsignifikanz konnte im Vergleich zu Codec 1, die Grösse der unkomprimierten Significance-Map halbiert werden.

Prüfung ist wiederum binär (enthält signifikante Koeffizienten bzw. enthält keine) und kann damit ebenfalls mit einen binären adaptiven Arithmetischen Kodierer kodiert werden. Diese Prüfung erfolgt aber nur, wenn der Block mindestens 256 Koeffizienten enthält – bei kleineren Blöcken lohnt sich der zusätzliche Overhead nicht. Bezogen auf `lena.pgm` mit einer Auflösung von 512×512 Pixeln und damit maximal 9 möglichen Transformationsstufen bedeutet dies, dass nur die Blöcke LH_i, HL_i, HH_i mit $i = 1 \ldots 5$ für die Kodierung der Blockinsignifikanz berücksichtigt werden, da sich mit jeder Transformationsstufe die Anzahl der Koeffizienten in den Blöcken viertelt und LH_5, HL_5 sowie HH_5 jeweils genau 256 Koeffizienten enthalten. Pro Bitplane fallen damit maximal 15 zusätzliche Bits an. Über alle Bitplanes werden im genannten Bild 90 unkomprimierte Bits benötigt, damit sind diese Bits in Anbetracht der Gesamtbitanzahl des Bildes zu vernachlässigen.

Vor allem in den oberen Bitplanes macht die Significance-Map den grössten Teil der komprimierten Bitmenge aus. Dies sei wiederum an `lena.pgm` (5/3 transformiert) veranschaulicht: wird bis einschliesslich Bitplane 5 kodiert, so benötigt die komprimierte Significance-Map 107.493 Bits, die Sign-Map 19.640 Bits und die Magnitude-Map 17.409 Bits. Bei den übrigen Testbildern sind diese Verhältnisse ähnlich. Um die Kodiereffizienz also weiter zu erhöhen, liesse sich mit einer verbesserten Komprimierung der Significance-Map am meisten erreichen.

	S+P	5/3	9/7
0,10 bpp	25,62 dB	25,41 dB	25,93 dB
0,25 bpp	28,66 dB	28,34 dB	29,08 dB
0,50 bpp	31,51 dB	31,23 dB	32,05 dB
0,75 bpp	33,55 dB	33,33 dB	34,17 dB
1,00 bpp	35,09 dB	34,89 dB	35,88 dB
2,00 bpp	39,91 dB	39,88 dB	41,26 dB
B bis $B-5$	0,02 bpp	0,03 bpp	0,02 bpp
B bis $B-6$	0,06 bpp	0,07 bpp	0,06 bpp
B bis $B-7$	0,15 bpp	0,17 bpp	0,15 bpp
B bis $B-8$	0,36 bpp	0,35 bpp	0,34 bpp
B bis $B-9$	0,71 bpp	0,66 bpp	0,66 bpp
B bis $B-10$	1,29 bpp	1,13 bpp	1,17 bpp
verlustfrei	4,85 bpp	4,85 bpp	-

Tabelle 5.5: Durchschnittswerte der mit Codec 2 komprimierten Testbilder.

Die Idee ist nun, mehrere Arithmetische Kontexte für die Kodierung der Significance-Map einzusetzen. Die Entscheidung, welcher Kontext für die Kodierung eines Koeffizienten verwendet werden soll, muss dabei auf Informationen basieren, die sowohl encoder- als auch decoderseitig zur Verfügung stehen. Aufgrund der Eigenschaften wavelet-transformierter Bilder ist es naheliegend die verfügbaren Informationen der benachbarten Koeffizienten zu nutzen, um einen Kontext auszuwählen und damit die Vorhersagegenauigkeit über Signifikanz bzw. Insignifikanz zu erhöhen.

5.3 Zusätzliche Kontexte für die Significance-Map

5.3.1 Auswahl benachbarter Koeffizienten

Zur Verbesserung der Kompression der Significance-Map sollen nun Koeffizienten in Abhängigkeit von benachbarten Koeffizienten kodiert werden. Benachbart heisst in diesem Zusammenhang sowohl benachbart in der gleichen als auch in der nächsthöheren Transformationsstufe desselben Subbands, so wie in Abbildung 5.2 dargestellt. Bei der Kodierung des Koeffizienten c sollen folgende Koeffizienten genutzt werden, um eine verbesserte Vorhersage der Signifikanz zu ermöglichen:

$n_i, i = 0 \ldots 11$ - benachbart in der gleichen Transformationsstufe
$e_i, i = 0 \ldots 4$ - benachbart in der nächsthöheren Transformationsstufe

oder als 17-Tupel ausgedrückt:

$$t = (e_0, e_1, e_2, e_3, e_4, n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11})$$

Es werden damit also sogar noch mehr Koeffizienten berücksichtigt als in ECECOW.

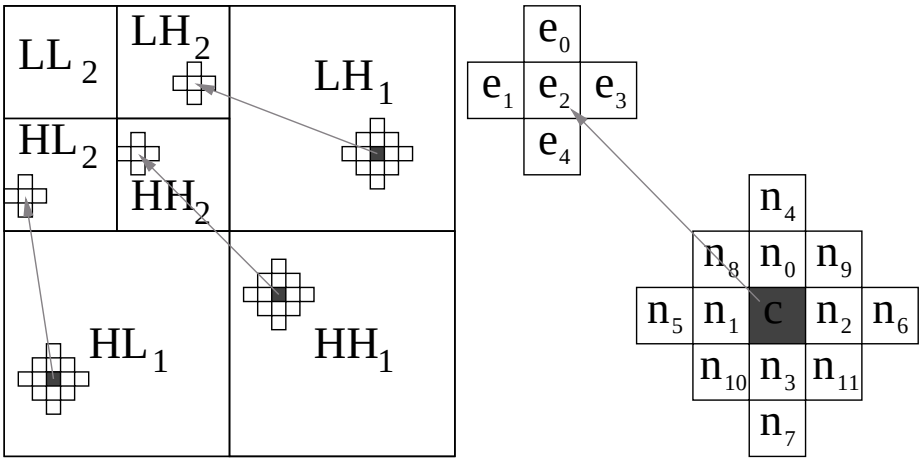


Abbildung 5.2: Zusammensetzung der 17-Tupel: um die Vorhersagegenauigkeit der Signifikanz des gerade betrachteten Koeffizienten c zu erhöhen, werden die auf Encoder- und Decoderseite verfügbaren Informationen über benachbarte Koeffizienten genutzt. Dabei werden sowohl Koeffizienten aus der gleichen Transformationsstufe ($n_i, i = 0 \ldots 11$), als auch der nächsthöheren Transformationsstufe ($e_i, i = 0 \ldots 4$) herangezogen.

Wie bereits erwähnt können nur Informationen verwendet werden, die sowohl auf En- als auch auf Decoderseite verfügbar sind. Wird gerade Bitplane i durchlaufen, so können aufgrund der verwendeten Scanorder von einigen Koeffizienten alle Bits bis Bitplane i verwendet werden, von anderen hingegen nur Bits bis Bitplane $i + 1$:

1. bis Bitplane i: e_j mit $j = 0 \ldots 4$ sowie n_j mit $j \in \{0, 1, 4, 5, 8, 9\}$, da diese Koeffizienten bereits in Bitplane i durchlaufen wurden.

2. bis Bitplane $i + 1$: n_j mit $j \in \{2, 3, 6, 7, 10, 11\}$, da diese Koeffizienten erst noch in Bitplane i durchlaufen werden.

Ein Tupel t kann mehrfach in einem Bild auftreten und im Zusammenhang mit t kann der gerade betrachtete Koeffizient c signifikant werden, oder insignifikant bleiben. Sei $ZERO(t)$ die Anzahl der Koeffizienten, die beim Auftreten von t insignifikant bleiben und $ONE(t)$ die Zahl derer, die signifikant werden. Ferner sei

$$p_t = \frac{ONE(t)}{ZERO(t) + ONE(t)}$$

die Signifikanzwahrscheinlichkeit des Tupels t.

In Bezug auf eine Menge von Tupeln T seien diese Funktionen wie folgt definiert:

$$\begin{aligned} ZERO(T) &= \sum_{\forall t \in T} ZERO(t) \\ ONE(T) &= \sum_{\forall t \in T} ONE(t) \\ p_T &= \frac{\sum_{\forall t \in T} ONE(t)}{\sum_{\forall t \in T} [ZERO(t) + ONE(t)]} \end{aligned}$$

Der Einfachheit halber, sei zusätzlich noch folgende Schreibweise zur Definition von Tupelmengen vereinbart:

$$T[e_2 = 0] = \text{(x,x,0,x,x,x,x,x,x,x,x,x,x,x,x,x)}$$

formal bedeutet dies:

$$t \in T[e_2 = 0] \quad \Leftrightarrow \quad t = (e_0, e_1, 0, e_3, e_4, n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_9, n_{10}, n_{11})$$

mit $e_0, e_1, e_3, e_4, n_i, i = 0 \ldots 11$ beliebig. Diese Schreibweise gilt analog auch für die übrigen e- und n-Koeffizienten.

5.3.2 Restriktionen bei der Implementierung der 17-Tupel

Um die Verarbeitung der Tupel einfach zu gestalten, wurde für jeden benachbarten Koeffizienten eine feste Bitanzahl reserviert: 3 Bit für n-Koeffizienten bzw. 4 Bit für e-Koeffizienten. Damit lässt sich ein 17-Tupel mit 56 Bit darstellen und passt somit in einen 64 Bit Integerwert.

Es werden dabei nur die relevanten Bits ab Bitplane i abgespeichert, ist der Wert zu gross wird er auf 7 für n-Koeffizienten bzw. 15 für e-Koeffizienten gesetzt. Um zu beurteilen, ob die 56 Bit für die Speicherung des 17-Tupel ausreichend sind, wurde geprüft wie häufig die Werte 7 bzw. 15 für jeweils einen n- bzw. e-Koeffizienten vorkommen. In Tabelle 5.6 ist die Verteilung für das LH-Subband von `lena.pgm` (5/3 transformiert) jeweils für e_2 und n_0 aufgeführt, für die übrigen 11 n- bzw. 4 e-Koeffizienten sieht die Verteilung ähnlich aus, dies gilt ebenso für die anderen Subbänder und die übrigen Testbilder.

Es ist ersichtlich, dass insbesondere bei n-Koeffizienten kaum grössere Werte existieren, die Restriktion auf 7 als grösstmöglichen Wert wirkt sich deshalb kaum aus. Die Verteilung der e-Koeffizienten ist etwas gestreckter und der Sprung von der 14 zur 15 ist vom Verhältnis her grösser als der Sprung von der 6 zur 7 bei den n-Koeffizienten. Es würde jedoch kaum Vorteile bringen, wenn man den grösstmöglichen Wert auf z.B. 31 hinaufsetzt, da sich die Signifikanzwahrscheinlichkeien für c bei grösseren Werten kaum unterscheiden.

Aus der Verteilung kann man bereits jetzt schlussfolgern, dass die weitaus meisten Koeffizienten spätestens dann signifikant werden, wenn auch benachbarte Koeffizienten in einer der vorigen Bitplanes signifikant geworden sind.

T	$ZERO(T)$	$ONE(T)$	p_T
$T[e_2 = 0]$ =(x,x,0,x,x,x,x,x,x,x,x,x,x,x,x,x)	307.495	12.891	4,02%
$T[e_2 = 1]$ =(x,x,1,x,x,x,x,x,x,x,x,x,x,x,x,x)	60.243	12.062	16,68%
$T[e_2 = 2]$ =(x,x,2,x,x,x,x,x,x,x,x,x,x,x,x,x)	27.932	13.656	32,34%
$T[e_2 = 3]$ =(x,x,3,x,x,x,x,x,x,x,x,x,x,x,x,x)	13.263	6.020	31,22%
$T[e_2 = 4]$ =(x,x,4,x,x,x,x,x,x,x,x,x,x,x,x,x)	9.663	7.764	44,55%
$T[e_2 = 5]$ =(x,x,5,x,x,x,x,x,x,x,x,x,x,x,x,x)	4.773	2.965	38,32%
$T[e_2 = 6]$ =(x,x,6,x,x,x,x,x,x,x,x,x,x,x,x,x)	4.560	4.558	49,99%
$T[e_2 = 7]$ =(x,x,7,x,x,x,x,x,x,x,x,x,x,x,x,x)	2.448	1.697	40,94%
$T[e_2 = 8]$ =(x,x,8,x,x,x,x,x,x,x,x,x,x,x,x,x)	2.590	2.797	51,92%
$T[e_2 = 9]$ =(x,x,9,x,x,x,x,x,x,x,x,x,x,x,x,x)	1.279	987	43,56%
$T[e_2 = 10]$ =(x,x,10,x,x,x,x,x,x,x,x,x,x,x,x,x)	1.592	1.776	52,73%
$T[e_2 = 11]$ =(x,x,11,x,x,x,x,x,x,x,x,x,x,x,x,x)	749	656	46,69%
$T[e_2 = 12]$ =(x,x,12,x,x,x,x,x,x,x,x,x,x,x,x,x)	942	1.258	57,18%
$T[e_2 = 13]$ =(x,x,13,x,x,x,x,x,x,x,x,x,x,x,x,x)	597	479	44,52%
$T[e_2 = 14]$ =(x,x,14,x,x,x,x,x,x,x,x,x,x,x,x,x)	671	936	58,25%
$T[e_2 = 15]$ =(x,x,15,x,x,x,x,x,x,x,x,x,x,x,x,x)	4.794	6.057	55,82%
$T[n_0 = 0]$ =(x,x,x,x,x,0,x,x,x,x,x,x,x,x,x,x)	394.862	31.740	7,44%
$T[n_0 = 1]$ =(x,x,x,x,x,1,x,x,x,x,x,x,x,x,x,x)	31.234	22.504	41,88%
$T[n_0 = 2]$ =(x,x,x,x,x,2,x,x,x,x,x,x,x,x,x,x)	9.070	10.184	52,59%
$T[n_0 = 3]$ =(x,x,x,x,x,3,x,x,x,x,x,x,x,x,x,x)	3.716	5.028	57,50%
$T[n_0 = 4]$ =(x,x,x,x,x,4,x,x,x,x,x,x,x,x,x,x)	1.797	2.587	59,01%
$T[n_0 = 5]$ =(x,x,x,x,x,5,x,x,x,x,x,x,x,x,x,x)	1.003	1.513	60,14%
$T[n_0 = 6]$ =(x,x,x,x,x,6,x,x,x,x,x,x,x,x,x,x)	590	884	59,97%
$T[n_0 = 7]$ =(x,x,x,x,x,7,x,x,x,x,x,x,x,x,x,x)	1.319	2.119	61,63%

Tabelle 5.6: Auftrittshäufigkeit von Tupeln mit $e_2 = 0 \ldots 15$ bzw. $n_0 = 0 \ldots 7$ am Beispiel von `lena.pgm` (5/3 transformiert), die Beschränkung auf 15 bzw. 7 als höchstmöglichen Wert, führt zwar zu einer Stauchung der Verteilung am oberen Ende, jedoch erscheint das Hinaufsetzen des Maximums wenig vorteilhaft, da sich die Signifikanzwahrscheinlichkeiten für grosse Werte kaum unterscheiden.

5.3.3 Analyse und Klassifizierung der 17-Tupel

Bei der folgenden Untersuchung wird wie zuvor auch nach dem bereits bekannten Schema durch die Bitplane gelaufen. Es wird für jeden Koeffizient das zugehörige 17-Tupel ermittelt und es wird vermerkt ob der Koeffizient bei diesem Tupel signifikant wurde oder insigifikant blieb. Bei dieser Untersuchung werden die LH-, HL- und HH-Subbänder getrennt betrachtet, LL kann aufgrund seiner Grösse vernachlässigt werden. Denn bereits nach 5 Transformationsstufen machen die Koeffizienten in LL gerade noch 0,1% aller Koeffizienten im transformierten Bild aus:

$$\frac{1}{2^5} \cdot \frac{1}{2^5} = \frac{1}{1024}$$

t	$ZERO(t)$	$ONE(t)$	p_t
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)	207.434	194	0,09%
(0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0)	7.463	70	0,93%
(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0)	9.704	94	0,96%
(0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0)	5.896	82	1,37%
(0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)	9.455	90	0,94%
(1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)	7.065	85	1,19%

Tabelle 5.7: Die am häufigsten auftretenden 17-Tupel. Die genaue Auftrittshäufigkeit am Beispiel von `lena.pgm` (5/3 transformiert) für das LH-Subband.

Darüber hinaus wird bei allen hier verwendeten Testbildern die Wavelet-Transformation neun mal angewendet, so dass sich sogar 9 Transformationsstufen ergeben und der Anteil der LL-Koeffizienten am Gesamtbild damit noch geringer ausfällt.

Zunächst wird untersucht, ob es Tupel gibt die häufiger auftreten als andere. Es zeigt sich, dass das Tupel $(0,\ldots,0)$ von allen vorkommenden Tupeln am weitaus häufigsten auftritt. Des Weiteren ist die Signifikanzwahrscheinlichkeit für Koeffizienten, die solch eine Nachbarschaft haben, sehr gering: im Bereich 0,1% - 0,5%. Das heisst mit mehr als 99% Wahrscheinlichkeit bleiben solche Koeffizienten insignifikant. Am zweithäufigsten treten die Tupel $(0,0,0,0,1,0,\ldots,0)$, $(0,0,0,1,0,\ldots,0)$, $(0,0,1,0,\ldots,0)$, $(0,1,0,\ldots,0)$ und $(1,0,\ldots,0)$ auf. Also jene Tupel bei denen ein e-Koeffizient in der gerade betrachteten Bitplane signifikant geworden ist, auch hier ist die Wahrscheinlichkeit für c insignifikant zu bleiben hoch, nämlich grösser 98%. Für `lena.pgm` sind die konkreten Zahlen in Tabelle 5.7 dargestellt.

Bei genauerer Betrachtung ist dieses Resultat nicht weiter verwunderlich, da vor allem in den oberen Bitplanes sehr viele Koeffizienten insignifikant bleiben. Je mehr Bitplanes durchlaufen werden, desto mehr Koeffizienten sind bereits signifikant, das heisst jedoch auch, dass es kaum noch Koeffizienten mit einer weitgehend insignifikanten Nachbarschaft gibt. Es zeigt sich deshalb, dass wenn man nur die unteren Bitplanes betrachtet, es keine Tupel mehr gibt, die deutlich häufiger auftreten als andere. Allgemein kann man sagen, dass sehr häufig auftretende Tupel viele $e_i = 0$ und $n_j = 0$ enthalten, solche Tupel werden folgend auch als *active* bezeichnet. Insgesamt betrachtet treten von den vorkommenden Tupeln nur wenige sehr häufig auf, die allermeisten Tupel kommen nur einige Male vor.

5.3.4 Codec 3

In Codec 3 sollen nun für die häufig auftretenden Tupel gesonderte Kontexte verwendet werden. Die Significance-Map wird jetzt also nicht mehr mit nur einem Arithmetischen Kontext kodiert, sondern mit mehreren. Da die Subbänder jetzt getrennt betrachtet werden erhöht sich die Kontextanzahl zusätzlich. Zur Komprimierung der Significance-Map

	S+P	5/3	9/7
0,10 bpp	26,32 dB	26,02 dB	26,67 dB
0,25 bpp	29,45 dB	29,14 dB	29,91 dB
0,50 bpp	32,36 dB	32,06 dB	32,96 dB
0,75 bpp	34,44 dB	34,28 dB	35,21 dB
1,00 bpp	35,96 dB	35,74 dB	36,86 dB
2,00 bpp	40,88 dB	40,79 dB	42,39 dB
B bis $B-5$	0,01 bpp	0,02 bpp	0,01 bpp
B bis $B-6$	0,05 bpp	0,06 bpp	0,05 bpp
B bis $B-7$	0,13 bpp	0,14 bpp	0,12 bpp
B bis $B-8$	0,30 bpp	0,30 bpp	0,28 bpp
B bis $B-9$	0,61 bpp	0,57 bpp	0,57 bpp
B bis $B-10$	1,14 bpp	1,00 bpp	1,04 bpp
verlustfrei	4,65 bpp	4,66 bpp	-

Tabelle 5.8: Durchschnittswerte der mit Codec 3 komprimierten Testbilder.

werden in Codec 3 alle auftretenden Tupel in folgende fünf Kontexte eingeteilt:

$$K_3(t) = \begin{cases} 0 & \text{wenn} \quad E(t) = 0 \wedge N(t) = 0 \\ 1 & \text{wenn} \quad E(t) = 1 \wedge N(t) = 0 \\ 2 & \text{wenn} \quad E(t) \leq 5 \wedge N(t) = 0 \\ 3 & \text{wenn} \quad E(t) = 0 \wedge N(t) = 1 \\ 4 & \text{sonst} \end{cases} \tag{5.1}$$

$$\text{mit} \quad E(t) = \sum_{i=0}^{4} e_i$$

$$\text{und} \quad N(t) = \sum_{i=0}^{11} n_i$$

Eine solche Einteilung wird auch häufig als *Kontext-Quantisierung* bezeichnet. In diesem Kapitel wird aber weiterhin der Begriff Quantisierung verwendet, wobei jedoch immer die Quantisierung in Bezug auf Kontexte gemeint ist. Die Kodierung der Sign- und Magnitude-Bits, sowie die der Blockinsignifikanz bleibt gegenüber Codec 2 unverändert. Verglichen zu Codec 2 ergibt sich dadurch schon eine deutliche Verbesserung (vgl. Tabelle 5.5 und Tabelle 5.8).

Noch eine Anmerkung zur Bezeichung: die Menge $ACTIVE$ mit

$$t \in ACTIVE \Leftrightarrow K_3(t) \leq 3$$

umfasst genau jene Tupel, die in Codec 3 in den Kontexten $0 \ldots 3$ liegen, also die häufigsten *active*-Tupel. Wenn also von Tupeln aus $ACTIVE$ die Rede ist, so sind immer genau die Tupel bezogen auf die Quantisierung K_3 gemeint. Der Begriff *active*-Tupel hingegen ist

allgemeiner und weniger präzise, denn es gibt durchaus noch mehr *active*-Tupel als in *ACTIVE* enthalten sind, diese unterscheiden sich aber von Bild zu Bild oder sind vom Umfang her zu gering, als dass es Sinn machen würde sie gesondert zu berücksichtigen.

5.3.5 Beurteilung der Quantisierung von Codec 3

Um nun beurteilen zu können, ob diese Kontextaufteilung sinnvoll ist sollen drei Schranken verwendet werden. Sei ALL die Menge aller vorkommenden 17-Tupel, ferner seien die Mengen $FULL$ und $SPARSE$ eine Partitionierung von ALL, mit

$$\begin{aligned} t \in FULL &\Leftrightarrow ZERO(t) > 0 \wedge ONE(t) > 0 \\ t \in SPARSE &\Leftrightarrow (ZERO(t) = 0 \wedge ONE(t) > 0) \vee (ZERO(t) > 0 \wedge ONE(t) = 0) \end{aligned} \quad .$$

Die angesprochenen drei Schranken (Trivial Bounds) ergeben sich nun wie folgt:

$$\begin{aligned} TB_1 &= \sum_{\forall t \in FULL} H(p_t) \cdot [ZERO(t) + ONE(t)] \\ TB_2 &= H(p_{FULL}) \cdot [ZERO(FULL) + ONE(FULL)] \\ TB_3 &= H(p_{ALL}) \cdot [ZERO(ALL) + ONE(ALL)] \end{aligned} \quad .$$

TB_1 stellt dabei eine untere Schranke und TB_3 eine obere Schranke für die Bitmenge dar, die von einem Arithmetischen Kodierer benötigt würde, wenn die vorkommenden Tupel zur Bildung von Kontexten benutzt würden. Bei TB_1 wird davon ausgegangen, dass alle Tupel in *SPARSE* kein einziges Bit zur Kodierung benötigen, da im Zusammenhang mit diesen Tupeln immer nur Nullen oder Einsen auftreten, zusätzlich wird davon ausgegangen, dass alle Tupel in $FULL$ mit der jeweiligen Signifikanzwahrscheinlichkeit kodiert werden. TB_3 hingegen berechnet die durchschnittliche Signifikanzwahrscheinlichkeit aller Tupel. Das Ergebnis entspricht also der Bitanzahl, die ein statischer Arithmetischer Kodierer benötigen würde. Mit TB_2 lässt sich in gewisser Weise die Varianz der $FULL$-Tupel eines Kontextes bestimmen, denn die Berechnung erfolgt wie in TB_3, jedoch werden nur Tupel in $FULL$ berücksichtigt. Je grösser der Unterschied zwischen TB_1 und TB_2 desto mehr variieren die Signifikanzwahrscheinlichkeiten der einzelnen $FULL$-Tupel in einem Kontext.

Wie bereits erwähnt treten die *ACTIVE*-Tupel nur in den oberen Bitplanes auf, deshalb wurden in Tabelle 5.9 auch nur Tupel bis einschliesslich Bitplane 5 berücksichtigt. Wären alle Bitplanes durchlaufen worden, so hätte sich in den Kontexten $0 \ldots 3$ nicht mehr viel geändert, hingegen würde Kontext 4 wesentlich mehr Tupel enthalten.

Die unteren und oberen Schranken der Kontexte 0, 1 und 3 unterscheiden sich kaum, bei Kontext 2 hingegen ist die Differenz schon etwas grösser. Im wesentlichen liegt das an den variierenden Signifikanzwahrscheinlichkeiten der einzelnen enthaltenen $FULL$-Tupel, denn die Differenz zwischen TB_1 und TB_2 ist grösser als die zwischen TB_2 und TB_3. Mit einer weiteren Aufteilung von Kontext 2 könnte man zwar diesem Umstand Rechnung tragen, jedoch liessen sich damit experimentell kaum Bits sparen, so dass darauf verzichtet wurde. Am auffälligsten ist jedoch, dass bei Kontext 4 die untere und obere Schranke sehr weit auseinander liegen. Eine sinnvolle Abschätzung welche Reduzierungen durch weitere

K_3	$\|T\|$	$ZERO(T)$	$ONE(T)$	p_T	TB_1	TB_2	TB_3	Codec 3
0	1	190.573	59	0,03%	773	773	773	764
1	5	26.534	98	0,37%	931	934	934	946
2	244	34.497	498	1,42%	3.408	3.713	3.769	3.769
3	6	493	21	4,09%	121	127	127	134
4	31.982	38.107	8.924	18,97%	3.726	4.543	32.966	32.915
$\sum$	32.238	290.204	9.600	-	8.959	10.090	38.569	38.528

Tabelle 5.9: Am Beispiel des LH-Subbands (bis einschliesslich Bitplane 5) von `lena.pgm` (5/3 transformiert) einige Details zu den zusätzlichen Kontexten in Codec 3. Die Kontexte $K_3 = 0 \ldots 3$ enthalten die quantisierten *active*-Tupel. Der Abstand zwischen unterer und oberer Schranke für diese Kontexte ist verhältnismässig gering, somit ist die Quantisierung nahe am Optimum.

Quantisierungen bei diesen, hauptsächlich in *SPARSE* liegenden, Tupeln möglich sind, lässt sich somit mit Hilfe der verwendeten Schranken leider nicht machen. Da aber der Grossteil der verbrauchten Bits in diesen Kontext fällt und die $ACTIVE$-Tupel schon nahe am Optimum quantisiert sind, ist eine feinere Aufteilung dieses Kontextes die einzige Möglichkeit, um die Kompression der Significance-Map nochmals zu verbessern. Die Schlussfolgerungen, die hier am Beispiel von `lena.pgm` gemacht wurden, gelten analog auch für die übrigen Testbilder.

In Codec 3 werden für die einzelnen Subbänder getrennte Kontexte für die Kodierung der Significance-Map verwendet, die auftretenden Signifikanzwahrscheinlichkeiten unterscheiden sich dabei nur geringfügig, so dass es auch möglich wäre alle Subbänder dieselben Kontexte nutzen zu lassen. Es ist jedoch anzumerken, dass das HH-Subband bei der Wavelet-Transformation einen geringeren Wichtungsfaktor hat, als die LH- und HL-Subbänder, so dass das HH-Subband mit der Signifikanzwahrscheinlichkeit etwas "hinterhinkt". Obwohl bei der verwendeten Testbildmenge eine Zusammenlegung der Kontexte der Subbänder keine entscheidend nachteiligen Veränderungen der Kodiereffizienz bewirkt hat, wird darauf verzichtet, da bei Verwendung von anderem Bildmaterial durchaus grössere Unterschiede in den Signifikanzwahrscheinlichkeiten der Subbänder auftreten könnten und sich eine Zusammenlegung damit negativ auswirken würde.

5.3.6 Quantisierung der übrigen Tupel

Nachdem die meisten *active*-Tupel quantisiert wurden, soll nun als erstes geprüft werden, ob sich in den verbleibenden Tupeln einige der e- bzw. n-Koeffizienten gut zur Vorhersage der Signifikanz eignen. In Tabelle 5.6 lässt sich bereits erkennen, dass es Sinn machen könnte den Wert eines e- bzw. n-Koeffizienten als Kontext für die verbleibenden Tupel zu nehmen. Dazu soll die Quantisierung aus Gleichung (5.1) jetzt erweitert werden,

um folgende 17 Quantisierungen zu testen:

$$K_i^{(test1)}(t) = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4+j & \text{wenn} \quad t \in (T[e_i = j] \setminus ACTIVE) \qquad j = 0 \ldots 15 \end{cases} \quad \text{für} \quad i = 0 \ldots 4$$

$$K_{i+5}^{(test1)}(t) = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4+j & \text{wenn} \quad t \in (T[n_i = j] \setminus ACTIVE) \qquad j = 0 \ldots 7 \end{cases} \quad \text{für} \quad i = 0 \ldots 11$$

Weil sich für die Kontexte $0 \ldots 3$ nichts ändert interessiert eigentlich nur, wieviel Bits in den Kontexten ≥ 4 benötigt werden. Da jede der 17 Quantisierungen genau die gleiche Datenmenge kodiert, wird die kleinste benötigte Bitmenge gesucht.

$$S_i = \frac{1}{\sum_{j=0}^{15} ONE(T_i)} \cdot \sum_{j=0}^{15} H(p_{T_i}) \cdot [ZERO(T_i) + ONE(T_i)] \qquad (5.2)$$
$$\text{mit} \quad T_i = T[e_i = j] \setminus ACTIVE \qquad \text{für} \quad i = 0 \ldots 4$$

$$S_{i+5} = \frac{1}{\sum_{j=0}^{7} ONE(T_{i+5})} \cdot \sum_{j=0}^{7} H(p_{T_{i+5}}) \cdot [ZERO(T_{i+5}) + ONE(T_{i+5})] \qquad (5.3)$$
$$\text{mit} \quad T_{i+5} = T[n_i = j] \setminus ACTIVE \qquad \text{für} \quad i = 0 \ldots 11$$

Der Ausdruck hinter der Summe berechnet wieviel Bits ein statischer Arithmetischer Kodierer benötigen würde, wenn er genau einen der benachbarten Koeffizienten des 17-Tupels zur Kontextbildung heranziehen würde. Da sich damit aber schlecht ein Durchschnitt über mehrere Bilder errechnen lässt, wird dies noch durch die Anzahl der signifikant gewordenen Koeffizienten dividiert. Man erhält dadurch ein Ergebnis in Bits per Significance. Die Berechnung erfolgt wiederum für jedes Subband getrennt.

In Tabelle 5.10 sind die Durchschnittswerte der Gleichungen (5.2) und (5.3) von allen Testbildern dargestellt. Es ist zu erkennen, dass nicht alle benachbarten Koeffizienten gleichgut zur Quantisierung der übrigen Tupel geeignet sind. Dabei spielt auch das Subband eine Rolle, aus welchem das Tupel stammt. Während im LH- und HH-Subband die Werte für S_5 die kleinsten sind, hat im HL-Subband S_6 den kleinsten Wert. Das heisst für Tupel aus LH und HH eignet sich n_0 am besten zur Vorhersage der Signifikanz von c und für Tupel aus HL n_1 – wenn man sich auf einen benachbarten Koeffizienten beschränken würde. Mit der ermittelten Reihenfolge lässt sich ausserdem die Auswahl der Koeffizienten in ECECOW nachvollziehen.

Zum Vergleich: in Codec 3 werden alle verbleibenden Tupel in Kontext 4 quantisiert und benötigen dort bei `lena.pgm` im LH-Subband 32.915 Bits (siehe Tabelle 5.9). Durch

LH-Subband					
i	S_i	i	S_i	i	S_i
5	3,32 bps	13	3,47 bps	16	3,50 bps
2	3,42 bps	14	3,47 bps	3	3,51 bps
9	3,43 bps	7	3,48 bps	10	3,51 bps
4	3,45 bps	0	3,49 bps	1	3,52 bps
6	3,45 bps	12	3,49 bps	11	3,52 bps
8	3,45 bps	15	3,50 bps		

HL-Subband					
i	S_i	i	S_i	i	S_i
6	3,36 bps	3	3,49 bps	9	3,53 bps
2	3,45 bps	13	3,50 bps	16	3,53 bps
5	3,45 bps	8	3,51 bps	0	3,54 bps
10	3,47 bps	1	3,52 bps	4	3,54 bps
7	3,48 bps	11	3,52 bps	12	3,55 bps
14	3,48 bps	15	3,52 bps		

HH-Subband					
i	S_i	i	S_i	i	S_i
5	3,61 bps	7	3,70 bps	4	3,73 bps
6	3,63 bps	9	3,70 bps	12	3,73 bps
2	3,65 bps	10	3,71 bps	1	3,74 bps
14	3,66 bps	15	3,71 bps	3	3,74 bps
13	3,67 bps	16	3,72 bps	11	3,74 bps
8	3,69 bps	0	3,73 bps		

Tabelle 5.10: Durchschnitt aller Testbilder (5/3 transformiert) gemäss den Gleichungen (5.2) und (5.3). Die obere Tabelle enthält die Bits per Significance für das LH-Subband, die mittlere für das HL- Subband und die untere für das HH-Subband (nach bps sortiert). Es wurde bis Bitplane 5 kodiert. Es fällt auf, dass die Werte für S_5 immer zu den kleinsten gehören. Das heisst n_0 eignet sich gut, um daraus weitere Kontexte für die übrigen Tupel abzuleiten.

den zusätzlichen Einsatz von Kontexten wie in $K_5^{(test1)}$ lassen sich dieselben Bits der Significance-Map mit 30.539 Bits kodieren, also knapp 2.400 Bits weniger.

Zur weiteren Verbesserung wäre es nun naheliegend nicht nur einen benachbarten Koeffizienten zu nehmen sondern die jeweils besten eines Subbands zu kombinieren. Dabei muss jedoch berücksichtigt werden, dass die verwendete Kontextanzahl nicht zu gross wird, da sonst in jedem Kontext nur wenige Bits kodiert werden und der adaptive Arithmetische Kodierer somit keine Möglichkeit hat sich sinnvoll an die Signifikanzwahrscheinlichkeit anzupassen. Dadurch würde der Overhead pro Kontext den möglichen Gewinn überwiegen. Es ist also immer ein Tradeoff zwischen einer möglichst feinen Kontextaufspaltung, um sich an die Charakteristik des Bildmaterial bestmöglich anpassen zu können, einerseits und

$K_5^{(test1)}$	T	$ZERO(T)$	$ONE(T)$	p_T
4	$T[n_0 = 0] \setminus ACTIVE$	31.257	4.646	12,9%
5	$T[n_0 = 1] \setminus ACTIVE$	5.024	2.674	34,7%
6	$T[n_0 = 2] \setminus ACTIVE$	1.030	850	45,2%
7	$T[n_0 = 3] \setminus ACTIVE$	357	303	45,9%
8	$T[n_0 = 4] \setminus ACTIVE$	157	158	50,2%
9	$T[n_0 = 5] \setminus ACTIVE$	96	87	47,5%
10	$T[n_0 = 6] \setminus ACTIVE$	59	55	48,2%
11	$T[n_0 = 7] \setminus ACTIVE$	127	151	54,3%

Tabelle 5.11: Verteilung der übrigen Tupel bezüglich n_0 bei `lena.pgm` (5/3 transformiert) bis Bitplane 5.

einer nicht zu hohen Kontextanzahl, um dem adaptiven Arithmetischen Kodierer genau diese Anpassung auch zu ermöglichen, andererseits.

Bereits in Tabelle 5.6 konnte man erkennen, dass es eine Korrelation zwischen benachbarten Koeffizienten und dem Signifikantwerden von c gibt, jedoch wurden dabei alle Tupel, also auch jene aus $ACTIVE$ mit berücksichtigt. In Tabelle 5.11 sind hingegen diese Tupel nicht mehr enthalten. Es ist ersichtlich, dass die meisten Tupel in $T[n_0 = 0]$ und $T[n_0 = 1]$ liegen. Anstatt nun alle 8 möglichen Werte der n-Koeffizienten zuzulassen reicht es sogar aus, sich auf zwei Werte zu beschränken, nämlich 0 und ≥ 1, ohne dabei merklich schlechter beim Kompressionsergebnis zu werden. Dies ist so ohne weiteres bei den e-Koeffizienten nicht möglich, eine Einschränkung der zulässigen Werte wirkt sich deutlicher aus.

Durch die Beschränkung auf 0 und ≥ 1 lassen sich nun einige n-Koeffizienten kombinieren. Bei vier solcher Koeffizienten bleibt man noch immer bei einer überschaubaren Kontextanzahl von 16. Es sollen vier Quantisierungen verglichen werden. Die erste $K_1^{(test2)}$ verwendet die direkt horizontal und vertikal benachbarten Koeffizienten n_0, n_1, n_2, n_3. Die übrigen drei Quantisierungen nutzen jeweils die n-Koeffizienten der LH-, HL- und HH-Subbänder mit den niedrigsten bps-Werten aus Tabelle 5.10.

$$
\begin{aligned}
K_1^{(test2)}(t) &= \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + F_4(n_0, n_1, n_2, n_3) & \text{sonst} \end{cases} \\
K_2^{(test2)}(t) &= \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + F_4(n_0, n_1, n_3, n_4) & \text{sonst} \end{cases} \\
K_3^{(test2)}(t) &= \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + F_4(n_0, n_1, n_2, n_5) & \text{sonst} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
K_4^{(test2)}(t) &= \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + F_4(n_0, n_1, n_8, n_9) & \text{sonst} \end{cases} \\
\text{mit} \quad F_1(x) &= \begin{cases} 0 & \text{wenn} \quad x = 0 \\ 1 & \text{wenn} \quad x \geq 1 \end{cases} \\
\text{und} \quad F_4(x_0, x_1, x_2, x_3) &= 8 \cdot F_1(x_0) + 4 \cdot F_1(x_1) + 2 \cdot F_1(x_2) + F_1(x_3)
\end{aligned}
$$

Mit diesen Quantisierungen ergeben sich für die einzelnen Subbänder aller Testbilder im Durchschnitt folgende Werte.

	S_{LH}	S_{HL}	S_{HH}
$K_1^{(test2)}$	3,19 bps	3,22 bps	3,44 bps
$K_2^{(test2)}$	3,19 bps	3,26 bps	3,46 bps
$K_3^{(test2)}$	3,24 bps	3,22 bps	3,46 bps
$K_4^{(test2)}$	3,23 bps	3,26 bps	3,44 bps
bester Wert Tabelle 5.10	3,32 bps	3,36 bps	3,61 bps

Zum Vergleich sind auch die jeweils besten Werte aus Tabelle 5.10 mit aufgeführt, wobei S_{LH}, S_{HL} und S_{HH} in ähnlicher Weise errechnet werden, wie die $S_i, i = 0 \ldots 16$ zuvor, bloss das jetzt die Kontextaufteilung durch $K_i^{(test2)}, i = 1, 2, 3, 4$ berücksichtigt wird. Die vier Quantisierungen sind in allen Subbändern besser als die Vergleichswerte aus Tabelle 5.10 – das ist wenig überraschend. Unerwartet hingegen ist, dass $K_1^{(test2)}$ am besten abschneidet, eigentlich hätte man erwarten können, dass die jeweils besten n-Koeffizienten eines Subbands auch für das jeweilige Subband das beste Ergebnis liefern, diese Ergebnisse werden aber auch durch $K_1^{(test2)}$ erreicht.

Zu berücksichtigen ist dabei jedoch, dass zuvor nur die Signifikanzwahrscheinlichkeiten in Abhängigkeit einzelner benachbarter Koeffizienten betrachtet wurden, jetzt sind sie in Abhängigkeit von mehreren Koeffizienten betrachtet worden und hierbei spielen die abhängigen Wahrscheinlichkeiten zwischen den Koeffizienten eine Rolle, die bei der zuvor erfolgten Betrachtung unberücksichtigt blieben. Insgesamt erscheint es daher plausibel, dass die horizontal und vertikal direkt benachbarten Koeffizienten am besten die Signifikanz von c vorhersagen können.

In $K_1^{(test2)}$ lässt sich die Kontextanzahl nochmals nahezu halbieren, indem davon ausgegangen wird, dass die beiden horizontalen und vertikalen Koeffizienten jeweils "gleichberechtigt" sind, aufgrund der Scanorder trifft das zwar nicht vollkommen zu, aber dennoch näherungsweise. Statt also für jeden n-Koeffizienten ein Bit zu reservieren, werden jetzt $F_1(n_0)$ und $F_1(n_3)$ bzw. $F_1(n_1)$ und $F_1(n_2)$ aufaddiert, dadurch reduziert sich die Kontextanzahl für die Tupel $t \notin ACTIVE$ von 16 auf 9.

$$
\begin{aligned}
K_1^{(test3)}(t) &= \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + 3 \cdot F_2(n_0, n_3) + F_2(n_1, n_2) & \text{sonst} \end{cases} \\
\text{mit} \quad F_2(x_0, x_1) &= F_1(x_0) + F_1(x_1)
\end{aligned}
$$

Die Ergebnisse haben sich im Vergleich zu $K_1^{(test2)}$ nur im LH-Subband geringfügig verschlechtert (siehe Tabelle 5.12). Vier dieser 9 Kontexte enthalten noch immer recht viele

Bits, so dass sich für sie eine weitere Aufspaltung lohnen könnte. Dabei handelt es sich um die Kontexte mit $F_2(n_0, n_3) \leq 1$ und $F_2(n_1, n_2) \leq 1$. Die folgenden erweiterten Quantisierungen sollen deshalb ebenfalls getestet werden.

$$K_i^{(test3)}(t) = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + G_i & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 0 \wedge F_2(n_1, n_2) = 0 \\ 7 + G_i & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 0 \wedge F_2(n_1, n_2) = 1 \\ 10 & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 0 \wedge F_2(n_1, n_2) = 2 \\ 11 + G_i & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 0 \\ 14 + G_i & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 1 \\ 17 & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 2 \\ 18 & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 0 \\ 19 & \text{wenn} \quad K_3(t) > 3 \wedge F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 1 \\ 20 & \text{sonst} \end{cases}$$

$$\text{für} \quad i = 2 \ldots 7$$

$$K_8^{(test3)}(t) = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 7 + K_4^{(test3)}(t) - 5 & \text{wenn} \quad K_4^{(test3)}(t) \geq 5 \\ 4 + G_7 & \text{sonst} \end{cases}$$

$$K_9^{(test3)}(t) = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + G_6 & \text{wenn} \quad K_4^{(test3)} = 4 \wedge G_7 = 0 \\ 6 + G_7 & \text{wenn} \quad K_4^{(test3)} = 4 \wedge G_7 \geq 1 \\ 9 + G_7 & \text{wenn} \quad K_4^{(test3)} = 5 \\ 12 + K_4^{(test3)}(t) - 6 & \text{wenn} \quad K_4^{(test3)} \geq 6 \end{cases}$$

$$\text{mit} \quad G_2 = F_2(n_4, n_9)$$

$$\text{und} \quad G_3 = F_2(n_8, n_9)$$

$$\text{und} \quad G_4 = F_2(n_8 + n_{10}, n_9 + n_{11})$$

$$\text{und} \quad G_5 = \begin{cases} 0 & \text{wenn} \quad F_1(n_8) + F_1(n_9) + F_1(n_{10}) + F_1(n_{11}) = 0 \\ 1 & \text{wenn} \quad F_1(n_8) + F_1(n_9) + F_1(n_{10}) + F_1(n_{11}) = 1 \\ 2 & \text{wenn} \quad F_1(n_8) + F_1(n_9) + F_1(n_{10}) + F_1(n_{11}) \geq 2 \end{cases}$$

$$\text{und} \quad G_6 = F_2(n_4 + n_7, n_5 + n_6)$$

$$\text{und} \quad G_7 = \begin{cases} 0 & \text{wenn} \quad e_2 \leq 1 \\ 1 & \text{wenn} \quad e_2 \in \{2, 3\} \\ 2 & \text{sonst} \end{cases}$$

Zusätzlich werden zum Vergleich ebenfalls die Werte für die in EBCOT verwendete Quantisierung mit angegeben, die Kontextnummern für $EBCOT_i(v, h, d)$ sind dabei aus der

	S_{LH}	S_{HL}	S_{HH}
$K_1^{(test3)}$	3,20 bps	3,22 bps	3,44 bps
$K_2^{(test3)}$	3,16 bps	3,20 bps	3,39 bps
$K_3^{(test3)}$	3,16 bps	3,19 bps	3,38 bps
$K_4^{(test3)}$	3,16 bps	3,19 bps	3,36 bps
$K_5^{(test3)}$	3,16 bps	3,19 bps	3,36 bps
$K_6^{(test3)}$	3,17 bps	3,20 bps	3,40 bps
$K_7^{(test3)}$	3,16 bps	3,18 bps	3,39 bps
$K_8^{(test3)}$	3,14 bps	3,17 bps	3,35 bps
$K_9^{(test3)}$	3,13 bps	3,16 bps	3,33 bps
$K_{LH}^{(EBCOT)}$	3,17 bps	3,20 bps	3,38 bps
$K_{HL}^{(EBCOT)}$	3,17 bps	3,20 bps	3,38 bps
$K_{HH}^{(EBCOT)}$	3,21 bps	3,23 bps	3,37 bps

Tabelle 5.12: Die durchschnittlich benötigten Bits per Significance verschiedener Quantisierungen bei der Anwendung auf alle 5/3 transformierten Testbilder.

Tabelle 4.2 des vorigen Kapitels zu entnehmen.

$$K_i^{(EBCOT)} = \begin{cases} K_3(t) & \text{wenn} \quad K_3(t) \leq 3 \\ 4 + EBCOT_i(v,h,d) & \text{sonst} \end{cases}$$
$$\text{für} \quad i \in \{LH, HL, HH\}$$
$$\text{mit} \quad v = F_2(n_0, n_3), \quad h = F_2(n_1, n_2), \quad d = \sum_{i=8}^{11} F_1(n_i)$$

In Tabelle 5.12 ist ersichtlich, dass $K_9^{(test3)}$ am besten abschneidet und damit auch die Quantisierung von EBCOT unterbietet. Hierbei müssen aber zwei Anmerkungen gemacht werden:

1. EBCOT teilt das wavelet-transformierte Bild in Blöcke, der Grösse 64×64 bzw. 32×32 auf. Diese Blöcke werden unabhängig voneinander kodiert, deshalb beschränkt sich die Quantisierung von EBCOT bewusst auf 9 Kontexte, $K_9^{(test3)}$ hat deutlich mehr Kontexte und ist aufgrund dessen nicht für die Kodierung von derartigen Teilblöcken geeignet.

2. Bei dem Vergleich wurden alle Quantisierungen nur auf Tupel angewendet, die nicht in $ACTIVE$ liegen, im EBCOT-Algorithmus gibt es keine derartige Unterscheidung.

Es ist also deshalb nicht überraschend, dass die EBCOT-Quantisierung etwas schlechter abschneidet, da sie für einen anderen Einsatzzweck konzipiert ist.

So wie auch in EBCOT, erfolgt auch hier eine Reduzierung der Nachbarschaftskoeffizienten auf $= 0$ und ≥ 1. Im Unterschied zu EBCOT wird bei der Quantisierung aber keine Unterscheidung nach der Subbandausrichtung gemacht. Das heisst es werden zwar für jedes Subband getrennte Kontexte verwendet, es kommt aber überall $K_9^{(test3)}$ zum Einsatz, da bereits implizit durch den Aufbau der Quantisierung horizontale und vertikale Komponenten gleichberechtigt sind.

5.3.7 Codec 4

Codec 4 nutzt nun die beste Quantisierung aus den vorangegangen Tests

$$K_4(t) \quad = \quad K_9^{(test3)}(t) \qquad . \tag{5.4}$$

Die erreichten Ergebnisse sind in Tabelle 5.14 aufgeführt. Noch einmal zum Vergleich: Codec 3 benötigte zur Kodierung der Significance-Map des LH-Subbands von `lena.pgm` bis Bitplane fünf 32.915 Bits (ohne die Kontexte $0 \ldots 3$), mit $K_5^{(test1)}$ konnte die benötigte Bitanzahl bereits auf 30.539 reduziert werden, K_4 verringert die Grösse abermals auf nun 28.644 Bits, insgesamt also eine Reduktion von über 13% gegenüber Codec 3. Vergleicht man die bpp-Zahlen aus Tabelle 5.14 mit denen aus Tabelle 5.8, so fällt die Verbesserung nicht ganz so gross aus, zum einen weil in den übrigen Subbändern die Reduktionen moderater ausgefallen sind und zum zweiten, weil noch die Bits der Kontexte $0 \ldots 3$ hinzukommen, sowie die unveränderte Sign- und Magnitude-Map.

5.4 Mehrere Durchläufe pro Bitplane

Tabelle 5.13 zeigt, dass die Kontexte der Tupel $t \notin ACTIVE$ von Codec 4 eine weite Spanne von Signifikanzwahrscheinlichkeiten umfassen. Wenn man nun die Reihenfolge der Übertragung der Koeffizienten daran ausrichtet, zu welchem Kontext das 17-Tupel eines Koeffizienten gehört und man mit den Kontexten hoher Signifikanzwahrscheinlichkeit beginnt, so sollte sich innerhalb einer Bitplane der PSNR schneller steigern lassen. In allen bisherigen Codecs wird jede Bitplane genau einmal durchlaufen, stattdessen könnte man nun eine Bitplane mehrmals durchlaufen und im ersten Durchlauf nur die Signifikanz derjenigen Koeffizienten kodieren, deren 17-Tupel in einem Kontext mit beispielsweise mindestens 40% Signifikanzwahrscheinlichkeit liegen. In den weiteren Durchlaufen wird dann diese Schwelle weiter abgesenkt, bis schliesslich in einem letzten Durchlauf alle verbleibenden Koeffizienten kodiert werden.

Es lässt sich jedoch nicht einfach die bisherige Quantisierung als Ausgangspunkt nehmen, denn durch die geänderte Scanorder treten auch Rückwirkungen auf die 17-Tupel auf, die letztendlich wiederum Einfluss auf die Quantisierung selbst nehmen. Bisher war die Menge der benachbarten Koeffizienten, deren Signifikanz in der gerade betrachteten Bitplane bereits kodiert wurde, stets die gleiche, nämlich $e_i, i = 0 \ldots 4$ sowie $n_i, i \in \{0, 1, 4, 5, 8, 9\}$ Durch die angedachte Änderung der Scanorder ist dies nicht mehr der Fall. Dies muss aber nicht unbedingt von Nachteil sein. Konnte bisher die Signifikanz von n_3 in derselben Bitplane bei der Quantisierung nicht berücksichtigt werden, so ist dies jetzt möglich, jedoch mit der Einschränkung, dass n_3 in einem früheren Durchlauf kodiert wurde.

K_4	$\|T\|$	$ZERO(T)$	$ONE(T)$	p_T	TB_1	TB_2	TB_3	Codec 4
0	1	190.573	59	0,03%	773	773	773	764
1	5	26.534	98	0,37%	931	934	934	946
2	244	34.497	498	1,42%	3.408	3.713	3.769	3.769
3	6	493	21	4,09%	121	127	127	134
$\sum$	256	252.097	676	-	5.233	5.547	5.603	5.613
4	1.196	5.811	240	3,97%	726	841	1.456	1.467
5	1.891	3.792	200	5,01%	307	404	1.145	1.154
6	208	211	15	6,64%	2	2	79	85
7	1.992	5.393	448	7,67%	935	1.078	2.280	2.289
8	1.122	1.508	317	17,37%	292	315	1.215	1.222
9	2.752	3.804	393	9,36%	387	454	1.882	1.889
10	1.636	1.680	295	14,94%	150	161	1.201	1.209
11	1.261	1.103	238	17,75%	50	50	904	911
12	675	520	161	23,64%	6	6	537	543
13	2.153	2.397	422	14,97%	230	277	1.717	1.724
14	3.598	3.107	733	19,09%	77	86	2.700	2.707
15	859	611	252	29,20%	2	2	751	758
16	485	326	159	32,78%	0	0	442	446
17	2.206	2.006	834	29,37%	488	550	2.480	2.489
18	2.272	1.571	757	32,52%	39	40	2.118	2.126
19	584	352	237	40,24%	0	0	572	575
20	725	490	258	34,49%	13	13	695	698
21	2.567	1.520	1.070	41,31%	15	15	2.533	2.538
22	1.196	641	555	46,40%	0	0	1.191	1.196
23	821	414	407	49,57%	0	0	820	826
24	407	197	210	51,60%	0	0	406	410
25	856	397	459	53,62%	0	0	852	857
26	520	256	264	50,77%	0	0	519	525
$\sum$	31.982	38.107	8.924	-	3.719	4.294	28.495	28.644
$\sum$	32.238	290.204	9.600	-	8.952	9.841	34.098	34.257

Tabelle 5.13: Am Beispiel des LH-Subbands (bis einschliesslich Bitplane 5) von `lena.pgm` (5/3 transformiert) einige Details zu den zusätzlichen Kontexten in Codec 4. Die Kontexte $K_4 = 0 \ldots 3$ sind die gleichen wie in Codec 3. Die benötigte Bitmenge für die Kontexte grösser drei konnte von 32.915 auf 28.644 Bits reduziert werden. Die Abweichung von TB_1 um 7 Bits im Vergleich zu Tabelle 5.9 ist durch Rundungsfehler bedingt.

	S+P	5/3	9/7
0,10 bpp	26,46 dB	26,18 dB	26,80 dB
0,25 bpp	29,65 dB	29,34 dB	30,11 dB
0,50 bpp	32,62 dB	32,30 dB	33,23 dB
0,75 bpp	34,70 dB	34,53 dB	35,49 dB
1,00 bpp	36,24 dB	36,02 dB	37,17 dB
2,00 bpp	41,19 dB	41,22 dB	42,68 dB
B bis $B-5$	0,01 bpp	0,02 bpp	0,01 bpp
B bis $B-6$	0,04 bpp	0,06 bpp	0,04 bpp
B bis $B-7$	0,12 bpp	0,14 bpp	0,12 bpp
B bis $B-8$	0,29 bpp	0,29 bpp	0,27 bpp
B bis $B-9$	0,59 bpp	0,55 bpp	0,54 bpp
B bis $B-10$	1,10 bpp	0,96 bpp	1,00 bpp
verlustfrei	4,59 bpp	4,58 bpp	-

Tabelle 5.14: Durchschnittswerte der mit Codec 4 komprimierten Testbilder.

Auch wenn jetzt von mehreren Durchläufen für ein und dieselbe Bitplane die Rede ist, wird auch weiterhin die Signifikanz bzw. Insignifikanz eines Koeffizienten pro Bitplane genau einmal kodiert. Es wird jedoch jetzt mehrmals das 17-Tupel überprüft, um zu entscheiden in welchem Durchlauf dies zu geschehen hat, dies führt letztendlich zu einer längeren Laufzeit.

Wenn die Significance-Map in der Reihenfolge der Signifikanzwahrscheinlichkeit der Koeffizienten kodiert werden soll, stellt sich die Frage, wie die Magnitude-Map hierbei einzuordnen ist. Die Signifikanz eines Koeffizienten c_s in Bitplane i bedeutet, dass der bisherige Wert 0 durch den zu erwartenden Wert mit

$$c_s = (-1)^{sign} \cdot (2^i + 2^{i-1})$$

approximiert wird. Ein Magnitude-Bit in Bitplane i bewirkt eine weitere Verfeinerung eines Koeffizienten c_m. Ist der Koeffizient in der vorigen Bitplane signifikant geworden, so ist er bereits auf

$$c_m = (-1)^{sign} \cdot (2^{i+1} + 2^i)$$

gesetzt – zur Erinnerung: Magnitude-Bits werden erstmalig in der Bitplane nach der Signifikanz kodiert, da die Signifikanz ja bereits das erste 1-Bit beinhaltet. Das Magnitude-Bit enthält nun das Bit der Position i, ausserdem wird wiederum die Intervallmitte gebildet, um den mittleren Fehler zu minimieren. Fasst man beides in einem Schritt zusammen, so ergibt sich folgendes:

$$\begin{aligned} c_m &= \begin{cases} (-1)^{sign} \cdot (2^{i+1} + 2^{i-1}) & \text{wenn} \quad magn_i = 0 \\ (-1)^{sign} \cdot (2^{i+1} + 2^i + 2^{i-1}) & \text{wenn} \quad magn_i = 1 \end{cases} \\ &= (-1)^{sign} \cdot \left(2^{i+1} + 2^i - (-1)^{magn_i} \cdot 2^{i-1}\right) \quad . \end{aligned}$$

Bei der Kodierung der Signifikanz ändert sich ein Koeffizient also von 0 auf $\pm(2^i + 2^{i-1})$, bei einem Magnitude-Bit hingegen von c auf $c \pm 2^{i-1}$. Bekanntlich geht die quadratische Abweichung – also der MSE – zwischen Originalkoeffzient und rekonstruiertem Koeffizient in den PSNR ein. Die Kodierung der Signifikanz in Bitplane i reduziert also den MSE stärker als ein Magnitude-Bit, dabei ist jedoch auch zu berücksichtigen, dass ein Magnitude-Bit auch nur mit einem Bit kodiert wird während eine Signifikanz mit Vorzeichen wohl mindestens 3 Bits benötigen wird (bei einer angenommenen Signifikanzwahrscheinlichkeit von 50%). Die folgende Rechnung soll eine Abschätzung liefern, wie weit der Approximationsfehler reduziert werden kann. Dabei bezeichnet $\hat{c}$ den Wert für c in Bitplane $i+1$ und $c^\star$ den Wert in der jetzigen Bitplane i. Im besten Fall ist $c^\star = c$:

Signifikanz (best case):

Approximationsfehler in Bitplane $i+1$:

$$\begin{aligned}(\hat{c} - c)^2 &= \left(0 \pm (2^i + 2^{i-1})\right)^2 \\ &= \left(2^i + 2^{i-1}\right)^2 \\ &= \left(2 \cdot 2^{i-1} + 2^{i-1}\right)^2 \\ &= \left(3 \cdot 2^{i-1}\right)^2 \\ &= 9 \cdot \left(2^{i-1}\right)^2\end{aligned}$$

Approximationsfehler in Bitplane i:

$$(c^\star - c)^2 = 0^2$$

Reduktion des Approximationsfehlers: $9 \cdot \left(2^{i-1}\right)^2$

Magnitude (best case):

Approximationsfehler in Bitplane $i+1$:

$$\begin{aligned}(\hat{c} - c)^2 &= \left(\pm(2^{i+1} + 2^i) - \pm(2^{i+1} + 2^i + 2^{i-1})\right)^2 \\ &= \left(2^{i-1}\right)^2\end{aligned}$$

Approximationsfehler in Bitplane i:

$$(c^\star - c)^2 = 0^2$$

Reduktion des Approximationsfehlers: $\left(2^{i-1}\right)^2$

Im schlechtesten Fall ist $c_s = 2^i$ bzw. $c_m = 2^{i+1}$:

Signifikanz (worst case):

Approximationsfehler in Bitplane $i+1$:

$$\begin{aligned}(\hat{c} - c)^2 &= \left(0 \pm 2^i\right)^2 \\ &= \left(2^i\right)^2 \\ &= \left(2 \cdot 2^{i-1}\right)^2 \\ &= 4 \cdot \left(2^{i-1}\right)^2\end{aligned}$$

Approximationsfehler in Bitplane i:

$$\begin{aligned}(c^\star - c)^2 &= \left(\pm(2^i + 2^{i-1}) \pm 2^i\right)^2 \\ &= \left(2^{i-1}\right)^2\end{aligned}$$

Reduktion des Approximationsfehlers: $4 \cdot \left(2^{i-1}\right)^2 - \left(2^{i-1}\right)^2 = 3 \cdot \left(2^{i-1}\right)^2$

Magnitude (worst case):
Approximationsfehler in Bitplane $i+1$:

$$\begin{aligned}(\hat{c} - c)^2 &= \left(\pm(2^{i+1} + 2^i) - \pm 2^{i+1}\right)^2 \\ &= \left(2^i\right)^2 \\ &= \left(2 \cdot 2^{i-1}\right)^2 \\ &= 4 \cdot \left(2^{i-1}\right)^2\end{aligned}$$

Approximationsfehler in Bitplane i:

$$\begin{aligned}(c^\star - c)^2 &= \left(\pm(2^{i+1} + 2^{i-1}) - \pm 2^{i+1}\right)^2 \\ &= \left(2^{i-1}\right)^2\end{aligned}$$

Reduktion des Approximationsfehlers: $4 \cdot \left(2^{i-1}\right)^2 - \left(2^{i-1}\right)^2 = 3 \cdot \left(2^{i-1}\right)^2$

Das heisst im besten denkbaren Fall kann der MSE mit der Kodierung einer Signifikanz neunmal mehr reduziert werden, als mit der Kodierung eines Magnitude-Bits, im schlechtesten denkbaren Fall sind die Reduktionen für beide gleich. In der Realität werden die Werte wohl irgendwo dazwischen liegen, der optimale Zeitpunkt zur Übertragung der Magnitude-Map soll deshalb später experimentell bestimmt werden.

5.4.1 Codec 5

In Codec 4 wurden für die Quantisierung der übrigen Tupel im wesentlichen die benachbarten Koeffizienten $n_i, i = 0 \ldots 3$ benutzt. Die vom Bitumfang her grössten Kontexte wurden dann unter Zuhilfenahme von $n_i, i = 8 \ldots 11$, e_2 und $n_i, i = 4 \ldots 7$ weiter aufgespalten. Genau nach diesem Prinzip soll jetzt auch die nach Signifikanzwahrscheinlichkeit gestaffelte Quantisierung erfolgen. Die Basis bilden wiederum die Koeffizienten $n_i, i = 0 \ldots 3$ und sofern Kontexte gross genug sind werden sie weiter mit Hilfe zusätzlicher Koeffizienten aufgespalten.

Experimentell hat sich die Aufteilung in 6 Durchläufe zur Kodierung der Significance-Map als sinnvoll erwiesen. Die Aufteilung ist so angelegt, dass pro Durchlauf die durchschnittliche Signifikanzwahrscheinlichkeit um etwa zehn Prozentpunkte sinkt. Dies variiert aber auch von Bild zu Bild, da die Signifikanzwahrscheinlichkeiten der einzelnen Kontexte zwischen verschiedenen Bildern auch durchaus differieren. Die Durchläufe sind mit (a) bis (f) bezeichnet, die verwendeten Quantisierungen sehen dabei wie folgt aus:

(a) 40% bis 50%

$$
\begin{aligned}
F_1'(x) &= \begin{cases} 0 & \text{wenn} \quad x \leq 1 \\ 1 & \text{wenn} \quad x \geq 2 \end{cases} \\
F_2'(x_0, x_1) &= F_1'(x_0) + F_1'(x_1) \\
K_5^{(a)}(t) &= \begin{cases} L_1 & \text{wenn} \quad F_2'(n_0, n_3) = 1 \wedge F_2'(n_1, n_2) = 1 \\ 3 & \text{wenn} \quad F_2'(n_0, n_3) = 1 \wedge F_2'(n_1, n_2) = 2 \\ 4 & \text{wenn} \quad F_2'(n_0, n_3) = 2 \wedge F_2'(n_1, n_2) = 0 \\ 5 & \text{wenn} \quad F_2'(n_0, n_3) = 2 \wedge F_2'(n_1, n_2) = 1 \\ 6 & \text{wenn} \quad F_2'(n_0, n_3) = 2 \wedge F_2'(n_1, n_2) = 2 \\ -1 & \text{sonst} \end{cases} \\
\text{mit} \quad L_1 &= F_2'\left(F_2'(n_8, n_{10}), F_2'(n_9, n_{11})\right)
\end{aligned}
$$

In dieser Quantisierung wird nicht auf $= 0$ und ≥ 1 getestet sondern auf ≤ 1 und ≥ 2, um nur die Koeffizienten mit möglichst hoher Signifikanzwahrscheinlichkeit "einzufangen".

(b) 30% bis 40%

$$
\begin{aligned}
K_5^{(b)}(t) &= \begin{cases} 7 + L_2 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 0 \\ 10 + L_2 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 1 \\ 13 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 2 \\ 14 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 0 \\ 15 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 1 \\ 16 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 2 \\ -1 & \text{sonst} \end{cases} \\
\text{mit} \quad L_2 &= F_2(n_8 + n_{10}, n_9 + n_{11})
\end{aligned}
$$

(c) 20% bis 30%

$$
K_5^{(c)}(t) = \begin{cases} 17 + L_2 & \text{wenn} \quad F_2(n_0, n_3) = 0 \wedge F_2(n_1, n_2) = 1 \\ 20 & \text{wenn} \quad F_2(n_0, n_3) = 0 \wedge F_2(n_1, n_2) = 2 \\ 21 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 0 \\ 22 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 1 \\ 23 & \text{wenn} \quad F_2(n_0, n_3) = 1 \wedge F_2(n_1, n_2) = 2 \\ 24 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 0 \\ 25 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 1 \\ 26 & \text{wenn} \quad F_2(n_0, n_3) = 2 \wedge F_2(n_1, n_2) = 2 \\ -1 & \text{sonst} \end{cases}
$$

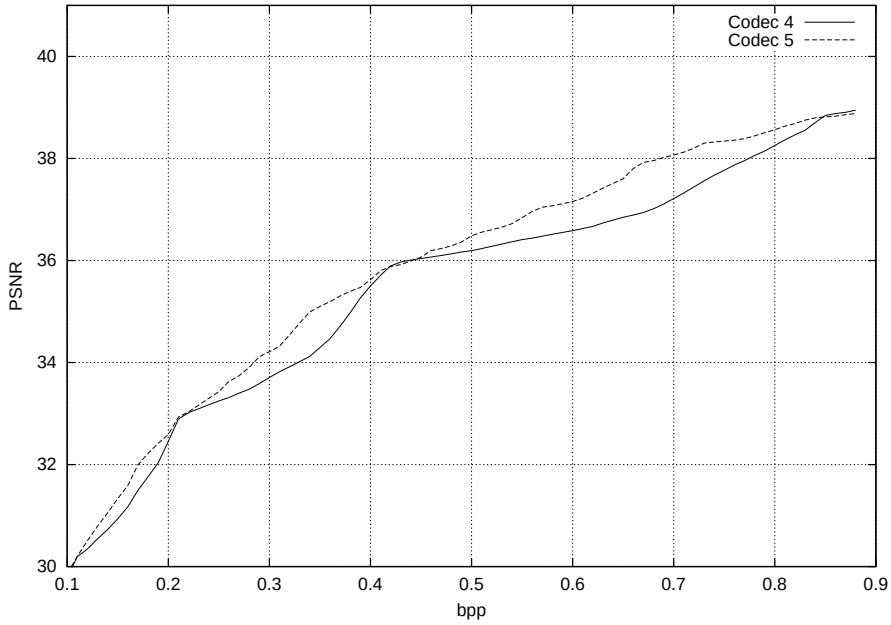


Abbildung 5.3: Am Beispiel von `lena.pgm`, (5/3 transformiert) wird der Unterschied zwischen Codec 4 und Codec 5 verdeutlicht. An den Stellen, wo sich die PSNR-Werte von Codec 4 und 5 treffen, ist jeweils eine Bitplane zu Ende. Im gezeigten Ausschnitt sind dies Bitplane 7, 6, 5, und 4. Es ist ersichtlich, dass in Codec 5 die PSNR-Kurve auch innerhalb einer Bitplane schneller steigt.

(d) 10% bis 20%

$$K_5^{(d)}(t) \quad = \quad \begin{cases} 27 + K_4(t) & \text{wenn} \quad K_4(t) \geq 6 \\ -1 & \text{sonst} \end{cases}$$

(e) 1% bis 10%

$$K_5^{(e)}(t) \quad = \quad \begin{cases} 27 + K_4(t) & \text{wenn} \quad K_4(t) \geq 1 \\ -1 & \text{sonst} \end{cases}$$

(f) der Rest (hauptsächlich Tupel $(0, \ldots, 0)$**)**

$$K_5^{(f)}(t) \quad = \quad 27 + K_4(t)$$

Die Kontextnummer -1 bei der Quantisierung bedeutet, dass der Koeffizient in diesem Durchlauf nicht kodiert wird, er wird somit im folgenden Durchlauf erneut betrachtet. Der letzte Durchlauf (f) muss deshalb grundsätzlich alle verbliebenen Koeffizienten kodieren.

Bitplane	vor (f)	vor (e)	vor (d)
8	+0.029942 dB	+0.000208 dB	-0.117803 dB
7	+0.037256 dB	-0.102039 dB	-0.294865 dB
6	+0.040623 dB	-0.126493 dB	-0.402671 dB
5	+0.062177 dB	-0.073672 dB	-0.469299 dB
4	+0.094459 dB	+0.055894 dB	-0.342969 dB
3	+0.094181 dB	+0.250100 dB	-0.114051 dB
2	+0.005958 dB	+0.372367 dB	+0.365220 dB
1	+0.000248 dB	+0.067768 dB	+0.344033 dB
0	-0.000191 dB	+0.004541 dB	+0.029423 dB

Tabelle 5.15: Gewinn/Verlust, wenn die Übertragung der Magnitude-Map vor Durchlauf (f), (e) bzw. (d) erfolgt. Es sind die Durchschnittswerte über alle 5/3 transformierten Testbilder aufgeführt.

Wurde ein Koeffizient in einem Durchlauf kodiert, so wird dies vermerkt, damit derselbe Koeffizient in einem späteren Durchlauf nicht nochmal kodiert wird.

Abbildung 5.3 zeigt, den Unterschied zwischen Codec 4 und Codec 5 am Beispiel von `lena.pgm`. Während Codec 4 den PSNR "schubweise" steigert und dabei innerhalb einer Bitplane zurückfällt, wird bei Codec 5 infolge der geänderten Übertragungsreihenfolge der PSNR innerhalb einer Bitplane schneller gesteigert. Durch eine noch feinere Staffelung liesse sich sicherlich an der einen oder anderen Stelle der PSNR innerhalb der Bitplane noch schneller steigern, jedoch würde das auch die Laufzeit weiter erhöhen. Da die zu erwartende Steigerung des PSNR nicht sehr gross sein dürfte wurde darauf verzichtet.

Es bleibt nun noch zu klären, nach welchem Durchlauf die Magnitude-Map übertragen werden soll, dazu soll folgendes untersucht werden:

1. kodiere genau bis zum Ende eines festgelegten Durchlaufes
2. bestimme die benötigte Dateigrösse für das komprimierte Bild
3. füge jetzt vor den letzten Durchlauf die Kodierung der Magnitude-Map ein, wobei die resultierende Datei nur genau so gross werden darf wie in 2.
4. vergleiche den PSNR zwischen 1. und 3., ist er in 3. höher so ist es vorteilhaft die Kodierung der Magnitude-Map vor dem getesteten Durchlauf durchzuführen.

Dieses Vorgehen wurde auf alle 5/3 transformierten Testbilder auf die Bitplanes 8 bis 0 angewendet, dabei wurden die Durchläufe (f), (e) und (d) getestet. Die Ergebnisse sind in Tabelle 5.15 dargestellt. Auf jeden Fall sollte die Magnitude-Map vor Durchlauf (f) übermittelt werden, denn der PSNR steigt dadurch in fast allen Bitplanes, vor allem in den unteren Bitplanes erscheint eine Kodierung der Magnitude-Map sogar noch vor Durchlauf (e) vorteilhaft zu sein. Ein ähnliches Bild ergibt sich für die S+P-Transformation. Für die 9/7 transformierten Bilder sind die Ergebnisse nicht ganz so eindeutig, dennoch erscheint es insgesamt am vorteilhaftesten die Kodierung der Magnitude-Map noch vor Durchlauf

	Codec 5			SPIHT	
	S+P	5/3	9/7	S+P	9/7
0,10 bpp	26,65 dB	26,49 dB	27,01 dB	26,53 dB	26,81 dB
0,25 bpp	29,92 dB	29,70 dB	30,44 dB	29,78 dB	30,20 dB
0,50 bpp	32,92 dB	32,81 dB	33,64 dB	32,82 dB	33,38 dB
0,75 bpp	34,91 dB	34,95 dB	35,83 dB	34,93 dB	35,58 dB
1,00 bpp	36,55 dB	36,58 dB	37,63 dB	36,58 dB	37,39 dB
2,00 bpp	41,15 dB	41,57 dB	43,21 dB	41,74 dB	43,03 dB
B bis $B-5$	0,01 bpp	0,02 bpp	0,01 bpp	-	-
B bis $B-6$	0,04 bpp	0,06 bpp	0,04 bpp	-	-
B bis $B-7$	0,12 bpp	0,14 bpp	0,12 bpp	-	-
B bis $B-8$	0,29 bpp	0,29 bpp	0,27 bpp	-	-
B bis $B-9$	0,58 bpp	0,54 bpp	0,54 bpp	-	-
B bis $B-10$	1,10 bpp	0,96 bpp	0,99 bpp	-	-
verlustfrei	4,58 bpp	4,57 bpp	-	4,60 bpp	-

Tabelle 5.16: Durchschnittswerte der mit Codec 5 komprimierten Testbilder, zum Vergleich die Werte der SPIHT-Implementierung [20], da die SPIHT-Binaries keine Möglichkeit bieten, die Werte für $B-x$ zu ermitteln, wurden diese ausgelassen.

(e) der Kodierung der Significance-Map durchzuführen. Die der Abbildung 5.3 zugrundeliegenden Werte berücksichtigen dies bereits. Insgesamt bestätigt dies auch das Vorgehen bei EBCOT, denn dort wird die Magnitude-Map auch nicht zu allerletzt übertragen.

In Tabelle 5.16 sind zum Vergleich auch die Werte von SPIHT mit aufgeführt. Es ist ersichtlich, dass Codec 5 vor allem bei dem 9/7-Wavelet immer um mindestens 0,2 dB besser abschneidet als SPIHT, ausserdem ergeben sich auch bei der verlustfreien Kompression leichte Vorteile. Im Vergleich zu Codec 4 ist anzumerken, dass sich die bpp-Zahlen am Ende einer Bitplane kaum ändern. Hingegen verbessern sich die PSNR-Werte für einen vorgegebenen bpp-Wert, da dieser zumeist in einer Bitplane liegt und Codec 5 ja gerade innerhalb einer Bitplane den PSNR schneller steigert als Codec 4.

5.5 Detailverbesserungen

Bisher wurde das Hauptaugenmerk ausschliesslich auf die Verbesserung der Kompression der Significance-Map gelegt, im Vergleich zu Codec 1 konnte hier auch einiges erreicht werden. Nun soll es unter anderem auch darum gehen die Sign-Map und die Magnitude-Map besser zu komprimieren, wie aber bereits zuvor angedeutet, wird man damit keine so grossen Verbesserungen erzielen können, wie bei der Significance-Map. Zuvor aber noch zwei Detailverbesserungen an anderer Stelle.

5.5.1 Optimierung der Arithmetischen Kodierung

Alle in diesem Kapitel vorgestellten Codecs arbeiten mit der Implementierung der adaptiven binären Arithmetischen Kodierung aus Kapitel 2. Im Verlauf der Untersuchungen wurde festgestellt, dass die einzelnen verwendeten Kontexte eine weite Spanne von Wahrscheinlichkeiten nutzen. Jeder Kontext startet dabei mit einer 1-Wahrscheinlichkeit von 50% und passt sich dann sukzessive an die Auftrittswahrscheinlichkeit der 1 in den Eingabedaten an. Um diesen Anpassungsprozess zu verkürzen, könnte man nun einige Kontexte mit Wahrscheinlichkeiten ungleich 50% vorbelegen, so dass die Anpassung schneller geht, dadurch lassen sich aber kaum messbare Verbesserungen erzielen. Denn das Problem ist weniger, einmal auf die 1-Wahrscheinlichkeit zu kommen, sondern vielmehr sich schnellstmöglich an Veränderungen selbiger anzupassen. Erfolgsversprechender ist deshalb eine "Beschleunigung" der Adaptivität des Arithmetischen Kodierers. Dazu wurde folgendes umgesetzt:

1. Das Update-Intervall zur Neuberechnung der 1-Wahrscheinlichkeit ist in der bisherigen Implementierung bis zu 64 Bit gross, führt man hingegen nach jedem Bit eine Neuberechnung durch, so führt dies zwar zu etwas mehr Rechenaufwand, verbessert aber die Adaptivität.

2. Die bisherige Implementierung arbeitet mit zwei 13 Bitcountern, um die Gesamtanzahl der Bits (`bitcount`) und den Anteil der Nullbits (`bitcount0`) zu zählen. Übersteigt die Gesamtanzahl den Wert 8191, so werden beide Counter halbiert. Bei 1-Wahrscheinlichkeiten nahe 0% oder 100% macht es Sinn den Wertebereich dieser Counter voll auszuschöpfen, jedoch ist es bei den übrigen 1-Wahrscheinlichkeiten weniger sinnvoll, stattdessen wäre es hier besser den Wertebereich zu verkleinern, dafür aber eine schnellere Anpassung zu erreichen. Die bisherige Bedingung (in C-Notation)

   ```
   if (bitcount >= 8192) /* Counter halbieren */;
   ```

   wurde deshalb wie folgt erweitert:

   ```
   bits = bitcount - bitcount0;
   if (2 * bits >= bitcount) bits = bitcount - bits;
   if (((bitcount >=  500) && (bitcount <=  50 * bits)) ||
           ((bitcount >= 1000) && (bitcount <= 100 * bits)) ||
           ((bitcount >= 2000) && (bitcount <= 200 * bits)) ||
           ((bitcount >= 4000) && (bitcount <= 400 * bits)) ||
           (bitcount >= 8192)) /* Counter halbieren */;
   ```

3. Das Zurücksetzen aller Kontexte auf die Startwahrscheinlichkeit von 50% zu Beginn einer neuen Bitplane. Experimentell hat sich gezeigt, dass dieses Vorgehen immer dann vorteilhaft ist, wenn man sich unterhalb von Bitplane 4 im transformierten Bild befindet bzw. mindestens ein Achtel aller Koeffizienten bereits signifikant sind. Die zwei Bitcounter werden dabei auf `bitcount = 2` bzw. `bitcount0 = 1` zurückgesetzt. In gewisser Weise ist dies also noch eine Steigerung von Punkt 2.

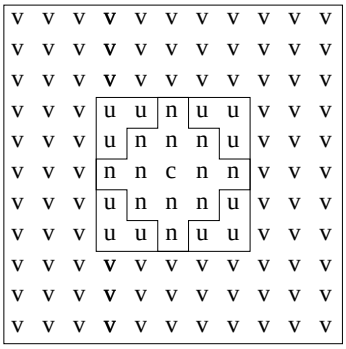


Abbildung 5.4: Ist das 17-Tupel $(0, \ldots, 0)$, so wird noch die weitere Umgebung derselben Transformationsstufe mit berücksichtigt.

5.5.2 Erweiterung des $(0, \ldots, 0)$-Tupels

In den bisherigen Untersuchungen hat sich gezeigt, dass Koeffizienten, deren 17-Tupel nur Nullen enthält mit sehr hoher Wahrscheinlichkeit insignifikant bleiben. Wenn man nun bei solchen Koeffizienten zusätzlich noch die weitere Umgebung derselben Transformationsstufe betrachtet, lässt sich eine zusätzliche Fallunterscheidung machen:

1. Alle Koeffizienten, die sich in dem Bereich $x \pm 5$ und $y \pm 5$ befinden sind insignifikant, bezogen auf Abbildung 5.4 heisst das alle n-, u- und v-Koeffizienten sind insignifikant. Die n-Koeffizienten werden auch in den übrigen beiden Fällen insignifikant bleiben, da sie bereits Teil des 17-Tupels sind und das 17-Tupel ja $(0, \ldots, 0)$ ist.

2. Alle Koeffizienten, die sich in dem Bereich $x \pm 2$ und $y \pm 2$ befinden sind insignifikant, im Bereich $x \pm 5$ und $y \pm 5$ gibt es aber mindestens einen signifikanten Koeffizienten, gemäss der in Abbildung 5.4 gewählten Bezeichnung ist also mindestens einer der v-Koeffizienten signifikant, während die u-Koeffizienten hingegen insignifikant sind.

3. In dem Bereich $x \pm 2$ und $y \pm 2$ befinden sich signifikante Koeffizienten, also mindestens einer der u-Koeffizienten ist signifikant.

Diese Fallunterscheidung macht aber nur Sinn, wenn in der aktuell bearbeiteten Bitplane nur relativ wenige Koeffizienten signifikant sind, deshalb wird sie nicht mehr durchgeführt, wenn die bearbeitete Bitplane kleiner 4 ist bzw. wenn mehr als ein Achtel der Koeffizienten signifikant sind. Diese Bedingung ist somit identisch mit der Prüfung ab wann Arithmetische Kontexte zurückgesetzt werden können, deshalb können beide Prüfungen auch zusammengefasst werden.

5.5.3 Verbesserte Kompression der Magnitude-Map

Bisher wurden alle Magnitude-Bits in einem Kontext kodiert, durch eine Aufteilung in mehrere Kontexte sind auch hier Verbesserungen möglich. Bei der Kontextbildung wird

zum einen berücksichtigt, in welcher Bitplane der gerade betrachtete Koeffizient signifikant geworden ist und zum zweiten die Summe aus $n_i, i = 0 \ldots 3$. Im Gegensatz zur Significance-Map werden jetzt gemeinsame Kontexte für alle Subbänder verwendet, da sich die 1-Wahrscheinlichkeiten der Kontexte zwischen den Subbändern nicht unterscheiden und die Bitanzahl pro Kontext nicht zu gering ausfallen soll. Es wird folgende Quantisierung verwendet:

$$
\begin{aligned}
K_{magn} &= \begin{cases} 6 \cdot M_1 + M_3 & \text{wenn} \quad M_1 \leq 1 \\ 12 & \text{wenn} \quad M_1 = 2 \wedge M_2 \leq 15 \\ 13 & \text{wenn} \quad M_1 = 2 \wedge M_2 \geq 16 \\ 14 & \text{sonst} \end{cases} \\
\text{mit} \quad M_1 &= \text{wieviele Magnitude-Bits für diesen Koeffizienten} \\
&\quad \text{bereits übertragen wurden} \\
\text{und} \quad M_2 &= \sum_{i=0}^{3} n_i \\
\text{und} \quad M_3 &= \begin{cases} 0 & \text{wenn} \quad M_2 = 0 \\ 1 & \text{wenn} \quad M_2 = 1 \\ 2 & \text{wenn} \quad M_2 = 2 \\ 3 & \text{wenn} \quad M_2 \geq 4 \wedge M_2 \leq 7 \\ 4 & \text{wenn} \quad M_2 \geq 8 \wedge M_2 \leq 11 \\ 5 & \text{sonst} \end{cases} .
\end{aligned}
$$

Im Vergleich zu EBCOT wird also eine wesentlich feinere Kontextaufspaltung durchgeführt. Es sei aber nochmals darauf verwiesen, dass EBCOT durch die Aufteilung in unabhängige Blöcke einem anderen Ansatz folgt und deshalb keine wesentlich grössere Kontextanzahl verwenden kann.

5.5.4 Kompression der Sign-Map

Die Kompression der Sign-Map läuft in EBCOT ähnlich ab, hier werden aber noch zusätzliche Kontexte eingefügt, falls $n_i, i = 0 \ldots 3$ keine sinnvolle Aussage über das Vorzeichen zulassen. Die Kontexte werden zwischen den Subbändern getrennt, so dass $4 \cdot 17$ Kontexte den bisher verwendeten Kontext ersetzen. Ausserdem werden auch hier ab dem gleichen Zeitpunkt, wie bei der Significance-Map, vor jeder Bitplane die Kontexte zurückgesetzt.

$$
K_{sign} = \begin{cases} I_4(n_0, n_1, n_2, n_3) & \text{wenn} \quad I_4(n_0, n_1, n_2, n_3) < 4 \\ I_4(n_0, n_1, n_2, n_3) - 1 & \text{wenn} \quad I_4(n_0, n_1, n_2, n_3) > 4 \\ I_4(n_4, n_5, n_6, n_7) + 8 & \text{sonst} \end{cases}
$$

	S+P	5/3	9/7
0,10 bpp	26,76 dB	26,57 dB	27,14 dB
0,25 bpp	30,07 dB	29,81 dB	30,63 dB
0,50 bpp	33,08 dB	32,94 dB	33,85 dB
0,75 bpp	35,09 dB	35,08 dB	36,08 dB
1,00 bpp	36,72 dB	36,79 dB	37,89 dB
2,00 bpp	41,34 dB	41,74 dB	43,54 dB
B bis $B-5$	0,01 bpp	0,02 bpp	0,01 bpp
B bis $B-6$	0,04 bpp	0,06 bpp	0,04 bpp
B bis $B-7$	0,12 bpp	0,13 bpp	0,12 bpp
B bis $B-8$	0,28 bpp	0,28 bpp	0,26 bpp
B bis $B-9$	0,57 bpp	0,53 bpp	0,52 bpp
B bis $B-10$	1,07 bpp	0,94 bpp	0,96 bpp
verlustfrei	4,48 bpp	4,48 bpp	-

Tabelle 5.17: Durchschnittswerte der mit Codec 6 komprimierten Testbilder.

$$\text{mit} \quad I_1(x) = \begin{cases} -1 & \text{wenn} \quad x < 0 \\ 0 & \text{wenn} \quad x \text{ insignifikant} \\ 1 & \text{wenn} \quad x \geq 0 \end{cases}$$

$$\text{und} \quad I_2(x_0, x_1) = \begin{cases} -1 & \text{wenn} \quad I_1(x_0) + I_1(x_1) < 0 \\ 0 & \text{wenn} \quad I_1(x_0) + I_1(x_1) = 0 \\ 1 & \text{wenn} \quad I_1(x_0) + I_1(x_1) > 0 \end{cases}$$

$$\text{und} \quad I_4(x_0, x_1, x_2, x_3) = 3 \cdot (I_2(x_1, x_2) + 1) + I_2(x_0, x_3) + 1$$

5.5.5 Codec 6

Die Detailverbesserungen entfalten unterschiedliche Wirkungen, während die weitere Aufspaltung des $(0, \ldots, 0)$-Tupels nur marginale Verbesserungen bringt, wirkt sich die höhere Kontextanzahl bei der Kodierung der Magnitude-Map schon stärker aus, die Hauptwirkung wird aber nur in den unteren Bitplanes erzielt, weil hier die Anzahl der Magnitude-Bits die der Significance-Bits übersteigt.

Die Verbesserung der Sign-Map Kompression wirkt sich über alle Bitplanes positiv aus, genauso wie die Optimierung der Arithmetischen Kodierung. Der Beitrag jeder einzelnen Detailverbesserung variiert von Bild zu Bild, bei der verlustlosen Kompression wurden aber im Mittel 1% bis 2% Verbesserung erzielt, Tabelle 5.17 zeigt die Ergebnisse.

5.6 Beschleunigung von Codec 6

In diesem Abschnitt soll es weniger um ausgefeilte Techniken zur Beschleunigung des Algorithmus gehen, sondern eher um triviale Verbesserungen, die ohne viel Aufwand Codec 6 beschleunigen helfen.

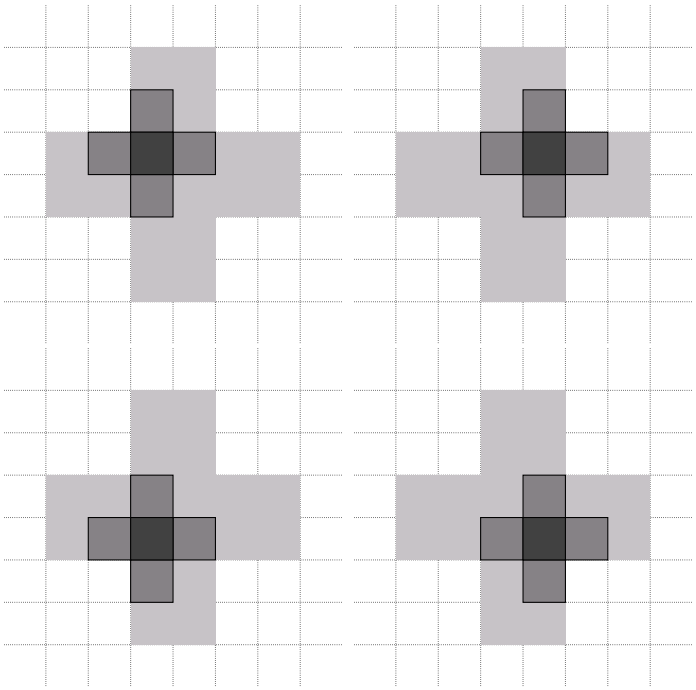


Abbildung 5.5: `signif2x2`$[\cdot,\cdot]$ enthält die Anzahl signifikanter Koeffizienten in einem 2x2 Block. In Bezug auf die Quantisierungen $K_5^{(a)}$, $K_5^{(b)}$ und $K_5^{(c)}$ lässt sich dadurch eine Beschleunigung erzielen, wenn statt der einzelnen Koeffizienten zunächst in `signif2x2`$[\cdot,\cdot]$ geprüft wird, ob überhaupt eine signifikante Nachbarschaft für alle vier Koeffizienten des 2x2 Blockes vorliegt.

Durch die sieben Durchläufe pro Bitplane (sechs für die Kodierung der Significance-Map, einer für die Magnitude-Map) hat sich die Laufzeit von Codec 5 und 6 gegenüber Codec 4 merklich verschlechtert, dieser Effekt soll nun verringert werden. Im wesentlichen geht es darum das 17-Tupel nur bei Bedarf zu erzeugen und nicht nur bei einem sondern gleich bei vier Koeffizienten zu testen, ob sie in diesem Durchlauf kodiert werden. Muss keiner der vier betrachteten Koeffizienten kodiert werden, so können weitere Aktionen entfallen.

Das Array `signif2x2`$[\cdot,\cdot]$ spielt dabei eine entscheidende Rolle. Es enthält die Anzahl der Signifikanzen in einem 2x2 Block. Wird an Position x, y ein Koeffizient signifikant, so wird `signif2x2` $\left[\left\lfloor\frac{x}{2}\right\rfloor, \left\lfloor\frac{y}{2}\right\rfloor\right]$ inkrementiert. Ein Element dieses Arrays kann also die Werte $0 \ldots 4$ annehmen.

Aus den Quantisierungen $K_5^{(a)}$, $K_5^{(b)}$ und $K_5^{(c)}$ lässt sich eine notwendige Bedingung für die Kodierung von Koeffizienten ableiten:

$$c \text{ muss kodiert werden} \quad \Rightarrow \quad \sum_{i=0}^{3} n_i > 0$$

	0,5 bpp	1,0 bpp	2,0 bpp	verlustfrei
SPIHT	0,92 spmp	1,30 spmp	1,72 spmp	2,56 spmp
Codec 1	5.26 spmp	5.68 spmp	5.99 spmp	6.90 spmp
Codec 2	2.98 spmp	3.43 spmp	3.74 spmp	4.77 spmp
Codec 3	8.58 spmp	10.87 spmp	13.12 spmp	16.21 spmp
Codec 4	8.77 spmp	11.06 spmp	13.47 spmp	16.63 spmp
Codec 5	26.47 spmp	32.54 spmp	37.00 spmp	41.96 spmp
Codec 6	28.46 spmp	34.71 spmp	39.79 spmp	45.66 spmp
Codec 7	13.85 spmp	18.23 spmp	23.04 spmp	30.17 spmp

Tabelle 5.18: Die Laufzeit aller Codecs mit S+P transformierten Bildern im Vergleich zu SPIHT (ebenfalls mit S+P-Transformation) am Beispiel von `lena.pgm`. Alle Werte wurden auf einem K6-2 mit 450 MHz ermittelt. Die Zeiten für Encoding und Decoding sind dabei nahezu gleich.

Im Umkehrschluss heisst das, ist die Summe gleich Null braucht keine nähere Betrachung zu erfolgen. Diese Bedingung lässt sich nun auch mit Hilfe von `signif2x2`$[\cdot,\cdot]$ prüfen

$$c_{x,y} \text{ muss kodiert werden} \quad \Rightarrow \quad \sum_{i=-1}^{1} \texttt{signif2x2}\left[\left\lfloor\frac{x}{2}\right\rfloor + i, \left\lfloor\frac{y}{2}\right\rfloor\right] + \sum_{i\in\{-1,1\}} \texttt{signif2x2}\left[\left\lfloor\frac{x}{2}\right\rfloor, \left\lfloor\frac{y}{2}\right\rfloor + i\right] \neq 0$$

Auch hier gilt, ist die Summe gleich Null braucht $c_{x,y}$ nicht kodiert werden. Das gilt auch für die übrigen drei Koeffizienten des 2x2 Blockes, wie Abbildung 5.5 veranschaulicht. Zusätzlich kann die Bearbeitung der vier Koeffizienten übersprungen werden, wenn `signif2x2` $\left[\left\lfloor\frac{x}{2}\right\rfloor, \left\lfloor\frac{y}{2}\right\rfloor\right] = 4$, denn dann sind bereits alle Koeffizienten des 2x2 Blockes signifikant. Analog lässt sich auch für die Durchläufe 4 und 5 eine Prüfung mit Hilfe von `signif2x2`$[\cdot,\cdot]$ finden, es müssen jedoch noch zusätzliche Werte berücksichtigt werden.

Um `signif2x2`$[\cdot,\cdot]$ sinnvoll nutzen zu können, wird es nötig die Scanorder zu ändern, es wird jetzt nicht mehr zeilenweise durch einen Subbandblock gelaufen sondern 2x2 blockweise.

bisherige Scanorder	
Zeile y	$c_{x,y}$, $c_{x+1,y}$, $c_{x+2,y}$, $c_{x+3,y}$, $c_{x+4,y}$, $c_{x+5,y}$, $c_{x+6,y}$, $\cdots$
Zeile $y+1$	$c_{x,y+1}$, $c_{x+1,y+1}$, $c_{x+2,y+1}$, $c_{x+3,y+1}$, $c_{x+4,y+1}$, $c_{x+5,y+1}$, $\cdots$

Scanorder mit `signif2x2`$[\cdot,\cdot]$	
Zeile y	$c_{x,y}$, $c_{x+1,y}$, $c_{x,y+1}$, $c_{x+1,y+1}$, $c_{x+2,y}$, $c_{x+3,y}$, $\cdots$
Zeile $y+2$	$c_{x+2,y+2}$, $c_{x+3,y+2}$, $c_{x+2,y+3}$, $c_{x+3,y+3}$, $c_{x+4,y+2}$, $c_{x+5,y+2}$, $\cdots$

In Tabelle 5.18 kommt Codec 2 von der Laufzeit her SPIHT am nächsten, teilweise ist er aber immer noch um Faktor drei langsamer. Dies mag sicherlich auch an der unoptimierten Implementierung der Arithmetischen Kodierung liegen. Der Hauptgrund ist

	Codec 7 (YAWICA)			SPIHT	
	S+P	5/3	9/7	S+P	9/7
0,10 bpp	26,76 dB	26,56 dB	27,14 dB	26,53 dB	26,81 dB
0,25 bpp	30,07 dB	29,81 dB	30,63 dB	29,78 dB	30,20 dB
0,50 bpp	33,08 dB	32,93 dB	33,85 dB	32,82 dB	33,38 dB
0,75 bpp	35,09 dB	35,08 dB	36,08 dB	34,93 dB	35,58 dB
1,00 bpp	36,72 dB	36,79 dB	37,89 dB	36,58 dB	37,39 dB
2,00 bpp	41,34 dB	41,74 dB	43,53 dB	41,74 dB	43,03 dB
B bis $B-5$	0,01 bpp	0,02 bpp	0,01 bpp	-	-
B bis $B-6$	0,04 bpp	0,06 bpp	0,04 bpp	-	-
B bis $B-7$	0,12 bpp	0,13 bpp	0,12 bpp	-	-
B bis $B-8$	0,28 bpp	0,28 bpp	0,26 bpp	-	-
B bis $B-9$	0,57 bpp	0,53 bpp	0,52 bpp	-	-
B bis $B-10$	1,07 bpp	0,94 bpp	0,96 bpp	-	-
verlustfrei	4,48 bpp	4,48 bpp	-	4,60 bpp	-

Tabelle 5.19: Durchschnittswerte der mit Codec 7 komprimierten Testbilder im Vergleich zu SPIHT.

aber wohl die geschickte Implementierung der Zerotree-Tests in SPIHT, die es erlaubt je Bitplane nur einen Teil der Koeffizienten betrachten zu müssen.

Ab Codec 3 werden die Laufzeiten durch die Kontextbetrachtung zusätzlich grösser. Es wird also nicht nur auf den Koeffizient selbst zugegriffen sondern auch noch auf die benachbarten Koeffizienten des zugehörigen 17-Tupels. Wird eine Bitplane dann noch mehrmals durchlaufen, wie ab Codec 5, steigt die Laufzeit noch weiter. Codec 7 kann diesen Effekt zumindest abschwächen, dieser Codec ist die durch `signif2x2`$[\cdot,\cdot]$ beschleunigte Version von Codec 6.

5.6.1 Codec 7 (YAWICA)

Tabelle 5.18 stellt die Laufzeit aller Codecs und die von SPIHT dar. Codec 7 benötigt teilweise die Hälfte der Laufzeit von Codec 6 ohne dabei messbar schlechtere Kompressionsergebnisse zu liefern, wie Tabelle 5.19 zeigt. Die in Codec 7 enthaltenen Verfahren sind für sich genommen alle nicht neu und teilweise auch in anderen existierenden Ansätzen verwendet worden, deshalb fällt die Namensgebung auch recht unspektakulär aus. Codec 7 soll zusätzlich auch als YAWICA bezeichnet werden, wobei diese Abkürzung für Yet Another Wavelet based Image Compression Algorithm steht.

In Tabelle 5.19 sind abermals die Vergleichswerte von SPIHT mit aufgeführt. Der Vorsprung konnte infolge der Detailverbesserungen von Codec 6 weiter ausgebaut werden und beträgt jetzt teilweise 0,5 dB. Die Ergebnisse sollen nun im folgenden Kapitel genauer analysiert werden.

Kapitel 6

Diskussion der Ergebnisse

In diesem Kapitel sollen die mit YAWICA erreichten Ergebnisse diskutiert und mit denen, der zuvor vorgestellten Verfahren, verglichen werden. Da alle hier vorgestellten Verfahren die Wavelet-Transformation einsetzen, spielt für einen Vergleich auch die Anzahl der Transformationsstufen eine Rolle. Sofern keine explizite Vorgabe erfolgt, verwendet YAWICA immer soviele Transformationsstufen wie möglich. Bei einem 512×512 Bild sind dies beispielsweise 9 ($2^9 = 512$). SPIHT, ECECOW und EBCOT verwenden teilweise nur maximal 5 Transformationsstufen, dennoch ergeben sich dadurch keine dramatischen Unterschiede. Wird YAWICA mit nur 5 Transformationsstufen verwendet, so verschlechtert sich der PSNR nur um etwa 0,005 dB. Somit bleiben die Ergebnisse weiterhin vergleichbar.

6.1 SPIHT

In Tabelle 5.19 wurden bereits die Ergebnisse von YAWICA mit denen von SPIHT gegenübergestellt. Sowohl bei verlustfreier Kompression, als auch bei verlustbehafteter Kompression mit dem 9/7-Wavelet erzielt YAWICA bessere Werte. Bei verlustbehafteter Kompression mit der S+P-Transformation sind die Ergebnisse aber nicht ganz so eindeutig, vor allem bei höheren bpp-Werten scheint YAWICA seinen Vorsprung einzubüssen. Vergleicht man die Ergebnisse für die einzelnen Bilder (siehe Anhang 8), so fallen vor allem bei 2,0 bpp deutliche Differenzen zwischen SPIHT und YAWICA auf. Welches Verfahren besser abschneidet, ist dabei von Bild zu Bild verschieden. Beachtenswert ist, dass bei den gleichen Bildern unter Verwendung des 9/7-Wavelets YAWICA immer den höheren PSNR erzielt. Der Ansatz von SPIHT scheint also bei Verwendung der S+P-Transformation vor allem in niedrigeren Bitplanes (und damit hohen bpp-Werten) bei einigen Bildern Vorteile zu haben.

Dass bei verlustfreier Kompression dennoch YAWICA bei allen Bildern besser abschneidet liegt sicherlich auch daran, dass YAWICA die Sign- und Magnitude-Map komprimiert, während SPIHT sie unkomprimiert abspeichert. Vergleicht man SPIHT nämlich mit Codec 5, der diese Detailverbesserungen noch nicht enthält, so liegen die Werte nahezu auf dem gleichen Niveau.

YAWICA ist mindestens um Faktor 10 langsamer als die verwendete SPIHT-Implementierung [20]. Vergleicht man die Werte in Tabelle 5.18, so fällt auf, dass vor allem die Betrachtung der benachbarten Koeffizienten und das mehrfache Durchlaufen jeder Bitplane die Laufzeit verschlechtert haben. Das mehrfache Durchlaufen einer Bitplane dient zur schnelleren Steigerung des PSNR innerhalb einer Bitplane, dabei sollen Koeffizienten mit hoher Wahrscheinlichkeit zuerst übertragen werden.

Auch SPIHT macht dies implizit durch die Abarbeitung seiner Listen in der Reihenfolge LIP, LIS, LSP. Aufgrund des Ablaufs des SPIHT-Algorithmus sind in LIP Koeffizienten enthalten die mit hoher Wahrscheinlichkeit bereits signifikante Nachbarkoeffizienten haben. In LIS werden Mengen von Koeffizienten auf Signifikanz getestet und gegebenenfalls einzelne Koeffizienten kodiert, der Aufwand für die Kodierung einer Signifikanz gemessen in Bits per Significance (bps) ist höher als in LIP. In LSP sind letztendlich alle bereits signifikanten Koeffizienten verzeichnet, für jene erfolgt also nur noch die Übertragung von Magnitude-Bits. Eine SPIHT-Implementierung, die auf Listen verzichtet, so wie sie in [14] beschrieben ist, erreicht deshalb innerhalb einer Bitplane schlechtere PSNR-Werte. Unter Laufzeitgesichtspunkten ist die Partitionierung der Koeffizienten in SPIHT wesentlich effizienter als der jetzige Ansatz in YAWICA.

6.2 ECECOW

Ein sinnvoller Vergleich zwischen ECECOW und YAWICA fällt schwer, da in [16] etliche Details offen gelassen wurden und ausserdem keine Programme mit einer Implementierung von ECECOW verfügbar sind. Ausserdem ist nicht klar ob die in [16] angegebenen Werte mit exakt denselben Testbildern ermittelt wurden, die auch hier verwendet wurden. Bei der verlustfreien Komprimierung erreicht ECECOW bei `lena.pgm` 4,09 bpp und bei `barbara.pgm` 4,57 bpp. Jedoch sind in [16] für SPIHT Werte von 4,16 bpp und 4,71 bpp aufgeführt. Wie in Anhang 8 zu ersehen, erreicht die hier verwendete SPIHT-Implementierung bei `lena.pgm` aber 4,19 bpp, nur bei `barbara.pgm` ergibt sich mit 4,71 bpp derselbe Wert. YAWICA erreicht für dieses Bild auch 4,57 bpp, genauso wie ECECOW, berücksichtigt man die Differenz von 0,03 bpp für die genannten SPIHT-Werte für `lena.pgm`, so liegen auch hier YAWICA und ECECOW gleichauf.

Erstaunlicher ist jedoch, dass ECECOW mit nur einem Durchlauf je Bitplane, laut [16] für `lena.pgm` und `barbara.pgm`, immer noch einen höheren PSNR-Wert erreicht als YAWICA. Würde man YAWICA auf nur einen Durchlauf reduzieren, so würde man Ergebnisse auf dem Niveau von Codec 4 erhalten, damit wird die Differenz zu ECECOW nochmals grösser. Die Quantisierung von YAWICA kann sicherlich noch optimiert werden, wenn jene von ECECOW aber soviel besser ist, dann hätte sich das eigentlich auch bei der verlustfreien Komprimierung niederschlagen müssen. Wie aber eingangs bereits erwähnt fällt ein Vergleich letztendlich schwer.

	YAWICA			EBCOT	
	S+P	5/3	9/7	5/3	(lossy)
0,10 bpp	26,76 dB	26,56 dB	27,14 dB	26,35 dB	26,86 dB
0,25 bpp	30,07 dB	29,81 dB	30,63 dB	29,74 dB	30,34 dB
0,50 bpp	33,08 dB	32,93 dB	33,85 dB	32,86 dB	33,58 dB
0,75 bpp	35,09 dB	35,08 dB	36,08 dB	35,00 dB	35,74 dB
1,00 bpp	36,72 dB	36,79 dB	37,89 dB	36,70 dB	37,55 dB
2,00 bpp	41,34 dB	41,74 dB	43,53 dB	41,62 dB	43,03 dB
verlustfrei	4,48 bpp	4,48 bpp	-	4,68 bpp	-

Tabelle 6.1: YAWICA im Vergleich zu EBCOT mit einem Qualitylayer über alle Testbilder.

6.3 EBCOT

Tabelle 6.1 zeigt die Resultate von YAWICA im Vergleich zu denen von EBCOT. Es wurde dabei die Implementierung von [21] verwendet. Aus der Dokumentation war nicht klar zu entnehmen, welche Transformation für den verlustbehafteten Fall verwendet wird, deshalb ist diese Spalte nur mit "(lossy)" überschrieben. Im verlustfreien Fall wird in [21] ebenfalls das 5/3-Wavelet eingesetzt.

Die Werte für EBCOT wurden so ermittelt, dass für den jeweiligen bpp-Wert ein eigener Datenstrom mit nur einem Quality-Layer erstellt wurde, damit werden die höchsten PSNR-Werte erzielt, weil der Overhead in EBCOT für zusätzliche Quality-Layer entfällt.

Auf den ersten Blick erreicht YAWICA bessere Ergebnisse als EBCOT, man darf dabei aber nicht die Möglichkeiten ausser Acht lassen, die EBCOT bietet. Ausserdem ist der EBCOT-Datenstrom, aufgrund der Aufteilung in unabhängige Blöcke resistenter gegen mögliche Übertragungsfehler, wohingegen ohne weitere Vorkehrungen bei YAWICA, ECECOW und SPIHT bereits ein gekipptes Bit ausreicht, um das rekonstruierte Bild unbrauchbar zu machen.

Aber genau diese Aufteilung in unabhängige Blöcke sorgt für zusätzlichen Overhead, der sich spätestens bei der verlustfreien Kompression bemerkbar macht, EBCOT schneidet hierbei nämlich auch im Vergleich zu SPIHT schlechter ab. Bei der verlustbehafteten Kompression, insbesondere bei kleinen bpp-Werten, kann EBCOT seine Blockaufteilung aber zum Vorteil einsetzen, da für jeden Block (typischerweise 32×32 oder 64×64) auch separate Kontexte für den Arithmetischen Kodierer einsetzt werden und damit lokale Unterschiede in einem Bild teilweise besser berücksichtigt werden können.

Lässt man die Blockaufteilung in EBCOT ausser Acht, so bestehen einige Gemeinsamkeiten mit YAWICA. Beide Verfahren laufen beispielsweise mehrfach durch eine Bitplane und ähneln sich auch an einigen Stellen bei der Kontextbildung für den Arithmetischen Kodierer. In Bezug auf die Laufzeit ist YAWICA jedoch deutlich langsamer als die verwendete EBCOT-Implementierung. YAWICA braucht mindestens zehnmal mehr Zeit als EBCOT. Ganz überraschend ist das Ergebnis aber nicht, weil die EBCOT-Implementierung bereits in Bezug auf Laufzeit und Speicherbedarf sehr optimiert wurde und einige Teile direkt in Assembler geschrieben sind, um die Multimedia-Instruktionen moderner Prozessoren

ausnutzen zu können.

6.4 Zusammenfassung

Im Verlauf der Untersuchungen in Kapitel 5 ist mit YAWICA ein Verfahren entstanden, dass bezüglich der Kompressionsleistung gegenüber den vorgestellten Vergleichsverfahren fast immer besser abschneidet. In gewisser Weise wurde dies aber durch eine grössere Komplexität und eine erhöhte Laufzeit erkauft. Wenn auch die jetzigen Vergleiche diesbezüglich noch nicht sehr aussagekräftig sind, weil YAWICA bisher nur eingeschränkt unter Laufzeitgesichtspunkten optimiert wurde. Dennoch stellt YAWICA einen guten Ausgangspunkt für weitere Untersuchungen dar. Hier einige Punkte, die zukünftig noch verbessert werden könnten:

1. Es könnte zunächst untersucht werden, ob es Vorteile bringt, weitere benachbarte Koeffizienten bei der Kontextbildung zu berücksichtigen. Die bisherigen Ergebnisse lassen aber darauf schliessen, dass sich damit wohl nur marginale Verbesserungen erzielen lassen.

2. Die Quantisierung der übrigen Tupel ist zwar das Resultat eines experimentellen Auswahlprozesses, jedoch könnte hier ein anderer Ansatz bessere Ergebnisse liefern.

3. In [19] wird für verlustbehaftete Kompression vorgeschlagen, nach abgeschlossener Dekodierung, den Wert insignifikanter Koeffizienten zu extrapolieren. Dies ist aber erst dann sinnvoll, wenn sich eine Aussage über das Vorzeichen des Koeffizienten machen lässt. Durch die Kompression der Sign-Map wird dies möglich.

4. Bei höheren bpp-Werten S+P transformierter Bilder scheint SPIHT bei einigen Bildern Vorteile gegenüber YAWICA zu haben. Es könnte geprüft werden, welche Änderungen an YAWICA nötig sind, um diesen Rückstand wettzumachen.

5. Beim Schritt von Codec 1 zu Codec 2 wurde die Insignifikanz von ganzen Blöcken seperat kodiert. Dadurch haben sich zwar nur geringfügige Verbesserungen bei der Kompression ergeben, weil diese Blöcke aber überhaupt nicht durchlaufen werden mussten, konnte eine spürbare Beschleunigung erzielt werden (siehe Tabelle 5.18). Dies wird umso wichtiger je häufiger die Bitplane durchlaufen wird. Mit einer feineren Blockaufteilung könnte so eine weitere Beschleunigung erzielt werden. Die Kodierung der Blocksignifikanz könnte dabei in Anlehnung an [12] erfolgen.

Kapitel 7

Inhalt der CD

Die beiliegende CD enthält folgende Dateien bzw. Verzeichnisse:

`images/`
: In diesem Verzeichnis sind alle Testbilder im PGM-Format abgelegt.

`diploma-thesis.ps`
: Diese Arbeit im Postscript-Format.

`diploma-thesis.pdf`
: Diese Arbeit als PDF.

`yawica-0.01-20060318.tar.gz`
: Die Quelltexte zu allen sieben, in dieser Diplomarbeit vorgestellten, Codecs. Die Entwicklung erfolgte unter Linux, so dass auch nur auf dieser Plattform die Kompilation getestet wurde. Die Quelltexte sollten aber auch auf anderen UNIX-Systemen kompilierbar und lauffähig sein.

Kapitel 8

Verwendete Testbilder

Die hier aufgeführten Tabellen geben die mit YAWICA, SPIHT und EBCOT ermittelten Ergebnisse für die einzelnen Testbilder wieder. Die Bilder `bike.pgm`, `cafe.pgm` und `woman.pgm` haben die Auflösung 2048×2560, alle übrigen Bilder sind 512×512 gross. Zur Ermittlung der Ergebnisse wurden verschiedene Programme verwendet, die folgende Auflistung zeigt die dabei verwendeten Parameter:

1. Vergleich zweier Bilder, zur Ermittlung des PSNR:
   `# yawica -C image1.pgm image2.pgm`

2. Kompression eines Bildes mit YAWICA unter Verwendung des 9/7-Wavelets, die Dateigrösse von `image.yaw` darf dabei 0,5 bpp nicht überschreiten
   `# yawica -E -t 9/7 -p 0.5 image1.pgm image.yaw`
   Mit der Option `-t 5/3` bzw. `-t s+p` wird stattdessen das 5/3-Wavelet bzw. die S+P-Transformation verwendet. Soll die Kompression mit einer anderen bpp-Zahl erfolgen ist entsprechend der Wert hinter der Option `-p` anzupassen, wird die Option `-p <bpp>` ganz weggelassen erfolgt die verlustfreie Kompression

3. Dekompression eines mit YAWICA komprimierten Bildes:
   `# yawica -D image.yaw image2.pgm`
   Auch hier ist die Option `-p <bpp>` möglich, um nur einen Teil des komprimierten Datenstromes zu nutzen, dies ist beispielsweise sinnvoll, um für dasselbe Bild, aber unterschiedliche bpp-Werte, die zugehöhrigen PSNR-Werte zu ermitteln.

4. Kompression eines Bildes mit SPIHT (S+P-Transformation):
   `# progcode image1.raw image.sphit <yres> <xres> 1 <bpp> 0`
   Das Bild ist dabei im RAW-Format, also ohne PGM-Header, zu übergeben. Die Auflösung des Bildes ist beim Aufruf des Befehls mit anzugeben.

5. Dekompression eines mit SPIHT komprimierten Bildes (S+P-Transformation):
   `# progdecd -s image.spiht image2.raw <bpp>`

6. Kompression bzw. Dekompression eines Bildes mit SPIHT (9/7-Wavelet) erfolgen analog mit den Programmen `codetree` und `decdtree`.

7. Kompression eines Bildes mit EBCOT:
   `# kdu_compress -i image1.pgm -o image.ebcot -rate <bpp> -no_info`
   Für verlustfreie Kompression ist für `<bpp>` ein `-` anzugeben und die Option `Creversible=yes` hinzuzufügen. Mit dem Parameter `-no_info` werden nicht benötigte Zusatzinformationen aus dem Datenstrom entfernt. Aus der Dokumentation zu [21] war nicht klar zu entnehmen, welche Transformation für den verlustbehafteten Fall verwendet wird, deshalb ist diese Spalte nur mit "(lossy)" überschrieben.

8. Dekompression eines mit EBCOT komprimierten Bildes:
   `# kdu_expand -i image.ebcot -o image2.pgm`

8.1 airplane.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	28,44 dB	28,01 dB	28,69 dB	28,14 dB	28,16 dB	27,68 dB	28,16 dB
0,25 bpp	32,50 dB	32,41 dB	33,05 dB	32,26 dB	32,61 dB	32,14 dB	32,75 dB
0,50 bpp	36,22 dB	36,26 dB	37,14 dB	36,05 dB	36,61 dB	35,96 dB	36,65 dB
0,75 bpp	38,68 dB	38,81 dB	39,77 dB	38,41 dB	39,22 dB	38,49 dB	39,42 dB
1,00 bpp	39,93 dB	40,30 dB	41,74 dB	40,19 dB	41,16 dB	40,24 dB	41,28 dB
2,00 bpp	44,08 dB	44,96 dB	47,39 dB	45,46 dB	46,94 dB	44,80 dB	46,93 dB
verlustfrei	3,85 bpp	3,85 bpp	-	3,94 bpp	-	4,01 bpp	-

8.2 barbara.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	24,22 dB	23,97 dB	24,70 dB	23,89 dB	24,26 dB	24,02 dB	24,66 dB
0,25 bpp	27,56 dB	26,86 dB	28,45 dB	26,89 dB	27,58 dB	27,38 dB	28,36 dB
0,50 bpp	31,15 dB	30,51 dB	32,25 dB	30,57 dB	31,40 dB	30,95 dB	32,31 dB
0,75 bpp	34,00 dB	33,27 dB	35,04 dB	33,53 dB	34,26 dB	33,59 dB	34,88 dB
1,00 bpp	36,00 dB	35,28 dB	37,32 dB	35,53 dB	36,41 dB	35,85 dB	37,19 dB
2,00 bpp	41,62 dB	41,53 dB	43,50 dB	41,02 dB	42,65 dB	41,38 dB	43,14 dB
verlustfrei	4,57 bpp	4,61 bpp	-	4,71 bpp	-	4,78 bpp	-

8.3 bike.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	25,19 dB	24,58 dB	25,57 dB	24,67 dB	24,92 dB	24,98 dB	25,51 dB
0,25 bpp	29,33 dB	28,85 dB	29,79 dB	28,79 dB	29,12 dB	29,11 dB	29,65 dB
0,50 bpp	33,15 dB	32,78 dB	33,66 dB	32,64 dB	33,01 dB	32,98 dB	33,55 dB
0,75 bpp	35,36 dB	35,45 dB	36,30 dB	35,18 dB	35,65 dB	35,42 dB	36,04 dB
1,00 bpp	37,37 dB	37,50 dB	38,33 dB	36,98 dB	37,70 dB	37,35 dB	38,12 dB
2,00 bpp	42,42 dB	42,98 dB	44,41 dB	42,61 dB	43,80 dB	42,79 dB	44,02 dB
verlustfrei	4,36 bpp	4,35 bpp	-	4,48 bpp	-	4,53 bpp	-

8.4 boat.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	27,26 dB	27,30 dB	27,68 dB	27,07 dB	27,37 dB	26,90 dB	27,32 dB
0,25 bpp	30,89 dB	30,71 dB	31,41 dB	30,63 dB	30,97 dB	30,51 dB	31,04 dB
0,50 bpp	34,38 dB	34,17 dB	35,02 dB	34,05 dB	34,45 dB	33,96 dB	34,60 dB
0,75 bpp	36,59 dB	36,55 dB	37,74 dB	36,32 dB	36,97 dB	36,47 dB	37,28 dB
1,00 bpp	38,55 dB	38,74 dB	39,77 dB	38,40 dB	39,12 dB	38,53 dB	39,28 dB
2,00 bpp	42,77 dB	43,36 dB	45,39 dB	43,65 dB	44,77 dB	43,42 dB	44,72 dB
verlustfrei	4,22 bpp	4,20 bpp	-	4,34 bpp	-	4,40 bpp	-

8.5 cafe.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	20,06 dB	19,93 dB	20,37 dB	19,79 dB	20,04 dB	19,78 dB	20,19 dB
0,25 bpp	23,19 dB	22,67 dB	23,43 dB	22,74 dB	23,03 dB	22,73 dB	23,16 dB
0,50 bpp	26,83 dB	26,46 dB	27,24 dB	26,18 dB	26,49 dB	26,35 dB	26,84 dB
0,75 bpp	29,60 dB	29,37 dB	30,07 dB	29,02 dB	29,34 dB	29,16 dB	29,55 dB
1,00 bpp	32,05 dB	32,03 dB	32,50 dB	31,36 dB	31,74 dB	31,56 dB	32,07 dB
2,00 bpp	38,49 dB	38,54 dB	39,79 dB	38,22 dB	38,91 dB	38,43 dB	39,11 dB
verlustfrei	5,11 bpp	5,11 bpp	-	5,28 bpp	-	5,35 bpp	-

8.6 goldhill.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	27,82 dB	27,67 dB	28,12 dB	27,70 dB	27,94 dB	27,40 dB	27,83 dB
0,25 bpp	30,34 dB	30,29 dB	30,80 dB	30,23 dB	30,56 dB	30,16 dB	30,54 dB
0,50 bpp	32,89 dB	32,88 dB	33,49 dB	32,60 dB	33,13 dB	32,78 dB	33,24 dB
0,75 bpp	34,48 dB	34,59 dB	35,33 dB	34,37 dB	34,95 dB	34,60 dB	35,03 dB
1,00 bpp	36,06 dB	36,20 dB	36,90 dB	35,80 dB	36,55 dB	35,95 dB	36,58 dB
2,00 bpp	40,66 dB	40,91 dB	42,35 dB	40,53 dB	42,02 dB	40,70 dB	41,92 dB
verlustfrei	4,66 bpp	4,65 bpp	-	4,78 bpp	-	4,83 bpp	-

8.7 lena.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	30,04 dB	29,91 dB	30,52 dB	29,92 dB	30,23 dB	29,40 dB	29,98 dB
0,25 bpp	33,99 dB	33,54 dB	34,60 dB	33,69 dB	34,15 dB	33,32 dB	34,17 dB
0,50 bpp	36,55 dB	36,55 dB	37,68 dB	36,60 dB	37,25 dB	36,38 dB	37,32 dB
0,75 bpp	38,12 dB	38,37 dB	39,37 dB	38,36 dB	39,09 dB	38,13 dB	39,03 dB
1,00 bpp	39,13 dB	39,44 dB	40,76 dB	39,52 dB	40,46 dB	39,32 dB	40,42 dB
2,00 bpp	42,81 dB	43,33 dB	45,45 dB	44,05 dB	45,12 dB	43,42 dB	44,85 dB
verlustfrei	4,12 bpp	4,12 bpp	-	4,19 bpp	-	4,31 bpp	-

8.8 mandrill.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	21,30 dB	21,10 dB	21,45 dB	21,14 dB	21,35 dB	21,06 dB	21,32 dB
0,25 bpp	23,02 dB	22,78 dB	23,36 dB	22,93 dB	23,27 dB	22,83 dB	23,16 dB
0,50 bpp	25,46 dB	25,05 dB	25,82 dB	25,21 dB	25,65 dB	25,07 dB	25,58 dB
0,75 bpp	27,24 dB	27,14 dB	27,71 dB	27,12 dB	27,52 dB	27,01 dB	27,43 dB
1,00 bpp	28,88 dB	28,55 dB	29,45 dB	28,72 dB	29,17 dB	28,66 dB	29,11 dB
2,00 bpp	34,28 dB	34,12 dB	35,38 dB	34,20 dB	34,98 dB	34,13 dB	34,80 dB
verlustfrei	5,85 bpp	5,85 bpp	-	5,96 bpp	-	6,11 bpp	-

8.9 peppers.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	29,78 dB	29,68 dB	30,12 dB	29,55 dB	29,78 dB	29,23 dB	29,73 dB
0,25 bpp	33,32 dB	33,31 dB	33,91 dB	33,09 dB	33,47 dB	33,08 dB	33,53 dB
0,50 bpp	35,31 dB	35,59 dB	36,19 dB	35,34 dB	35,92 dB	35,40 dB	35,93 dB
0,75 bpp	36,61 dB	36,89 dB	37,58 dB	36,40 dB	37,25 dB	36,64 dB	37,22 dB
1,00 bpp	37,37 dB	37,84 dB	38,68 dB	37,61 dB	38,53 dB	37,79 dB	38,35 dB
2,00 bpp	41,58 dB	42,09 dB	43,57 dB	41,57 dB	43,29 dB	41,55 dB	43,13 dB
verlustfrei	4,45 bpp	4,41 bpp	-	4,58 bpp	-	4,62 bpp	-

8.10 woman.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	26,50 dB	26,36 dB	26,91 dB	26,35 dB	26,73 dB	26,19 dB	26,77 dB
0,25 bpp	29,82 dB	29,53 dB	30,32 dB	29,51 dB	29,95 dB	29,30 dB	30,01 dB
0,50 bpp	33,35 dB	33,03 dB	33,95 dB	33,07 dB	33,59 dB	32,88 dB	33,65 dB
0,75 bpp	35,77 dB	35,48 dB	36,75 dB	35,57 dB	36,18 dB	35,48 dB	36,26 dB
1,00 bpp	37,68 dB	37,69 dB	38,84 dB	37,53 dB	38,28 dB	37,54 dB	38,45 dB
2,00 bpp	42,43 dB	43,07 dB	44,52 dB	43,03 dB	43,99 dB	42,77 dB	44,02 dB
verlustfrei	4,31 bpp	4,31 bpp	-	4,42 bpp	-	4,51 bpp	-

8.11 zelda.pgm

	YAWICA			SPIHT		EBCOT	
	S+P	5/3	9/7	S+P	9/7	5/3	(lossy)
0,10 bpp	33,71 dB	33,68 dB	34,41 dB	33,64 dB	34,19 dB	33,17 dB	33,98 dB
0,25 bpp	36,80 dB	36,93 dB	37,80 dB	36,79 dB	37,50 dB	36,63 dB	37,35 dB
0,50 bpp	38,54 dB	39,00 dB	39,87 dB	38,73 dB	39,66 dB	38,79 dB	39,66 dB
0,75 bpp	39,51 dB	39,97 dB	41,17 dB	39,92 dB	40,94 dB	40,03 dB	40,97 dB
1,00 bpp	40,88 dB	41,10 dB	42,48 dB	40,78 dB	42,13 dB	40,93 dB	42,18 dB
2,00 bpp	43,63 dB	44,26 dB	47,13 dB	44,82 dB	46,84 dB	44,42 dB	46,67 dB
verlustfrei	3,84 bpp	3,82 bpp	-	3,93 bpp	-	3,99 bpp	-

Literaturverzeichnis

[1] Amir Said: *Introduction to Arithmetic Coding Theory and Practice.* Hewlett-Packard Laboratories Report, HPL-2004-76, Palo Alto, CA, 2004, `http://www.hpl.hp.com/techreports/` (zuletzt besucht im März 2006).

[2] Amir Said: *Fast Arithmetic Coding Implementations.* `http://www.cipr.rpi.edu/~said/FastAC.html` (zuletzt besucht im März 2006).

[3] Tilo Strutz: *Bilddatenkompression.* Verlag Vieweg, 2005, ISBN 3-528-13922-6.

[4] Martin Vitterli, Jelena Kovacevic: *Wavelets and Subband Coding.* Prentice Hall, 1995.

[5] Colm Mulcahy: *Plotting and Scheming with Wavelets.* `http://www.spelman.edu/~colm/wav.ps` (zuletzt besucht im März 2006).

[6] Rob Polikar: *Wavelet Tutorial.* `http://users.rowan.edu/~polikar/WAVELETS/WTtutorial.html` (zuletzt besucht im März 2006).

[7] Wim Sweldens: *The Lifting Scheme: a construction of second generation wavelets.* Siam J. Math. Anal, Vol. 29, Nr. 2, pp 511-546, 1997.

[8] Wim Sweldens: *Factoring wavelet transforms into lifting steps.* J. Fourier Anal. Appl., Vol. 4, Nr. 3, pp. 247-269, 1998.

[9] Amir Said, William A. Pearlman: *An Image Multiresolution Representation for Lossless and Lossy Compression.* IEEE Transactions on Image Processing, Volume 5, pages 1303-1310, 1996.

[10] Jerome M. Shapiro: *Embedded image coding using zerotrees of wavelet coefficients.* IEEE Transactions on Signal Processing, Volume 11, pages 3115-3162, 1993.

[11] Amir Said, William A. Pearlman: *A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees.* IEEE Transactions on Circuits and Systems for Video Technology, Volume 6, 1996.

[12] Asad Islam, William A. Pearlman: *An embedded and efficient low-complexity hierarchical image coder.* Proceedings of SPIE, Vol. 3653, pages 294-305, 1999.

[13] Ulug Bayazit, William A. Pearlman: *Algorithmic Modifications To SPIHT*. IEEE International Conference on Image Processing (ICIP), 2001.

[14] Frederick W. Wheeler, William A. Pearlman: *SPIHT Image Compression without Lists*. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2000.

[15] Edwin S. Hong, Richard E. Ladner: *Group Testing for Image Compression*. IEEE Transactions on Image Processing, Volume 11, 2002.

[16] Xiaolin Wu: *High-Order Context Modeling and Embedded Conditional Entropy Coding of Wavelet Coefficients for Image Compression*. Proceedings of 31st Asilomar Conference on Signals, Systems and Computers, pages 1378-1382, 1997.

[17] David Taubman, Erik Ordentlich, Marcelo Weinberger, Gadiel Seroussi: *Embedded Block Coding in JPEG2000*. Signal Processing – Image Communication, Volume 17, pages 49-72, 2002.

[18] David Taubman: *High Performance Scalable Image Compression with EBCOT*. Proceedings of International Conference on Image Processing, pages 344-348, 1999.

[19] Aaron Deever, Sheila S. Hemami: *What's Your Sign ? Efficient Sign Coding for Embedded Wavelet Image Coding*. Proceedings of Data Compression Conference, 2000.

[20] Amir Said, William A. Pearlman: *SPIHT Binaries*. `http://www.cipr.rpi.edu/research/SPIHT/spiht3.html` (zuletzt besucht im März 2006).

[21] David Taubman: *Kakadu Software*. `http://www.kakadusoftware.com/` (zuletzt besucht im März 2006).