R. Brigola, TH Nürnberg Georg Simon Ohm,
Fakultät Angewandte Mathematik, Physik und Allgemeinwissenschaften
rolf.brigola@th-nuernberg.de

Oktober 2013

Mathematica - Notebooks als Bonusmaterial zum Lehrbuch

[1] Rolf Brigola Fourier-Analysis und Distributionen,
Eine Einführung mit Anwendungen,
edition swk, Hamburg 2013

Teil 3 Grundwissen über die diskrete Fouriertransformation Beispiele zur diskreten Fouriertransformation (DFT, FFT) und zur diskreten Cosinustransformation (DCT I und DCT II)

Das Notebook ist mit Mathematica 9 unter Windows 7 erstellt. Dieses und weitere Demo - Notebooks findet man unter folgender URL des Autors:
www.stiftung-swk.de/mathematica

Dieses Notebook ist eine Fortführung des Notebooks Fourierreihen-Teil4.nb, das auf der o.g. URL zur Verfügung steht. Hier soll als Anwendung der DCT-2D (vom Typ II) Bilddatenkompression mit dem verbreiteten JPEG-Format betrachtet werden. Sie lernen dabei auch einige grundlegende *Mathematica*-Befehle für den Umgang mit Grafiken kennen. Lesen Sie bei Bedarf die zugehörigen Hilfe-Seiten von *Mathematica*.

2. Anwendungsbeispiele von DFT und DCT

2.2 Anwendungsbeispiel der DCT-2D vom Typ II bei JPEG, Verfälschungen (Artefakte) bei verlustbehafteter Datenkompression und ein Beispiel zur Huffman-Codierung

Das Demo-Beispiel bezieht sich auf [1], Abschnitt 5.7, Seiten 95-97. Dort findet man auch eine Beschreibung der Datenkompression mit JPEG durch Verwendung von Luminanzmatrizen im Spektralbereich eines Bildes.

1) Zur Demonstration wird zunächst ein einfaches Bitmap-Testbild “Testbild.bmp” mit einem Rechteck und einem Kreis in Schwarz auf weißem Hintergrund geladen. (Das Testbild findet man zum Download in ein zu setzendes User-Verzeichnis unter der oben schon genannten URL dieses Notebooks.

2) Dann wird als Bildausschnitt die obere linke Ecke des gezeichneten Rechtecks betrachtet und der entsprechende Ausschnitt der Grauwert-Matrix mit Integer-Werten zwischen 0 und 255 gezeigt. Der Wert 255 korrespondiert zu Weiß, der Wert Null zu Schwarz. Man erkennt in dieser Bilddaten-Matrix das schwarze linke obere Eck des Rechtecks wiedergegeben. Der Bildausschnitt wird dann vergrößert noch einmal gezeigt.

3) Anschließend erzeugen wir **mit dem bei *Mathematica* bereits implementierten JPEG-Algorithmus** eine komprimierte Version des Bildes. Die Größe schrumpft damit von ursprünglich 500 KB auf 9 KB. Wir sehen deutlich hier schon die Verfälschungen in der Umgebung der Kanten der beiden Figuren Rechteck und Kreis. Die Gründe liegen bei den ganzzahligen Näherungen für die DCT-Werte nach der Division durch die Werte in der Luminanzmatrix, in der "Requantisierung" mit der Luminanzmatrix vor der IDCT-2D und darin, dass die dann zurückgewonnenen Bilddaten in der Regel auch Werte haben, die außerhalb der Grauwert-Skala [0,255] (bei Byte-Darstellung) liegen, so dass diese Daten noch einmal neu gerastert werden müssen, um eine brauchbare Grauwert-Matrix zu erhalten. *Mathematica* macht das erforderliche Rastern automatisch.

Man kann es auch so beschreiben, dass durch die veränderten DCT-Werte ein trigonometrisches Polynom in zwei Variablen gegeben wird, welches in der Regel nun einen niedrigeren Grad hat als jenes, das mit den Koeffizienten der ursprünglichen DCT beschrieben wäre. Das Gibbs-Phänomen bei Fourierreihen-Entwicklungen zeigt dann etwa anschaulich (vgl. Notebook Fourierreihen-Teil1.nb auf dieser URL), dass die "Buckel der Überschwingungen" weiter von den Kanten wegrücken und Werte erzeugen, die in der Umgebung von Schwarz-Weiß-Kanten kleiner als Null und größer als 255 werden ("Überschwinger"). Das trigonometrische Polynom mit den neuen Koeffizienten, die nun von der veränderten DCT geliefert werden, interpoliert an den vorherigen Stellen also in der Regel andere Werte als die ursprünglichen gegebenen Grauwerte.

4) Wir sehen diesen Effekt ganz deutlich, indem wir wieder den vorher schon betrachteten Ausschnitt der Bilddaten-Matrix herausgreifen und diesen Ausschnitt noch einmal vergrößert ansehen.

Durch verlustbehaftete Datenkompression entstehende Verfälschungen, auch als Artefakte bezeichnet, sind also unvermeidlich.

Aus diesem Grund ist es z.B. nicht erlaubt, Bilder im medizinischen Bereich verlustbehaftet zu speichern, da es bei Diagnosen mit verfälschten Bilddaten leicht zu folgenschweren Fehldiagnosen kommen könnte. Gerade an den Kanten in den Bildern, in diesem Gebiet etwa Gefäß- oder Organränder, käme es ja zu starken Verfälschungen und dann ggf. zu vielleicht fatalen Fehleinschätzungen bei der Bildinterpretation.

In[1]:=
```
ClearAll["Global`*"]; Remove["Global`*"]; ?Global`*
```

Remove::rmnsm : There are no symbols matching "Global`*". ≫

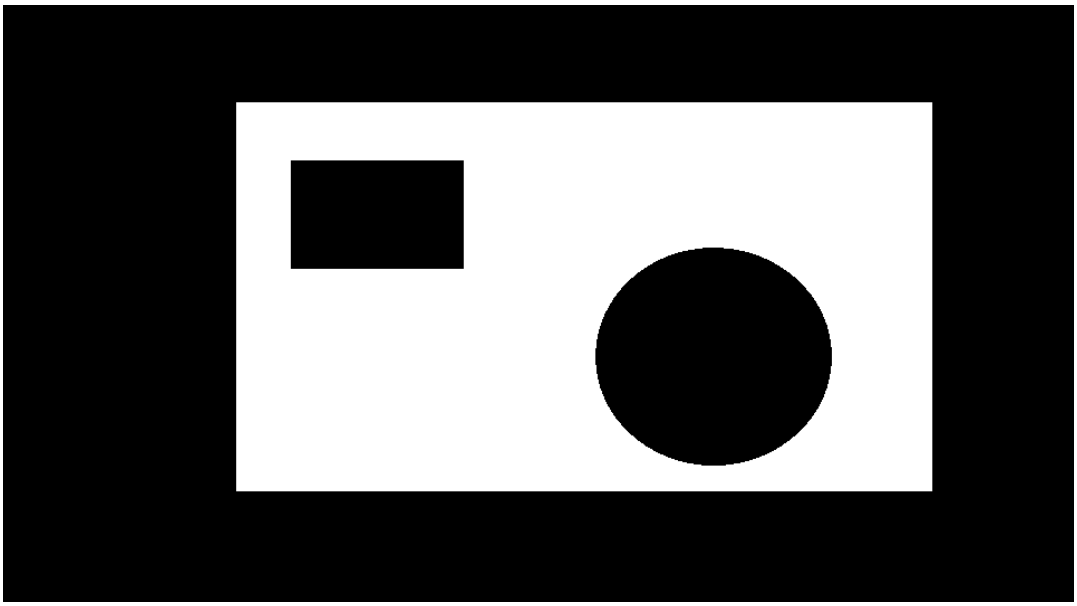
Information::nomatch : No symbol matching Global`* found. ≫

In[2]:=
```
SetDirectory["D:/Mathematica"]
(* Verwenden Sie Ihr eigenes spezifisches User-Verzeichnis *)
```

Out[2]=
```
D:\Mathematica
```

In[3]:=
```
testbild = Import["Testbild.bmp"] (* Wir importieren das Testbild *)
```

Out[3]=



In[4]:=
```
ImageDimensions[testbild] (* Check Pixel-Anzahl in beiden Richtungen *)
```

Out[4]=
```
{960, 534}
```

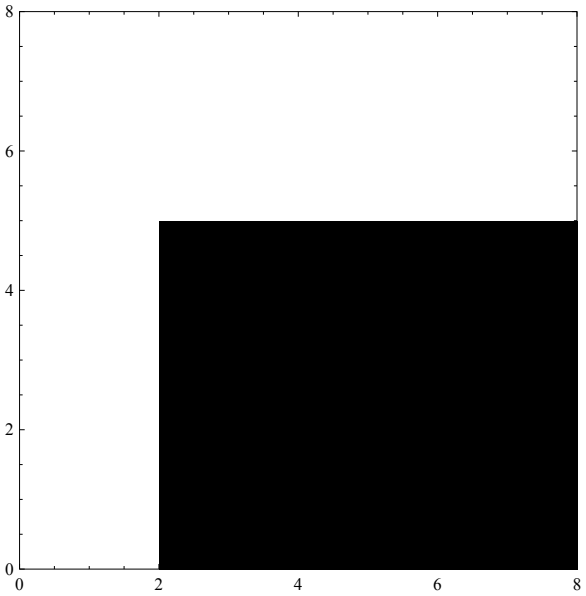
In[5]:=
```
bild2data = ImageData[ImageTake[testbild, {137, 144}, {257, 264}], "Byte"];
bild2data // MatrixForm (* 8x8-
  Pixelblock der Bilddaten: Ecke des Rechtecks links oben in Byte-Form *)
```

Out[6]//MatrixForm=

$$\begin{pmatrix} 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 255 & 255 & 255 & 255 & 255 & 255 \\ 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \\ 255 & 255 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
bild2 = Image[bild2data , Magnification → 20, ImageSize → 300];
Show[bild2, Frame → True, FrameStyle → Black]
(* Bildausschnitt Ecke des Rechtecks links oben,
vergrößert: Das ganze Rechteck zeigt 8x8 Pixel *)
```

Out[8]=



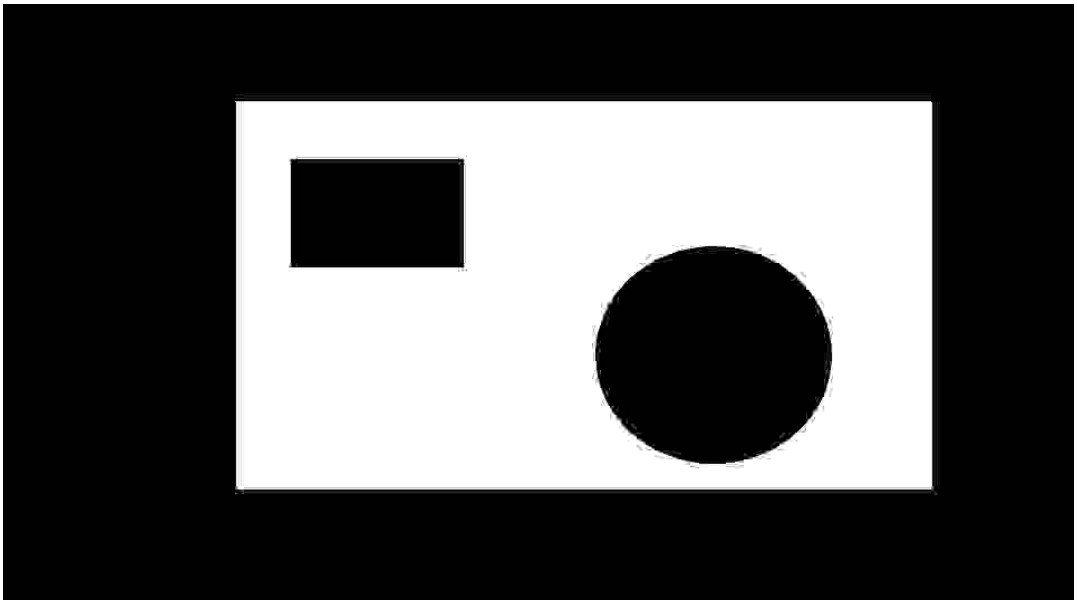
Nun das JPEG - komprimierte Testbild

In[9]:=
```
Export["testbild.jpeg", testbild, ColorSpace -> "Grayscale",
 "CompressionLevel" → 1.0] (* Exportiert als JPEG mit
  CompressionLevel 1 in das oben gesetzte User-Directory *)
```

Out[9]= testbild.jpeg

In[10]:=
```
testbildjpeg = Import["testbild.jpeg"]
 (* Größe des Bildes von ursprünglich 500 KB auf 9 KB komprimiert *)
 (* Hier das jpeg-komprimierte Bild. Man sieht
   deutlich Verfälschungen an den Rändern der Figuren *)
```

Out[10]=



In[11]:=
```
bild3data = ImageData[ImageTake[testbildjpeg, {137, 144}, {257, 264}], "Byte"];
bild3data // MatrixForm
 (*Der gleiche Ausschnitt wie oben, nun aus der neuen Bilddaten-
  Matrix. Auch hier sind die Veränderungen deutlich
   festzustellen: Vergleichen Sie mit den Grauwerten des Ausschnitts oben *)
```

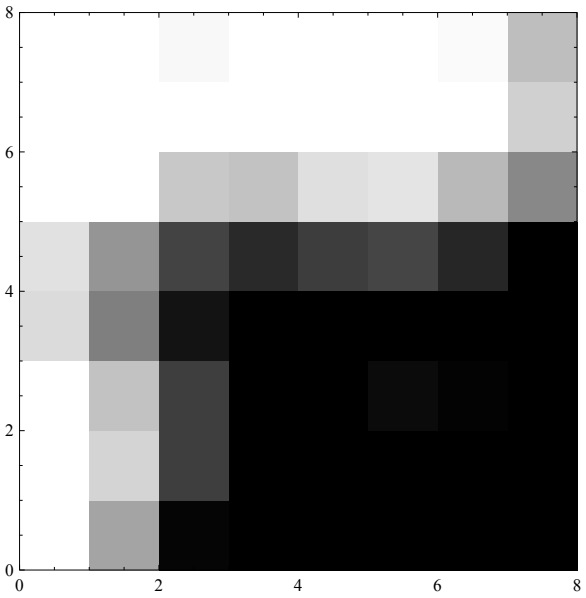
Out[12]//MatrixForm=

255	255	248	255	255	255	250	190
255	255	255	255	255	255	255	208
255	255	200	194	223	228	185	136
225	149	67	41	61	69	38	0
219	127	19	0	0	0	0	0
255	194	62	0	0	11	3	0
255	212	62	0	0	0	0	0
255	164	5	0	0	0	0	0

Hier noch einmal der Ausschnitt vergrößert als Bild zur Verdeutlichung der durch die verlustbehaftete JPEG - Kompression entstehenden Artefakte in der Umgebung der Bildkanten, hier des oberen linken Ecks unseres Rechtecks im Bild. Der Wechsel zwischen helleren und dunkleren Werten in dieser Umgebung entspricht dem “Oszillationsverhalten beim Gibbs-Phänomen” in der Umgebung von Sprungstellen bei Funktionen, wenn diese durch eine Partialsumme ihrer Fourierreihe approximativ ersetzt werden. Das JPEG-Bild liefert ein neues Raster von Abtastwerten einer solchen Näherung.

In[13]:=
```
bild3 = Image[bild3data, "Byte", ImageSize → 300] ;
Show[bild3, Frame → True, FrameStyle → Black]
(* Bildausschnitt Ecke des Rechtecks links oben *)
```

Out[14]=



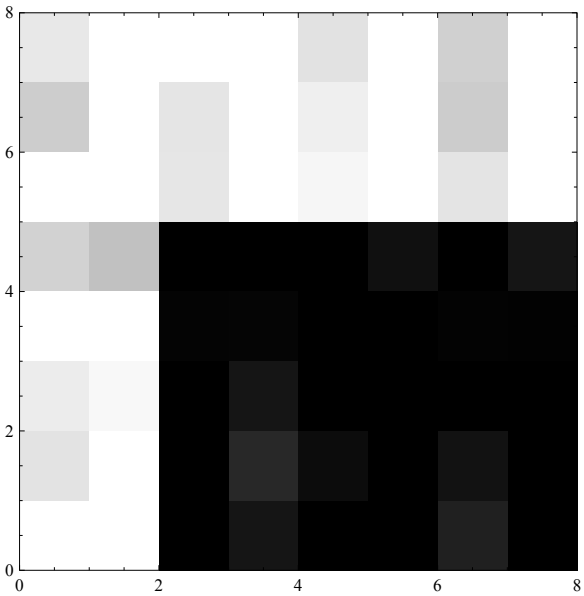
In[15]:=
```
Export["Bild3.bmp", bild3]
```

Out[15]= Bild3.bmp

Hier noch eine *Mathematica*-JPEG-Transformation nur des 8x8-Pixelblocks von bild2 oben. Auch hier sehen wir die Artefakte, diesmal wegen der Option “CompressionLevel->0.8 mit einer anderen Luminanzmatrix als weiter oben und daher mit veränderten Grauwerten im Vergleich.

In[16]:=
```
Export["bild2.jpeg", bild2, ColorSpace -> "Grayscale", "CompressionLevel" → 0.8];
bild2jpeg = Import["bild2.jpeg"];
Show[bild2jpeg, Frame → True, FrameStyle → Black, ImageSize → 300]
```

Out[17]=



Ganz analog werden Musikstücke verfälscht, wenn man sie komprimiert als MP3-Files speichert. Den Kanten in Bilddaten entsprechen in der Akustik starke Änderungen in der Dynamik. Entfernen von kleinen DCT-Koeffizienten - meist solche zu hohen Frequenzanteilen - verringert die Bandbreite und nimmt Obertöne weg. Für ein geschultes Gehör sind daher mp3-Wiedergaben schon eine rechte Qual. Um Ihnen einen Anhaltspunkt für die enormen Qualitätsunterschiede zwischen guten Analog-Aufnahmen von Musik, Digital-Aufnahmen mit 44.1 kHz Abtastfrequenz bei CD-Qualität im WAV-Format und MP3 zu geben, sei ein Bekannter von mir zitiert, der Tonmeister beim BR ist: Dort haben die Tonmeister einmal getestet, wie hoch die Abtastfrequenz einer Digitalaufnahme werden muss (etwa mit 24-Bit zur Quantisierung), so dass ein Tonmeister (mit gut geschultem Gehör) keinen Unterschied zu einer guten Analogaufnahme mehr hören kann. Das Ergebnis war, dass man dazu mit mehr als 200 kHz Abtastfrequenz arbeiten müsste. Gerade jungen Leuten rate ich daher, ihr Gehör nicht durch gewohnheitsmäßigen mp3-Konsum - und das im schlimmsten Fall dann auch noch laut - zu “verbilden”, sondern es durch echten live-Musikgenuss (am besten unplugged und nicht zu laut) zu erfreuen und gesund zu erhalten. Hörschädigungen sind nämlich leider irreversibel !

Die **Daten**, die tatsächlich im Datensegment **eines JPEG-Files** stehen, wenn Sie ein solches speichern oder an Freunde verschicken, **sind also DCT-Koeffizienten**.
Der Betrachter braucht daher immer ein Programm (Browser, Viewer), das mit einer IDCT aus den Daten wieder ein Bild generiert. Zur Rekonstruktion muss das Decoder-Programm natürlich auch die verwendete **Luminanzmatrix** kennen, die **in der JPEG-Datei mitgeliefert** wird (dort Segment FF DB).

Bei JPEG werden 8x8-Pixelblöcke wie oben transformiert, diese dann jeweils mit den Daten der für die gewünschte Kompressionsrate verwendeten Luminanzmatrix behandelt (dividiert und gerundet) und danach **mit einem Huffman-Code gespeichert**. Für Einzelheiten lese man bitte die JPEG-Seite von Wikipedia, die Referenz [5] unten und dort genannte weitere Quellen. *Erst durch die Codierung der Blöcke, die nun in der Regel viele Nullen enthalten, wird die eigentliche Datenkompression mit kleineren Bilddaten-Files erreicht.* Man siehe hierzu auch den Anhang am Ende des Notebooks.

Wir sehen uns das Prinzip am 8 x8 - Block unseres Bildausschnitts von oben mit einer der oft verwendeten Luminanzmatrizen noch einmal an :

In[18]:=
```
bild2data // MatrixForm
 (*Die Ausgangsdaten: Linke obere Ecke des scharzen Vierecks im Bild*)
```

Out[18]//MatrixForm=

255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	0	0	0	0	0	0
255	255	0	0	0	0	0	0
255	255	0	0	0	0	0	0
255	255	0	0	0	0	0	0
255	255	0	0	0	0	0	0

In[19]:=
```
dctblock = FourierDCT[bild2data]; dctblock // MatrixForm (* DCT des Blocks*)
```

Out[19]//MatrixForm=

1083.75	288.828	208.233	101.423	0.	−67.7686	−86.2531	−57.4515
452.847	−136.779	−98.6121	−48.0304	0.	32.0929	40.8465	27.207
176.692	−53.3685	−38.4765	−18.7405	0.	12.522	15.9375	10.6157
−65.8676	19.8948	14.3434	6.98613	0.	−4.66799	−5.94122	−3.95733
−135.234	40.8465	29.4487	14.3434	0.	−9.58393	−12.198	−8.12487
−44.0114	13.2933	9.58393	4.66799	0.	−3.11905	−3.9698	−2.6442
73.1882	−22.1059	−15.9375	−7.76258	0.	5.18679	6.60153	4.39715
90.0768	−27.207	−19.6152	−9.55383	0.	6.38367	8.12487	5.41181

In[20]:=
```
qLum = {{16, 11, 10, 16, 24, 40, 51, 61},
   {12, 12, 14, 19, 26, 58, 60, 55}, {14, 13, 16, 24, 40, 57, 69, 56},
   {14, 17, 22, 29, 51, 87, 80, 62}, {18, 22, 37, 56, 68, 109, 103, 77},
   {24, 35, 55, 64, 81, 104, 113, 92}, {49, 64, 78, 87, 103, 121, 120, 101},
   {72, 92, 95, 98, 112, 100, 103, 99}};
qLum // MatrixForm (* Hier eine Luminanzmatrix*)
```

Out[20]//MatrixForm=

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Jetzt die Division der DCT-Koeffizienten durch die entsprechenden Koeffizienten der Luminanzmatrix und Rundung auf ganze Zahlen (wir runden “zur Null hin”).

In[21]:=
```
jpegblock =
 Table[Sign[dctblock[[i, j]]] Floor[Abs[dctblock[[i, j]] / qLum[[i, j]]]],
  {i, 1, 8}, {j, 1, 8}]; jpegblock // MatrixForm
 (* Die resultierenden Daten für die JPEG-Speicherung mit nun vielen Nullen *)
```

Out[21]//MatrixForm=

$$\begin{pmatrix} 67 & 26 & 20 & 6 & 0 & -1 & -1 & 0 \\ 37 & -11 & -7 & -2 & 0 & 0 & 0 & 0 \\ 12 & -4 & -2 & 0 & 0 & 0 & 0 & 0 \\ -4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Genau diese Daten werden dann für das JPEG-Format mit einem **Huffman-Code** gespeichert. Man siehe dazu den Anhang am Ende des Notebooks.

Nur um eine erste Idee zu geben, sei dazu kurz gesagt: Durchläuft man die folgende Matrix jpegblock in einem Zickzack-Muster (wie man es von einer üblichen Nummerierung der rationalen Zahlen kennt), dann kommen im Beispiel am Ende 35 Nullen. Statt diese mit 35x8 =280 Bit zu speichern, können sie durch ein einziges “”EOB” Codewort (“end of block”) am Beginn der Null-Sequenz gespeichert werden, welches bedeutet, dass nun bis zum Blockende lauter Nullen kommen. Allein dadurch wird schon sehr viel Speicherplatz eingespart. Da ein Huffman-Code für eine Zeichenfolge nicht eindeutig bestimmt ist, hilft ein vereinbarter Standard für die Codierung und Decodierung.

Um einen solchen internationalen eindeutigen Standard für Codierung und Decodierung zu erreichen, wurden in CCITT-Gremien Codierungsempfehlungen vereinbart, mit denen jpeg-Bilder verarbeitet werden können. De facto wird im Header jeder JPEG-Datei im Segment FF C4 (DHT “Definition of Huffman Table”) genau definiert, mit welchem Huffman-Code die Daten codiert wurden. Jedes Viewer-Programm muss dann pro Bild damit eine zugehörige Code-Tabelle erzeugen, um richtig zu decodieren.

Die CCITT-Tabellen und weitere Einzelheiten, auch Matlab-Programme zur Huffman-Codierung und ihre Erläuterung, findet man gut beschrieben in den beiden Büchern [4] und [5] von R.C. Gonzales, R.E. Woods und S.L. Eddins, die am Ende des Notebooks angegeben sind. Weitere Informationen zur Huffman-Codierung finden Sie bei Interesse auch unter folgenden Links, beim letzten Link auch ein Notebook-Beispiel zu einer Huffman-Codierung eines Textes [6]:

http://mitpress.mit.edu/sicp/full-text/sicp/book/node41.html
http://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html
http://www.davidaltherr.net/mathematics/notebooks/huffman_encoding/huffman_encoding.php

Nun zur Bildrekonstruktion aus unserer obigen Matrix mit quantisierten DCT-Koeffizienten des Bildes:

Für die (nun verlustbehaftete) **Bildrekonstruktion** wird mit der gleichen Luminanztabelle requantisiert, d.h. die Elemente der letzten Matrix werden nun wieder elementweise mit den Koeffizienten der Luminanzmatrix multipliziert.

In[22]:=
```
blockrequant = Table[jpegblock qLum]; blockrequant // MatrixForm
  (* vergleichen Sie mit der Matrix dctblock weiter oben *)
```

Out[22]//MatrixForm=

1072	286	200	96	0	−40	−51	0
444	−132	−98	−38	0	0	0	0
168	−52	−32	0	0	0	0	0
−56	17	0	0	0	0	0	0
−126	22	0	0	0	0	0	0
−24	0	0	0	0	0	0	0
49	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0

In[23]:=
```
bildausschnitt = FourierDCT[blockrequant, 3];
bildausschnitt // MatrixForm
  (* Ansicht des Ergebnisses nach Rücktransformation *)
```

Out[24]//MatrixForm=

232.424	266.247	237.206	257.599	281.129	254.306	252.728	239.335
248.509	270.827	227.428	240.51	267.331	250.489	259.007	251.401
296.274	294.574	218.665	208.702	229.084	217.085	232.532	228.796
216.537	185.044	67.8716	26.0159	31.7552	16.3546	30.8313	25.7937
261.305	206.599	59.2583	−2.47993	−2.23629	−16.8252	−3.25102	−10.8073
274.018	207.74	48.739	−14.1592	−5.43848	−11.4411	4.25628	−5.12619
279.142	208.043	46.2578	−12.0838	5.72693	5.65908	20.5211	7.69597
270.516	196.841	33.1335	−24.2175	−4.27453	−4.80511	6.07901	−10.7755

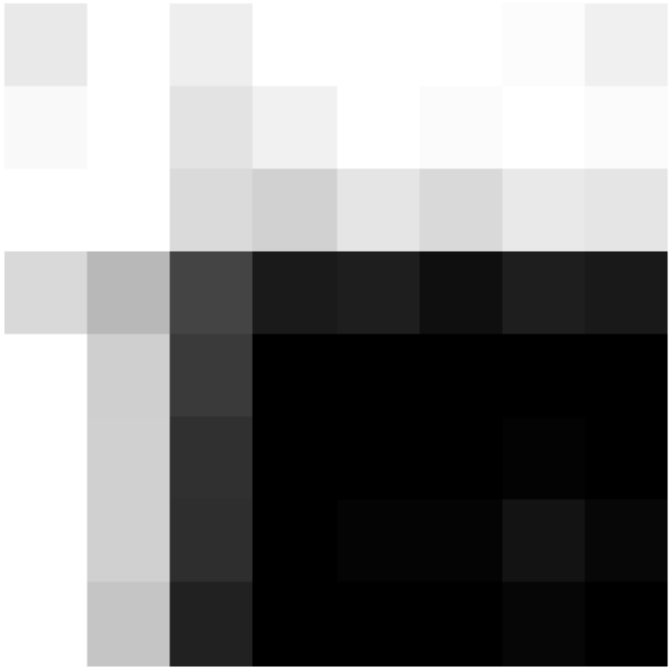
Wir sehen den bereits erwähnten “Gibbs-Effekt” mit Real-Werten außerhalb der Grauwert-Skala mit Integer-Zahlen von Null bis 255 (*negative Werte, nicht-ganzzahlige Werte, Werte größer als 255*). Die erforderliche neue Rasterung überlassen wir zunächst *Mathematica* mit dem **Raster-Befehl**.
Wir rastern danach selbst nochmal in einfacher Weise, sehen uns dann die nach diesen Rücktransformationen erzeugte Grauwertmatrix des betrachteten Bildausschnitts an und auch das zugehörige Bild in Vergrößerung.

Man vergleiche zum Abschluss die erreichte Grauwertmatrix unseres Ausschnitts auch noch einmal mit bild2data und bild3data weiter oben.

Zunächst der nun von *Mathematica* gerasterte Bildausschnitt:

In[25]:=
```
ergebnis = Image[Graphics[Raster[bildausschnitt, {{0, 8}, {8, 0}}, {0, 255}]],
  ColorSpace → "Grayscale"]
```

Out[25]=



Nun rastern wir selbst, indem wir die Werte in “bildausschnitt” auf ganze Zahlen abrunden, wobei Werte<0 einfach auf Null und Werte >255 auf 255 gesetzt werden. Damit erhalten wir eine Grauwert-Matrix unseres JPEG - Ergebnisses:

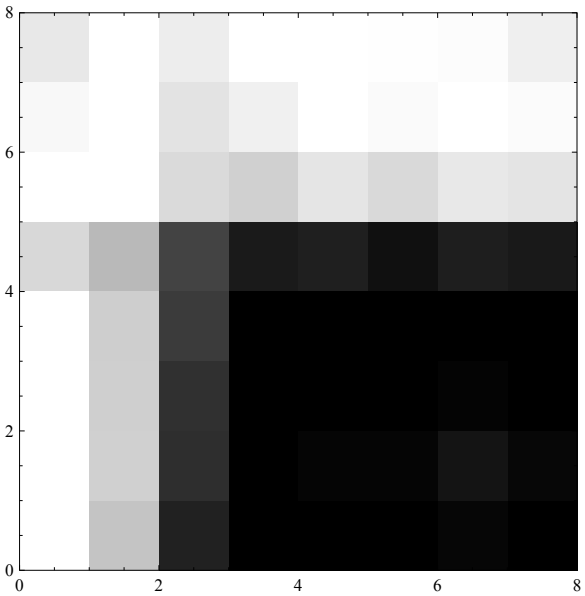
In[26]:=
```
bildraster1 = Table[Max[{bildausschnitt[[i, j]], 0}], {i, 1, 8}, {j, 1, 8}];
bildgerastert =
  Table[Floor[Min[{bildraster1[[i, j]], 255}]], {i, 1, 8}, {j, 1, 8}];
bildgerastert // MatrixForm
```

Out[28]//MatrixForm=

232	255	237	255	255	254	252	239
248	255	227	240	255	250	255	251
255	255	218	208	229	217	232	228
216	185	67	26	31	16	30	25
255	206	59	0	0	0	0	0
255	207	48	0	0	0	4	0
255	208	46	0	5	5	20	7
255	196	33	0	0	0	6	0

In[29]:=
```
jpegbildgerastert = Image[bildgerastert, "Byte", ImageSize → 300] ;
Show[jpegbildgerastert, Frame → True, FrameStyle → Black]
```

Out[30]=



Bei Farbbildern wird das Ausgangsbild, welches oft als RGB-Bild vorliegt, in den YCbCr-Farbraum umgerechnet und dann für jede der 3 zugehörigen Komponenten zur JPEG-Komprimierung ganz analog wie gezeigt verfahren.

==========

Anhang zur Huffman - Codierung

Wir wollen nachfolgend noch ein Beispiel ansehen, wie man einen möglichen Huffman-Code für die Elemente in “jpegblock” konstruieren kann. Wir berücksichtigen dabei keine Umordnung der Elemente und keine Lauflängen. Das Beispiel soll nur das **Prinzip einer Huffman-Codierung** erläutern.

Zunächst “flatten” wir unsere 8x8-Matrix jpegblock. Die resultierende Liste hat dann 64 Elemente.

In[31]:=
```
data = Flatten[jpegblock]
```

Out[31]=
```
{67, 26, 20, 6, 0, -1, -1, 0, 37, -11, -7, -2, 0, 0, 0, 0, 12,
 -4, -2, 0, 0, 0, 0, 0, -4, 1, 0, 0, 0, 0, 0, 0, -7, 1, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
```

Eine Huffman-Codierung geschieht in 3 Schritten, die Decodierung in einem Schritt.

1) Zuerst wird eine Häufigkeitstabelle (Character Frequency Table) der vorkommenden Koeffizienten in der Liste erstellt. Die Ergebnisliste enthält, nach wachsenden Häufigkeiten geordnet, die vorkommenden Koeffizienten und deren Häufigkeit, im Beispiel etwa {2,-7} als Listeneintrag, weil -7 zweimal in der Koeffizientenmatrix von jpegblock vorkommt.

Nachfolgend die dazu verwendete Funktion:

(Die Funktion ist eine Modifikation einer im oben schon genannten Link http://www.davidaltherr.net/mathematics/notebooks/huffman_encoding/huffman_encoding.php gezeigten Funktion, weil wir an Stelle von Textzeichen wie dort mit mehrstelligen Integer-Zahlen zu tun haben, die jeweils als ein einziges Zeichen codiert werden sollen.)

In[32]:=
```
getCharFreqTable[data_List] :=
  Sort[{Count[Flatten[# & /@ data], #[[1]]], #[[1]]} & /@
    Transpose[{Union[Flatten[# & /@ data]]}]];
```

Hier das Ergebnis mit unserem Beispiel:

In[33]:=
```
getCharFreqTable[data]
```

Out[33]=
```
{{1, -11}, {1, 6}, {1, 12}, {1, 20}, {1, 26}, {1, 37},
 {1, 67}, {2, -7}, {2, -4}, {2, -2}, {3, -1}, {4, 1}, {44, 0}}
```

2) Aus dieser Liste wird nun ein Baum (**Huffman Tree**) erzeugt, in dem Knoten und Kanten durch einen Binärstring dargestellt werden können. Es gibt stets genau 2 Pfade, die wir von einem Knoten aus zu einem Blatt oder einem weiteren Knoten wählen können. Diese werden dargestellt durch eine 1 oder eine 0. Die optimale Baumstruktur wird durch eine Rekursion (Reduktion) erreicht, welche die Koeffizienten anhand ihrer jeweiligen Häufigkeit verarbeitet.

In[34]:=
```
getHuffmanTree[data_List] :=
 Nest[Sort[Delete[ReplacePart[#, {Plus @@ (Transpose[Take[#, {1, 2}]][[1]]),
         Take[#, {1, 2}]}, {1}], {2}]] &,
   getCharFreqTable[data], Length[getCharFreqTable[data]] - 1][[1, 2]]
```

Diese Funktion arbeitet folgendermaßen:

Der Prozess startet damit, die ersten beiden Koeffizienten der geordneten Häufigkeits-tabelle oben in eine Zeichenmenge unter einen Knoten zu stellen. Die Häufigkeit dieser Zeichenmenge wird dann als die Summe der Häufigkeiten der enthaltenen Zeichen definiert. Im Beispiel mit {1,-11} und {1,6} ergibt sich also mit 2 als Summe der beiden Häufigkeiten in diesen Elementen

In[35]:=
```
{Plus @@ (Transpose[Take[#, {1, 2}]][[1]]), Take[#, {1, 2}]} &@
 getCharFreqTable[data]
```

Out[35]=
```
{2, {{1, -11}, {1, 6}}}
```

Das Ergebnis wird dann in die vorherige Häufigkeitstabelle an Stelle der beiden ersten Einträge substituiert. Das Ergebnis im Beispiel ist:

In[36]:=
```
(Delete[ReplacePart[#,
     {Plus @@ (Transpose[Take[#, {1, 2}]][[1]]), Take[#, {1, 2}]}, {1}], {2}] &@
  getCharFreqTable[data])(*[[Range[1,12]]]*)
```

Out[36]=
```
{{2, {{1, -11}, {1, 6}}}, {1, 12}, {1, 20}, {1, 26}, {1, 37},
 {1, 67}, {2, -7}, {2, -4}, {2, -2}, {3, -1}, {4, 1}, {44, 0}}
```

Nun wird neu nach wachsenden Häufigkeiten sortiert:

In[37]:=
```
(Sort[Delete[ReplacePart[#,
      {Plus @@ (Transpose[Take[#, {1, 2}]][[1]]), Take[#, {1, 2}]}, {1}], {2}]] &@
  getCharFreqTable[data])(*[[Range[1,12]]]*)
```

Out[37]=
```
{{1, 12}, {1, 20}, {1, 26}, {1, 37}, {1, 67}, {2, -7},
 {2, -4}, {2, -2}, {2, {{1, -11}, {1, 6}}}, {3, -1}, {4, 1}, {44, 0}}
```

Dieser Prozess wird fortgeführt, indem jeweils die beiden ersten resultierenden Einträge, ob einzelner Koeffizient oder schon eine Gruppe von solchen, unter einen Knoten gesetzt werden.

***Die Rekursion endet mit dem vollen Huffman Tree**, hier in Form einer "nested list", die wir "hTree" taufen.*

In[38]:=
```
hTree = getHuffmanTree[data]
```

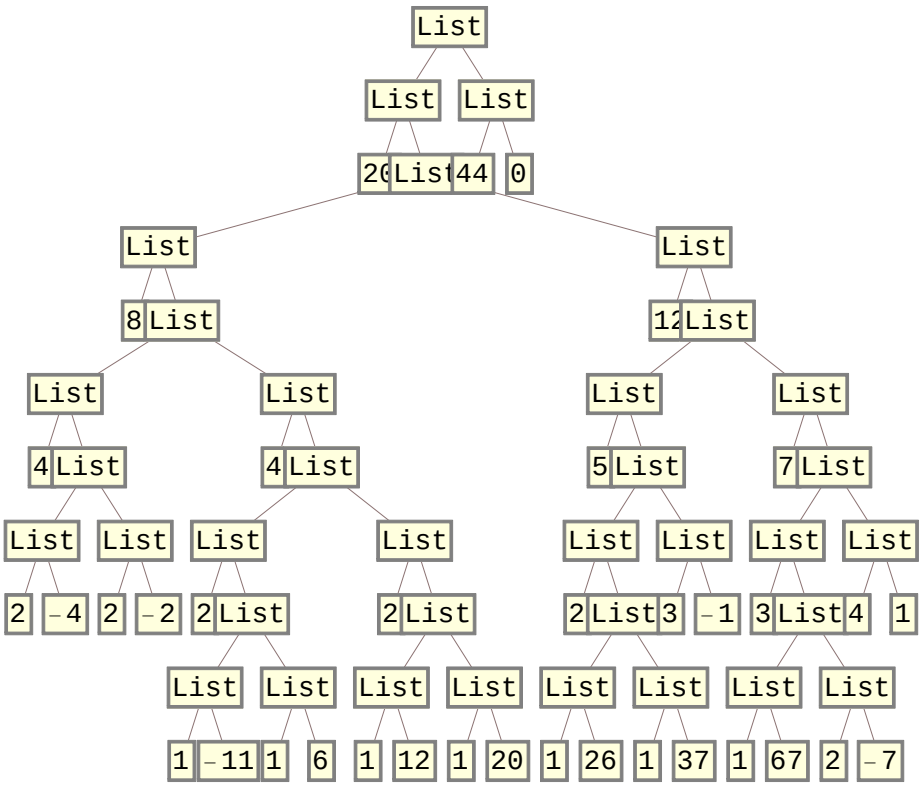
Out[38]=
```
{{20, {{8, {{4, {{2, -4}, {2, -2}}},
      {4, {{2, {{1, -11}, {1, 6}}}, {2, {{1, 12}, {1, 20}}}}}}}},
   {12, {{5, {{2, {{1, 26}, {1, 37}}}, {3, -1}}},
     {7, {{3, {{1, 67}, {2, -7}}}, {4, 1}}}}}}}, {44, 0}}
```

Diese "nested list" stellen wir nun mit *Mathematica* als Baum dar. Aus der Baumstruktur kann man direkt einen Huffman-Code erzeugen. Äquivalent dazu kann man natürlich direkt auch die Darstellung von hTree als "nested list" dafür verwenden.

Sie merken dabei, **dass der Code nicht eindeutig festgelegt ist**, weshalb es für einen einheitlichen Standard beim JPEG die CCITT-Tabellen gibt. Man könnte zum Beispiel den Baum anders anlegen und damit die Codewörter für Koeffizienten gleicher Häufigkeit untereinander austauschen oder auch alle 0er und 1er in jedem Codewort tauschen.

In[39]:=
```
TreeForm[hTree, ImageSize → Large]
```

Out[39]//TreeForm=



3) Nachfolgend, wie man daraus einen Huffman-Code generieren kann:

Der Code eines Zeichens unserer Daten ist eindeutig durch seine Position in hTree bestimmt. Aus den Positionen der Zeichen generieren wir Codewörter und damit eine zu "data" analoge "Codeliste".
Danach geben wir den Bitstrom aus, der die ganze Liste codiert. Sie sehen, dass Daten-Elemente umso kürzere Codewörter erhalten je größer ihre Häufigkeit in der zu codierenden Liste ist.
(**Aufgabe**: Analysieren Sie, was die beiden nachfolgenden Funktionen bewirken.)

In[40]:=
```
encodeChar[c_, tree_List] := (list = Flatten[Position[tree, {_, c}]];
  b = Table[ToString[list[[i]] - 1], {i, 1, Length[list], 2}]; StringJoin[b])
encode[charlist_, tree_] := Table[encodeChar[charlist[[n]], tree],
  {n, Length[charlist]}]
```

Mit diesen beiden Funktionen können wir die Daten nun codieren. Die erzeugte Liste enthält statt der ursprünglichen Matrix-Elemente von "jpegblock" nun deren Codewörter als Binärstrings.

In[42]:=
```
codelist = encode[data, hTree]
```

Out[42]=
```
{01100, 01000, 00111, 00101, 1, 0101, 0101, 1, 01001, 00100, 01101, 0001, 1, 1, 1, 1,
 00110, 0000, 0001, 1, 1, 1, 1, 1, 0000, 0111, 1, 1, 1, 1, 1, 1, 01101, 0111, 1, 1, 1,
 1, 1, 1, 0101, 1, 1, 1, 1, 1, 1, 1, 0111, 1, 1, 1, 1, 1, 1, 1, 0111, 1, 1, 1, 1, 1, 1, 1}
```

Schließlich der ganze benögte Bitstrom, der aus den Codewörtern entsteht und seine Länge:

In[43]:=
```
bitstream = {StringJoin[codelist]}
StringLength[bitstream[[1]]]
```

Out[43]=
```
{0110001000001110010110101010110100100100011010001111100110000000011111100000111111111011010111111111010111111111011111111110111111111}
```

Out[44]=
```
133
```

Kurzes Nachrechnen ergibt: Codierung der 64 Zahlen als 8-Bit-Wörter bräuchte **512 Bit**. Codieren wie oben würde einen Bitstrom mit **nur 133 Bits Länge** benötigen, also nur noch etwa 1/4 des vorherigen Speicherplatzes verbrauchen.

Eine geschicktere Anordnung nach einem Zickzackmuster wie oben schon erwähnt und Darstellung der dann resultierenden Abschluss-Sequenz mit 35 Nullen durch ein einziges Codewort würde noch einmal etwa 26% einsparen. Die CCITT-Codierung benutzt de facto Difference Coding, ein Zickzack-Schema pro Block und Lauflängen-Code (siehe Referenzen).

Nun zur Decodierung:

Wir erzeugen zunächst eine Liste, welche die Zuordnung der Matrix-Koeffizienten und ihrer Codewörter paarweise zeigt. Die Liste nennen wir “translationTable”.

In[45]:=
```
huffmanTable[data_List, code_List] :=
 Table[{data[[i]], code[[i]]}, {i, Length[data]}]
```

In[46]:=
```
translationTable = Union[huffmanTable[data, codelist]]
```

Out[46]=
```
{{-11, 00100}, {-7, 01101}, {-4, 0000}, {-2, 0001}, {-1, 0101}, {0, 1}, {1, 0111},
 {6, 00101}, {12, 00110}, {20, 00111}, {26, 01000}, {37, 01001}, {67, 01100}}
```

Unser nachfolgendes Decodierungsergebnis speichern wir in der Liste “ergebnis”. Sie muss im Beispiel 64 Einträge haben und wird nachfolgend angelegt. Um unseren Bitstream zu belassen, kopieren wir ihn insgesamt nach “stream”.

In[47]:=
```
ergebnis = Table[NULL, {n, 64}]; stream = bitstream;
```

Nun - als Schleife programmiert - arbeiten wir den Bitstream ab, indem wir solange Zeichen der 0-1-Folge mit dem StringTake-Befehl zusammenfassen bis wir das erste Codewort in translationTable finden. Es steht dort in der zweiten Komponente eines Treffers. Dann weisen wir die erste Komponente dieses Treffers - das codierte Zeichen - der Ergebniskomponente “ergebnis[[1]]” zu und löschen mit StringDrop die decodierte Bitfolge aus “stream”.
Das Verfahren wird dann weiter in einer Schleife so oft wiederholt bis alle Einträge “ergebnis[[n]]” für alle n, $1 \le n \le 64$ vorliegen.

In[48]:=
```
For[n = 1, n ≤ 64, n++,
 For[m = 1, m ≤ 64 && ergebnis[[n]] == NULL, m++,
  ( a := StringTake[stream, m];
   If[MemberQ[translationTable, {_, a[[1]]}],
     (b := Position[translationTable, {_, a[[1]]}];
      ergebnis[[n]] = translationTable[[b[[1]]]][[1]][[1]];
      stream = StringDrop[stream, m];)] )]]
```

Hier das erzielte Ergebnis in Form einer Liste, die anschließend wieder in eine 8x8-Matrix umgewandelt wird. Sie stimmt mit der Matrix “jpegblock//MatrixForm” überein, d.h. wir haben richtig decodiert. .

In[49]:=
```
ergebnis
```

Out[49]=
```
{67, 26, 20, 6, 0, -1, -1, 0, 37, -11, -7, -2, 0, 0, 0, 0, 12,
 -4, -2, 0, 0, 0, 0, 0, -4, 1, 0, 0, 0, 0, 0, 0, -7, 1, 0, 0, 0, 0, 0, 0,
 -1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0}
```

In[50]:=
```
ArrayReshape[ergebnis, {8, 8}] // MatrixForm
```

Out[50]//MatrixForm=

67	26	20	6	0	−1	−1	0
37	−11	−7	−2	0	0	0	0
12	−4	−2	0	0	0	0	0
−4	1	0	0	0	0	0	0
−7	1	0	0	0	0	0	0
−1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

In[51]:=
```
jpegblock // MatrixForm
```

Out[51]//MatrixForm=

67	26	20	6	0	−1	−1	0
37	−11	−7	−2	0	0	0	0
12	−4	−2	0	0	0	0	0
−4	1	0	0	0	0	0	0
−7	1	0	0	0	0	0	0
−1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Schlussbemerkung: Wenn Sie mit Ihren *Mathematica*-Kenntnissen schon fortgeschrittener sind, finden Sie vielleicht die oben verwendete Schleifenprogrammierung zur Decodierung zu “old-fashioned”. Dann überlegen Sie vielleicht, wie man das noch eleganter hinbekommt, etwa durch Verwendung eines rekursiven Algorithmus.

Zur vertiefenden Lektüre empfehle ich noch

[1] Rolf Brigola Fourier-Analysis und Distributionen, Eine Einführung mit Anwendungen, edition swk, Hamburg 2013

[2] Andrew B. Watson Image Compression Using the Discrete Cosine Transform *Mathematica* Journal, 4(1), 1994, p. 81-88, zu finden unter http://vision.arc.nasa.gov/publications/mathjournal94.pdf

[3] CCITT Recommendation T.81, zu finden unter http://www.jpeg.org/jpeg/

[4] R.C. Gonzales, R.E. Woods, S.L. Eddins Digital Image Processing Using Matlab
Pearson, Prentice Hall, 2004
[5] R.C. Gonzales, R.E. Woods Digital Image Processing, 3rd ed.
Pearson, Prentice Hall, 2008
[6] David A. Altherr huffman binary encoding
http://www.davidaltherr.net/mathematics/notebooks/huffman_encoding/huffman_encoding.php