operations on JPEG images

E.F. de Vries G.P. Kumara

September 20, 1994

Contents

List of Figures

Chapter 1

Introduction

This report describes the implementation of a family of algorithms, that can perform image operations directly on the compressed data of the image. The time needed, to execute these operations directly on the compressed data of the image, should be less than the so called 'brute force' version of the operation. This can be achieved, because of the following reasons:

- a compressed image contains substantially less data than its unpacked counterpart.
- no decompression and compression has to be performed.

The compression algorithm used in this project is developed by the Joint Photographic Expert Group (JPEG). The JPEG standard is developed for continuous-tone (real-life) images. Using a JPEG compression algorithm for non continuous-tone images might result in a 'compressed image' that is actually larger than its unpacked version. The JPEG standard includes two basic compression methods.

1. a predictive method for 'lossless' compression. (a description of this method can be found in [Wall])
2. a DCT-based method for lossy compression. (for a description of this method see Section 2.1)

The JPEG compression method used in the project is a DCT-based method.

1.1 The operations

scalar multiplication

Intuitively, scalar multiplication will give a change in contrast in the image. In the YCbCr colour space, the first component determines the (greyscale) 'scene' of the image. Differences of neighbouring values are scaled by a given factor, and therefore are increased (if the factor > 1) or decreased (if the factor < 1), with a change in contrast as a result. For an example, see Figure 1.1.

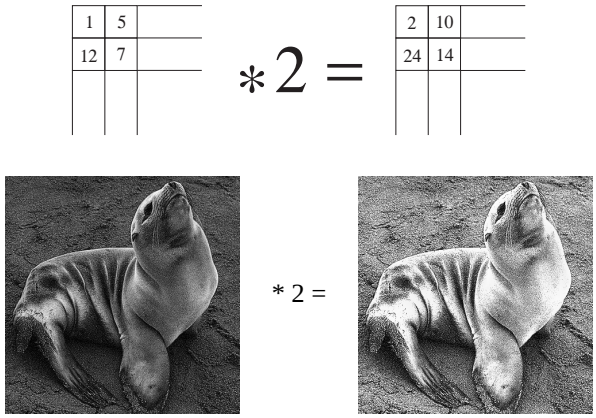


Figure 1.1: Scalar multiplication in practice.

scalar addition

Scalar addition adds the same factor β to all elements in a component (each component can have it's own add factor). This will result in a brighter (factor > 0) or darker image (factor < 0) (see Figure 1.2).

This operation can be used before scalar multiplication, to maximize the effect of contrast stretching.

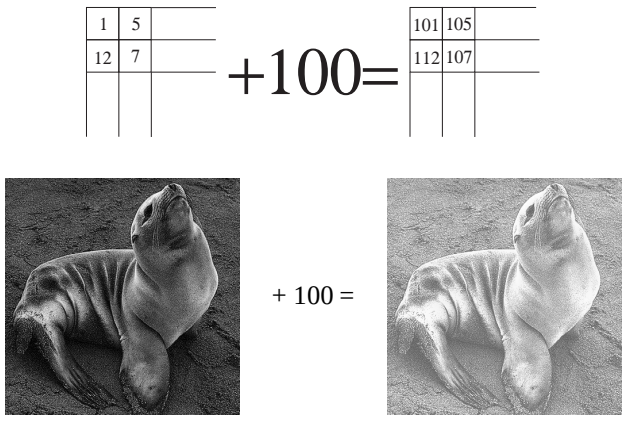


Figure 1.2: Scalar addition in practice.

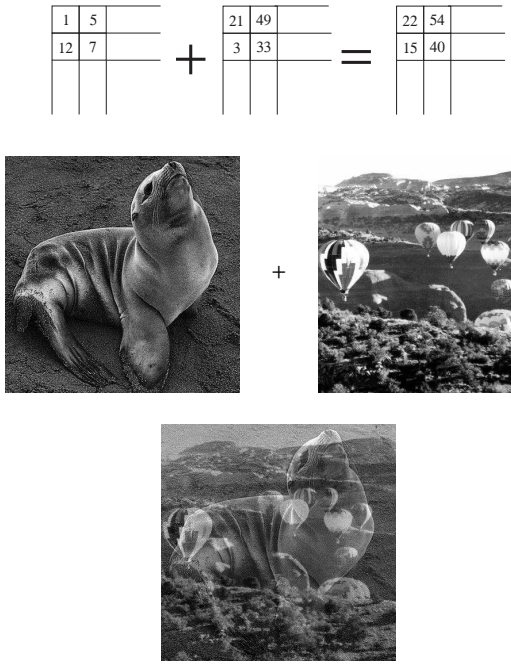


Figure 1.3: Pixel addition in practice.

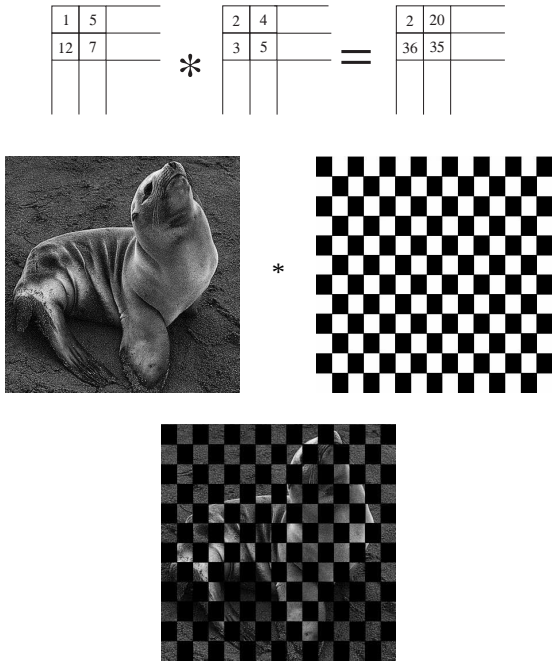


Figure 1.4: Pixel multiplication in practice.

pixel addition

Two images are added pixel by pixel. The resulting image is a mix of the two input images. This might be useful for subtitling (add an image of a real life scene to an image of the subtitle) or dissolving one image into another.

Figure 1.3 shows pixel addition in practice.

pixel multiplication

Pixel multiplication can be used to mask off regions in an image. Another use of pixel multiplication is image enhancement. Taking the 'square of an image' applying the correct scaling factors can give interesting results.

Figure 1.4 shows an example of pixel multiplication.

1.2 Terminology & notation

In this section we will introduce the notations and terminology used in this report. First we introduce the notations we use for (sub)images. With x_1, x_2 and y we denote the normalized 8x8 matrices in the spatial domain. X_1,X_2 and Y are used to represent x_1, x_2 and y resp. in the frequency domain. The subscript 'Q' is used whenever we are dealing with the quantized version of a matrix. The subscript 'zz' is used for zigzagged ordered versions of the matrices. Note that this subscript implies a vector instead of a matrix. For an example of our notation see Figure 1.5.

We use i and j for the indices of a block in the spatial domain ($x_1[i,j]$). u_1, u_2, v_1, v_2, w_1 and w_2 are used for indices in the frequency domain ($X_1[v_1,v_2]$). u is the zigzag ordered (Subsection 2.1.5) counterpart of (u_1,u_2). v and w are defined likewise (see Subsection 2.1.3 and Subsection 4.6.1).

x_1	:	spatial domain
X_1	:	frequency domain
$X_{1,Q}$	:	quantized X_1
$X_{1,Q,zz}$	:	zigzag ordered vector of $X_{1,Q}$

Figure 1.5: An example of our notation.

Chapter 2

JPEG coding

In this chapter we will present everything needed to understand our implementation of the operations given in [Smith]. This knowledge not only includes the JPEG algorithm, but also some insight in the Discrete Cosine Transform and quantization tables.

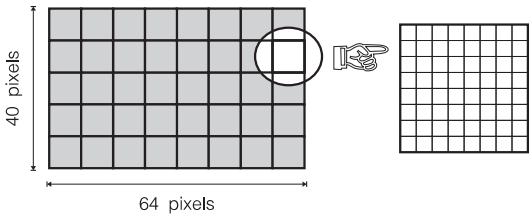
2.1 JPEG algorithm

In this section, we give an overview of the JPEG algorithm. A complete description can be found in [Wall].

Suppose we are dealing with images in the YCbCr colour space; the first component determines the luminance, the second and third component determine the chrominance. Suppose also that each value in all components is an eight bit value. These assumptions are realistic ones since most JPEG-images are distributed in the *JFIF-standard* which uses the YCbCr colour space as a standard. The JFIF-standard will be briefly described later in this report. See Chapter 6.

2.1.1 Block division

The first step divides all components in submatrices of 8x8. Components that do not have sizes that are a multiple of 8 are padded with zeroes on the right and bottom side of the image. These submatrices are called *blocks* in JPEG-terminology.



2.1.2 Normalization

The second step in the JPEG-algorithm is to *normalize* the blocks i.e. all values in all blocks of all components should be in the range $-128 \ldots 127$. In our colour space, we only need to normalize the blocks of the first component (which have values in the range $0 \ldots 255$), since the blocks of the second and third component are already in the correct range.

166	166	166	166	166	165	166	166
166	166	166	168	166	164	166	167
166	166	166	165	166	167	166	166
166	164	166	166	165	166	166	166
166	166	167	166	166	98	166	166
167	166	168	166	166	98	98	98
166	166	166	166	166	98	108	108
166	165	164	166	166	98	108	108

Normalisatie

38	38	38	38	38	37	38	38
38	38	38	40	38	36	38	39
38	38	38	37	38	39	38	38
38	36	38	38	37	38	38	38
38	38	39	38	38	-30	38	38
39	38	40	38	38	-30	-30	-30
38	38	38	38	38	-30	-20	-20
38	37	36	38	38	-30	-20	-20

Figure 2.1: Normalization.

2.1.3 Discrete cosine transform

The third step is the *Discrete Cosine Transform* (DCT) which transforms the original 8x8 matrices into blocks in the frequency domain in which we perform our operations. Let y be a normalized 8x8 matrix and let Y be the result of the DCT on y. Then by definition of the DCT

we have

$$Y[u,v] = \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)y[i,j] \tag{2.1}$$

for $u, v = 0 \ldots 7$ where

$$C(i,u) = A(u)\cos\frac{(2i+1)u\pi}{16} \tag{2.2}$$

and

$$A(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$$

This is done for every block in all components.

$Y[0,0]$ is called the *DC-coefficient* (Direct Current). The DC-coefficient represents the average intensity or amplitude of the block.

The $Y[u,v]$ for $u, v = 1 \ldots 7$ are called the *AC-coefficients* (Alternating Current). The AC-coefficients represent the fluctuations in the intensity of the block.

38	38	38	38	38	37	38	38
38	38	38	40	38	36	38	39
38	38	38	37	38	39	38	38
38	36	38	38	37	38	38	38
38	38	39	38	38	-30	38	38
39	38	40	38	38	-30	-30	-30
38	38	38	38	38	-30	-20	-20
38	37	36	38	38	-30	-20	-20

DCT

224	86	-24	-27	35	-3	-26	29
81	-89	30	19	-30	7	21	-23
-18	22	-16	8	1	-6	5	-5
-20	24	-4	-11	10	0	-10	11
16	-23	16	-7	0	6	-7	4
3	-1	-7	13	-8	-3	12	-9
-11	14	-10	6	-1	-4	5	-2
7	-11	15	-13	5	4	-13	10

Figure 2.2: DCT step.

2.1.4 Quantization

The fourth step is the *quantization step*. This quantization step is defined by:

$$Y_Q[u,v] = \text{IntegerRound}\left(\frac{Y[u,v]}{q[u,v]}\right)$$

for $u, v = 0 \ldots 7$.

Every element of each block is divided by a given value. The values, $q[u,v]$, that are used for this quantization process are stored in a matrix. This matrix is called a *quantization table*. The 64 element quantization table does not contain any values less or equal to zero.

The aim of this process is to get rid of small values in the frequency domain, which appear mostly at the lower right of the matrix (with the origin at the upper left corner) by scaling them to zero. This will cause runs of consecutive zeroes, which will result in a better compression (see 2.1.6).

Small entries in the matrix can be set to 0 by dividing them by larger values using integer rounding. At the decoding process (at the dequantization stage to be more precisely), these zero entries will stay zero. In other words, instead of the original matrix, a mutated matrix will be the result of dequantizing.

Typically, there is one quantization table for the luminance component and one quantization table for the chrominance components. A quantization table in the JFIF standard is always stored in zigzag order, see figure 2.4. The higher the values in the quantization table, the better the compression, but the bigger the loss of information will be. If all entries in the quantization tables would be 1, then no information loss due to quantization will occur, but a poor compression ratio will be the penalty.

224	86	-24	-27	35	-3	-26	29
81	-89	30	19	-30	7	21	-23
-18	22	-16	8	1	-6	5	-5
-20	24	-4	-11	10	0	-10	11
16	-23	16	-7	0	6	-7	4
3	-1	-7	13	-8	-3	12	-9
-11	14	-10	6	-1	-4	5	-2
7	-11	15	-13	5	4	-13	10

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

Quantisatie

14	8	-2	-2	3	0	-2	2
6	-6	2	1	2	0	1	-1
-1	1	-1	0	0	0	0	0
-1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 2.3: Quantization step.

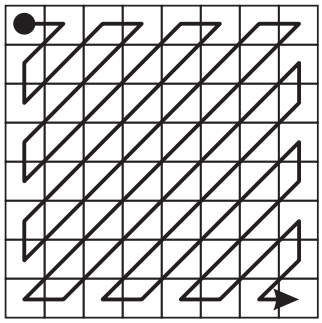
2.1.5 Zigzag

The fifth step of the JPEG algorithm is called the *zigzag scan step*. The 8x8 matrices Y_Q, the so called 'blocks', are converted into *zigzagged vectors* $Y_{Q,zz}$, containing the 64 matrix elements using the 'zigzag' ordering.

In most images this vector will contain a lot of sequential zeroes, especially at the end of the vector. Both the discrete cosine transform and the quantization process are responsible for this.

If a picture is compressed according to the JFIF standard, the quantization tables are stored in zigzag order. In that case steps 4 and 5 of the algorithm should be swapped.

14	8	-2	-2	3	0	-2	2
6	-6	2	1	2	0	1	-1
-1	1	-1	0	0	0	0	0
-1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0



Zigzag

14 8 6 -1 -6 -2 -2 2 1 -1 0 →
↪ 1 -1 1 3 0 2 0 0 0 0 0 →
↪ 0 0 0 0 0 -2 2 1 0 0 0 →
↪ 0 0 0 0 0 0 0 0 0 -1 0 →
↪ 0 0 0 0 0 0 0 0 0 0 0 →
↪ 0 0 0 0 0 0 0 0 0 0

Figure 2.4: Zigzag step.

2.1.6 Runlength encoding

The input of the *Run Length Encoding* (RLE) process is the 64 element zigzag vector. The output of this process is a 'Runlength Encoded block' (RLE-block). Note that a more precise term would be RLE-zigzag-vector. The first element is the DC-coefficient of the frequency domain, the other 63 elements are the AC-coefficients.

The DC-coefficient is treated differently from the AC-coefficients: Instead of the actual DC-coefficients, the difference of the DC-coefficients of two consecutive blocks is stored in the RLE representation.

14 8 6 -1 -6 -2 -2 2 1 -1 0 1 -1 1 3 0 →
↪ 2 0 0 0 0 0 0 0 0 0 0 -2 2 1 0 0 →
↪ 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 →
↪ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

RLE

0	14
0	8
0	6
0	-1
0	-6
0	-2
0	-2
0	2
0	1
0	-1
1	1
0	-1
0	1
0	3
1	2
10	-2
0	2
0	1
12	-1
0	0

Figure 2.5: RLE step.

RLE is used to represent a run of zeroes by the length of this run. This decreases storage space for the vector since most vectors end with a lot of zeroes.

The RLE representation for the DC-coefficient is defined as:

(SIZE)(AMPLITUDE)

where 'SIZE' is the number of bits that is needed to represent the difference of the DC-coefficient of the current block and the DC-coefficient of the previous block in the same component ('AMPLITUDE'). Differences are stored because in real life images, differences between consecutive DC-coefficients tend to be small, so we need fewer bits to store them.

The RLE representation for the AC-coefficients is defined as:

(RUNLENGTH, SIZE)
(AMPLITUDE)

where 'RUNLENGTH' represents the number of consecutive zeroes preceeding the current 'AMPLITUDE' (nonzero). This nonzero is represented by 'SIZE' bits. In Section 7.1 of [Wall] it is mentioned that the FDCT applied on a 8x8 point signal containing 8-bit integers results in DC-coefficients of at most 11-bits. So 'SIZE' can have integer values in the range [1, 11]; 'SIZE' needs 4 bits to represent values in this range.

'RUNLENGTH' represents zero-runs of length 0 to 15 (note that 'RUNLENGTH' can be stored using 4 bits). A run containing more than 16 consecutive zeroes, say n zeroes, is represented by a (0xF,0) marker, followed by the (RUNLENGTH, SIZE)(AMPLITUDE) representation for the remaining $n - 16$ zeroes. Up to three consecutive (0xF,0) markers can preceed the terminating (RUNLENGTH, SIZE) symbol.

The last step is the Huffman encoding step.

(0,0).

This marker is called the End of Block marker (EOB).

We end this subsection by mentioning that a (RUNLENGTH, SIZE) symbol can be stored in exactly one byte.

2.1.7 Huffman encoding

The last step is the Huffman encoding step. Huffman is used to decrease the storage space that is needed for the Runlength Encoded version of $Y_{Q,zz}$. The most frequent (RUNLENGTH, SIZE) combinations are given the shortest Huffman codes and less frequent combinations are given the larger codes. Huffman tables can vary between JPEG images that are generated by different applications.

2.2 Minimal Code Unit

The steps described in Subsection 2.1.3 through Subsection 2.1.7 describe what to do with each block of a component but not in what order the components (or rather, the blocks of the components) are to be processed.

Another problem is that different components can have a different number of blocks. This can be caused by the different *sampling factors* for each component.

For example, in the YCbCr colour space, the first components needs to be sampled more precise because the first component determines luminance, but the second and third component can be sampled roughly since these two components determine chrominance only.

For these purposes, the notion *Minimal Code Unit* (MCU) was introduced. The idea is to interleave the scan-data of the components. an MCU is defined to be the smallest unit of interleaved data.

First each component i is divided into *block-matrices* of H_i by V_i blocks where H_i and V_i are the horizontal and vertical sampling factors for component i.

Next, each block-matrix is transformed into a *block-vector* by a left to right, top to bottom ordering.

Now the j^{th} MCU can be formed by taking the j^{th} block-vector of the first component, then the j^{th} block-vector of the second component and so on. (See Figure 2.6).

The implementation of the steps in Subsection 2.1.3 through Subsection 2.1.7 is MCU oriented; each step operates on a complete MCU.

2.3 Some more DCT

In this section, we will highlight some features of the DCT which will be useful further on.

Consider equation 2.1 in Section 2.1. Suppose $y[i,j] = n$ for $i,j = 0 \ldots 7$ and for some value n. If the DCT would be applied on matrix y then

$$Y[u,v] = \begin{cases} 8n & \text{for } u,v = 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

This result can be generalized; now $y[i,j] = n_{i,j}$ for $i,j = 0 \ldots 7$. The DC-coefficient can still be calculated without the difficult equation 2.1:

$$Y[0,0] = \frac{1}{8} \sum_{i=0}^{7} \sum_{j=0}^{7} y[i,j]$$

which could be rewritten as:

$$Y[0,0] = 8\frac{1}{64} \sum_{i=0}^{7} \sum_{j=0}^{7} y[i,j].$$

This is the average of all matrix entries in y multiplied by 8, so determination of the DC-coefficient is a linear process.

This may not be a surprising result if you're familiar with the Discrete Cosine Transform, but it's a useful fact.

Again, consider equation 2.1, and the entries in the matrix are $y[i,j] = n_{i,j}$ for $i,j = 0 \ldots 7$. Y is the result of the DCT applied to y. Suppose we want to add n to every entry in y but we only have Y.

We could apply the inverse DCT to Y and add n to every entry in y. But if we keep in mind that the DC-coefficient is the average value of all $y[i,j]$, we could 'shift' this average.

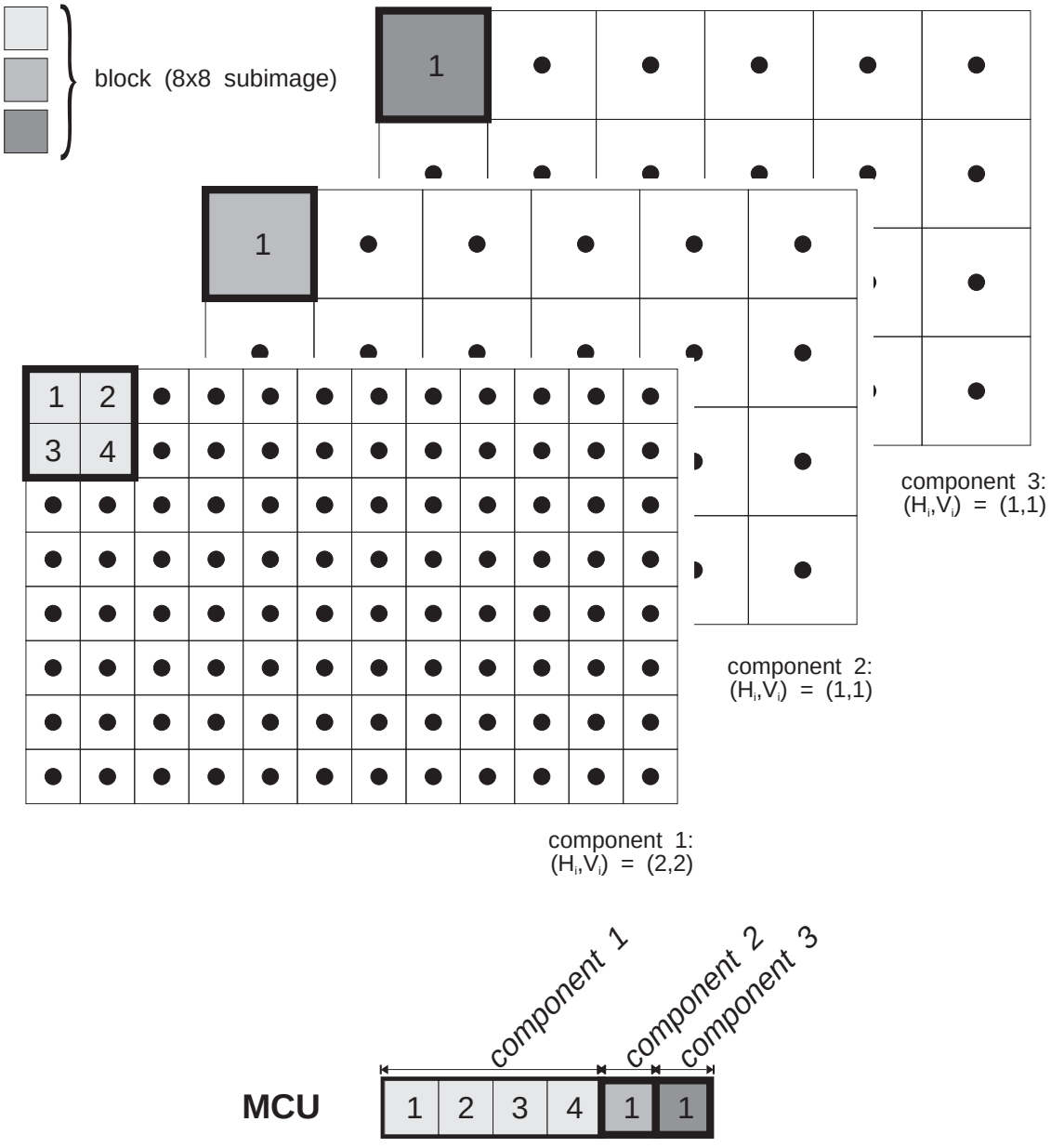


Figure 2.6: The construction of one MCU.

If a is the average and we would add n to every $y[i,j]$ then the new average would be $a+n$. The DC-coefficient is the average value multiplied by 8, so we get $8a + 8n$. We see that we only have to multiply n by 8 and add this to $Y[0,0]$. So our new version of Y (let's call it Y') looks like:

$$Y'[u,v] = \begin{cases} Y[0,0] + 8n & \text{for } u,v = 0 \\ Y[u,v] & \text{otherwise} \end{cases} \quad (2.4)$$

This property is not only useful for scalar addition, but also for denormalizing the first component when performing scalar multiplication or pixel multiplication.

To finish this Section, we will present the inverse DCT (IDCT), which will be used in pixel multiplication.

$$y[i,j] = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(i,u)C(j,v)Y[u,v] \quad (2.5)$$

2.4 Quantization tables

As already mentioned in Section 2.1.4, in a colour — 3 component — image there is a separate quantization table for the luminance component, and one quantization table for both the chrominance components. A greyscale image consists of only one (luminance) component and therefore contains only one quantization table.

The quantization tables determine the quality and the compression ratio of the resulting image. The higher the values in the quantization table, the better the compression, but the bigger the loss of information will be. If all entries in the quantization tables would be 1, then no information loss due to quantization will occur, but a poor compression ratio will be the penalty. Because of the influence of the quantization tables on the file size and the quality of the resulting image most JPEG compression programs let you pick a file size versus image quality trade off by selecting a quality setting. This quality determines the contents of the quantization tables. A quality range that is used in most JPEG compression programs is 1. . . 100. For every different quality, different (quality based) quantization tables should be used. For a maximum quality (of 100) all elements of the quantization tables should be 1.

To determine a *quality based quantization table* the only additional information needed is a *basic quantization table*. For a greyscale image only a basic luminance quantization table is required. For a colour image two basic quantization table are needed to determine the quality based luminance and chrominance quantization tables. For some example basic quantization tables see Figure 2.7. These basic tables quantization tables are (as all good quantization tables) stored in zigzag order.

luminance quantization table:

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

chrominance quantization table:

17	18	18	24	21	24	47	26
26	47	99	66	56	66	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Figure 2.7: Basic quantization tables.

Every quality based quantization table can now be extracted from this basic table by using the user selected quality. The relationship between an element in the basic quantization table (basic_tbl[i]) and an element in a quality based quantization table q_tbl[i] can be described according to the algorithm in figure 2.8.

if quality$<$50 then

$$\text{q\_table[i]} = \frac{\left(\text{basic\_tbl[i]} * \frac{5000}{\text{quality}} + 50\right)}{100}$$

else

$$\text{q\_table[i]} = \frac{(\text{basic\_tbl[i]} * (200 - 2 * \text{quality}) + 50)}{100}$$

fi

Figure 2.8: Quality conversion.

If the selected quality is 100 then all elements in the quality based quantization table will be 1. Use of this maximum quality is not recommended because the resulting file will be two or three times as large as with quality 95, but of hardly any better quality. If the selected quality is 50 the quality based quantization table will be the same as the basic quantization table.

The lower the quality, the larger the elements in the quality based quantization tables. As a result, the quantization step will produce more zero-entries. More zero-entries will reduce the average number of entries in an RLE block.

Chapter 3

Algebraic operations: the mathematical side

The sections in [Smith] that deal with the operations only give a short mathematical explanation (except for pixel multiplication). A complete mathematical description will be given for each operation in the following sections, where we won't involve quantization yet. We will only show how the algebraic operations would look like in the frequency domain.

Furthermore, in the following sections of this chapter, we will work with the actual DC-coefficients instead of the DC-differences that are stored in the bitstream by the JPEG method. How to deal with the DC-differences will be discussed in Chapter 4.

3.1 Scalar multiplication

Let's recall the Discrete Cosine Transform (Eq. 2.1). Suppose y is the result of scalar multiplication by factor α on the original 8x8 submatrix and x is our original matrix. In other words:

$$y[i,j] = \alpha x[i,j] \tag{3.1}$$

for $i, j = 0, \ldots, 7$.

We now substitute Eq. 3.1 into Eq. 2.1 to obtain:

$$\begin{aligned} & Y[u,v] \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)y[i,j] \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)\alpha x[i,j] \\ = \; & \alpha\frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)x[i,j] \\ = \; & \alpha X[u,v] \end{aligned}$$

This implies that scalar multiplication in the frequency domain is equivalent to scalar multiplication in the spatial domain.

3.2 Scalar addition

Scalar addition adds the same factor β to all elements in a block. If we recall Section 2.3, scalar addition is an easy operation.

Let y be the result of scalar addition on x with factor β in the spatial domain. So we have

$$y[i,j] = x[i,j] + \beta \tag{3.2}$$

for $i, j = 0, \ldots, 7$.

Substitution of Eq. 3.2 into Eq. 2.1 gives

$$\begin{aligned} & Y[u,v] = \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)y[i,j] \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{j} C(i,u)C(j,v)\{x[i,j]+\beta\} \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} \{C(i,u)C(j,v)x[i,j] + \\ & C(i,u)C(j,v)\beta\} \\ = \; & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)x[i,j] + \\ & \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)\beta \\ = \; & X[u,v] + Z[u,v] \end{aligned}$$

for $u, v = 0, \ldots, 7$.

Let $Z[u,v]$ be the result of the DCT applied on matrix z, with $z[i,j] = \beta$ for $i, j = 0, \ldots, 7$. If we use the property, given in Eq. 2.3, we see that

$$\begin{aligned} & Y[u,v] = \\ = \; & X[u,v] + Z[u,v] \\ = \; & \begin{cases} X[u,v] + 8\beta & \text{for } u,v = 0 \\ X[u,v] & \text{otherwise} \end{cases} \end{aligned} \tag{3.3}$$

Another way of looking at scalar addition is to notice that adding β to all entries in x is the same as adding β

to the average of all entries in x. If we define a to be the average of all values in x then $a + \beta$ would be the average of all entries in y, so $Y[0,0]$ (the DC coefficient of Y) would be $8(a + \beta) = 8a + 8\beta$, which is $X[0,0] + 8\beta$.

We conclude this section with the remark that we only have to perform one addition and one multiplication per matrix in the frequency domain (namely $Y[0,0] = X[0,0]+8\beta$), instead of 64 additions in the spatial domain.

3.3 Pixel addition

Pixel addition (or rather matrix addition) is defined by

$$y[i,j] = x_1[i,j] + x_2[i,j] \tag{3.4}$$

for $i, j = 0, \ldots, 7$.

Substitution of Eq. 3.4 into Eq. 2.1 gives

$$\begin{aligned}
Y[u,v] = & \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)y[i,j] \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)\,\{x_1[i,j] \\
&\quad + x_2[i,j]\} \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} \{C(i,u)C(j,v)x_1[i,j] + \\
&\quad C(i,u)C(j,v)x_2[i,j]\} \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)x_1[i,j] + \\
&\quad \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)x_2[i,j] \\
&= X_1[u,v] + X_2[u,v]
\end{aligned} \tag{3.5}$$

for $u, v = 0, \ldots, 7$

So if we compare Eq. 3.4 to Eq. 3.5, we see that pixel addition in the frequency domain is the same as pixel addition in the spatial domain.

3.4 Pixel multiplication

Pixel multiplication in the spatial domain is defined by

$$y[i,j] = x_1[i,j]x_2[i,j] \tag{3.6}$$

for $i, j = 0, \ldots, 7$.

In [Smith], a factor α is introduced to scale the products in order to keep the values within the correct range (depending on the colour space and component we're using). Since this is not important for the mathematical discussion on pixel multiplication in the frequency domain, we will leave it out for now, and we will introduce this α in Section 4.5.

Substitution of Eq. 3.6 into Eq. 2.1 gives:

$$\begin{aligned}
Y[u_1,u_2] = & \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u_1)C(j,u_2)y[i,j] \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u_1)C(j,u_2)x_1[i,j]x_2[i,j] \\
&= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u_1)C(j,u_2) \\
&\quad \left(\frac{1}{4}\sum_{v_1=0}^{7}\sum_{v_2=0}^{7} C(i,v_1)C(j,v_2)X_1[v_1,v_2]\right) \\
&\quad \left(\frac{1}{4}\sum_{w_1=0}^{7}\sum_{w_2=0}^{7} C(i,w_1)C(j,w_2)X_2[w_1,w_2]\right)
\end{aligned} \tag{3.7}$$

$$\begin{aligned}
&= \sum_{v_1,v_2,w_1,w_2=0}^{7} (X_1[v_1,v_2]X_2[w_1,w_2]* \\
&\quad *M[v_1,v_2,w_1,w_2,u_1,u_2])
\end{aligned} \tag{3.8}$$

where

$$\begin{aligned}
M[v_1,v_2,w_1,w_2,u_1,u_2] = & \\
&= \frac{1}{64}W[u_1,v_1,w_1]W[u_2,v_2,w_2]
\end{aligned}$$

with

$$W[u,v,w] = \sum_{i=0} C(i,u)C(i,v)C(i,w)$$

for $u_1, u_2 = 0, \ldots, 7$.

Notice that the IDCT, as defined in Eq. 2.5, is used to get Eq. 3.7.

spatial domain	frequency domain
$y[i,j] = \alpha x[i,j]$	$Y[u,v] = \alpha X[u,v]$
$y[i,j] = x[i,j] + \beta$	$\begin{cases} X[u,v] + 8\beta & \text{als } u,v = 0 \\ X[u,v] & \text{als } u,v \neq 0 \end{cases}$
$y[i,j] = x_1[i,j] + x_2[i,j]$	$Y[u,v] = X_1[u,v] + X_2[u,v]$
$y[i,j] = x_1[i,j]x_2[i,j]$	$Y[u_1,u_2] = \sum_{v_1,v_2,w_1,w_2=0}^{7} (X_1[v_1,v_2]X_2[w_2,w_2]* M[v_1,v_2,w_1,w_2,u_1,u_2])$

Table 3.1: An overview of the operations.

Chapter 4

Algebraic operations: the implementation side

In this chapter we will adjust the theory, given in Chapter 3 to properties of JPEG compressed images.

First we will extend the results obtained in Chapter 3; Eq. 3.2, Eq. 3.3, Eq. 3.5 and Eq. 3.8 are modified to quantization, DC-differences and normalization separately. Then these modifications are combined.

quantization

A complete mathematical description for each operation was given in Chapter 3, in which we didn't use quantization. Now we will use quantization but we will make the following assumptions:

Assumption 4.1 *The quantization tables for the output image of scalar multiplication and scalar addition are the same as for the input image.*

Assumption 4.2 *The quantization tables for the output image of pixel multiplication and pixel addition are the same as for the first input image that is selected in our program.*

In the operations scalar addition, pixel addition and pixel multiplication, we introduce an additional scaling factor. With this factor, we try to keep the outcome in the correct range (depending on the colour space and component we're using).

DC-differences

In Chapter 3 we didn't use the DC-differences. In this chapter we will examine the influence of using DC-differences, instead of DC-coefficients, on the equations in Chapter 3.

In [Smith], DC-coefficients are used instead of DC-differences. Reasons for the use of DC-coefficients could be:

- Source code; The source code provided by the [Ijg] automatically converts the quantized DC-differences into the quantized DC-coefficients.
- Simplicity; Using DC-coefficients makes the theories in Chapter 3 easier to understand. However some increase in performance can be expected when the DC-differences are used.
- Rounding errors; Theoretically in some operations it doesn't make any difference whether DC-differences or DC-coefficients are used to calculate the output image. In pratice however, rounding errors made during calculation of the output image (using DC-differences) might influence further calculations of the output DC-differences. Rounding errors might pile up.

normalization

In the JPEG method, all values are brought into the range $-128 \ldots 127$, before the DCT is applied. Normally, we would bring our data back into the spatial domain, denormalize it when necessary, perform our operation, normalize it when necesarry and bring it back into the frequency domain.

Now we want to perform the operations directly in the frequency domain. This leaves us with the problem of combining denormalization, applying our operation and normalization in the frequency domain.

As we're assuming our JPEG files to be formatted in the JFIF style, this process only needs to be done for the first component.

One remark can be made here on the combination of normalization and the use of DC-differences: When DC-differences are used to perform an operation on a normalized component, then the denormalization and normalization only needs to be done for the first DC-difference (which is in fact a DC-coefficient); the difference between two consecutive DC-coefficients stays the same, whether the component containing the DC-coefficients was normalized or not.

scaling factor

In the operations pixel addition and pixel multiplication, the values of the resulting pixels could easily go out of the valid ranges. To avoid this, we introduced a scaling factor α. The principle is basicly the same as scalar multiplication, but now we have to combine the scaling factor with pixel addition and pixel multiplication.

combined

In practice, the above properties of JPEG images are combined, so the underlying theories have to be combined as well. This has consequences for the operations scalar multiplication and pixel addition, in particular for the calculation of the resulting DC-differences or DC-coefficients. The calculation of the DC-differences or DC-coefficients, will be discussed in these subsections

4.1 DC-differences, quantization and rounding errors

Before we describe the implementation issues of the operations, we will take a look at the use of DC-differences in combination with quantization tables.

One general remark on DC-differences can be made : Regardless of the operation to be performed, every rounding error made on a DC-difference works accumulative because the DC-coefficients are sums of DC-differences.

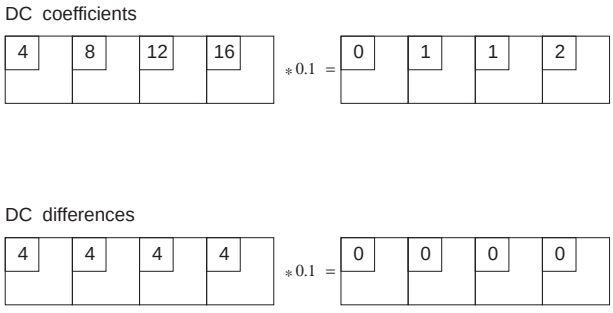


Figure 4.1: Example of DC-differences and rounding errors

The DC-coefficient of the i^{th} block is the sum of the first i DC-differences. If during the calculations of the first i DC-differences k ($k < i$) rounding errors were made, then these k rounding errors are used to calculate the i^{th} DC-coefficient. So the larger the image, the more likely this effect will start to show.

This unpleasant effect is amplified by the use of quantization tables. Suppose a quantized DC-difference d_Q should have been stored in the JPEG bitstream, but $d_Q - 1$, a mutated version of d_Q, was stored because of a rounding error made during the calculations and integer rounding.

After this incorrect value $d_Q - 1$ is read from the bitstream during decompressing, it will be dequantized by $q[0,0]$. As a result, the DC difference that will be used to calculate the current DC-coefficient will be

$$(d_Q - 1) * q[0,0] = d_Q * q[0,0] - q[0,0]$$

But the correct DC-difference should have been

$$d_Q * q[0,0]$$

So the rounding error made during the calculations using DC-differences results in an error of $q[0,0]$ in the dequantized frequency domain. This causes an error of $\frac{1}{8}q[0,0]$ *for every element* in the spatial domain (see Section 2.3).

Now we see the following: The bigger $q[0,0]$ — or the lower the quality, see Section 2.4 — the less time the operation takes, since the RLE lists are smaller, but the more influence a rounding error has on the resulting data in the spatial domain. So we expect our DC-difference oriented versions not to be useful for low quality images. Figure 4.2 shows scalar multiplication, using α = 0.5 with an input image that has a 10% quality.

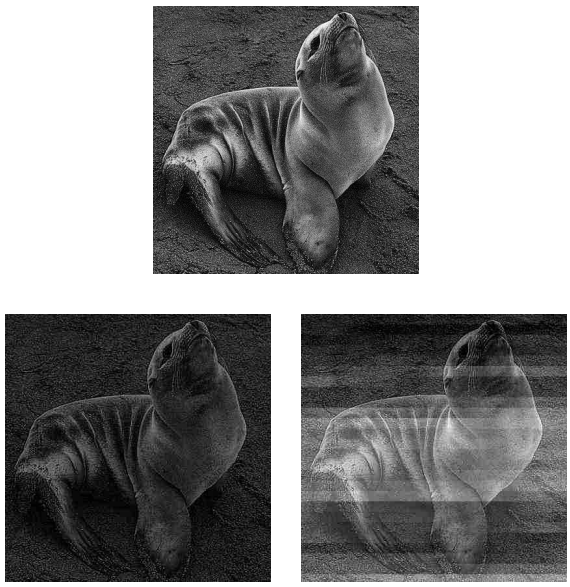


Figure 4.2: Scalar multiplication: input, result of DC-coefficients, result of DC-differences.

4.2 Scalar multiplication

quantization

In Section 3.1 we saw

$$Y[u,v] = \alpha X[u,v] \tag{4.1}$$

However, the values obtained after Huffman decoding are quantized. Instead of $X[u,v]$, as in Eq. 4.1, we have $X_Q[u,v]$. The relationship between $X[u,v]$ and $X_Q[u,v]$ can be described as

$$X[u,v] = X_Q[u,v]q_X[u,v] \tag{4.2}$$

where $q_X[u,v]$ is the $(u,v)^{th}$ quantization value for every block in the current component.

This result substituted in Eq. 4.1 gives

$$Y[u,v] = \alpha X_Q[u,v]q_X[u,v] \tag{4.3}$$

Now we need to quantize the result; we want an equation for $Y_Q[u,v]$. The relationship between $Y[u,v]$ and $Y_Q[u,v]$ is a similar one as in Eq. 4.2. After substitution of $Y[u,v]$ of Eq. 4.3 we obtain

$$Y_Q[u,v]q_Y[u,v] = \alpha q_X[u,v]X_Q[u,v]$$

which leads to

$$Y_Q[u,v] = \alpha \frac{q_X[u,v]}{q_Y[u,v]} X_Q[u,v] \tag{4.4}$$

Because of Assumption 4.1 on the quantization tables, $q_X[u,v] = q_Y[u,v]$, Eq. 4.4 can be rewritten as

$$Y_Q[u,v] = \alpha X_Q[u,v] \tag{4.5}$$

As a result of Eq. 4.5, we can see that scalar multiplication can be performed directly on the quantized coefficients in the frequency domain.

DC-differences

Suppose the DC-coefficient of the first block is d_1 and the DC-coefficient of the second block in the same component is $d_1 + d_2$. So the JPEG algorithm will store d_1 as the first DC-difference and d_2 as the second DC-difference.

Let's assume we are multiplying by factor α.

Performing scalar multiplication on the DC-coefficients would result in αd_1 for the first DC-coefficient and $\alpha(d_1 + d_2)$ for the second DC-coefficient. So the DC-difference for the second block is

$$\alpha(d_1 + d_2) - \alpha d_1 = \alpha d_2.$$

Performing scalar multiplication directly on the DC-differences would give αd_1 for the first block and αd_2 for the second block.

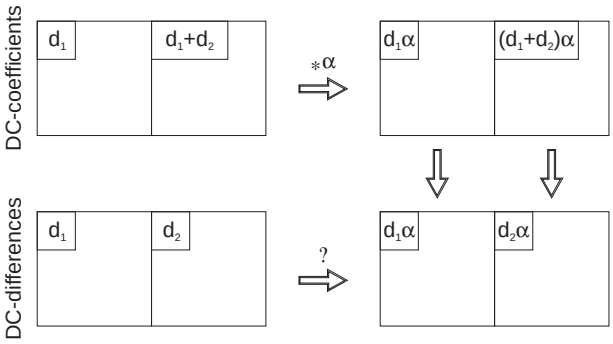


Figure 4.3: Scalar multiplication with DC-coefficients and with DC-differences.

Figure 4.3 shows the process for DC-coefficients and for DC-differences.

The first component however was normalized, so in that case we cannot just multiply the DC-differences by the given factor.

So it is proven that theoretically, it doesn't make any difference for the calculation of the output image data if DC-differences are used instead of DC-coefficients (if normalization is not taken into account).

In practice however, rounding errors on DC-differences can have too big an impact on the output image data; rounding errors might occur, if the multiplication factor α is not an integer.

normalization

The first component should be denormalized before performing scalar multiplication, otherwise we would also scale the '-128' that is used in the normalization process. The only problem we have is that we are working in a quantized frequency domain as calculated by the DCT and the quantization table.

But if we recall that the DC-coefficient is the same as the average of all original values multiplied by 8, then 'denormalizing' should not be difficult.

Suppose d is our DC-coefficient and $q[0,0]$ is the quantization coefficient for the DC-coefficients. Then the quantized DC-coefficient d_Q — the value that is actualy stored in the image bitstream — would be

$$d_Q = \frac{d}{q[0,0]}$$

In the ordinary case, i.e. when we would transform our blocks back into the spatial domain, we would have added 128 to each value in the block.

If we were to denormalize in a quantized frequency domain, the proces of denormalizing in the frequency domain would simply look like

$$d_Q q[0,0] + 128 * 8 = d_Q q[0,0] + 1024$$

combined

A block which is input of the scalar multiplication operation can have one of the following combinations of properties:

1. The block is DC-difference oriented, normalized and it is the very first block of the component.
2. The block is DC-difference oriented, normalized and it is not the first block of the component.
3. The block is DC-difference oriented and not normalized
4. The block is DC-coefficient oriented and normalized.
5. The block is DC-coefficient oriented and not normalized

Note that in addition to each combination, each block is quantized.

DC-difference, normalized, first block

This combination occurs when we are working with a DC-difference oriented algorithm in the first component, and we're about to perform scalar multiplication on the first block.

Because we're working on the first block of the component, the quantized DC-difference of this block is actually a quantized DC-coefficient. Suppose d is that DC-coefficient and d' is the DC-coefficient to be calculated.

First we need to dequantize the DC-coefficient, then denormalization should be applied before scalar multiplication can be applied. After that, normalization and quantization should be done. This and Assumption 4.1 gives the following combination:

$$\begin{aligned} d' &= \frac{\alpha\,(dq_X[0,0]+1024)-1024}{q_X[0,0]} \\ &= \frac{dq_X[0,0]\alpha+1024\alpha-1024}{q_X[0,0]} \\ &= d\alpha+\frac{1024(\alpha-1)}{q_X[0,0]} \end{aligned}$$

DC-difference, normalized, not first block

It was shown that scalar multplication can be applied directly on the DC-differences. What we need to do normally is to dequantize, denormalize, perform scalar multiplication, normalize and quantize the DC-difference (in that order).

However, since we're working with DC-differences, we don't need to denormalize; this was already done in the first block of this component. Whether we are working in a normalized frequency domain or in an ordinary (not) normalized frequency domain, the difference of two consecutive DC-coefficients stays the same; the DCT is a linear processes. So we need to dequantize the DC-difference, perform scalar multiplication and to quantize the result. Using Assumption 4.1, the calculation of d' looks like:

$$\begin{aligned} d' &= \frac{\alpha(dq_X[0,0])}{q_X[0,0]} \\ &= \alpha d \end{aligned} \tag{4.6}$$

DC-difference oriented and not normalized

This is a similar case as the one described above. The only difference is that we do not have to denormalize the block because it was not normalized at all. So the method to be used for such DC-differences is the same as described in Eq. 4.6

DC-coefficient oriented and normalized

After dequantization of the DC-coefficient, we need to denormalize it. After the scalar multiplication, normalization and then quantization is needed.

Using Assumption 4.2 gives the following calculation:

$$\begin{aligned} d' &= \\ &= \frac{((dq_X[0,0]+1024)\,\alpha)-1024}{q_X[0,0]} \\ &= \frac{\alpha dq_X[0,0]+1024\alpha-1024}{q_X[0,0]} \\ &= \alpha d+\frac{1024(\alpha-1)}{q_X[0,0]} \end{aligned}$$

DC-coefficient oriented and not normalized

A DC-difference of this combination only needs to be dequantized before the scalar multiplication operation. Afterwards quantization is to be applied. Using Assumption 4.1 the calculation for d' would look like:

$$\begin{aligned} d' &= \\ &= \alpha\frac{dq_X[0,0]}{q_X[0,0]} \\ &= \alpha d \end{aligned}$$

4.3 Scalar addition

quantization

In Section 3.2 we found Eq. 3.3. This equation needs to be quantized, but as in Eq. 3.3 can be seen, there are two cases: $u, v = 0$ and $u, v \neq 0$.

For $u, v = 0$ we had $Y[u,v] = X[u,v] + 8\beta$. Again, we substitute Eq. 4.2.

$$Y[u,v] = X_Q[u,v]q_X[u,v] + 8\beta \tag{4.7}$$

A similar substitution for $Y[u,v]$ is done:

$$Y_Q[u,v]q_Y[u,v] = X_Q[u,v]q_X[u,v] + 8\beta$$

But $Y_Q[u,v]$ is what we're looking for:

$$\begin{aligned} Y_Q[u,v] = \\ &= \frac{X_Q[u,v]q_X[u,v] + 8\beta}{q_Y[u,v]} \\ &= \frac{q_X[u,v]}{q_Y[u,v]}X_Q[u,v] + \frac{8\beta}{q_Y[u,v]} \end{aligned}$$

For $u, v \neq 0$ we had $Y[u,v] = X[u,v]$. After doing the same as $u, v = 0$ for $Y[u,v] = X[u,v]$ we get

$$Y_Q[u,v] = \frac{q_X[u,v]}{q_Y[u,v]}X_Q[u,v]$$

As $q_Y[u,v] = q_X[u,v]$ (Assumption 4.1) we can now write

$$Y_Q[u,v] = \begin{cases} X_Q[u,v] + \frac{8\beta}{q_X[u,v]} & \text{for } u,v = 0 \\ X_Q[u,v] & \text{otherwise} \end{cases} \tag{4.8}$$

Eq. 4.8 shows us that only $q_X[0,0]$ is needed to calculate $Y_Q[0,0]$. The remaining entries of Y are just copies of X.

DC-difference

As already mentioned in Section 3.2, we only have to perform one addition and one multiplication per block if we use the (non)quantized frequency domain with DC-coefficients. An additional division is needed if we're using the quantized frequency domain with DC-coefficients. But what if we're using DC-differences now?

Let's inspect the process using DC-coefficients. Suppose the DC-coefficient of the first block is d_1 and of the DC-coefficient of the second block is $d_1 + d_2$. After scalar addition using factor β we get

$$d_1 + \frac{8\beta}{q_Y[0,0]}$$

for the first block and

$$(d_1 + d_2) + \frac{8\beta}{q_Y[0,0]}$$

for the second block. Converting these actual DC-coefficients into DC-differences would give

$$d_1 + \frac{8\beta}{q_Y[0,0]}$$

as the first DC-difference and

$$d_1 + \frac{8\beta}{q_Y[0,0]} - \left((d_1 + d_2) + \frac{8\beta}{q_Y[0,0]}\right) = d_2$$

as the second.

Now we will examine the process using DC-differences. Using the same DC-coefficients as above, we would store d_1 as the first DC-difference and d_2 as the second. Applying scalar addition on the first block gives

$$d_1 + \frac{8\beta}{q_Y[0,0]}.$$

Note that the DC-difference in the second block is d_2. This would also be the result if the DC-coefficients were used for scalar addition (see Figure 4.4).

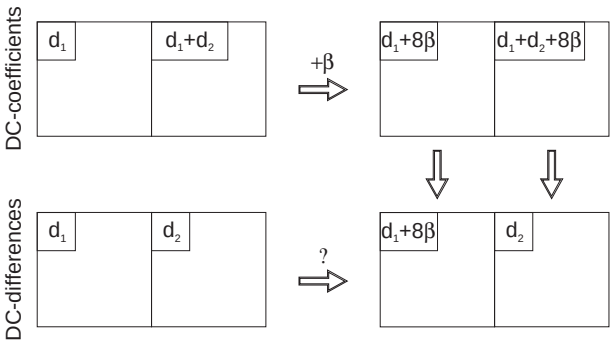


Figure 4.4: Scalar addition with DC-coefficients and with DC-differences.

So scalar addition can be done very efficiently if we use the DC-differences. We only need to perform this operation on the first DC-coefficient of each component. In this way, every component only needs one addition, one multiplication and one division. If DC-coefficients were used, a lot more work would be needed.

Integer rounding does not give any trouble, because for every component only one rounding error might occur (the operation is only performed on the first DC-difference — which is acutally a DC-coefficient). So a piling up of rounding errors onto rounding errors will not occur since only one rounding error per component can be made.

normalization

Another nice property of scalar addition is that we don't need to denormalize the first component.

Again, d is the DC-coefficient of a block in the first component. If we were to denormalize the first component in a nonquantized frequency domain, it would look like this:

$d + 1024$

but since we're dealing with a quantized frequency domain, the normalization needs to be quantized:

$$\frac{d + 1024}{q_Y[0,0]}$$

After the addition (with factor β) is performed it would look like this:

$$\frac{d + 1024}{q_Y[0,0]} + \frac{8\beta}{q_Y[0,0]}$$

Now we need to 'normalize' the result:

$$\left(\frac{d + 1024}{q_Y[0,0]} + \frac{8\beta}{q_Y[0,0]}\right) - 1024 = d + \frac{8\beta}{q_Y[0,0]}$$

which is basicly the same as described in Eq. 4.8.

4.4 Pixel addition

quantization

In Eq. 3.5 we found

$$Y[u,v] = X_1[u,v] + X_2[u,v]$$

After substitution of Eq. 4.2 into this equation we get:

$$\begin{aligned} Y[u,v] &= \\ &= X_{1,Q}[u,v]q_{X_1}[u,v] + X_{2,Q}[u,v]q_{X_2}[u,v] \end{aligned}$$

A similar substitution for Y[u,v] is done:

$$\begin{aligned} Y_Q[u,v]q_Y[u,v] &= \\ &= X_{1,Q}[u,v]q_{X_1}[u,v] + X_{2,Q}[u,v]q_{X_2}[u,v] \end{aligned}$$

After division by $q_Y[u,v]$ we get:

$$\begin{aligned} Y_Q[u,v] &= \\ &= \frac{X_{1,Q}[u,v]q_{X_1}[u,v] + X_{2,Q}[u,v]q_{X_2}[u,v]}{q_Y[u,v]} \\ &= \frac{q_{X_1}[u,v]}{q_Y[u,v]}X_{1_Q}[u,v] + \frac{q_{X_2}[u,v]}{q_Y[u,v]}X_{2,Q}[u,v] \end{aligned}$$

As $q_Y[u,v] = q_{X_1}[u,v]$ (Assumption 4.2) we can now write:

$$Y_Q[u,v] = X_{1,Q}[u,v] + \frac{q_{X_2}[u,v]}{q_{X_1}[u,v]}X_{2,Q}[u,v]$$

This leads to the conclusion that the quantization tables q_{X_1} and q_{X_2} are extensively used.

DC-difference

Suppose the DC-coefficient in the first block of the first input image is c_1 and the DC-coefficient of the second block is $c_1 + c_2$ Doing the same for the second input image gives a DC-coefficient of d_1 for the first block and a DC-coefficient of $d_1 + d_2$ for the second block.

Pixel addition, using DC-coefficients, would give

$$c_1 + d_1$$

as the first DC-coefficient of the output image, and

$$c_1 + d_1 + c_2 + d_2$$

as the second DC-coefficient. Converting these actual DC-coefficients into DC-differences would give

$$c_1 + d_1$$

as the first DC-difference of the output image, and

$$c_1 + d_1 - (c_2 + d_1 + c_2 + d_2) = c_2 + d_2$$

as the second.

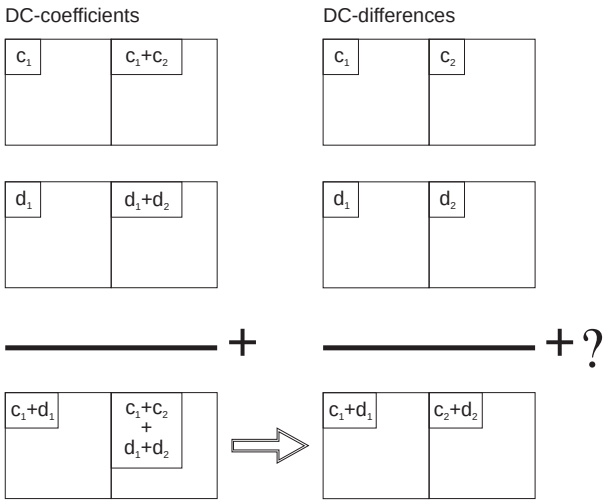


Figure 4.5: Pixel addition with DC-coefficients and with DC-differences.

Let's inspect the process using DC-differences instead of DC-coefficients. Suppose the DC-coefficients of the input images are the same as above. The DC-difference of the first block of the first input image would be c_1 and the DC-difference of the second block would be c_2. Calculation of the DC-differences of the second input image gives d_1 as the first DC-difference, and d_2 as the second. Addition of the two images, using DC-differences, would give $c_1 + d_2$ as the first DC-difference of the output image, and $d_1 + d_2$ as the second.

Figure 4.5 shows the comparison of DC-coefficients and DC-differences with respect to pixel addition.

So we see that theoretically it doesn't make any difference for the calculations whether we use DC-coefficients or DC-differences. For performance it could make a difference since the differences are mostly small numbers. Of course, in practice, rounding errors are still a problem and the use of actual DC-coefficients would be a solution.

normalization

The first component is normalized, this means that we have to denormalize this component. This is done by adding 1024 to the dequantized DC-coefficients.

(Working with DC-differences instead of DC-coefficients means that the denormalization can be done by denormalizing the first DC-coefficient of the first MCU of the first component!) So using the DC-difference instead of the actual DC-coefficient would save a lot of work.

The first components of the two images that are added are normalized. This means that we have to denormalize the first components of the two images before performing our operation.

Dequantization and denormalization of the first component of the first image gives

$$d_1 * q_{X_1}[0,0] + 1024.$$

Dequantization and denormalization of the first component of the second image gives

$$d_2 * q_{X_2}[0,0] + 1024.$$

Pixelwise addition gives

$$d_1 * q_{X_1}[0,0] + d_2 * q_{X_2}[0,0] + 2048.$$

After the addition, the result needs to be normalized and quantized again:

$$\frac{\left(d_1 * q_{X_1}[0,0] + d_2 * q_{X_2}[0,0] + 2048\right) - 1024}{q_Y[0,0]}.$$

Using Assumption 4.2 and rewriting the normalization and denormalization makes it

$$d_1 + \frac{d_2 * q_{X_2}[0,0] + 1024}{q_{X_1}[0,0]}$$

scaling

Scaling the result found in 3.5 with α results in

$$Y[u,v] = \alpha(X_1[u,v] + X_2[u,v])$$

A typical value for α could be $1/2$. In this case, the resulting image could be considered an average image of the two input images.

combined

A block which is input of the pixel addition operation can have one of the following combinations of properties:

1. The blocks are DC-difference oriented, normalized and they are the very first blocks of their component.
2. The blocks are DC-difference oriented, normalized and they are not the first blocks of the component.
3. The blocks are DC-difference oriented and not normalized
4. The blocks are DC-coefficient oriented and normalized.
5. The blocks are DC-coefficient oriented and not normalized

DC-difference, normalized, first block

Because this is the first DC-difference of the first component, this DC-difference is also the DC-coefficient of the first block of the component. So for both DC-coefficients d_1 and d_2, we need to dequantize them, with $q_{X_1}[0,0]$ and $q_{X_2}[0,0]$ respectively, denormalize them, perform pixel addition, normalize the sum and finaly, quantize the normalized sum:

$$\begin{aligned} d' &= \\ &= \frac{\alpha\left(d_1 q_{X_1}[0,0] + 1024 + d_2 q_{X_2}[0,0] + 1024\right) - 1024}{q_{X_1}[0,0]} \\ &= \frac{\alpha\left(d_1 q_{X_1}[0,0] + d_2 q_{X_2}[0,0] + 2048\right) - 1024}{q_{X_1}[0,0]} \end{aligned}$$

DC-difference, normalized, not first block

It was already shown that pixel addition could be performed directly on the DC-differences. And because we are working with real DC-differences — unlike the very first DC-difference of a normalized component, which is in fact a DC-coefficient — we do not have to bother with the denormalization stage.

So what we need to do is to dequantize the quantized DC-difference, perform pixel addition, scale the sum and quantize the scaled sum:

$$d' = \frac{\alpha\left(d_1 q_{X_1}[0,0] + d_2 q_{X_2}[0,0]\right)}{q_{X_1}[0,0]}$$

DC-difference, not normalized

DC-differences that are not normalized are easily dealt with. Again, we use the fact that we can perform pixel addition and scaling directly on the DC-differences.

So the only thing to do is dequantization, pixel addition, scaling and quantization.

$$d' = \frac{\alpha\left(d_1 q_{X_1}[0,0] + d_2 q_{X_2}[0,0]\right)}{q_{X_1}[0,0]}$$

DC-coefficient, normalized

This situation occurs when the used algorithm is a DC-coefficient oriented algorithm, and the first component is to be processed. In this case the steps to be taken are, dequantization, denormalization, pixel addition, scaling, normalization and quantization.

$$\begin{aligned} d' &= \frac{\alpha\left(d_1 q_{X_1}[0,0] + 1024 + d_2 q_{X_2}[0,0] + 1024\right) - 1024}{q_{X_1}[0,0]} \\ &= \frac{\alpha\left(d_1 q_{X_1}[0,0] + d_2 q_{X_2}[0,0] + 2048\right) - 1024}{q_{X_1}[0,0]} \end{aligned}$$

DC-coefficient, not normalized

This occurs when the DC-coefficient oriented algorithm is processing a block from the second or third component. Calculation of such a DC-coefficient consists of the following steps: dequantization, pixel addition, scaling, quantization.

$$d' = \frac{\alpha\left(d_1 q_{X_1}[0,0] + d_2 q_{X_2}[0,0]\right)}{q_{X_1}[0,0]}$$

4.5 Pixel multiplication

quantization

Pixel multiplication in the frequency domain is given by Eq. 3.6. Substituting Eq. 4.2 into Eq. 3.6 and doing a similar substituting for $Y[u,v]$, and using Assumption 4.2, gives

$$\begin{aligned} &Y_Q[u_1,u_2] = \\ &= \sum_{v_1,v_2,w_1,w_2}^{7} (X_1[v_1,v_2]X_2[w_1,w_2]* \\ &\quad *M_Q[v_1,v_2,w_1,w_2,u_1,u_2]) \end{aligned} \tag{4.9}$$

where

$$\begin{aligned} &M_Q[v_1,v_2,w_1,w_2,u_1,u_2] = \\ &= \frac{q_{X_1}[v_1,v_2]q_{X_2}[w_1,w_2]}{64 q_Y[u_1,u_2]} * \\ &\quad *W[u_1,v_1,w_1]W[u_2,v_2,w_2] \end{aligned} \tag{4.10}$$

with

$$W[u,v,w] = \sum_i C(i,u)C(i,v)C(i,w) \tag{4.11}$$

with $C(i,u)$ as defined in Eq. 2.2.

In Eq. 4.10 it can be seen that the quantization table of the second input image is neccesary.

DC-differences

Pixel multiplication is done with the DC-coefficients instead of DC-differences. Using DC-differences would not work here.

normalization

As we're using actual DC-coefficients, instead of DC-differences, we need to denormalize every block in the first component. Denormalizing a block in the frequency domain can be done by adding 1024 to dequantized DC-coefficient of the block. After the operation, normalization can be done by subtracting 1024 of the not yet quantized DC-coefficient.

scaling

Scaling Eq. 4.5 can be done in two ways: the complete sum can be scaled or the matrix M can be scaled. Advantage of the second way is that scaling only needs to be done during initialization of M:

$$\begin{aligned} &Y[u_1,u_2] = \\ &= \sum_{v_1,v_2,w_1,w_2=0}^{7} (X_1[v_1,v_2]X_2[w_1,w_2]* \\ &\quad *M[v_1,v_2,w_1,w_2,u_1,u_2] \end{aligned}$$

where

$$\begin{aligned} &M[v_1,v_2,w_1,w_2,u_1,u_2] = \\ &= \frac{\alpha}{64} W[u_1,v_1,w_1]W[u_2,v_2,w_2] \end{aligned}$$

with

$$W[u,v,w] = \sum_{i=0} C(i,u)C(i,v)C(i,w)$$

for $u_1, u_2 = 0, \ldots, 7$.

4.6 Combination array

This section will be spent on the performance optimization of the process given in Section 4.5. In [Smith] two remarks are made on the summation in Eq. 4.9.

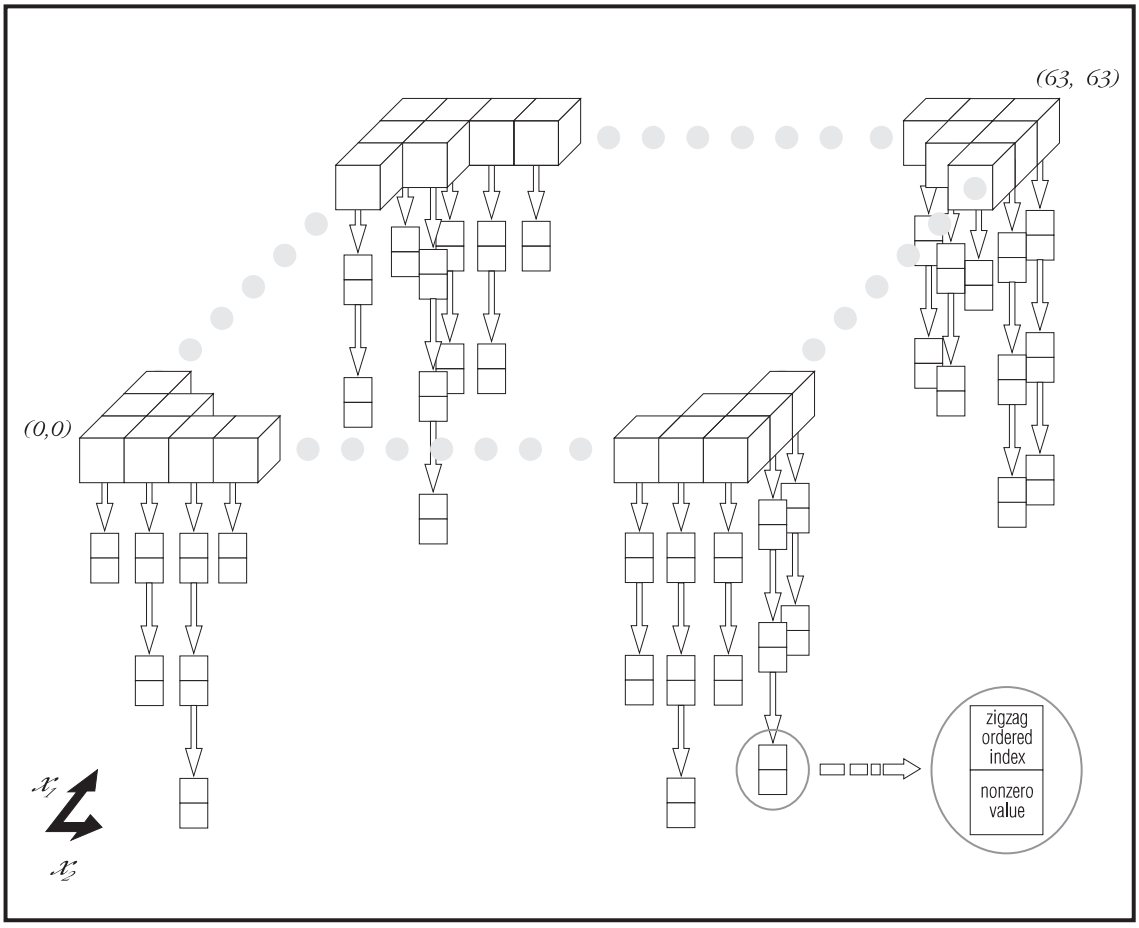


Figure 4.6: The combination array.

1. Eq. 4.9 is a rather large summation. However, in practice many elements in $X_{1,Q}$ and $X_{2,Q}$ are zero.
2. The matrix M_Q has 8^6 elements. However, about 4% are nonzero entries, so M_Q is a sparse matrix.

Using the runlength encoded representation of the blocks takes care of the first remark. To make use of the second remark [Smith] introduced the *combination array*.

4.6.1 The datastructure

Because we want to perform pixelwise multiplication on the RLE blocks, and because the entries in RLE blocks are in zigzag order, we want to rewrite M_Q into a zigzag version.

We introduce x_1, x_2 and z as the zigzag ordered indices of (v_1, v_2), (w_1, w_2), (u_1, u_2) respectively. Define $M_{Q,zz}$ as the zigzag ordered version of M_Q, so now we have $M_{Q,zz}[x_1, x_2, z]$. Furthermore, we define $Y_{Q,zz}$, $X_{1,zz}$ and $X_{2,zz}$ as the zigzag ordered version of Y_Q, X_1 and X_2 respectively.

These new notations are used to rewrite Eq. 4.9 as

$$Y_{Q,zz}[z] = \sum_{x_1,x_2=0}^{63} X_{1,Q,zz}[x_1] X_{2,Q,zz}[x_2] M_{Q,zz}[x_1, x_2, z] \tag{4.12}$$

for $z = 0 \ldots 63$.

$M_{Q,zz}[x_1, x_2, z]$ with $x_1, x_2, z = 0 \ldots 63$ is a sparse matrix. In order to store this matrix efficiently and in order to calculate Eq. 4.12 efficiently, [Smith] introduced a datastructure called *the combination array*.

A combination array is a 64x64 matrix, one entry for each (x_1, x_2) combination, where $x_1, x_2 = 0, \ldots, 63$. Each element in a combination array is a *combination list*.

We call the $(x_1, x_2)^{th}$ entry in the combination array *the combination list of* (x_1, x_2).

A combination list is a list of *combination elements*. There is one combination element for every nonzero value in the matrix M. A combination element holds two items:

1. a value indicating a nonzero entry in M.

$$\begin{array}{rcl} Y_{Q,zz}[0] & = & \sum_{x_1,x_2=0}^{63} X_{1,Q,zz}[x_1] X_{2,Q,zz}[x_2] M_{Q,zz}[x_1, x_2, 0] \\ \vdots & & \vdots \\ Y_{Q,zz}[63] & = & \sum_{x_1,x_2=0}^{63} X_{1,Q,zz}[x_1] X_{2,Q,zz}[x_2] M_{Q,zz}[x_1, x_2, 63] \end{array}$$

Figure 4.7: The most likely way to evalute $Y_{Q,zz}$.

2. an integer z value indicating the zigzag ordered index of the nonzero element $M_{Q,zz}[x_1, x_2, z]$. Note that presence of a combination element in the list of (x_1, x_2) implies the zigzag ordered indices x_1 and x_2.

See Figure 4.6 for an illustration of this datastructure.

4.6.2 The convolution algorithm

In this subsection we will explain how to use the combination array to calculate Eq. 4.12.

The most likely way to evaluate Eq. 4.12 is to evaluate it one z value at a time, as in Figure 4.7.

But the equations could also be evaluated as given in Figure 4.8. Notice that for each iteration in the alternative evaluation, the (x_1, x_2) combination is constant within an iteration.

This is exactly why the method explained in Figure 4.8 is more suited for our purposes than the first method (Figure 4.7). If we would use the first method, we would have to visit all (x_1, x_2) combinations for every $z \in \{0, \ldots 63\}$ Keeping (x_1, x_2) constant during an iteration of the second method, means that we don't have to traverse the RLE lists (during that iteration); in the entire proces, each (x_1, x_2) is visited once at most.

Let's have another look at Figure 4.8. To compute $Y_{Q,zz}$ efficiently, we want to avoid unuseful calculations of $Y^{(i)}_{Q,zz}[z]$. To make the discussion easier, we define the notions (possibly) useful evaluation and useful iteration.

Iteration 0: $(x_1, x_2) = (0, 0)$

$$\begin{aligned}
Y^{(0)}_{Q,zz}[0] &= X_{1,Q,zz}[0]X_{2,Q,zz}[0]M_{Q,zz}[0,0,0] \\
Y^{(0)}_{Q,zz}[1] &= X_{1,Q,zz}[0]X_{2,Q,zz}[0]M_{Q,zz}[0,0,1] \\
&\vdots \quad\quad \vdots \\
Y^{(0)}_{Q,zz}[63] &= X_{1,Q,zz}[0]X_{2,Q,zz}[0]M_{Q,zz}[0,0,63]
\end{aligned}$$

Iteration 1: $(x_1, x_2) = (0, 1)$

$$\begin{aligned}
Y^{(1)}_{Q,zz}[0] &= Y^{(0)}_{Q,zz}[0] + X_{1,Q,zz}[0]X_{2,Q,zz}[1]M_{Q,zz}[0,1,0] \\
Y^{(1)}_{Q,zz}[1] &= Y^{(0)}_{Q,zz}[1] + X_{1,Q,zz}[0]X_{2,Q,zz}[1]M_{Q,zz}[0,1,1] \\
&\vdots \quad\quad \vdots \\
Y^{(1)}_{Q,zz}[63] &= Y^{(0)}_{Q,zz}[63] + X_{1,Q,zz}[0]X_{2,Q,zz}[1]M_{Q,zz}[0,1,63]
\end{aligned}$$

$\vdots \quad \vdots \quad \vdots$

Iteration i**:** $(x_1, x_2) = (i \textbf{ div } 64, i \bmod 64)$

$$\begin{aligned}
Y^{(i)}_{Q,zz}[0] &= Y^{(i-1)}_{Q,zz}[0] + X_{1,Q,zz}[i \text{ div } 64]X_{2,Q,zz}[i \bmod 64]M_{Q,zz}[i \text{ div } 64, i \bmod 64, 0] \\
Y^{(i)}_{Q,zz}[1] &= Y^{(i-1)}_{Q,zz}[1] + X_{1,Q,zz}[i \text{ div } 64]X_{2,Q,zz}[i \bmod 64]M_{Q,zz}[i \text{ div } 64, i \bmod 64, 1] \\
&\vdots \quad\quad \vdots \\
Y^{(i)}_{Q,zz}[63] &= Y^{(i-1)}_{Q,zz}[63] + X_{1,Q,zz}[i \text{ div } 64]X_{2,Q,zz}[i \bmod 64]M_{Q,zz}[i \text{ div } 64, i \bmod 64, 63]
\end{aligned}$$

$\vdots \quad \vdots \quad \vdots$

Iteration $64^2 - 1 = 4095$**:** $(x_1, x_2) = (63, 63)$

$$\begin{aligned}
Y^{(4095)}_{Q,zz}[0] &= Y^{(4094)}_{Q,zz}[0] + X_{1,Q,zz}[63]X_{2,Q,zz}[63]M_{Q,zz}[63,63,0] \\
Y^{(4095)}_{Q,zz}[1] &= Y^{(4094)}_{Q,zz}[1] + X_{1,Q,zz}[63]X_{2,Q,zz}[63]M_{Q,zz}[63,63,1] \\
&\vdots \quad\quad \vdots \\
Y^{(4095)}_{Q,zz}[63] &= Y^{(4094)}_{Q,zz}[63] + X_{1,Q,zz}[63]X_{2,Q,zz}[63]M_{Q,zz}[63,63,63]
\end{aligned}$$

where $Y_{Q,zz}[z] = Y^{(4095)}_{Q,zz}[z]$ for $z = 0 \ldots 63$.

Figure 4.8: Another way to evaluate $Y_{Q,zz}$.

```
for every x1 for which X1,Q,zz[x1] ≠ 0 do
    for every x2 for which X2,Q,zz[x2] ≠ 0 do
        if combination list of (x1, x2) not empty then
            for every entry z in combination list do
                calculate Y_{Q,zz}^{x1*64+x2}[z]
            od
        fi
    od
od
```

Figure 4.9: The convolve algorithm.

Definition 4.1 The evaluation of $Y^{(i)}_{Q,zz}[z]$ for some $z \in \{0, 1, \ldots, 63\}$ during some iteration $i \in \{0, 1, \ldots, 4095\}$ is a *possibly useful evaluation* if and only if

$X_{1,Q,zz}[i \text{ div } 64] \neq 0$ and $X_{2,Q,zz}[i \bmod 64] \neq 0$

A *useful evaluation* is a possible useful evaluation for which also holds that

$M_{Q,zz}[i \text{ div } 64, i \bmod 64, 1] \neq 0$

Definition 4.2 An iteration i is a *useful iteration* if and only if for some $z \in \{0, 1, \ldots, 63\}$, the evaluation of $Y^{(i)}_{Q,zz}[z]$ is useful. Otherwise the iteration i is *not useful*.

If we project Definition 4.1 and Definition 4.2 onto the combination array, we can conclude that an iteration is useful if the combination list of (x_1, x_2) — where x_1 and x_2 are entries in the RLE block of $X_{1,Q,zz}$ and $X_{2,Q,zz}$ respectively — is not an empty list.

The entries in the combination list for a certain x_1 and x_2 are exactly those entries of $M_{Q,zz}$ with $M_{Q,zz}[x_1, x_2, z] \neq 0$. The order in which the z coordinates occur in the combination list is not important for the method used in Eq. 4.8.

To make use of the fact that many entries of $X_{1,Q,zz}$ and $X_{2,Q,zz}$ are zero, we use the Runlength Encoded representation of $X_{1,Q,zz}$ and $X_{2,Q,zz}$. In this way, we skip iterations that are not useful at all. Note that the entries in an RLE block are ordered by the index of the nonzeroes within the 8x8 blocks.

Now we can give the idea of how to evaluate Eq. 4.9. For every $(x1, x2)$ combination — where x_1 and x_2 are the indices of nonzero values in $X_{1,Q,zz}$ and $X_{2,Q,zz}$ — we traverse the combination list of (x_1, x_2). Every combination element in the list represents an evaluation of iteration $x_1 * 64 + x_2$. See Figure 4.9.

Chapter 5

The modules

5.1 block.c

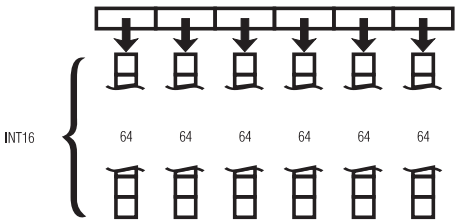


Figure 5.1: An MCU of blocks.

In this module, all functions concerning blocks are defined, such as allocating, freeing and setting to zero. These functions perform directly on a complete MCU, so the datastructure should be designed to store a complete MCU. The datastructure for an MCU is an array of pointers. Each pointer points to an array of 64 integer elements; this represents one block.

5.2 calc.c

In this module a function for integer rounding is given. This function was designed for quantization, but is used in other calculations as well.

5.3 huffman.c

The Huffman tables are provided by the image file; different image files could contain different Huffman tables. Mostly, separate Huffman tables are given for the DC and AC-coefficients and for the different components. In practice, the Huffman tables for the second and third component are the same.

As already mentioned in section 2.1.7, the aim of Huffman coding is to assign short codes to frequent (RUN, SIZE) combinations. If we were to take full advantage of Huffman coding, we would have to count all (RUN, SIZE) combinations in every image we would process, to construct an optimal Huffman table for that image. This is of course too much work; determination of frequencies of (RUN, SIZE) combinations, constructing the Huffman table and encoding the image would require two scans of the image. To save runtime, we use predefined tables.

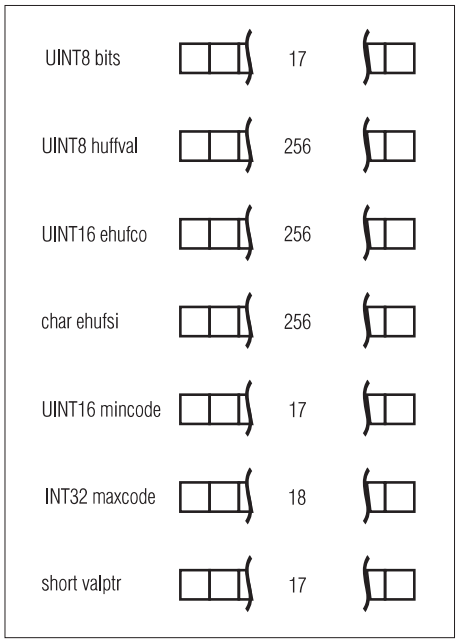


Figure 5.2: The Huffman structure.

5.3.1 Setting up Huffman tables

Setting up the Huffman tables is handled in the function fix_huff_tbl. The Huffman tables of JPEG images in the JFIF style are stored in Huffman table segments (see Chapter 6). A Huffman table segment contains the following:

- an array `bits[].bits[i]` represents the number of Huffman codes with i bits.

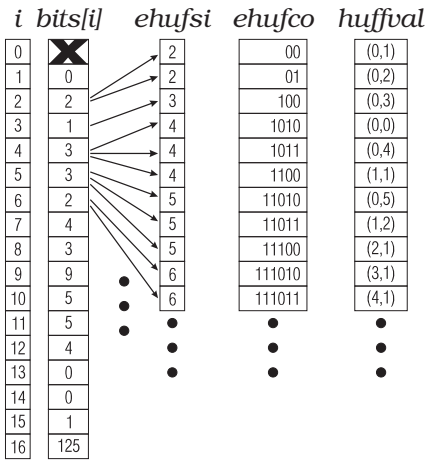


Figure 5.3: Part of a hufftable.

- an array `huffval[]`. `huffval[i]` gives the i^{th} (RUN,SIZE) combination (see Section 2.1.7) in the Huffman table.

The Huffman tables can be computed out of `bits[]` and `huffval[]`. The Huffman tables are stored by `bits[]` and `huffval[]` to decrease storage space.

Now we will generate the Huffman codes (the codes are to be stored in the array `ehufco[]`): Since we have the array `bits[]`, we know how many Huffman codes of length i should be generated. Suppose we are generating the j^{th} code of length i, and suppose the value of this code is C. The $(j+1)^{th}$ code of length i is $(C+1)$. This process should be repeated untill `bits[i]` tells us that we have generated all Huffman codes of length i.

We define C^i_{last} to be the value of the last Huffman code of length i and C^i_{first} to be the value of the first Huffman code of length i.

The following equation shows how the next Huffman code is generated if we have completed generating Huffman codes of length i:

$$C^{k+\alpha}_{first} = 2C^k_{last}$$

where

$$\alpha = \min\{\alpha \mid \alpha \geq 1 \wedge \texttt{bits[k+}\alpha\texttt{]} \neq 0\}.$$

See Figure 5.3. The first Huffman code to be used is 0.

For decoding purposes, it is useful to know where the codes of length i begin in the array `huffval[]`. This data is stored in the array `valptr[]`, `valptr[i]` is the index in the array `huffval[]`, where `huffval[valptr[i]]` is the first code of length i (see Figure 5.4). To calculate `valptr[]`, both `huffval[]` and `bits[]` are needed.

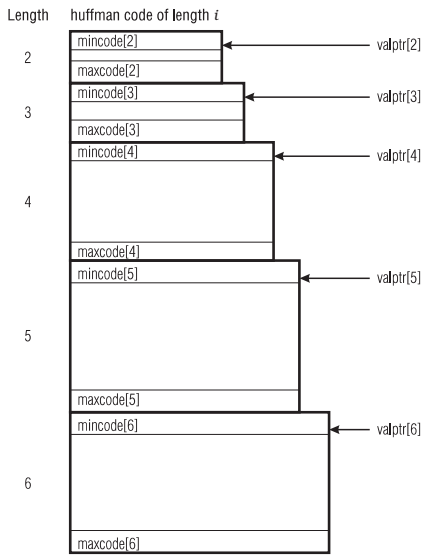


Figure 5.4: Mincode, maxcode and valptr.

5.3.2 Decoding

The input to the decoding step is a Huffman coded bitstream. The first couple of bits from the inputstream form an encoded DC-difference. The next values to be decoded are AC values, until 63 AC values have been read, or until an end of block marker has been read. The process of reading DC-differences and AC values is repeated until an end of image marker is encountered.

extracting Huffman codes

The process of extraction is implemented in the function huff_DECODE. Suppose the i^{th} bit has just been read. These i bits form a binary number $codevalue$. $codevalue$ is compared to `maxcode[`i`]`. If $codevalue >$ `maxcode[`i`]` then the i bits read so far don't form a Huffman code yet, so a next bit has to be read from the input stream. If $codevalue \leq$ `maxcode[`i`]` then the i bits form a valid Huffman code. Since matching Huffman values and Huffman codes have the same index to `huffval[]` and `ehufco[]` respectively, we need to compute the index of the Huffman code in the array `ehufco[]`. This computation makes use of the property that the values of consecutive Huffman codes of the same length have a difference of one. Within all codes of length i, the Huffman code just found is the $(codevalue - \texttt{mincode[i]})^{th}$ entry. `valptr[i]` gives the index of the first Huffman code of length i within the complete array `ehufco[]`. Now we can compute the index of the Huffman code, which is also the index of the matching Huffman value:

`valptr[i]` + $codevalue$ - `mincode[i]`.

decoding DC-differences

All Huffman codes are stored in the array `ehufco[]` and with `valptr[]`, `mincode[]` and `maxcode[]` it is easy to determine the index of the Huffman code in the array `ehufco` as described in Section 5.3.2. The index found in the Extracting step is used in the array `huffval[]`, to find the matching SIZE with this Huffman code. This SIZE is the number of bits, that has to be read from the input stream, to find the two's complements representation of the DC-difference.

decoding AC values

Again bits are extracted (from the proper AC Huffman table) until a Huffman code is found. The matching `huffval[]` gives a (RUN, SIZE) combination. RUN is the number of zeroes preceding the AMPLITUDE. Again SIZE is the number of bits to be read to find the two's complement representation of the AC value.

5.3.3 Encoding

The input to the Huffman encoding process is a number of runlength encoded vectors see Subsection 2.1.6.

encoding DC-differences

Suppose a DC-difference is to be encoded, and that the number of bits needed to represent this DC-difference is SIZE. The proper base code (the first couple of bits of the Huffman code) is SIZE. The last part of the code can be calculated by taking the two's complement representation of the DC-difference.

encoding AC values

Suppose an AC-coefficient is to be encoded, and that the number of bits needed to represent this AC-coefficient is SIZE. And suppose that the number of zeroes previous to the AC-coefficient is RUN. The proper base code (the first bits of the Huffman code) is the 4 bits binary representation of RUN, followed by a 4 bits SIZE (The (RUN, SIZE) pair is stored in one byte). The next part of the Huffman code can be calculated by taking the two's complement representation of the AC value.

A special base code is inserted for runs of 15 zeroes followed by a zero. This is stored under (RUN, SIZE) pair (0xF, 0). Another special base code to mark an end of block exists: (RUN, SIZE) pair is (0, 0).

5.4 image.c

In the function allocate_image_struct, memory is allocated to store a complete image structure. An image structure not only contains data such as size, number of components and the filename of the image, but also data needed for Huffman coding and quantization.

All pointers in this structure are initialized with NULL in case some pointers are not used; in order for release_image to work properly, unused pointers are assumed to be NULL. See Section 5.6 for details.

In release_image, all memory used for the image structure is released. Before the structure itself can be released, all memory allocated to the pointers in the image structure has to be released.

The function init_output_image initializes the output image structure by copying the input image structure. In case of an image operation with two input images, the structure of the first image is used to initialize the structure for the output image.

There are two ways to copy a structure in C. The first possibility is to copy the contents of the structure one field at a time.

The second (and faster) solution, is to copy the entire memory occupied by the structure, using memmcpy. In this case, special care has to be taken of the pointers in the structure. The pointers in the output image structure point to the same memory areas as the pointers in the input image structure. Changing some memory in the input image structure would result in a change in the output image structure and vice versa.

To avoid this, we allocate a different area of memory and copy the allocated memory areas of the input image structure into these newly allocated areas.

The function images_compatible checks whether two images are suitable for pixel addition and pixel multiplication.

Two images are *compatible* if the heigth, the width and the number of components are the same.

The functions open_image and close_image are used for file I/O.

5.5 jfif.c

The routines in this module, rely heavily on the JFIF standard. See Chapter 6 for an overview of the JFIF standard.

In copy2bytes, two bytes are read from the input stream and copied to the output stream. The two bytes form a 16 bits integer. This function is mostly used to read (and write) the length of the segment to be processed.

copy_next_marker looks for the next segment marker. A segment is marked by two bytes: 0xFF followed by

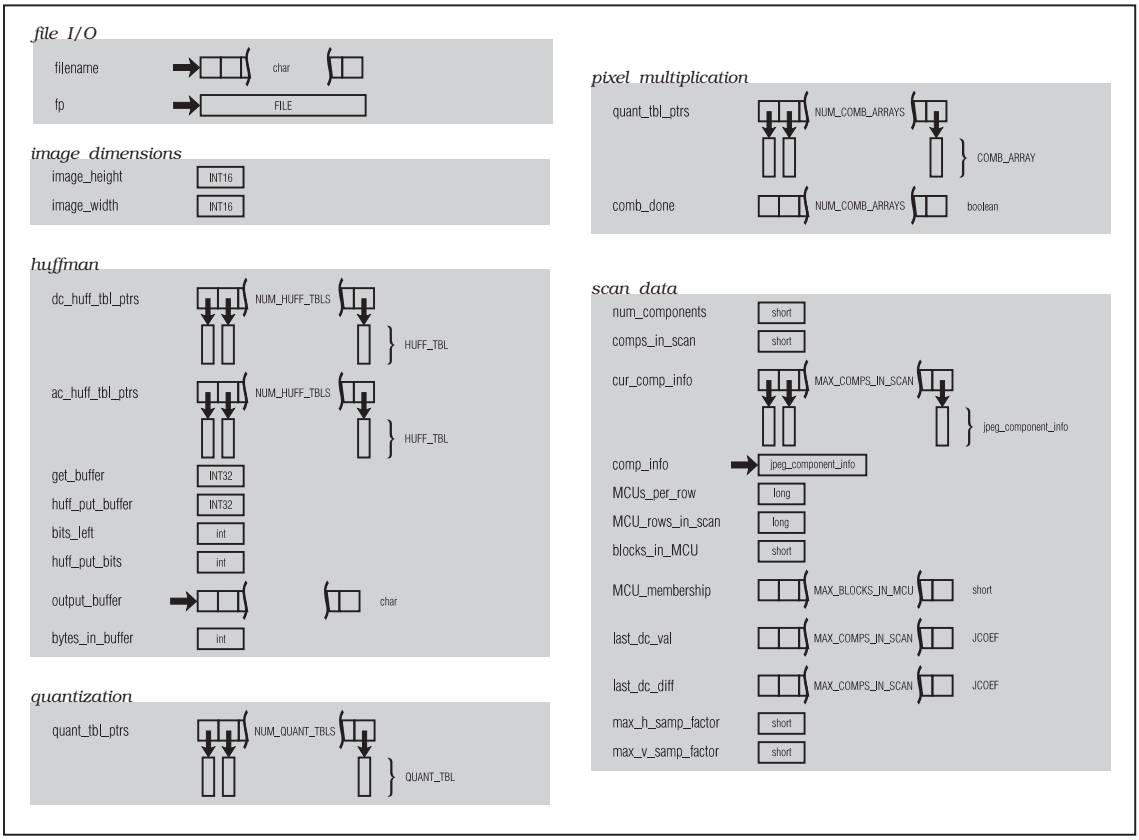


Figure 5.5: Image_struct: the image structure.

another byte which specifies the segment. See Chapter 6 for a list of markers.

The copy_segment function copies an entire segment without processing the stored information. This function is used for segments that are not needed for our operations.

get_dqt is used to read and write a quantization segment. One quantization segment can contain several quantization tables. Each table has a unique number to identify this table. This number is also stored in this segment.

get_dqt2 reads a quantization segment but writes quantization tables in which all entries are set to one.

read_dqt only reads a quantization segment. This is used to process the quantization segment of the second input image of pixel addition and pixel multiplication.

get_dht reads and writes a Huffman segment (read_dht only reads the segment). First, the number of the table is read from the input stream. This number also determines whether the following table is an AC or DC table. Following this identifying number are the arrays `bits[]` and `huffval[]`. See Section 5.3.1 for an explanation on these arrays.

5.6 memmgr.c

In this module, the allocation and release routines are defined.

In order for the release routine to work properly, all pointers, not attached to an allocated piece of memory, should be NULL; on some systems, releasing non-defined pointers is not allowed.

There are two allocation routines: allocate and large_allocate. This is done to support MS-DOS machines, which do not have enough memory for some operations. Using ‘far memory’ solves this problem, but this makes memory management more difficult for the operating system and therefore less efficient. Another drawback is that allocation of ‘near memory’ requires a different function than allocation of ‘far memory’. Therefore two different allocation routines are needed for MS-DOS machines.

UNIX based machines (and most other operating systems) do not divide memory into ‘far memory’ and ‘near memory’; all memory can be considered to be one contiguous area. As a consequence, one function suffices to allocate all memory available in the system. For UNIX based machines, the routines allocate and large_allocate

are the same, two different routines were made for portability between MSDOS and non-MSDOS systems.

5.7 pmul.c

In this module, all functions concerning combination arrays and the convolution algorithm (as described in [Smith]) are defined. These functions are needed for pixel multiplication. The functions C_init and W_init calculate Eq. 2.2 and Eq. 4.11 respectively.

Because of the finite precision used in computers, small values cannot be trusted. Therefore, in W_init, small values in a specified range are set to zero to increase performance in convolve_init. This thresholding is not neccesary for C_init as the calculated values do not suffer from finite precision. The results of C_init and W_init are stored in tables and these tables are used in the function convolve_init_prec.

In convolve_init_prec the combination arrays are built. Our implementation is much the same as described in [Smith] but in order to limit the number of entries in the combination lists, small values outside a specified range (user defined, see Figure 5.6) are discarded.

prec	2	3	4	5
factor= $2^{4prec-1}$	128	2048	32768	524288
range	$-\frac{1}{factor} \cdots \frac{1}{factor}$			

Figure 5.6: The relation between prec, factor and range.

Small combination lists result in a faster convolution algorithm, but discarding too many values would result into inaccurate calculations. For example, precision=2 will most likely give a black output picture. precision=5 will probably take more time than precision=3,but the output image will not differ too much.

To avoid floating point calculations, we multiplied every W value (see Figure 4 in [Smith]) by a user defined factor (see Figure 5.6) and used integer rounding on the result. Because of performance reasons, we use $factor > 10^{prec}$, where $factor$ a power of 2. See Figure 5.6.

Convolve_init is almost the same as convolve_init_prec, the only difference is that precision is now a constant value.

5.8 rle.c

The function rle2block converts a runlength encoded MCU into an MCU of 8x8 blocks. Actually the blocks are 64 element vectors, but thinking of them as blocks makes things easier to understand. block2rle converts an MCU of 8x8 blocks into a runlength encoded MCU. These two functions are used in the brute force versions of scalar multiplication, scalar addition, pixel addition and pixel multiplication. When performing brute force operations, we would like our operations to be performed on the bitmaps. So it should be obvious to use 8x8 blocks.

The function block2rle is also used for pixelwise addition and for scalar- and pixelwise multiplication (for a further explanation see Section 5.9).

Other functions in this module are allocate_RLE, init_RLE and release_RLE.

5.9 The main modules

For an overview of the main modules see Figure 5.7 and Figure 5.13. The following is handled in every main function described in this section:

- The input and output files are opened.
- It is checked whether the first input image is in the JFIF standard, and the header is copied.
- This step is only needed for pixelwise operations:
  - The second input image is checked for the JFIF standard.
  - The first and second input files are checked on compatibility, see Section 5.4
- The config file is read. If there is no config file, default values are used.
- The actual operations are handled (this part differs for every operation).
- the structures that were used used are freed

For an overview of the steps taken in the non-brute-force implementation of the operations see Figure 5.8. Step 2 also involves quantization and if needed, denormalization and normalization.

Now we take a closer look at scalar multiplication, pixel addition, pixel multiplication and we will make some general remarks for the brute force implementations. For the different implementations of scalar addition, it suffices to refer to Section 4.3.

5.9.1 Scalar multiplication

There are four different flavours for scalar multiplication. There is a DC-coefficient oriented version and a DC-difference oriented version. Both versions can be either RLE-in-RLE-out or RLE-in-block-out.

OPERATIONS	
smval1.c	scalar multiplication using DC-coefficients, RLE-in-RLE-out oriented
smval2.c	scalar multiplication using DC-coefficients, RLE-in-block-out oriented
smdif1.c	scalar multiplication using DC-differences, RLE-in-RLE-out oriented
smdif2.c	scalar multiplication using DC-differences, RLE-in-block oriented
saval.c	scalar addition using DC-coefficients
sadif.c	scalar addition using DC-differences
paval1.c	pixel addition using DC-coefficients, block-in-block-out oriented
paval2.c	pixel addition using DC-coefficients, RLE-in-block-out oriented
padif1.c	pixel addition using DC-differences, block-in-block-out oriented
padif2.c	pixel addition using DC-differences, RLE-in-block-out oriented
pm.c	pixel multiplication
pmprec.c	pixel multiplication using user defined precision

Figure 5.7: An overview of the main modules.

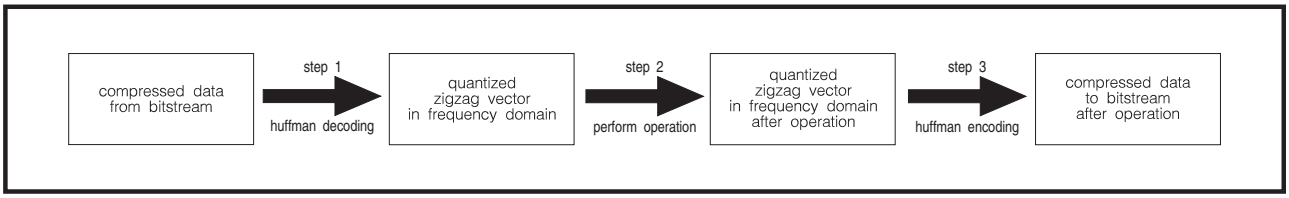


Figure 5.8: The smart way.

A discusion of DC-coefficient oriented versus DC-difference oriented was given in Section 4.2.

an RLE-in-RLE-out oriented implementation operates directly on the runlength encoded block (RLE-block). This could be dangerous, specially if a small scaling factor is applied to a small amplitude in the RLE-block; if the scaled amplitude is in the range $0 \ldots \frac{1}{2}$, then integer rounding will set the resulting amplitude to 0 (See Figure 5.9). This would mean that a zero-amplitude would appear in the RLE-block. Even though the RLE-block was introduced to leave out zeros! We call these zero-amplitudes *ghost-amplitudes*.

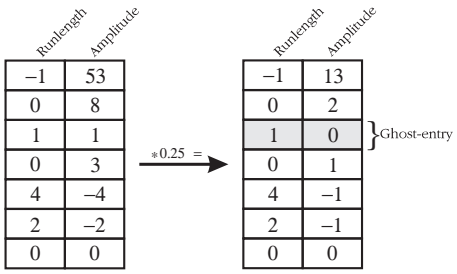


Figure 5.9: Scalar multiplication: RLE-in-RLE-out and ghost-amplitudes.

An option could be to rearrange the RLE-block, but we expect this to cost too much overhead. Therefore, another option was chosen; we read the incoming amplitudes from the RLE-blocks, but we store the outgoing amplitudes directly in a block (See Figure 5.10). The index of a resulting amplitude in the output block can be calculated, using the runlengths of the current and previous amplitudes.

Ofcourse this method has a drawback; an additional conversion — runlength encoding, from block to RLE-block — has to be applied for each MCU. This takes time. But the result is mostly better — a better compression ratio is achieved.

5.9.2 Pixel addition

The implementation of pixel addition comes in four different flavours. Again there is a DC-coefficient oriented version and a DC-difference oriented version. Now both versions can be either block-in-block-out or RLE-in-block-out.

Note that we left out an RLE-in-RLE-out version. We expect this version to take too much overhead because we have to merge the incoming RLE-blocks; Often, two matching block entries — for example, the two Y-components of the pixels at (x_i, y_i) of the input images — do not match in the RLE-blocks because one of the two is missing in the RLE-block, due to an amplitude of zero.

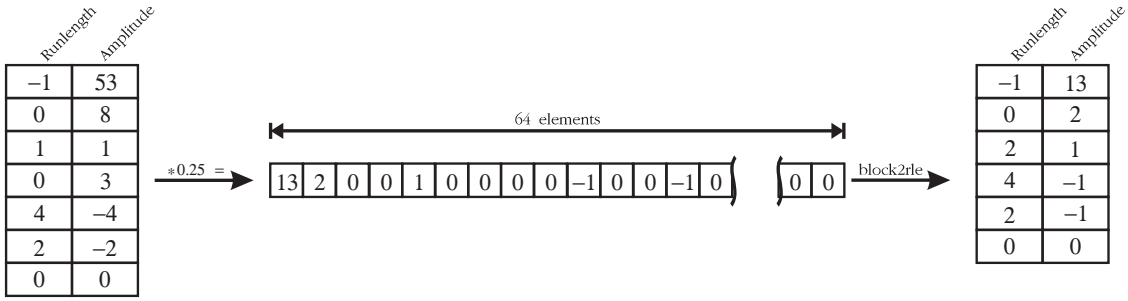


Figure 5.10: Scalar multiplication: RLE in zigzagged vector out.

The most straightforward way to overcome this problem is to convert the incoming RLE-blocks to normal blocks and perform pixel addition in a matrix addition like manner. Figure 5.11 illustrates this method with all quantization table entries set to 1. The drawback of this method is that lots of unuseful additions could be done; many entries in the blocks are zero, so it is likely that two zeroes are added.

This leads to another implementation. We return to the idea of merging the RLE-blocks, but now we perform the merging process during the pixel addition, and the merging takes place in a block; The first input RLE-block is taken and its amplitudes are stored in the output block using the runlenghts for each amplitude (an RLE-block to block conversion). Then the second input RLE-block is taken and its amplitudes are added to the matching entries in the output block, again using the runlenghts for each amplitude. An example can be found in Figure 5.12. Again, all quantization table entries are set to one.

Section 7.3 shows that the RLE-in-block-out approach often saves time.

5.9.3 Pixel multiplication

For pixel multiplication there is one ‘normal version’ and one version where a user defined precision is needed (for an explanation see below). For both pixel multiplication versions, DC-coefficients are being used. The module pmprec.c contains the main function of pixel multiplication with a user defined precision.

The actual work for pixel multiplication is handled in pmul.c, see Section 5.7. The main modules for pixel multiplication — pm.c and pmprec.c — are almost the same, the only difference is that pm.c uses a constant precision for the calculations instead of a user defined precision. This constant precision (3) was found by experimenting and suffices for most images. The advantage of a constant precision over a user defined precision is that less overhead — i.e. less calculations — is needed to compute one block entry. As a result, time is saved.

5.9.4 The brute force operations

A brute force algorithm can be performed as displayed in Figure 5.14. As can be seen in Figure 5.13 there are four versions of every brute force operation.

For an overview of the steps needed for a brute force implementation of an operation see Figure 5.14. The reverse- and forward zigzag steps (steps 3 and 7) can be skipped. These steps can be omitted, because the position within the input matrix is not important for the performance of the operations. Applying zigzag increases execution time, because more work is needed. Steps 4 and 6 in the algorithm (the reverse- and forward DCT steps) can also be executed using a fast DCT algorithm. The smart implementation is based on an algorithm described by [Loef].

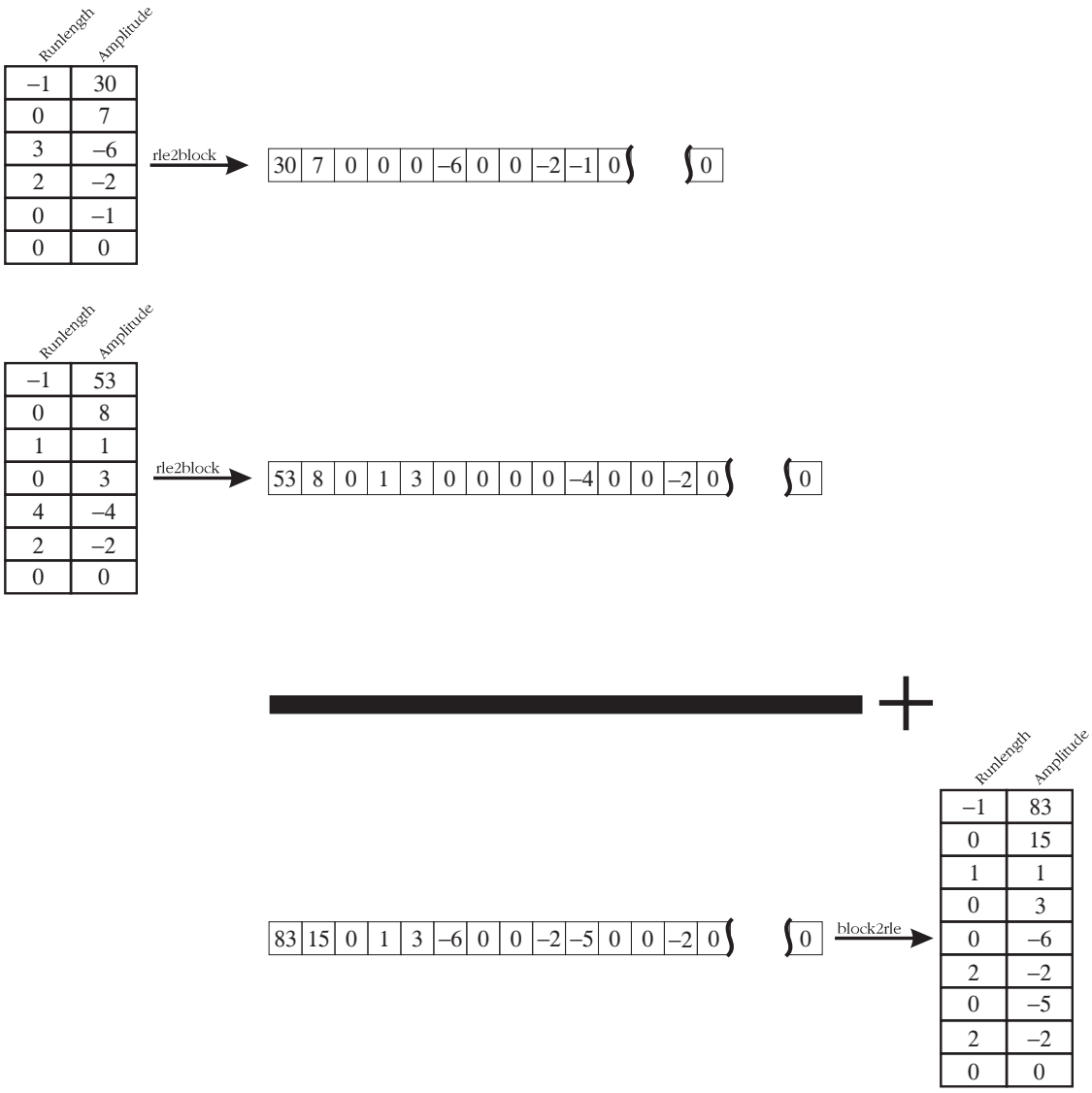


Figure 5.11: Pixel addition: block in block out.

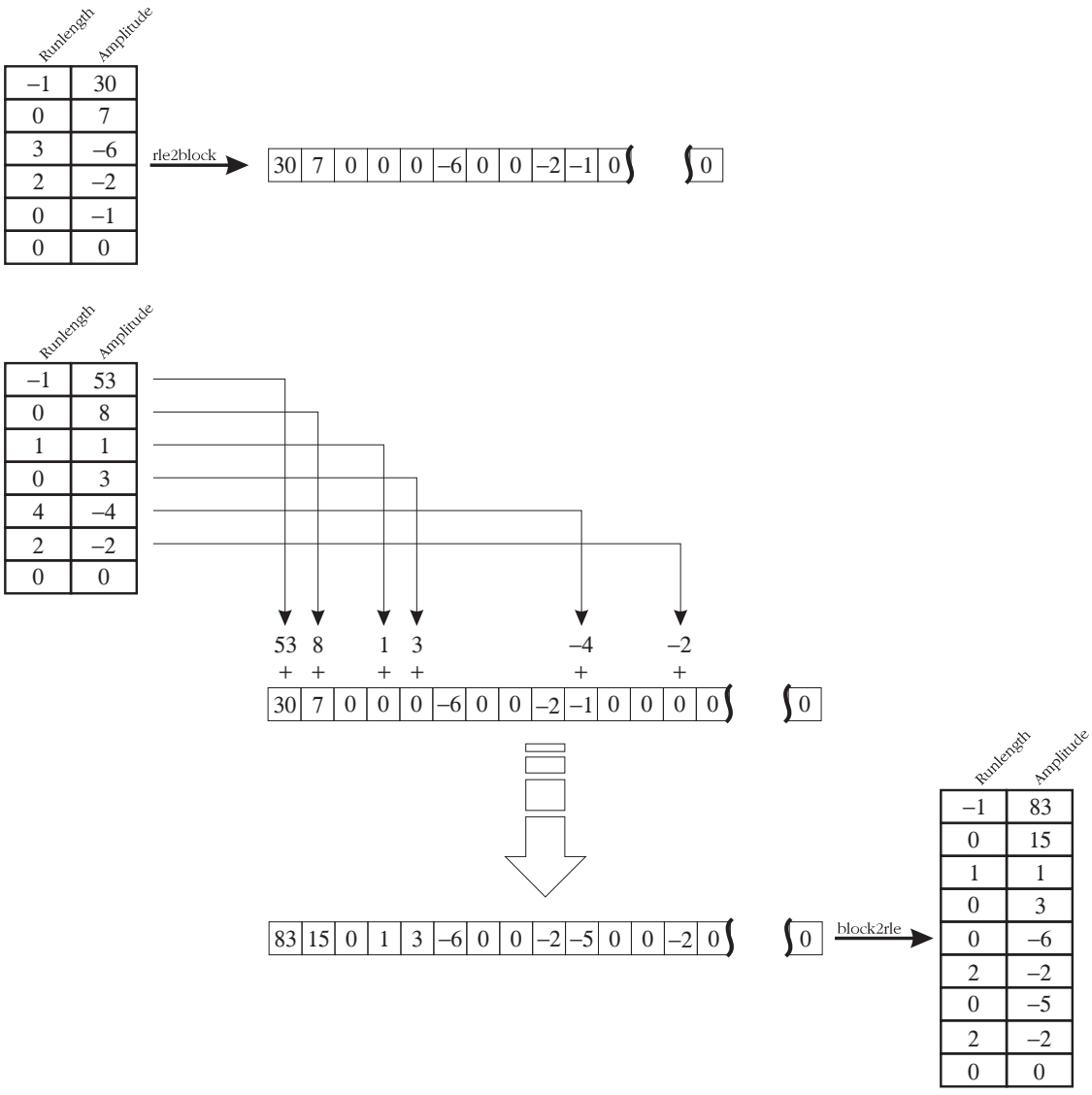


Figure 5.12: Pixel addition: RLE in block out.

BRUTE FORCE	
smbd.c	brute force scalar multiplication using brute force DCT
smzzbd.c	brute force scalar multiplication with zigzag using brute force DCT
smsd.c	brute force scalar multiplication using smart DCT
smzzsd.c	brute force scalar multiplication with zigzag using smart DCT
sabd.c	brute force scalar addition using brute force DCT
pazzbd.c	brute force scalar addition with zigzag using brute force DCT
sasd.c	brute force scalar addition using smart DCT
pazzsd.c	brute force scalar addition with zigzag using smart DCT
pabd.c	brute force pixelwise addition using brute force DCT
pazzbd.c	brute force pixelwise addition with zigzag using brute force DCT
pasd.c	brute force pixelwise addition using smart DCT
pazzsd.c	brute force pixelwise addition with zigzag using smart DCT
pmbd.c	brute force pixelwise multiplication using brute force DCT
pmzzbd.c	brute force pixelwise multiplication with zigzag using brute force DCT
pmsd.c	brute force pixelwise multiplication using smart DCT
pmzzsd.c	brute force pixelwise multiplication with zigzag using smart DCT

Figure 5.13: An overview of the main brute force modules.

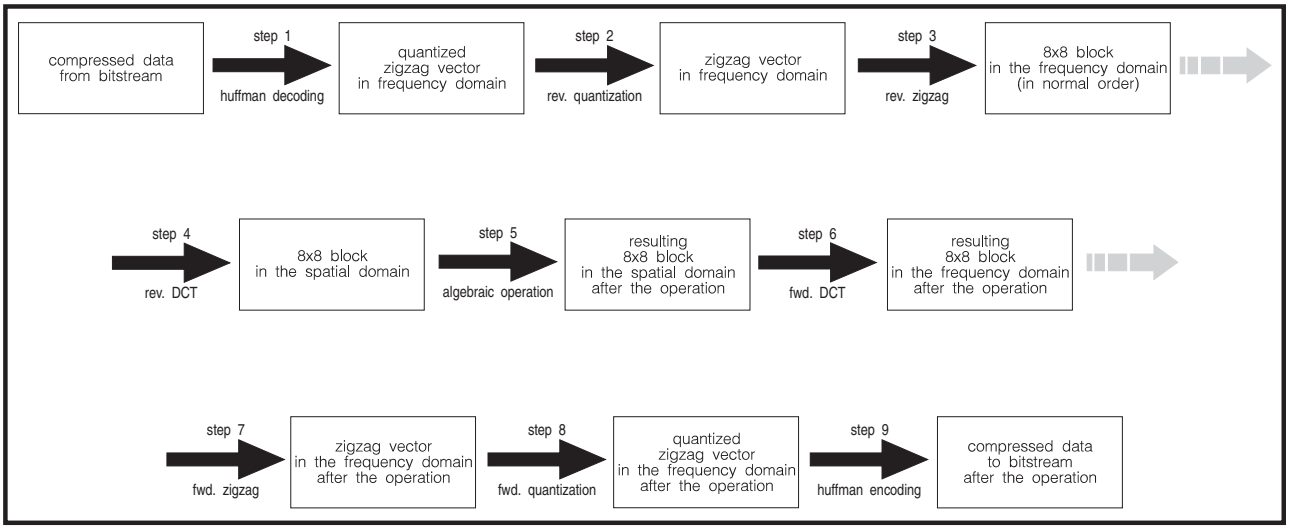


Figure 5.14: The brute force way.

Chapter 6

JFIF

Most JPEG-images are distributed in the *JFIF* (JPEG File Interchange Format) *standard*. The purpose of the JFIF standard is to allow the exchange of JPEG compressed images. The JFIF standard is compatible with the standard JPEG interchange format, and meets with the requirements of the JPEG Draft International Standard.

The JPEG interchange format requires that all tables needed in the encoding process are put in the bitstream before they are used. This is also required by the JFIF standard.

The colour space used in the JFIF standard is YCbCr. (if needed the YCbCr colour space can be converted into the RGB colour space, or vice versa). The first component of the YCbCr colour space determines the luminance, the second and third component determine the chrominance. A component contains 8-bit values. The first component (luminance) is in the range [0...255], and the second and third components (chrominance) have values in the range of [-128...127]. A picture containing only one component (the Y of the YCbCr colour space) will be a greyscale image.

6.1 Markers

A JFIF marker is defined by 0xFF followed by the marker code. In the file jfif.h the definitions of the codes can be found. A marker is always followed by two bytes which give the length of the total segment (except for the SOI marker, which is directly followed by the APP0 marker) . All markers, belonging to useful segments of the project, are briefly described below. For a list of markers of useful segments (a useful segment contains data that is needed for the project) see Table 6.1. The other markers and segments are just copied into the output image data.

SOI marker

According to the JFIF standard, an image has to begin with a start of image marker (SOI). Directly after the SOI marker should be a next marker (APP0). If the segment (identified by this marker) doesn't contain any useful information, it is just copied into the output image data, and a next marker is read. If the segment does contain useful information, the needed data is extracted, and then copied into the output image data. This process of reading and copying data is repeated until the SOS marker is encountered.

```
M_SOF0 = 0xc0, Start of Frame marker
M_DHT  = 0xc4, Def. Huffman Table(s)
M_SOI  = 0xd8, Start of Image
M_EOI  = 0xd9, End of Image
M_SOS  = 0xda, Start of Scan
M_DQT  = 0xdb, Def. Quantization Table
M_APP0 = 0xe0, Application marker
```

Figure 6.1: A list of used markers.

APP0 marker

In order to identify a JFIF compressed image, an APP0 marker is used. This marker has to be added right after the SOI marker. The JFIF APP0 marker provides an image with additional data, such as:

- version number
- X and Y pixel density (dots per inch or dots per cm)
- pixel apect ratio
- thumbnail.

Additional APP0 marker segment(s) can be used to specify JFIF extensions and for application-specific information. The additional APP0 marker segment(s) is/are optional.

The additional information provided by the APP0 marker is not used in the project.

SOF0 marker

A start of frame segment is recognized. A start of frame segment contains useful data, such as:

- image height,
- image width,
- number of components,
- horizontal sampling factors,
- vertical sampling factors.

Horizontal and vertical sampling factors are used to calculate the number of blocks per MCU.

DHT marker

A Huffman table segment is identified. The data needed for setting up the Huffman tables is stored in this segment.

- index,
- `bits[]`,
- `huffval[]`.

Index is needed to identify the Huffman table that is to be processed. The use of `bits[]` and `huffval[]` is explained in section 5.3.1.

DQT marker

A quantization table segment marker is recognized. The data, concerning the quantization table(s), is stored in this segment:

- index,
- quantization table.

Every quantization table has its own index. The use of the quantization tables is explained in section 2.1.4

SOS marker

A start of scan marker is recognized. A SOS segment contains information on components in scan. As we assume that a file contains only one image, we don't really use this information

After a start of scan segment has been processed all data needed for encoding the Huffman coded bitstream has been processed, (according to the JFIF standard). The remaining data following the start of scan segment, forms the Huffman encoded image in the frequency domain. Within the scan data, no marker should appear. If by any change a 0xFF should appear in the scan data, the processing application could interpret this as the beginning of a marker. To tell the application that the 0xFF is not part of a marker but part of the scan data, 0xFF is followed by a zero byte. Whenever 0xFF is encountered, the application should check for a folllowing zero byte. If this byte is not present, the 0xFF and the non zero byte are interpreted as a JFIF marker. If the zero byte is present, this zero byte is discarded and the next byte is used for the scandata.

EOI marker

The End of Image (EOI) marker is used to define the end of the scandata.

6.2 A JFIF example

In this section we will illustrate the JFIF format with an example. As mentioned in Section 6.1, each block (or segment) starts with four bytes: two bytes to indicate what kind of information is stored in the block (the marker) and two bytes to indicate the length of the block, including the two bytes representing the length but excluding the two bytes that form the marker. A program *readblock* was written which takes a JFIF style JPEG image as input, and identifies the blocks in the file. The output of readblock for each segment is a JFIF marker, the length of the segment and the data in the segment (all in hexadecimal representation). The picture Senna.jpg was input to readblock, and the output is given in Figure 6.2.

First, readblock checks the first two bytes of the JFIF file. These two bytes should be 0xFF and 0x8D, together they form the Start of Image marker (M_SOI). The SOI marker is discarded.

The first marker following SOI is 0xFF 0xE0 (M_APP0, see Figure 6.1). An APP0 segment starts with the hexadecimal codes 0x4A, 0x46, 0x49, 0x46, 0x00. These codes form the ASCII representation of the zero terminated string "JFIF". The next two bytes indicate the JFIF version of the file. Senna.jpg is in the JFIF style version 1.01. The following bytes are not needed for our purposes, but for an overview see [Hami].

The order of the segments after APP0 is not important, so the order in which the segments of the example are discussed is not a mandatory order.

The next two segments store the quantization tables. After the marker (0xFF 0xDB) and the length (next two bytes), one byte is used to specify both the precision of the table entries (quantization table entries can be one byte values or two byte values), as well as a unique identification number for the table to be read. The following 64 or 128 bytes (depending on the precision) are the quantization table entries. In this example each

quantization table has its own segment, but it is possible to store several tables in the same segment. The first quantization table has index 0x00 (0), its first entry is 0x02 (2), and its last entry is 0x0A (10). The second quantization table has index 0x01 (1), its first entry is 0x02 (2), and its last entry is 0x0A (10).

Following the quantization segments is the SOF0 segment. After the 4 bytes that form the marker and the length specification (0xFF 0xC0 0x00 0x11) is one byte to define the number of bits per pixel component value (0x08). The next 4 bytes define the heigth and width of the image. In our example, the image is 0x018B (395) pixels high and 0x029B (667) pixels wide. The next bytes specify the number of components of the image, which in our example is 0x03 (3). This segment is closed with information for each component of the image. The information for each component consists of an index for each component, the horizontal and vertical sampling factors for each component and the index of the quantization table to be used with this component. The first component of our image has index 0x01 (1) and has sampling factor 0x22 which means a horizontal sampling factor of 2 and a vertical sampling factor of 2. The next byte tells that quantization table 0x00 is to be used with component 1. The information for component 2 and component 3 is dealt with in a similar manner.

The next four segments are Huffman table segments. As mentioned in Section 5.3, two arrays (`bits[]` and `huffval[]`) are needed to construct the Huffman tables. The first byte after the marker and the length bytes, indicates both the index of the Huffman table and the type of the Huffman table; a Huffman table for DC-coefficients or a Huffman table for AC-coefficients. The next 16 bytes are the entries of the `bits[]` array. Define n as $n = \sum_{i=1}^{16} \texttt{bits}[i]$. Then the next n bytes are the entries for the `huffval[]` array. In Senna.jpg, each table has its own segment, but it is allowed to store several tables in one segment.

The last segment before the scan data is the Start of Scan segment. After the marker and length bytes, the number of components are specified in one byte. Then for each component the following information is supplied: the component index, the index of the matching Huffman table for the DC-coefficients and the index of the matching Huffman table for the AC-coefficients. In our example, there are three components (the first 0x03 in the segment). The next 0x01 (1) and 0x00 (0) tell that component 1 uses AC-Huffman table 0 and DC-Huffman table 0. Component 2 and 3 use AC-Huffman table 1 and DC-Huffman table 1 (0x02 0x11 0x03 0x11). The rest of the bytes in this segment have an unknown purpose.

```
Filename     : /home/gkumara/pic/Senna.jpg
Total length : 95094 (decimal)
==============


Marker           : FF E0
Length of block  : 0010 (=16)
 4A  46  49  46  00  01  01  00  00  01  00  01  00  00


Marker           : FF DB
Length of block  : 0043 (=67)
 00  02  01  01  01  01  01  02  01  01  01  02  02  02  02  02
 04  03  02  02  02  02  05  04  04  03  04  06  05  06  06  06
 05  06  06  06  07  09  08  06  07  09  07  06  06  08  0B  08
 09  0A  0A  0A  0A  0A  06  08  0B  0C  0B  0A  0C  09  0A  0A
 0A


Marker           : FF DB
Length of block  : 0043 (=67)
 01  02  02  02  02  02  02  05  03  03  05  0A  07  06  07  0A
 0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A
 0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A
 0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A  0A
 0A


Marker           : FF C0
Length of block  : 0011 (=17)
 08  01  8B  02  9B  03  01  22  00  02  11  01  03  11  01


Marker           : FF C4
Length of block  : 001F (=31)
 00  00  01  05  01  01  01  01  01  01  00  00  00  00  00  00
 00  00  01  02  03  04  05  06  07  08  09  0A  0B


Marker           : FF C4
Length of block  : 00B5 (=181)
 10  00  02  01  03  03  02  04  03  05  05  04  04  00  00  01
 7D  01  02  03  00  04  11  05  12  21  31  41  06  13  51  61
 07  22  71  14  32  81  91  A1  08  23  42  B1  C1  15  52  D1
 F0  24  33  62  72  82  09  0A  16  17  18  19  1A  25  26  27
 28  29  2A  34  35  36  37  38  39  3A  43  44  45  46  47  48
 49  4A  53  54  55  56  57  58  59  5A  63  64  65  66  67  68
 69  6A  73  74  75  76  77  78  79  7A  83  84  85  86  87  88
 89  8A  92  93  94  95  96  97  98  99  9A  A2  A3  A4  A5  A6
 A7  A8  A9  AA  B2  B3  B4  B5  B6  B7  B8  B9  BA  C2  C3  C4
 C5  C6  C7  C8  C9  CA  D2  D3  D4  D5  D6  D7  D8  D9  DA  E1
 E2  E3  E4  E5  E6  E7  E8  E9  EA  F1  F2  F3  F4  F5  F6  F7
 F8  F9  FA


Marker           : FF C4
Length of block  : 001F (=31)
 01  00  03  01  01  01  01  01  01  01  01  01  00  00  00  00
 00  00  01  02  03  04  05  06  07  08  09  0A  0B


Marker           : FF C4
Length of block  : 00B5 (=181)
 11  00  02  01  02  04  04  03  04  07  05  04  04  00  01  02
 77  00  01  02  03  11  04  05  21  31  06  12  41  51  07  61
 71  13  22  32  81  08  14  42  91  A1  B1  C1  09  23  33  52
 F0  15  62  72  D1  0A  16  24  34  E1  25  F1  17  18  19  1A
 26  27  28  29  2A  35  36  37  38  39  3A  43  44  45  46  47
 48  49  4A  53  54  55  56  57  58  59  5A  63  64  65  66  67
 68  69  6A  73  74  75  76  77  78  79  7A  82  83  84  85  86
 87  88  89  8A  92  93  94  95  96  97  98  99  9A  A2  A3  A4
 A5  A6  A7  A8  A9  AA  B2  B3  B4  B5  B6  B7  B8  B9  BA  C2
 C3  C4  C5  C6  C7  C8  C9  CA  D2  D3  D4  D5  D6  D7  D8  D9
 DA  E2  E3  E4  E5  E6  E7  E8  E9  EA  F2  F3  F4  F5  F6  F7
 F8  F9  FA


Marker           : FF DA
Length of block  : 000C (=12)
 03  01  00  02  11  03  11  00  3F  00


This program has come to an unknown block-structure.
Nothing to worry about, I probably found an SOS
(Start Of Scan) Marker, so in fact this is good news!
```

Figure 6.2: An example of the output of *readblock* which identifies blocks within a JFIF file.

Chapter 7

Results

In this chapter we will give our results from several tests. Our testset is given in Figure 7.2. For our testset we tried to use testimages of a wide variety. In order to accomplish this we used one big and one small one-component (greyscale) image as well as one big and one small colour (three components) image as testimages.

In order to see if 'smoothness' would make any difference to the output image or the execution time of our smart algorithm, we have also included a 'raw' and a 'smooth' version. A raw image can display details with pixel precision whereas a smooth image tends to smear out such details.

For every testimage we also used three different quality settings to determine whether a different quality setting would make any difference to the performance of the algorithms. The input images of the lowest quality,used in our testset, are still of a good visual quality.

Our program runs on a multiuser system, but our tests were done during off peak periods. The computer used was a Hewlett Packard 9000/720 running version E of release A.09.03 of the HP-UX operating system.

With the results we want to show how the different methods perform on a particular image. The times in the tables are averages of ten runs and are given in seconds.

As the times on the HP's were fluctuating due to an inconsistent systemload, we also timed the algorithms on an Indy workstation of Silicon Graphics Inc. (using release 5.2 of the IRIX operation system), in the hope to find more consistent values. As the Silicon Graphics Indy is not as busy as the HPs and the systemload is more consistent, this hope was justified.

The results are given in Appendix B. The results of the pixel multiplication algorithms are omitted; we did not succeed to run the pm and pmprec algorithms on the Silicon Graphics Indy, due to porting problems. As the results on the Indy are fairly consistent, the times in the tables are averages of five, instead of ten, runs.

We decided to use the results found using the HPs, even though these times are less consistent, because of the absence of the results of the pixel multiplication algorithms.

val	-	DC-coefficients
diff	-	DC-differences
bd	-	without zigzag with brute force DCT
zzbd	-	with zigzag and brute force DCT
sd	-	without zigzag with smart DCT
zzsd	-	with zigzag and smart DCT
valprec 3	-	3 digits precision
valprec 4	-	4 digits precision
valprec 5	-	5 digits precision

Table 7.1: List of abbreviations.

image	quality	type	bytes	size	number of RLEs	total RLE length	average RLE length
dont.panic.smooth.50.jpg	50%	Greyscale	60720	1280 x 852	17120	103971	6.07
dont.panic.smooth.75.jpg	75%	Greyscale	96096	1280 x 852	17120	155169	9.06
dont.panic.smooth.100.jpg	100%	Greyscale	520990	1280 x 852	17120	806592	47.11
dont.panic.smooth.upsidedown.50.jpg	50%	Greyscale	60521	1280 x 852	17120	103826	6.06
dont.panic.smooth.upsidedown.75.jpg	75%	Greyscale	95947	1280 x 852	17120	155338	9.07
dont.panic.smooth.upsidedown.100.jpg	100%	Greyscale	364687	1280 x 852	17120	509327	29.75
dont.panic.raw.50.jpg	50%	Greyscale	119372	1280 x 852	17120	183489	10.72
dont.panic.raw.75.jpg	75%	Greyscale	211376	1280 x 852	17120	334446	19.54
dont.panic.raw.100.jpg	100%	Greyscale	897012	1280 x 852	17120	1037651	60.61
dont.panic.raw.upsidedown.50.jpg	50%	Greyscale	119088	1280 x 852	17120	183617	10.73
dont.panic.raw.upsidedown.75.jpg	75%	Greyscale	211089	1280 x 852	17120	334185	19.52
dont.panic.raw.upsidedown.100.jpg	100%	Greyscale	896924	1280 x 852	17120	1038480	60.66
sphynx.smooth.50.jpg	50%	Greyscale	3228	149 x 199	475	4734	9.97
sphynx.smooth.75.jpg	75%	Greyscale	4550	149 x 199	475	6525	13.74
sphynx.smooth.100.jpg	100%	Greyscale	16489	149 x 199	475	22114	46.56
sphynx.smooth.upsidedown.50.jpg	50%	Greyscale	3198	149 x 199	475	4726	9.95
sphynx.smooth.upsidedown.75.jpg	75%	Greyscale	4520	149 x 199	475	6481	13.64
sphynx.smooth.upsidedown.100.jpg	100%	Greyscale	12807	149 x 199	475	16153	34.01
sphynx.raw.50.jpg	50%	Greyscale	4447	149 x 199	475	6678	14.06
sphynx.raw.75.jpg	75%	Greyscale	7022	149 x 199	475	10282	21.65
sphynx.raw.100.jpg	100%	Greyscale	25806	149 x 199	475	28711	60.44
sphynx.raw.upsidedown.50.jpg	50%	Greyscale	4389	149 x 199	475	6571	13.83
sphynx.raw.upsidedown.75.jpg	75%	Greyscale	6950	149 x 199	475	10162	21.39
sphynx.raw.upsidedown.100.jpg	100%	Greyscale	26003	149 x 199	475	28876	60.79
wongat.smooth.50.jpg	50%	Colour	14716	282 x 349	2376	23600	9.93
wongat.smooth.75.jpg	75%	Colour	20711	282 x 349	2376	30703	12.92
wongat.smooth.100.jpg	100%	Colour	79707	282 x 349	2376	100378	42.25
wongat.smooth.upsidedown.50.jpg	50%	Colour	14709	282 x 349	2376	23491	9.89
wongat.smooth.upsidedown.75.jpg	75%	Colour	20690	282 x 349	2376	30623	12.89
wongat.smooth.upsidedown.100.jpg	100%	Colour	64359	282 x 349	2376	80511	33.89
wongat.raw.50.jpg	50%	Colour	20117	282 x 349	2376	31869	13.41
wongat.raw.75.jpg	75%	Colour	30297	282 x 349	2376	44899	18.90
wongat.raw.100.jpg	100%	Colour	113652	282 x 349	2376	122084	51.38
wongat.raw.upsidedown.50.jpg	50%	Colour	20224	282 x 349	2376	31795	13.38
wongat.raw.upsidedown.75.jpg	75%	Colour	30299	282 x 349	2376	44614	18.78
wongat.raw.upsidedown.100.jpg	100%	Colour	78529	282 x 349	2376	98532	41.47
sunset.smooth.50.jpg	50%	Colour	49360	967 x 810	18666	92287	4.94
sunset.smooth.75.jpg	75%	Colour	79822	967 x 810	18666	140000	7.50
sunset.smooth.100.jpg	100%	Colour	351833	967 x 810	18666	489782	26.24
sunset.smooth.upsidedown.50.jpg	50%	Colour	49627	967 x 810	18666	92687	4.97
sunset.smooth.upsidedown.75.jpg	75%	Colour	77769	967 x 810	18666	136502	7.31
sunset.smooth.upsidedown.100.jpg	100%	Colour	386060	967 x 810	18666	558947	29.94
sunset.raw.50.jpg	50%	Colour	73863	967 x 810	18666	128427	6.88
sunset.raw.75.jpg	75%	Colour	126836	967 x 810	18666	208952	11.19
sunset.raw.100.jpg	100%	Colour	375238	967 x 810	18666	324514	17.38
sunset.raw.upsidedown.50.jpg	50%	Colour	74178	967 x 810	18666	129084	6.92
sunset.raw.upsidedown.75.jpg	75%	Colour	126610	967 x 810	18666	210670	11.29
sunset.raw.upsidedown.100.jpg	100%	Colour	573603	967 x 810	18666	755925	40.50

Table 7.2: table of testimages.

7.1 Scalar multiplication

Input : dont.panic.jpg
Scale : 1.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	1.4900	2.1920	11.2010	2.7250	4.7080	15.4790
val2	2.1390	2.9910	12.2670	3.4130	5.5790	16.7650
diff1	1.4270	2.1350	10.8860	2.6170	4.6450	15.1950
diff2	2.1260	2.8930	12.3200	3.4100	5.5990	16.7900
bd	977.5530	977.8660	990.0960	979.9280	983.2200	990.5800
zzbd	978.6370	979.3130	993.9470	980.6840	981.7600	992.0590
sd	10.9230	12.2140	22.2780	12.9610	15.4920	26.3540
zzsd	12.9470	13.9910	23.9310	14.6420	17.1170	27.8220

Input : sphynx.jpg
Scale : 1.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.0750	0.0960	0.3220	0.0980	0.1510	0.4240
val2	0.0860	0.1150	0.3520	0.1200	0.1700	0.4710
diff1	0.0650	0.0870	0.3090	0.0940	0.1430	0.4220
diff2	0.0860	0.1200	0.3490	0.1150	0.1720	0.4720
bd	27.2050	27.2090	27.3790	27.2080	27.2520	27.7570
zzbd	27.2230	27.2480	27.4120	27.2450	27.2920	27.5270
sd	0.3640	0.3860	0.6340	0.3890	0.4460	0.7310
zzsd	0.4150	0.4300	0.6850	0.4390	0.4910	0.7890

Input : wongat.jpg
Scale : (1.5, 1.5, 1.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.3320	0.4270	1.5140	0.4430	0.6300	1.8690
val2	0.4190	0.5340	1.6200	0.5380	0.7430	2.0510
diff1	0.3240	0.4130	1.4490	0.4200	0.6100	1.8580
diff2	0.4190	0.5290	1.6200	0.5410	0.7380	2.0650
bd	135.7480	135.9280	136.9730	135.8550	136.1670	137.3610
zzbd	135.9460	136.2020	137.2150	136.0590	136.3420	137.5250
sd	1.6440	1.7700	3.0250	1.7840	1.9920	3.4100
zzsd	1.9060	2.0300	3.2070	2.0360	2.2500	3.6170

Input : sunset.jpg
Scale : (1.5, 1.5, 1.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	1.2570	1.9050	7.0380	1.7770	2.8920	5.6850
val2	1.9320	2.6220	8.0570	2.4850	3.6770	6.5760
diff1	1.1980	1.8540	6.9580	1.7190	2.8340	5.6310
diff2	1.9100	2.6130	8.0680	2.4700	3.6590	6.5780
bd	1065.5060	1065.4340	1071.4760	1066.5190	1070.5100	1071.3240
zzbd	1067.5440	1071.0160	1072.7800	1068.2330	1072.4710	1072.4530
sd	11.1170	12.1170	18.8510	11.8800	13.3110	17.7720
zzsd	13.1300	14.0790	20.2210	13.7950	15.2250	18.9860

7.2 Scalar addition

Input : dont.panic.jpg
Scale : 50.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	1.2980	1.9060	9.5140	2.3310	4.0820	13.5710
diff	1.2280	1.8330	9.5120	2.2980	3.9960	13.3070
bd	977.5260	979.5880	986.0200	980.1320	989.5400	990.3250
zzbd	979.6040	978.6400	988.2840	980.8300	993.7520	999.5690
sd	10.1650	11.6250	20.9860	12.4380	14.8790	25.1340
zzsd	12.4180	13.4360	22.3680	14.0710	16.4990	26.9600

Input : sphynx.jpg
Scale : 50.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.0650	0.0850	0.2710	0.0810	0.1240	0.3880
diff	0.0600	0.0820	0.2700	0.0810	0.1220	0.3780
bd	27.1650	27.1920	27.3710	27.1930	27.2350	27.4940
zzbd	27.2000	27.2230	27.4060	27.2530	27.2910	27.5310
sd	0.3460	0.3780	0.5970	0.3650	0.4340	0.7170
zzsd	0.3980	0.4200	0.6400	0.4130	0.4820	0.7630

Input : wongat.jpg
Scale : (50.5, 50.5, 50.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.2260	0.3710	1.2620	0.3830	0.5450	1.6370
diff	0.2720	0.3590	1.2510	0.3750	0.5330	1.6220
bd	135.6900	135.8610	136.7500	136.6670	136.0610	137.1040
zzbd	135.8850	136.0830	136.9360	136.8760	136.2750	137.1880
sd	1.5750	1.7050	2.8420	1.7280	1.9260	3.2410
zzsd	1.8340	1.9720	3.0850	1.9640	2.1840	3.4390

Input : sunset.jpg
Scale : (50.5, 50.5, 50.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val.	1.1020	1.6540	6.0750	1.5510	2.4950	5.0710
diff.	1.0270	1.5740	6.0020	1.4660	2.4140	4.9710
bd	1064.5080	1064.8000	1070.5910	1065.7200	1065.8130	1070.5740
zzbd	1066.1520	1066.4600	1071.7850	1067.3340	1067.4500	1071.7200
sd	10.6490	11.6000	17.8050	11.4080	12.7990	16.4900
zzsd	12.6350	13.5710	19.1990	13.3120	14.7160	17.8430

7.3 Pixel addition

Input 1 : dont.panic.jpg
Input 2 : dont.panic.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	6.5510	7.6950	18.9370	8.6120	11.9280	27.2970
val2	5.3680	6.5560	18.4080	7.5420	11.0560	27.2210
diff1	6.5180	7.6510	18.8940	8.5790	11.9010	27.2860
diff2	5.3270	6.5300	18.3600	7.5050	11.0220	27.2040
bd	1464.2830	1465.5470	1475.7640	1467.9590	1470.6160	1486.2760
zzbd	1466.4540	1467.7170	1479.0440	1469.6270	1472.6700	1488.9500
sd	14.2650	16.4000	29.1620	18.0050	21.9340	38.9550
zzsd	17.5320	19.2420	31.7220	20.4350	24.3860	40.8900

Input 1 : sphynx.jpg
Input 2 : sphynx.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.2190	0.2620	0.5580	0.2710	0.3450	0.7650
val2	0.1930	0.2300	0.5420	0.2430	0.3170	0.7650
diff1	0.2220	0.2600	0.5530	0.2660	0.3420	0.7650
diff2	0.1900	0.2300	0.5390	0.2390	0.3210	0.7630
bd	40.7190	40.7620	41.1800	40.7590	40.8200	41.3960
zzbd	40.7710	40.8150	41.2110	40.8110	40.9010	41.4710
sd	0.4770	0.5290	0.8380	0.5360	0.6240	1.0870
zzsd	0.5600	0.6000	0.8990	0.6050	0.6920	1.1390

Input 1 : wongat.jpg
Input 2 : wongat.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	1.0840	1.2440	2.6690	1.2720	1.5710	3.1620
val2	0.9300	1.1020	2.6030	1.1330	1.4360	3.1180
diff1	1.0710	1.2380	2.6610	1.2620	1.5540	3.1570
diff2	0.9260	1.0940	2.5870	1.1300	1.4330	3.1070
bd	203.4080	203.6960	205.3860	203.5700	204.0070	205.3100
zzbd	203.7030	204.0240	205.8220	203.8680	204.5440	205.5810
sd	2.2110	2.3970	4.0520	2.4290	2.7730	4.5430
zzsd	2.6040	2.8270	4.3590	2.8160	3.1820	4.9030

Input 1 : sunset.jpg
Input 2 : sunset.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	6.5760	7.5840	16.6930	7.4690	9.1060	18.5430
val2	5.2420	6.2870	15.8970	6.1860	8.1060	17.7710
diff1	6.5240	7.5220	16.6350	7.4160	9.2300	18.5000
diff2	5.1970	6.2480	15.8540	6.1520	8.0710	17.7300
bd	1597.4410	1596.3660	1607.0680	1597.3220	1597.9150	1608.8940
zzbd	1598.9550	1598.6210	1610.0900	1601.4550	1600.4850	1610.9290
sd	14.4110	15.9600	26.9560	15.8950	18.1400	28.8342
zzsd	17.5450	19.1270	29.2970	18.7640	21.1380	31.3108

7.4 Pixel multiplication

Input 1 : dont.panic.jpg
Input 2 : dont.panic.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	4.7170	7.0403	69.5400	9.2190	21.6670	166.4350
valprec 3	6.1090	10.2830	108.8300	13.1890	33.8330	267.7280
valprec 4	6.1520	10.3680	114.7570	13.2120	33.8960	276.4950
valprec 5	6.1550	10.3960	116.1480	13.2380	33.9500	278.7790
bd	1466.8800	1465.9260	1478.9040	1466.6290	1469.4790	1486.7091
zzbd	1469.1830	1468.3030	1480.2190	1469.7700	1471.7760	1490.2050
sd	14.9450	17.0650	30.3940	18.2750	22.2240	39.6050
zzsd	18.2030	19.9340	32.4200	20.8770	24.6660	41.6090

Input 1 : sphynx.jpg
Input 2 : sphynx.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.5850	0.7080	2.4780	0.7230	1.0720	5.0760
valprec 3	0.6840	0.8880	3.6690	0.8960	1.4880	7.7490
valprec 4	0.6900	0.8920	3.8720	0.9080	1.5010	8.1640
valprec 5	0.6900	0.9040	3.8960	0.9090	1.5070	8.2590
bd	40.7470	40.7950	41.2170	40.7720	40.8650	41.5110
zzbd	40.7900	40.8520	41.2200	40.8350	40.9180	41.5740
sd	0.5060	0.5460	0.8690	0.5500	0.6460	1.0920
zzsd	0.5770	0.6200	0.9360	0.6200	0.7190	1.1760

Input 1 : wongat.jpg
Input 2 : wongat.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	1.8780	2.4280	10.9940	2.6170	4.5090	15.8170
valprec 3	2.4440	3.3380	16.7020	3.6570	6.0480	24.3630
valprec 4	2.4530	3.3680	17.5820	3.6680	6.0890	25.6310
valprec 5	2.4620	3.3840	17.6870	3.7600	6.1090	25.6790
bd	203.4110	203.5600	205.4220	203.4920	203.7170	206.4700
zzbd	203.7340	204.2390	205.8190	204.0830	204.4220	207.0700
sd	2.2240	2.4080	3.9350	2.4160	2.7490	4.4610
zzsd	2.7020	2.8400	4.2370	2.8200	3.1600	4.7660

Input 1 : sunset.jpg
Input 2 : sunset.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	4.2320	6.3370	46.6280	6.0520	11.2540	44.5560
valprec 3	5.3310	8.6800	72.4840	8.2220	16.6910	67.8880
valprec 4	5.3620	8.7540	75.2640	8.2440	16.7980	70.7610
valprec 5	5.3590	8.7880	75.5130	8.2670	16.8220	71.2150
bd	1595.7220	1596.4120	1606.0380	1596.0330	1597.6960	1607.7820
zzbd	1597.8670	1598.8400	1613.6960	1598.5490	1600.0980	1612.5458
sd	14.5810	16.0650	26.1180	15.8530	17.8690	28.2125
zzsd	17.7630	19.2380	28.5670	18.8120	20.8750	30.4780

Chapter 8

Conclusions

In this chapter we will discuss the results given in Chapter 7. Each section will deal with a specific operation.

First, it is important to note that the results found in Chapter 7 are inaccurate, even though the timing of the operations was done during off-peak hours.

During a run of ten tests the times found, fluctuate within a reasonable margin, but when the same test is done on another day, the resulting average can differ a little. For this reason, we tested the operations we wanted to compare in one testsession (e.g. smval1, smdif1, smval2 and smdif2 were tested in one testsession).

We emphasize that we are more interested in the relative times than in the absolute times. Table 8.1 shows how an overall performance comparison per operation, for both smooth and raw versions. In this table, the smart operation to be preferred (see Section 8.2, Section 8.3, Section 8.4 and Section 8.5), was compared with its brute force version that uses that advanced DCT algorithm. Table 8.1 shows that scalar multiplication is about 4.8 times faster than its brute force counterpart in case of the 50% quality smooth images.

smooth	50%	75%	100%
scalar mul.	4.8	3.9	2.0
scalar add.	7.8	5.8	2.4
pixel add.	2.6	2.4	1.6
pixel mul.	2.2	1.7	0.5
raw	50%	75%	100%
scalar mul.	3.8	2.9	1.9
scalar add.	5.6	4.1	2.3
pixel add.	2.3	2.0	1.5
pixel mul.	1.6	1.0	0.3

Table 8.1: Relative excecution times.

8.1 General remarks

In the brute force versions, it is clear that the zigzag versions are slower than the ones where the zigzag step was omitted. The reason for this can be found in the 64 memory references in each block. As it makes no difference for the output image whether the zigzag version or the version without zigzag is used, the brute force versions without zigzag steps are to be preferred.

In the brute force versions it is also clear that the smart DCT functions are remarkably faster than its brute force DCT counterparts. As the smart DCT versions do not suffer much from rounding errors, the smart DCT versions are to be preferred to the brute force DCT versions.

From Chapter 7 it can be seen that the 'smooth' images result in a better performance than the 'raw' images. This is caused by the RLE-blocks which are longer for the 'raw' versions. One exception to this is sunset.jpg with 100% quality. Although the smooth image-file is smaller, the average length of the RLE-blocks is higher (see Appendix C). This is possible because in the smooth version, the RLE entries are assigned shorter Huffman codes.

It is also clear that the quality of the image determines the performance of the algorithms; the lower the quality of the image, the lower the average length of the RLE-blocks (see Section 2.4), and therefore less work is needed in the Huffman decoding and Huffman encoding phases. In case of the smart algorithms, less work is also needed during the operation phase since the smart algorithms are RLE-based.

8.2 Scalar multiplication

In the scalar multiplication operation, we see that val1 (RLE-in-RLE-out) is faster than val2 (RLE-in-block-out). This is also true for the DC-difference oriented versions. Nevertheless, the RLE-in-block-out versions are to be preferred to the RLE-in-RLE-out versions because of the possible appearances of ghost-entries (see Subsection 5.9.1).

The differences in time between the DC-difference oriented versions and the DC-coefficient versions are very small, and the differences could be caused by fluctuations in the systemload of the computer on which our program was executed.

The DC-coefficient oriented versions are preferred to the DC-difference oriented versions, because a piling up of rounding errors in the DC-difference oriented versions might result in a distorted output image.

A promising fact is that the brute force algorithms are indeed slower than the smart algorithms. Even the brute force versions that make use of the smart DCT are slower than smval2, which was to be preferred to the other smart algorithms, in most cases smval2 is more than three times faster than smsd.

8.3 Scalar addition

The first thing that comes to mind is that the times for saval (DC-coefficient oriented) and sadif (DC-difference oriented) don't differ very much. These small differences could even be caused by fluctuations in the systemload, but because the sadif algorithm is faster for every image than the saval algorithm, it might be justified to conclude that sadif is a faster algorithm indeed.

If we remind that saval needs only one operation for each component, and sadif needs one operation for every block, it is obvious that saval is a faster algorithm. See Section 4.3. Taking this into consideration we expected a better performance from sadif compared to saval. Probably, the algorithms (both sadif and saval) spend most of the time in the huff-decoding and huff-encoding phase. The additional calculations needed for saval don't add much to the total time using current hardware.

The brute force algorithms are still considerably slower than the smart algorithm.

The output images of any scalar addition algorithm are of the same visual quality as the input images. This is why we prefer the fastest algorithm, sadif.

8.4 Pixel addition

In the pixel addition operation, we see that paval2 (RLE-in-block-out) is faster than paval1 (block-in-block-out). This is also the case for the DC-difference oriented versions. The block-in-block-out versions are slower because in most cases a lot of unnecessary additions are needed. See Section 5.9.2

The DC-difference oriented versions are faster than the DC-coefficient versions. Even though the DC-difference versions are faster, the DC-coefficient versions are preferred to the DC-difference oriented versions, because a piling up of rounding errors in the DC-difference oriented versions might result in a distorted output image. It can now be concluded that of the smart algorithms paval2, is the best smart algorithm (fastest version that is not troubled by a distorted output image).

The brute force algorithms are indeed slower than the smart algorithms. Even the brute force versions that make use of the smart DCT are slower than paval2, which was to be preferred to the other smart algorithms.

8.5 Pixel multiplication

Comparison of the execution times of the smart algorithms, pmval (pixelwise multiplication using DC-coefficients) and pmprec (pixelwise multiplication with a user defined precision), shows that of the smart algorithms, pmval is the faster one. This can be explained by the fact that for the pmprec algorithms more overhead is needed. As the execution time of pmval is less than any of the pmprec versions, and the output image of the pmval version has the same visual quality as the output image of a pmprec version, it is clear that the pmval version is to be preferred.

A brute force algorithm, compared to a smart algorithm, provides a more detailed output image. This might be caused by the fact that boundaries were set to the values that are entered in the combination list. This was done to increase performance. See Section 5.7.

If the execution time of the fastest smart algorithm (pmval) is compared to the execution time of the fastest brute force algorithm (pmsd), the following can be concluded:

1. The smaller the input image, the worse pmval performs in comparison to pmsd.
2. The higher the quality, the worse pmval, in comparison to pmsd, performs.

This can be explained by the fact that in the pmval algorithm a combination array initialization is needed. The size of this combination array does not depend on the size of the image but on the quantization tables of the input image. As the time needed for initialization of the combination array only depends on the quantization tables, it is obvious that the impact of this initialization is bigger on a small image. Because the quantization tables depend on the quality of the input image, it can be concluded that the time needed for the initialization of the combination array depends on the quality of the input images.

Chapter 9

Further ideas for further research

9.1 Pixel division

In this section we will discuss the division operation. Pixel division can be used to reduce the noise in images of microscopic scenes. A version that works on the compressed data, has not been implemented yet, but we will present a suggestion for the implementation of pixel multiplication. We will use DC-coefficients only.

In a first attempt to solve this problem, we wanted to develop a method similar to the one followed for pixel multiplication (see Section 3.4).

Pixel division is defined by:

$$y[i,j] = \frac{x_1[i,j]}{x_2[i,j]} \tag{9.1}$$

Like in Chapter 3 we substitute the operation into Eq. 2.1. In this way we get:

$$\begin{aligned} & Y[u_1,u_2] \\ &= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u_1)C(j,u_2)y[i,j] \\ &= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u_1)C(j,u_2)\frac{x_1[i,j]}{x_2[i,j]} \end{aligned}$$

Now we apply the IDCT on x_1 and x_2.

$$\begin{aligned} & Y[u_1,u_2] \\ &= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)\frac{x_1[i,j]}{x_2[i,j]} \\ &= \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v) * \\ &\qquad * \frac{\frac{1}{4}\sum_{v_1=0}^{7}\sum_{v_2=0}^{7} C(i,v_1)C(j,v_2)X_1[v_1,v_2]}{\frac{1}{4}\sum_{w_1=0}^{7}\sum_{w_2=0}^{7} C(i,w_1)C(j,w_2)X_2[w_1,w_2]} \end{aligned}$$

Now we face a problem. The key to the solution offered in Section 3.4 — collecting all $C(i,u)$ and putting those terms in M — cannot be applied here; the division ruins everything.

So another approach had to be taken. Our next idea was to rewrite pixel division into pixel multiplication, and considering the problem as a linear system. To keep the notations simple, we use the zigzag ordered notation: x_1, x_2 and z are the zigzag ordered indices of $X_{1,Q}$, $X_{2,Q}$ and Y_Q respectively. We start to rewrite Eq. 9.1:

$$x_1[i,j] = y[i,j]x_2[i,j]$$

for $i,j = 0,\ldots,8$

Combining Section 3.4 with the notation used in Eq. 4.12 leads to

$$X_{1,Q,zz}[u] = \sum_{v,w}^{63} Y_{Q,zz}[v]X_{2,Q,zz}[w]M_{Q,zz}[v,w,u]$$

for $u = 0,\ldots,63$.

Remember, our original problem was Eq. 9.1, so note that $X_{1,Q,zz}$ and $X_{2,Q,zz}$ are known and $Y_{Q,zz}$ is unknown.

Now we rewrite the equation:

$$\begin{aligned} & X_{1,Q,zz}[u] \\ &= \sum_{v}^{63}\left\{ Y_{Q,zz}[v]\sum_{w}^{63} X_{2,Q,zz}[w]M_{Q,zz}[v,w,u]\right\} \end{aligned}$$

To keep our notation short, we define

$$P[u,v] = \sum_{w}^{63} X_{2,Q,zz}[w]M_{Q,zz}[v,w,u] \tag{9.2}$$

Figure 9.1 is the 'expanded' version of Eq. 9.2 and was included to illustrate the underlying linear system of pixel division.

Now we define

$$\vec{x} = (X_{1,Q,zz}[0], X_{1,Q,zz}[1], \ldots, X_{1,Q,zz}[63])^T$$

$$\vec{y} = (Y_{Q,zz}[0], Y_{Q,zz}[1], \ldots, Y_{Q,zz}[63])^T$$

and

$$P = \begin{pmatrix} P[0,0] & P[0,1] & \cdots & P[0,63] \\ P[1,0] & P[1,1] & \cdots & P[1,63] \\ \vdots & \vdots & & \vdots \\ P[63,0] & P[63,1] & \cdots & P[63,63] \end{pmatrix}$$

$$
\begin{array}{rcccccccc}
X_{1,Q,zz}[0] & = & P[0,0]Y_{Q,zz}[0] & + & P[0,1]Y_{Q,zz}[1] & + & \cdots & + & P[0,63]Y_{Q,zz}[63] \\
X_{1,Q,zz}[1] & = & P[1,0]Y_{Q,zz}[0] & + & P[1,1]Y_{Q,zz}[1] & + & \cdots & + & P[1,63]Y_{Q,zz}[63] \\
\vdots & & \vdots & & \vdots & & & & \vdots \\
X_{1,Q,zz}[63] & = & P[63,0]Y_{Q,zz}[0] & + & P[63,1]Y_{Q,zz}[1] & + & \cdots & + & P[63,63]Y_{Q,zz}[63]
\end{array}
$$

Figure 9.1: Pixel division as a linear system.

Now the system in Figure 9.1 can be written as

$$\vec{x} = P\vec{y} \tag{9.3}$$

where $\vec{x}$ and P are given and $\vec{y}$ is the vector we're looking for.

In order to solve this linear system, we must know the entire matrix P; we have to evaluate every $P[u,v]$. Evaluation could be done efficiently by using the RLE representation of $X_{2,Q,zz}$ and by using a datastructure and convolution algorithm similar to the one used for pixel multiplication.

Some remarks have to be made on the construction of matrix P:

1. Eq. 9.3 has to be done for every block in every component since P depends on $X_{2,Q,zz}$.
2. The number of calculations needed to construct P depends on the length of the RLE representation of $X_{2,Qzz}$. Since the length of the RLE representation strongly depends on the quantization tables of the image, we conclude that the quantization tables of an JPEG-image influence the number of calculations needed to construct P.

If P is finaly constructed, then Eq. 9.3 has to be solved. For this we need a very fast solver, since we expect the solving of Eq. 9.3 to be the bottleneck of this operation.

Appendix A

Auxiliary tools

During the process of understanding the knowledge that was needed for this project, we developed some auxiliary tools to try to concentrate completely on a certain subject or to generate testdata. In this chapter we will discuss the tools briefly.

A.1 readblock.c

After we studied the JPEG baseline method, we wanted to start implementing the operations. This is exactly the point where we found out that the file format for JPEG images is not specified in the JPEG standard itself. After reading [Lane] we found out about the existence of the JFIF standard and we studied [Hami].

But in [Hami], only the APP0 segments are explained, but it gave us a good idea of how the data needed for decompression is stored in the file.

Then we studied the original source code supplied by [Ijg]. At this stage, we were interested in the module jrdjfif.c. In this module all needed segments are identified and all needed data is extracted from these segments.

But now we wanted to work out some examples manually. To do this we developed the program *readblock*. With the output of readblock — which we like to call *segment dumps* — , we were able to study jfif.c better because now we had a way to trace the working of jfif.c. A demonstration of readblock was already given in Section 6.2.

A.2 huffload.c

When we were able to extract the meta data of an image — such as heigth, width, quantization tables and Huffman tables — it became time to extract the (scan)data itself.

The scandata is Huffman encoded, so we needed to understand the Huffman decoding and encoding phase completely. In [Gon] complete Huffman tables are given and the example that uses those tables gave us a lot of insight in the algorithm used.

But in order to embed the Huffman modules from [Ijg] in our own implementation, we spent some time examining those implementations of the Huffman process. We wanted to trace the original Huffman modules and for that purpose we created *huffload*.

```
==========================================
Defining Huffman Table 0x00 which is a DC table
==========================================

huffcode[p] = (decimal) = (binary)    huffsize[p]
huffcode[  0]   =    0 = 00     huffsize[  0] = 2
huffcode[  1]   =    2 = 010     huffsize[  1] = 3
huffcode[  2]   =    3 = 011     huffsize[  2] = 3
huffcode[  3]   =    4 = 100     huffsize[  3] = 3
huffcode[  4]   =    5 = 101     huffsize[  4] = 3
huffcode[  5]   =    6 = 110     huffsize[  5] = 3
huffcode[  6]   =   14 = 1110     huffsize[  6] = 4
huffcode[  7]   =   30 = 11110     huffsize[  7] = 5
huffcode[  8]   =   62 = 111110     huffsize[  8] = 6
huffcode[  9]   =  126 = 1111110     huffsize[  9] = 7
huffcode[ 10]   =  254 = 11111110     huffsize[ 10] = 8
huffcode[ 11]   =  510 = 111111110     huffsize[ 11] = 9

Encoding table

[category/run] - huffman code - # of bits in code
[0/0] = 00 -   2
[0/1] = 010 -   3
[0/2] = 011 -   3
[0/3] = 100 -   3
[0/4] = 101 -   3
[0/5] = 110 -   3
[0/6] = 1110 -   4
[0/7] = 11110 -   5
[0/8] = 111110 -   6
[0/9] = 1111110 -   7
[0/A] = 11111110 -   8
[0/B] = 111111110 -   9

Total number of codes: 12


maxcode[ 1] = -1

mincode[2] = 00 = 0
maxcode[2] = 00 = 0

mincode[3] = 010 = 2
maxcode[3] = 110 = 6

mincode[4] = 1110 = 14
maxcode[4] = 1110 = 14

mincode[5] = 11110 = 30
maxcode[5] = 11110 = 30

mincode[6] = 111110 = 62
maxcode[6] = 111110 = 62

mincode[7] = 1111110 = 126
maxcode[7] = 1111110 = 126

mincode[8] = 11111110 = 254
maxcode[8] = 11111110 = 254

mincode[9] = 111111110 = 510
maxcode[9] = 111111110 = 510

maxcode[10] = -1

maxcode[11] = -1

maxcode[12] = -1

maxcode[13] = -1

maxcode[14] = -1

maxcode[15] = -1

maxcode[16] = -1
```

Figure A.1: Huffload in action.

The program huffload prints the contents of some arrays that contain important information for the Huffman decoding and encoding phase. Where necessary, the contents of the array are given in binary representation. See Figure A.1 for an example (the output has been edited because the complete output is too long).

A.3 dct.c

When we wanted to implement the operations, we noticed that in [Smith], normalization was not dealt with explicitly; it is mentioned that normalization needs to be done, but is is not mentioned how this affects the implementation.

So we needed to find out more about the Discrete Cosine Transform. We could have done some extensive studying on the Discrete Cosine Transformation, but we only wanted to know what we needed to know.

```
Enter value to fill input-block with >5


ORIG
====

 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000

FDCT
====

 +40.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000
 +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000  +0.0000

IDCT
====

 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
 +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000  +5.0000
```

Figure A.2: Example of a dct session.

Another option was to study it by 'observing' the behaviour of the Discrete Cosine Transform by implementing and running it. Because the DCT modules supplied by [Ijg] (which are actually designed by [Loef]) work on integers only, and because of the difficulty we had understanding the given modules, we implemented the Discrete Cosine Transfrom for floats in a straightforward way. The results was the program *dct*.

We 'played' with this program by manipulating the data in the spatial or frequency domain, and studying the results. The outputs gave us some clues that lead us to our needed knowledge described in Section 2.3.

In Figure A.2 the program dct can be seen in action; the original matrix (ORIG) is transformed into the frequency domain (FDCT), this result is transformed back into the spatial domain (IDCT).

A.4 setcomp.c

When our first implementations were ready, we wanted to test the correctness of the output images. During the implementation phase we embedded a *trace facility* in our program with which we could keep track of several aspects of the program, such as memory, Huffman decoding, Huffman encoding and RLEs. This facility however, turned out to be too tedious to keep track of the pixels.

We wanted images in which predetermined pixel-values appeared. For instance, we wanted to generate an image in which all pixel-values were set to 1 (image 1). A similar image with the value 2 (image 2) and another image with the value 3 (image 3). Then we could apply pixel addition on image 1 and image 2, and compare the output image with image three with the UNIX command 'cmp' or the DOS command 'compare'.

A.5 rlestat.c

When we studied the results and when we wanted to write something about pixel division, we felt it was necessary to know something about the (average) length of an RLE in an image.

For that reason a program *rlestat* was written that determines the total length of all RLEs — this is an indication for the total amount of work needed for the operation — and the average RLE length — this indicates the average amount of work to be done per block. Also the frequencies are calculated for each RLE-length.

It is interesting to see how most frequent RLE-lengths appear at the small RLE-blocks (10 or less RLE-entries) for 50 percent quality images, at the medium sized RLE-blocks (10 to 30 RLE-entries) for 75 percent quality images and at the large sized RLE-blocks (30 or more RLE-entries) for 100 percent quality images.

We programmed rlestat to format the output in a LaTeX style table. Appendix C shows some statistics for all testimages.

A.6 Timing tools

During the timing phase of this project, we discovered that our JPEG program was not suited for extensive timing. It was only possible to test one image at a time, so when we wanted to run an operation with the same input data for 10 times, we had to start the operation and select the input data 10 times.

We looked for a batch-like version of our program. We wanted our program to read in a *jobfile*, in which the following is defined for each operation to be tested:

- the name of the operation
- 2 input image filenames; if only one input image is needed, the second input image is ignored.
- an output image filename.
- the name of the configuration file (in which scaling factors are defined).
- number of input images needed.
- a boolean to indicate the need to use a configuration file.
- the number of times to run the operation.
- the precision, in case of pixel multiplication with user defined precision.

In addition, we wanted to specify how many jobs are to be processed.

We realize that some unnecessary information was included (e.g. the number of input files needed, could be derived from the name of the operation), but we wanted to implement the modifications in a straigthforward and quick way.

Furthermore, we wanted the output to be written in a file, so we could create a huge jobfile, keep the computers

```
3
sazzbd
sunset.raw.75.jpg
sunset.raw.upsidedown.75.jpg
results.jpg
sadd.cfg
1
1
7
padif1
dont.panic.raw.50.jpg
dont.panic.raw.upsidedown.50.jpg
results.jpg
sadd.cfg
2
0
12
pmprec
dont.panic.raw.75.jpg
dont.panic.raw.upsidedown.75.jpg
results.jpg
sadd.cfg
2
0
12
4
```

Figure A.3: Example of a jobfile.

busy for a weekend and collect and process all data after the weekend.

The adjustments are only in callback.c, the rest of the modules are unaffected. In order to start timing, the user must select an operation and the input data. This operation won't be executed, instead all the operations in the jobfile were executed.

A.6.1 alljobs.c

Creating a jobfile can be tedious, so why not let the computer prepare a jobfile? The program *alljobs* creates a jobfile and uses the images that are given in the file files.lst to create the jobs that are given in the source code of alljobs. The user must also specify the number of times to run an operation in the source code.

Figure A.3 shows a small jobfile in which 3 jobs are defined. Figure A.4 shows the outputfile after 2 timings, of the operation 'sazzbd', have been completed.

A.6.2 checkjob.c

Sometimes, the jobfile created by alljobs needs to be altered, or some operations need to be added. A mistake is easy to make, and can be disastrous. For that reason a program *checkjob* was programmed. This program processes a jobfile, and checks for non-existent files, invalid operations and also checks for the validity of the number of jobs, specified in the jobfile.

```
operation               : sazzbd
experiment #     :0
input file            : sunset.raw.75.jpg
output file           : morpheus.jpg
scale file            : sadd.cfg
resolution            : 967 x 810
components            : 3
MCU's                 : 3111
elapsed time          : 1069.1100 seconds

-------************-----------


operation               : sazzbd
experiment #     :1
input file            : sunset.raw.75.jpg
output file           : morpheus.jpg
scale file            : sadd.cfg
resolution            : 967 x 810
components            : 3
MCU's                 : 3111
elapsed time          : 1067.6000 seconds

-------************-----------
```

Figure A.4: Example output of a timing session.

Appendix B

Timing results SGI Indy

B.1 Scalar multiplication

Execution times are given in seconds.

Input : dont.panic.jpg
Scale : 1.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	1.1080	1.6060	8.3220	1.9320	3.4180	11.8800
val2	1.4480	1.9380	7.8340	2.4640	3.6560	11.0140
diff1	1.4080	1.9080	8.3900	2.1520	3.6500	12.1880
diff2	2.1040	2.6720	10.1780	3.0560	4.7580	14.0260
bd	1357.9958	1357.0760	1363.6180	1357.3519	1360.0599	1364.9680
zzbd	1359.4399	1358.4459	1365.0400	1358.7041	1361.5120	1366.3359
sd	11.1100	11.9480	18.3140	12.2780	13.7780	21.1320
zzsd	10.9460	11.6680	17.9440	11.9380	13.4220	20.7740

Input : sphynx.jpg
Scale : 1.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.0460	0.0600	0.2380	0.0720	0.1060	0.3300
val2	0.0580	0.0800	0.2320	0.0760	0.1080	0.3100
diff1	0.0500	0.0720	0.2360	0.0720	0.1120	0.3280
diff2	0.0760	0.0940	0.2920	0.0980	0.1440	0.3880
bd	37.7020	37.6720	37.8480	37.6740	37.7480	37.8780
zzbd	37.7380	37.7120	37.8860	37.7120	37.6900	37.9140
sd	0.3480	0.3660	0.5280	0.3660	0.3940	0.5980
zzsd	0.3380	0.3540	0.5080	0.3500	0.3800	0.5760

Input : wongat.jpg
Scale : (1.5, 1.5, 1.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.2100	0.2860	0.9880	0.2940	0.4220	1.2660
val2	0.2820	0.3580	1.0780	0.3660	0.4960	1.3560
diff1	0.2140	0.2840	0.9700	0.2900	0.4180	1.2620
diff2	0.2840	0.3600	1.0940	0.3680	0.5040	1.3840
bd	185.2160	185.0160	185.9000	185.2120	185.1280	186.1580
zzbd	178.5860	178.6480	179.2340	178.6500	178.7500	179.4940
sd	1.3280	1.3940	2.0780	1.4020	1.5220	2.3200
zzsd	1.1080	1.1780	1.8300	1.1860	1.2960	2.0740

Input : sunset.jpg
Scales (1.5, 1.5, 1.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.8280	1.2620	4.7720	1.1820	1.9420	4.0260
val2	1.3240	1.7620	5.5540	1.7160	2.4240	4.6420
diff1	0.8560	1.2820	4.7300	1.2000	1.9480	4.0040
diff2	1.3260	1.7960	5.4800	1.7100	2.5220	4.6960
bd	1454.3259	1452.5680	1457.9960	1454.6500	1453.1960	1458.0260
zzbd	1402.1820	1402.5740	1405.7600	1404.2440	1403.1460	1405.6599
sd	9.5640	10.0820	13.7940	9.9860	10.7380	13.3160
zzsd	7.7780	8.2840	11.7380	8.1460	8.8980	11.1960

B.2 Scalar addition

Execution times are given in seconds.

Input : dont.panic.jpg
Scale : 50.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.7620	1.1240	5.6740	1.3780	2.4040	8.1100
diff	0.7080	1.0620	5.5720	1.3220	2.3460	7.9340
bd	1295.4500	1295.9381	1301.1149	1296.3940	1297.5940	1304.3519
zzbd	979.6040	978.6400	988.2840	980.8300	993.7520	999.5690
sd	8.8580	9.7040	15.8140	10.0100	11.5100	18.7900
zzsd	10.1340	10.8520	16.8100	11.1020	12.5700	19.8100

Input : sphynx.jpg
Scale : 50.5

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.0340	0.0500	0.1660	0.0540	0.0760	0.2320
diff	0.0400	0.0480	0.1600	0.0520	0.0740	0.2280
bd	35.9720	35.9780	36.1120	35.9780	36.0060	36.1960
zzbd	27.2000	27.2230	27.4060	27.2530	27.2910	27.5310
sd	0.2780	0.3000	0.4440	0.3000	0.3300	0.5260
zzsd	0.3120	0.3240	0.4740	0.3300	0.3620	0.5560

Input : wongat.jpg
Scale : (50.5, 50.5, 50.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val	0.1600	0.2120	0.7420	0.2200	0.3140	0.9600
diff	0.1520	0.2060	0.7220	0.2100	0.3020	0.9420
bd	178.4220	178.4820	179.0680	178.4960	178.5940	179.3120
zzbd	178.5400	178.6040	179.1740	178.6140	178.7120	179.4120
sd	0.9340	0.9980	1.6380	1.0060	1.1200	1.8880
zzsd	1.0580	1.1240	1.7420	1.1240	1.2400	1.9880

Input : sunset.jpg
Scale : (50.5, 50.5, 50.5)

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val.	0.6220	0.9460	3.5260	0.8840	1.4360	3.0560
diff.	0.5820	0.8940	3.4380	0.8320	1.3760	2.9740
bd	1401.9000	1402.2720	1405.2200	1402.0281	1402.6560	1405.0360
zzbd	1066.1520	1066.4600	1071.7850	1067.3340	1067.4500	1071.7200
sd	6.4040	6.9120	10.2640	6.8040	7.5500	9.7540
zzsd	7.3800	7.8700	11.0360	7.7420	8.4800	10.4800

B.3 Pixel addition

Execution times are given in seconds.

Input 1 : dont.panic.jpg
Input 2 : dont.panic.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	5.1100	5.8620	12.9180	6.5000	8.5900	18.8100
val2	4.2200	4.9020	11.8940	5.6000	7.5460	17.4500
diff1	4.6960	5.4240	12.4980	6.0680	8.3280	18.3080
diff2	4.2880	5.1860	13.5480	5.8980	8.5000	20.9240
bd	1464.2830	1465.5470	1475.7640	1467.9590	1470.6160	1486.2760
zzbd	1466.4540	1467.7170	1479.0440	1469.6270	1472.6700	1488.9500
sd	14.2650	16.4000	29.1620	18.0050	21.9340	38.9550
zzsd	17.5320	19.2420	31.7220	20.4350	24.3860	40.8900

Input 1 : sphynx.jpg
Input 2 : sphynx.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.1620	0.1900	0.3740	0.1920	0.2400	0.5260
val2	0.1540	0.1720	0.3580	0.1740	0.2380	0.5020
diff1	0.1560	0.1760	0.3720	0.1840	0.2320	0.5160
diff2	0.1500	0.1760	0.4080	0.1820	0.1820	0.5860
bd	40.7190	40.7620	41.1800	40.7590	40.8200	41.3960
zzbd	40.7710	40.8150	41.2110	40.8110	40.9010	41.4710
sd	0.4770	0.5290	0.8380	0.5360	0.6240	1.0870
zzsd	0.5600	0.6000	0.8990	0.6050	0.6920	1.1390

Input 1 : wongat.jpg
Input 2 : wongat.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	0.7120	0.8120	1.7060	0.8240	1.0020	2.0200
val2	0.6420	0.7520	1.6660	0.7800	0.9280	1.9780
diff1	0.7080	0.8060	1.7040	0.8180	1.0000	2.0200
diff2	0.6240	0.7240	1.6380	0.7380	0.9240	1.9540
bd	203.4080	203.6960	205.3860	203.5700	204.0070	205.3100
zzbd	203.7030	204.0240	205.8220	203.8680	204.5440	205.5810
sd	2.2110	2.3970	4.0520	2.4290	2.7730	4.5430
zzsd	2.6040	2.8270	4.3590	2.8160	3.1820	4.9030

Input 1 : sunset.jpg
Input 2 : sunset.upsidedown.jpg
Scale : ＜default＞

	smooth			raw		
	50%	75%	100%	50%	75%	100%
val1	4.4780	5.0880	10.7540	5.0120	6.1160	11.9860
val2	3.8320	4.4460	10.2460	4.2980	5.5660	11.5300
diff1	4.4260	5.0300	10.6860	4.9540	6.0460	11.9140
diff2	3.7160	4.3480	10.1900	4.2740	5.4140	11.3640
bd	1597.4410	1596.3660	1607.0680	1597.3220	1597.9150	1608.8940
zzbd	1598.9550	1598.6210	1610.0900	1601.4550	1600.4850	1610.9290
sd	14.4110	15.9600	26.9560	15.8950	18.1400	28.8342
zzsd	17.5450	19.1270	29.2970	18.7640	21.1380	31.3108

Appendix C

Run Length Encoding - statistics

dont.panic.smooth.50.jpg

length	freq.
2	1903
3	1902
4	2386
5	2590
6	2190
7	1533
8	1113
9	914
10	684
11	548
12	506
13	350
14	233
15	145
16	77
17	30
18	11
19	4
21	1
#RLEs	17120
tot.	103971
avg.	6.07

dont.panic.smooth.75.jpg

length	freq.
2	455
3	646
4	619
5	922
6	1440
7	1947
8	2100
9	2024
10	1658
11	1355
12	1031
13	743
14	606
15	545
16	380
17	290
18	178
19	105
20	45
21	23
22	4
23	2
24	1
25	1
#RLEs	17120
tot.	155169
avg.	9.06

dont.panic.smooth.100.jpg

length	freq.
2	10
5	3
6	1
9	1
10	1
11	2
12	7
13	3
14	4
15	10
16	7
17	10
18	8
19	21
20	26
21	24
22	26
23	29
24	32
25	42
26	36
27	47
28	60
29	75
30	77
31	84
32	101
33	141
34	180
35	234
36	273
37	310
38	412
39	447
40	536
41	556
42	664
43	693
44	693
45	762
46	750
47	778
48	805
49	784
50	821
51	847
52	803
53	814
54	788
55	768
56	707
57	649
58	459
59	371
60	198
61	91
62	29
63	10
#RLEs	17120
tot.	806592
avg.	47.11

dont.panic.smooth.upsidedown.50.jpg

length	freq.
2	1918
3	1854
4	2517
5	2537
6	2110
7	1585
8	1155
9	834
10	704
11	585
12	475
13	342
14	232
15	137
16	76
17	37
18	15
19	5
20	2
#RLEs	17120
tot.	103826
avg.	6.06

dont.panic.smooth.upsidedown.75.jpg

length	freq.
2	453
3	605
4	617
5	922
6	1458
7	2025
8	2087
9	1964
10	1701
11	1337
12	1012
13	767
14	624
15	464
16	413
17	301
18	181
19	93
20	63
21	23
22	7
23	3
#RLEs	17120
tot.	155338
avg.	9.07

dont.panic.smooth.upsidedown.100.jpg

length	freq.
2	9
4	1
5	2
6	1
7	6
8	4
9	9
10	9
11	16
12	32
13	34
14	53
15	97
16	116
17	146
18	204
19	273
20	325
21	393
22	532
23	607
24	695
25	859
26	919
27	1031
28	968
29	970
30	1011
31	1018
32	955
33	845
34	848
35	758
36	664
37	574
38	516
39	459
40	323
41	268
42	196
43	139
44	97
45	59
46	38
47	18
48	11
49	5
50	2
51	3
52	2
#RLEs	17120
tot.	509327
avg.	29.75

dont.panic.raw.50.jpg

length	freq.
2	901
3	695
4	649
5	878
6	1121
7	1209
8	1266
9	1217
10	1123
11	1012
12	945
13	890
14	883
15	792
16	762
17	659
18	529
19	427
20	331
21	221
22	178
23	158
24	91
25	75
26	55
27	22
28	12
29	7
30	5
31	5
32	2
#RLEs	17120
tot.	183489
avg.	10.72

dont.panic.raw.75.jpg

length	freq.
2	42
3	74
4	153
5	277
6	340
7	383
8	439
9	545
10	606
11	712
12	735
13	757
14	755
15	838
16	783
17	808
18	764
19	773
20	731
21	617
22	555
23	415
24	382
25	336
26	295
27	277
28	296
29	288
30	331
31	318
32	312
33	348
34	310
35	278
36	281
37	192
38	204
39	161
40	127
41	86
42	70
43	47
44	29
45	18
46	20
47	6
48	6
#RLEs	17120
tot.	334446
avg.	19.54

dont.panic.raw.100.jpg

length	freq.
2	18
17	2
18	1
25	2
29	5
31	1
32	3
33	9
41	1
44	3
46	3
47	2
48	7
49	14
50	30
51	26
52	39
53	69
54	133
55	255
56	423
57	691
58	1099
59	1668
60	2254
61	2835
62	3249
63	2835
64	1443
#RLEs	17120
tot.	1037651
avg.	60.61

dont.panic.raw.upsidedown.50.jpg

length	freq.
2	908
3	636
4	681
5	883
6	1150
7	1222
8	1261
9	1185
10	1106
11	1057
12	930
13	892
14	882
15	775
16	770
17	617
18	572
19	412
20	337
21	258
22	172
23	131
24	95
25	73
26	48
27	21
28	24
29	14
30	4
31	3
32	1
#RLEs	17120
tot.	183617
avg.	10.73

dont.panic.raw.upsidedown.75.jpg

length	freq.
2	45
3	70
4	145
5	239
6	384
7	388
8	471
9	527
10	624
11	717
12	741
13	746
14	773
15	820
16	757
17	752
18	816
19	700
20	746
21	638
22	559
23	481
24	385
25	330
26	315
27	253
28	260
29	287
30	311
31	325
32	363
33	325
34	310
35	286
36	267
37	207
38	195
39	161
40	115
41	91
42	70
43	49
44	37
45	16
46	11
47	9
48	1
51	1
53	1
#RLEs	17120
tot.	334185
avg.	19.52

dont.panic.raw.upsidedown.100.jpg

length	freq.
2	8
7	2
8	2
10	2
11	1
12	1
16	1
17	1
26	1
27	1
29	3
33	7
34	3
36	1
40	2
41	1
42	2
43	2
46	5
47	3
48	8
49	16
50	19
51	24
52	33
53	77
54	125
55	233
56	400
57	688
58	1052
59	1655
60	2264
61	2930
62	3264
63	2844
64	1439
#RLEs	17120
tot.	1038480
avg.	60.66

sphynx.smooth.50.jpg

length	freq.
4	3
5	3
6	22
7	39
8	76
9	69
10	73
11	60
12	55
13	40
14	27
15	3
16	5
#RLEs	475
tot.	4734
avg.	9.97

sphynx.smooth.75.jpg

length	freq.
6	2
7	1
8	9
9	12
10	23
11	49
12	62
13	60
14	80
15	45
16	54
17	49
18	14
19	11
20	4
#RLEs	475
tot.	6525
avg.	13.74

sphynx.smooth.100.jpg

length	freq.
27	1
30	1
31	1
32	1
34	4
35	4
36	7
37	7
38	15
39	14
40	13
41	19
42	27
43	33
44	35
45	29
46	24
47	30
48	36
49	23
50	26
51	23
52	20
53	23
54	14
55	16
56	12
57	9
58	4
59	2
60	1
62	1
#RLEs	475
tot.	22114
avg.	46.56

sphynx.smooth.upsidedown.50.jpg

length	freq.
3	1
4	4
5	8
6	17
7	35
8	65
9	70
10	81
11	73
12	65
13	25
14	17
15	11
16	2
17	1
#RLEs	475
tot.	4726
avg.	9.95

sphynx.smooth.upsidedown.75.jpg

length	freq.
6	3
7	1
8	5
9	12
10	35
11	41
12	59
13	58
14	85
15	71
16	50
17	22
18	20
19	7
20	4
21	1
22	1
#RLEs	475
tot.	6481
avg.	13.64

sphynx.smooth.upsidedown.100.jpg

length	freq.
16	1
23	3
24	1
25	3
26	8
27	8
28	25
29	20
30	27
31	40
32	37
33	41
34	40
35	45
36	41
37	39
38	27
39	23
40	20
41	8
42	7
43	6
44	4
45	1
#RLEs	475
tot.	16153
avg.	34.01

sphynx.raw.50.jpg

length	freq.
5	2
6	2
7	3
8	10
9	21
10	23
11	34
12	49
13	56
14	59
15	60
16	52
17	42
18	30
19	15
20	11
21	5
24	1
#RLEs	475
tot.	6678
avg.	14.06

sphynx.raw.75.jpg

length	freq.
9	1
10	1
11	2
12	5
13	4
14	13
15	12
16	32
17	30
18	26
19	33
20	32
21	41
22	42
23	45
24	28
25	25
26	16
27	32
28	18
29	19
30	12
31	1
32	1
33	2
34	1
35	1
#RLEs	475
tot.	10282
avg.	21.65

sphynx.raw.100.jpg

length	freq.
48	1
51	3
52	2
53	2
54	3
55	9
56	14
57	20
58	45
59	41
60	63
61	79
62	98
63	65
64	30
#RLEs	475
tot.	28711
avg.	60.44

sphynx.raw.upsidedown.50.jpg

length	freq.
4	1
5	2
6	4
7	1
8	14
9	15
10	28
11	39
12	42
13	56
14	84
15	55
16	45
17	38
18	23
19	12
20	9
21	5
22	2
#RLEs	475
tot.	6571
avg.	13.83

sphynx.raw.upsidedown.75.jpg

length	freq.
8	1
9	1
10	3
11	1
12	3
13	9
14	11
15	19
16	17
17	29
18	27
19	38
20	42
21	48
22	37
23	45
24	33
25	33
26	15
27	15
28	15
29	12
30	7
31	4
32	6
33	1
34	2
35	1
#RLEs	475
tot.	10162
avg.	21.39

sphynx.raw.upsidedown.100.jpg

length	freq.
52	3
53	2
54	2
55	4
56	8
57	17
58	27
59	53
60	70
61	79
62	102
63	79
64	29
#RLEs	475
tot.	28876
avg.	60.79

wongat.smooth.50.jpg

length	freq.
2	281
3	264
4	188
5	102
6	53
7	40
8	45
9	67
10	65
11	117
12	138
13	150
14	195
15	176
16	179
17	155
18	90
19	43
20	17
21	8
22	3
#RLEs	2376
tot.	23600
avg.	9.93

wongat.smooth.75.jpg

length	freq.
2	65
3	142
4	193
5	199
6	138
7	76
8	51
9	36
10	30
11	31
12	42
13	55
14	88
15	93
16	120
17	176
18	185
19	166
20	188
21	132
22	89
23	47
24	24
25	4
26	4
27	2
#RLEs	2376
tot.	30703
avg.	12.92

wongat.smooth.100.jpg

length	freq.
2	1
6	1
7	2
8	3
9	1
10	4
11	5
12	8
13	2
14	4
15	6
16	5
17	16
18	16
19	12
20	24
21	27
22	36
23	26
24	37
25	40
26	57
27	43
28	72
29	64
30	47
31	57
32	40
33	49
34	33
35	32
36	37
37	22
38	31
39	46
40	37
41	35
42	43
43	43
44	38
45	53
46	59
47	64
48	67
49	97
50	109
51	116
52	109
53	104
54	101
55	104
56	90
57	68
58	56
59	33
60	24
61	12
62	7
63	1
#RLEs	2376
tot.	100378
avg.	42.25

wongat.smooth.upsidedown.50.jpg

length	freq.
2	255
3	292
4	182
5	89
6	50
7	62
8	41
9	86
10	83
11	127
12	135
13	152
14	175
15	179
16	162
17	115
18	100
19	51
20	23
21	14
22	1
23	2
#RLEs	2376
tot.	23491
avg.	9.89

wongat.smooth.upsidedown.75.jpg

length	freq.
2	60
3	151
4	192
5	199
6	134
7	81
8	35
9	23
10	40
11	33
12	62
13	72
14	91
15	99
16	114
17	186
18	153
19	172
20	182
21	117
22	81
23	57
24	26
25	13
26	1
27	2
#RLEs	2376
tot.	30623
avg.	12.89

wongat.smooth.upsidedown.100.jpg

length	freq.
2	2
6	1
8	4
10	5
11	6
12	8
13	3
14	7
15	10
16	17
17	20
18	27
19	38
20	43
21	54
22	63
23	70
24	82
25	59
26	64
27	76
28	76
29	69
30	53
31	57
32	64
33	67
34	66
35	85
36	70
37	100
38	112
39	99
40	122
41	116
42	116
43	90
44	96
45	79
46	64
47	47
48	27
49	23
50	9
51	6
52	2
53	1
54	1
#RLEs	2376
tot.	80511
avg.	33.89

wongat.raw.50.jpg

length	freq.
2	275
3	269
4	167
5	94
6	38
7	24
8	33
9	29
10	21
11	41
12	26
13	53
14	55
15	65
16	101
17	114
18	104
19	118
20	119
21	144
22	111
23	84
24	80
25	65
26	44
27	33
28	23
29	23
30	12
31	4
32	6
33	1
#RLEs	2376
tot.	31869
avg.	13.41

wongat.raw.75.jpg

length	freq.
2	64
3	124
4	187
5	193
6	138
7	70
8	34
9	13
10	10
11	17
12	15
13	30
14	26
15	26
16	15
17	31
18	24
19	44
20	49
21	50
22	76
23	78
24	95
25	80
26	97
27	92
28	102
29	79
30	106
31	78
32	47
33	66
34	42
35	45
36	37
37	20
38	19
39	16
40	14
41	12
42	5
43	3
44	4
45	2
46	1
#RLEs	2376
tot.	44899
avg.	18.90

wongat.raw.100.jpg

length	freq.
2	1
6	1
7	1
8	4
9	3
10	1
11	1
12	4
13	6
14	2
15	3
16	7
17	13
18	15
19	2
20	8
21	17
22	11
23	26
24	29
25	30
26	24
27	28
28	40
29	31
30	33
31	47
32	32
33	50
34	35
35	24
36	19
37	28
38	33
39	23
40	23
41	19
42	19
43	22
44	18
45	12
46	13
47	11
48	14
49	5
50	10
51	16
52	10
53	10
54	15
55	19
56	26
57	50
58	52
59	90
60	148
61	231
62	331
63	348
64	232
#RLEs	2376
tot.	122084
avg.	51.38

wongat.raw.upsidedown.50.jpg

length	freq.
2	241
3	292
4	180
5	80
6	35
7	24
8	34
9	31
10	27
11	41
12	56
13	55
14	58
15	78
16	89
17	87
18	102
19	109
20	119
21	149
22	106
23	103
24	81
25	67
26	45
27	30
28	19
29	14
30	10
31	8
32	5
33	1
#RLEs	2376
tot.	31795
avg.	13.38

wongat.raw.upsidedown.75.jpg

length	freq.
2	55
3	138
4	203
5	172
6	154
7	58
8	26
9	14
10	9
11	16
12	13
13	24
14	25
15	26
16	29
17	35
18	44
19	43
20	63
21	43
22	67
23	61
24	80
25	112
26	92
27	92
28	99
29	100
30	90
31	67
32	60
33	52
34	45
35	39
36	30
37	29
38	17
39	18
40	12
41	12
42	3
43	4
44	1
45	3
46	1
#RLEs	2376
tot.	44614
avg.	18.78

wongat.raw.upsidedown.100.jpg

length	freq.
2	2
6	2
7	1
9	1
10	5
11	5
12	4
13	5
14	9
15	1
16	9
17	25
18	17
19	22
20	31
21	49
22	41
23	42
24	52
25	50
26	73
27	43
28	52
29	46
30	56
31	40
32	42
33	32
34	34
35	26
36	28
37	23
38	26
39	30
40	33
41	35
42	38
43	41
44	51
45	49
46	72
47	67
48	78
49	100
50	96
51	117
52	102
53	124
54	102
55	88
56	84
57	61
58	43
59	32
60	24
61	11
62	4
#RLEs	2376
tot.	98532
avg.	41.47

sunset.smooth.50.jpg

length	freq.
2	5993
3	2850
4	1921
5	1610
6	1280
7	1093
8	1000
9	831
10	664
11	453
12	320
13	183
14	121
15	120
16	87
17	74
18	42
19	17
20	5
21	2
#RLEs	18666
tot.	92287
avg.	4.94

sunset.smooth.75.jpg

length	freq.
2	3222
3	1566
4	1539
5	1422
6	1314
7	1239
8	1182
9	1128
10	1084
11	934
12	912
13	904
14	682
15	502
16	354
17	185
18	150
19	122
20	100
21	58
22	50
23	14
24	3
#RLEs	18666
tot.	140000
avg.	7.50

sunset.smooth.100.jpg

length	freq.
2	2376
3	16
4	19
5	115
6	198
7	331
8	371
9	207
10	226
11	243
12	318
13	265
14	265
15	299
16	321
17	290
18	332
19	363
20	322
21	325
22	323
23	343
24	311
25	334
26	361
27	318
28	299
29	323
30	334
31	375
32	407
33	410
34	452
35	442
36	448
37	484
38	496
39	538
40	553
41	547
42	485
43	510
44	431
45	378
46	309
47	276
48	196
49	155
50	114
51	117
52	106
53	92
54	69
55	48
56	40
57	25
58	6
59	5
60	4
#RLEs	18666
tot.	489782
avg.	26.24

sunset.smooth.upsidedown.50.jpg

length	freq.
2	5956
3	2754
4	2029
5	1586
6	1306
7	1118
8	951
9	828
10	684
11	485
12	295
13	194
14	138
15	108
16	105
17	68
18	45
19	10
20	6
#RLEs	18666
tot.	92687
avg.	4.97

sunset.smooth.upsidedown.75.jpg

length	freq.
2	3218
3	1848
4	1778
5	1531
6	1243
7	1079
8	1042
9	1088
10	1028
11	938
12	891
13	834
14	683
15	480
16	309
17	186
18	138
19	122
20	98
21	73
22	37
23	21
24	1
#RLEs	18666
tot.	136502
avg.	7.31

sunset.smooth.upsidedown.100.jpg

length	freq.
2	1413
3	1
4	9
5	8
6	71
7	310
8	174
9	233
10	141
11	173
12	231
13	232
14	253
15	275
16	252
17	257
18	261
19	303
20	286
21	343
22	344
23	352
24	404
25	412
26	424
27	429
28	406
29	401
30	419
31	432
32	417
33	416
34	420
35	468
36	444
37	470
38	480
39	486
40	576
41	547
42	564
43	522
44	528
45	495
46	454
47	374
48	362
49	275
50	212
51	171
52	154
53	161
54	114
55	89
56	77
57	49
58	30
59	30
60	21
61	7
62	4
#RLEs	18666
tot.	558947
avg.	29.94

sunset.raw.50.jpg

length	freq.
2	5326
3	2297
4	1419
5	1026
6	903
7	797
8	754
9	754
10	759
11	733
12	740
13	628
14	567
15	517
16	372
17	247
18	185
19	104
20	104
21	77
22	74
23	76
24	52
25	47
26	41
27	36
28	18
29	12
30	1
#RLEs	18666
tot.	128427
avg.	6.88

sunset.raw.75.jpg

length	freq.
2	3160
3	1457
4	1307
5	1056
6	906
7	677
8	532
9	414
10	411
11	428
12	464
13	577
14	548
15	564
16	580
17	644
18	650
19	668
20	623
21	551
22	549
23	421
24	372
25	252
26	156
27	112
28	97
29	65
30	78
31	48
32	46
33	62
34	45
35	44
36	32
37	30
38	16
39	9
40	4
41	5
42	4
43	2
#RLEs	18666
tot.	208952
avg.	11.19

sunset.raw.100.jpg

length	freq.
2	2720
3	19
4	18
5	185
6	413
7	407
8	404
9	248
10	368
11	470
12	607
13	644
14	593
15	669
16	702
17	769
18	822
19	852
20	853
21	813
22	811
23	789
24	786
25	622
26	592
27	469
28	370
29	261
30	206
31	135
32	90
33	51
34	48
35	37
36	38
37	31
38	27
39	27
40	26
41	26
42	22
43	28
44	25
45	23
46	12
47	18
48	36
49	28
50	28
51	28
52	47
53	42
54	28
55	56
56	46
57	55
58	49
59	29
60	29
61	14
62	3
63	1
64	1
#RLEs	18666
tot.	324514
avg.	17.39

sunset.raw.upsidedown.50.jpg

length	freq.
2	5353
3	2151
4	1452
5	1022
6	903
7	831
8	794
9	741
10	770
11	717
12	717
13	665
14	598
15	474
16	351
17	295
18	163
19	121
20	108
21	85
22	72
23	63
24	75
25	54
26	50
27	23
28	10
29	5
30	1
31	1
34	1
#RLEs	18666
tot.	129084
avg.	6.92

sunset.raw.upsidedown.75.jpg

length	freq.
2	3060
3	1623
4	1465
5	1107
6	829
7	568
8	435
9	384
10	408
11	444
12	463
13	470
14	543
15	567
16	631
17	609
18	664
19	618
20	612
21	549
22	502
23	482
24	342
25	289
26	216
27	133
28	99
29	79
30	84
31	71
32	61
33	48
34	43
35	49
36	43
37	22
38	17
39	21
40	11
41	1
42	1
43	1
45	1
46	1
#RLEs	18666
tot.	210670
avg.	11.29

sunset.raw.upsidedown.100.jpg

length	freq.
2	1380
3	1
4	5
5	2
6	39
7	241
8	151
9	261
10	112
11	131
12	173
13	233
14	212
15	209
16	201
17	224
18	207
19	222
20	267
21	246
22	247
23	269
24	265
25	249
26	255
27	211
28	229
29	212
30	210
31	203
32	165
33	153
34	170
35	139
36	122
37	126
38	119
39	128
40	100
41	130
42	113
43	111
44	121
45	141
46	150
47	167
48	143
49	177
50	189
51	218
52	282
53	317
54	380
55	441
56	558
57	701
58	838
59	996
60	1106
61	1186
62	1092
63	692
64	328
#RLEs	18666
tot.	755925
avg.	40.50

Bibliography

[Gon] R.C. Gonzales and R.E. Woods
''Digital Image Processing'',
Addison Wesley,
June 1992, page 394 – 403.

[Hami] E. Hamilton
''JPEG File Interchange Format version 1.02'',
C-Cube Microsystems,
September 1, 1992.

[Ijg] The Independent JPEG Group
''The Independent JPEG Group's JPEG software, release 4'',
December 1-th, 1992.

[Lane] T. Lane
''JPEG image compression: Frequently Asked Questions'',
Internet: ftp://rtfm.mit.edu/pub/usenet/news.answers/jpeg-faq,
October 18th, 1993.

[Loef] C. Loeffler, A. Ligtenberg and G. Moschytz
''Practical Fast 1-D DCT Algorithms with 11 Multiplications'',
Proceedings International Conference on Acoustics, Speech, and Signal Processing 1989,
page 988 – 991.

[Smith] B. C. Smith and L. A. Rowe
''Algorithms for Manipulating Compressed Images'',
IEEE Computer Graphics & Applications,
September 1993, page 36 – 39.

[Wall] G. K. Wallace
''The JPEG Still Picture Compression Standard'',
Communications of the ACM,
Volume 34, Number 4, April 1991, page 30 – 44.