Experiments with a Lossless JPEG Codec

Kongji Huang

Cornell University

kongji@cs.cornell.edu

Thesis Advisor: Brian Smith

1.0 Introduction

The ISO JPEG Still Image Compression Standard is popularly known for its DCT-based compression technique, but its non-DCT based lossless mode of operation is less well known. Lossless JPEG uses a form of discrete pulse code modulation (DPCM) [3]. That is, a linear combination of a pixel's left, upper and upper left neighbors is used to predict the pixel's value, and the difference between the pixel and its predictor is coded through either Huffman or Arithmetic coding. Lossless JPEG defines seven linear combination known as prediction selection values (PSV).

Although the lossless JPEG standard has been available for several years, few experimental studies using the technique's performance have been reported in the literature. This paper describes an implementation of lossless JPEG and the results of experiments with the codec. In particular, we show that some classes of PSVs, namely two dimensional PSVs (using at least two neighboring pixels in predictor calculation) result in better compression for most "real world" images.

The rest of this paper is organized as follows. The lossless JPEG encoding and decoding algorithms are briefly reviewed in section 2. Our implementation is described in section 3. Besides the Huffman encoding and optimal Huffman encoding algorithm, we have introduced and implemented an effective automatic PSV selection algorithm which guarantees the best possible compression ratio for lossless JPEG on a given image.

Section 4 compares the speed of our codec with other popular lossless compression programs and demonstrates that our codec runs at comparable speeds. In section 4 we also show the breakdown of CPU time used by each compression phase. Section 5 compares lossless JPEG's compression ratio with other lossless compression programs, and studies the effect of different PSVs on image compression ratio.

2.0 Lossless JPEG Algorithm

This section describes the steps of the lossless JPEG process and gives three algorithms for implementing this process, one for decoding and two for encoding. A procedure for automatically selecting the PSV is also described.

2.1 The lossless JPEG encoding process

For simplicity of discussion, consider a source image as a two-dimensional array of samples. Each sample is a pixel containing either three components represent the red, green, and blue levels (rgb) for a color image or single component for a grayscale image. In lossless JPEG, each component specified as an integer with 2 to 16 bits of precision.

A point transformation parameter, *Pt*, can be used to reduce the source image precision. If *Pt* is non zero, each input sample is shifted right by *Pt* bits. The result is input to encoder as new samples.

The JPEG standard defines eight PSVs, given in table 1. PSV 0, which reserved for differential coding in the hierarchical progressive mode, is not used in our lossless JPEG codec. For each pixel, $P_{x,y}$, a linear combination of its left ($P_{x,y-1}$), upper ($P_{x-1,y}$), and upper left ($P_{x-1,y-1}$) neighboring pixels is used to calculate a predictor of $P_{x,y}$. PSV 1, 2 and 3 are one dimensional predictors (i.e., they use only one neighbor), and 4, 5, 6 and 7 are two dimensional (i.e., they use two or more neighbors).

PSVs	**Predictor**
0	no prediction (differential coding)
1	$P_{x,y-1}$
2	$P_{x-1,y}$
3	$P_{x-1,y-1}$
4	$P_{x,y-1}+P_{x-1,y}-P_{x-1,y-1}$
5	$P_{x,y-1}+(P_{x-1,y}-P_{x-1,y-1})/2$
6	$P_{x-1,y}+(P_{x,y-1}-P_{x-1,y-1})/2$
7	$(P_{x,y-1}+P_{x-1,y})/2$

TABLE 1. Predictors for lossless JPEG

The first row and column of the image are treated specially. PSV 1 is used for the first row of samples and PSV 2 is used for the first column. For the pixel in the upper left corner of the image, the predictor 2^{P-Pt-1} is used, where P is the sample precision bits.

The derived predictor is component-wise subtracted from the pixel value and the difference is encoded by either a Huffman or an Arithmetic entropy coder. Because Arithmetic entropy coder is patented, we have implemented only the Huffman entropy coder.

The Huffman entropy model partitions the differences into 16 difference categories, denoted as SSSS, where SSSS is a integer from 0 to 15. Each of the difference category is assigned a Huffman code using either a default Huffman table defined in [1] or one customized to the image. The entropy model takes advantage of the small set of difference category symbols to minimize the overhead of specifying and storing the Huffman table. The difference category table is listed below.

SSSS	Difference values
0	0
1	-1,1
2	-3,-2,2,3
3	-7..-4,4..7
4	-15..-8,8..15
5	-31..-16,16..31
6	-63..-32,32..63
7	-127..-64,64..127
8	-255..-128,128..255
9	-511..-256,256..511
10	-1023..-512,512..1023
11	-2047..-1024,1024..2047
12	-4095..-2048,2048..4095
13	-8191..-4096,4096..8191
14	-16383..-8192,8192..16383
15	-32767..-16384,16384..32767
16	32768

To fully specify a difference, SSSS additional bits are appended to the Huffman coded difference category to identify the sign and the magnitude of the difference. The rule for the additional bits is specified as follows. If the difference is positive, append the SSSS low order bits of the difference; if the difference is negative, subtract one from the difference and append the SSSS low-order bits of the resulting value. The additional bits are appended most significant bit first.

The following example illustrates the encoding process. The source image is a 2x3 grayscale image. We use PSV 7 ($(P_{x,y-1} + P_{x-1,y})/2$) as predictor and an optimal Huffman table to code the difference categories. Pt is set to zero.

source image		
120	100	100
80	90	100

diff = pixel - predictor		
-8	-20	0
-40	0	5

difference category and additional bits		
4 0111	5 01011	0
6 010111	0	3 101

output bit stream (spaces are not part of bit stream)					
101 0111	110 01011	0	111 010111	0	100 101

To decode the above encoded bits steam, the Huffman code 101 is first extracted and Huffman decoded to get the category symbol 4. Then 4 additional bits, 0111, are extracted from the bitstream. The leading 0 means the difference value is a negative number, so the magnitude of the difference value is 1 plus binary 0111. Adding the reconstructed -8 to its predictor, 128, the original pixel value 120 is recovered. This inverse encoding process is repeated until the bits stream is decoded.

2.2 Lossless Encoding

The procedure LEncode, shown in Figure 1, implements the lossless encoding process described in previous section, using a default Huffman table given in K.3.1 of [1] to encode each category symbol.

```
LEncode(PSV,image)

1. Output File Header and Scan header
2. for each pixel do
3.     D=image[x,y]-Predict(PSV,image[x-1,y],image[x,y-1],image[x-1,y-1]
4.     EmitBits(Hufco[D],Hufsi[D])
5.     if (D<0) then D=D-1
6.     if (D!=0) then EmitBits(D, NumOfAddBits(D))
7. od
```

FIGURE 1. LEncode - The lossless JPEG encoding algorithm

LEncode is passed a target image and one of the seven PSVs. It first outputs the file header information. Then, for each pixel, image[x,y], in the target image, a predictor is calculated by Predictor() using the PSV and neighboring pixels, image[x-1,y], image[x,y-1] and image[x-1,y-1]. The predictor is subtracted from image[x,y], and the difference, D, is encoded as a pair of values: *a difference category symbol* and *additional bits*. The difference category symbol is output at line 4 using EmitBits() and the default Huffman table stored in the arrays Hufco and Hufsi. The additional bits are output at line 6.

2.3 Optimal Lossless Encoder

The procedure OLencode, shown in Figure 2, is similar to LEncode with one difference: OLencode builds and uses a customized Huffman table to encode the difference categories. The generation of the customized Huffman table is frequency of each difference category. Thus, the inner loop of the LEncode is performed twice in OLencode: once to count the number of difference symbols (in Lines 1-4) and once to encode the image (in Lines 7-12).

```
OLEncode(PSV,image)

1.for each pixel do
2.    D=image[x,y]-Predict(PSV,image[x-1,y],image[x,y-1],image[x-1,y-1])
3.    Count(D,freqTbl)
4.od
5.BuildHuffmanTable(HuffTbl,freqTbl)
6.Output File Header and Scan header
7.for each pixel do
8.    D=image[x,y]-Predict(PSV,image[x-1,y],image[x,y-1],image[x-1,y-1]
9.    EmitBits(Hufco[D],Hufsi[D])
10.   if (D<0) then D=D-1
11.   if (D!=0) then EmitBits(D,NumOfAddBits(D))
12.od
```

FIGURE 2. OLEncode - The optimal lossless JPEG encoding algorithm

In both lossless encoding and optimal lossless encoding, the compression ratio varies significantly depending on choice of PSVs, and no single PSV performs best on all images. Thus our encoding program uses an automatic PSV selection procedure, described next, to pick the best PSV for a specific image.

2.4 Automatic PSV selection

The automatic PSV selection procedure can be incorporated in both LEncode and OLencode. The algorithm is shown in Figure 3. For each PSV in a PSV set, a frequency table is built to count the number of times each difference category appears. Then, the total bits of category symbols and additional bits are calculated using the frequency table and a Huffman table. The PSV which generates the least total number of bits is selected.

```
PSVSelection(PSVSet,image)

for each PSV in PSVSet do
    for each pixel do
        D=image[x,y]-Predict(PSV,image[x-1,y],image[x,y-1],image[x-1,y-1])
         Count(D,freqTable[PSV])
    od
    curTotalBits=TotalBits(freqTable[PSV],huffTbl)
    if (curTotalBits<curLeastTotalBits) then
       curLeastTotalBits=curTotalBits
       curBestPSV=PSV
    fi
od
```

FIGURE 3. PSVSelection - automatic PSV selection procedure

2.5 Lossless Decoder

The procedure LDecode, shown in Figure 4, implements the decoding algorithm. LDecode reads the header information from the input file, including 1 to 4 Huffman tables. It then extracts the encoded difference category symbol from the bits stream, decodes the symbol using the corresponding Huffman table, reconstructs a difference value, calculates predictor using the PSV (stored in the scan header) and the previously decoded neighboring pixels, and adds the predictor to the difference value to reconstruct the pixel. If the point transformation parameter, Pt, (also stored in the header) is non zero, each pixel value is shifted left Pt bits before being output.

```
LDecode(PSV,bits)


while (bitstream != NULL) do
    DC=Next difference category symbol
    nbits=Decode(DC, huffTbl)
    D=Extract(nbits)
    P=Predict(PSV,image[x-1,y],image[x,y-1],image[x-1,y-1])
    image[x,y]=D+P
od
```

FIGURE 4. LDecode - The lossless decoding algorithm

3.0 Implementation

This section briefly describes our implementation of the lossless JPEG algorithms described in the previous section.

The encoder implements lossless encoding on portable pixmap (PPM) images [4]. Command line options allow the user

1) to direct the encoder to select the best PSV from a user specified set of PSVs,

2) to supply a point transform value,

3) to select Huffman table optimization,

4) to output a variety of statistics.

The PSVSelection procedure selects the best PSV among the set of PSVs supplied by the user. If this set is not supplied, the PSVSelection selects the best among all seven PSVs. The encoder contains 3776 lines of C code, including comments, and memory usage is approximately equal to the size of the source image.

The encoder has been highly optimized. Profiling revealed that most of time was spent in the frequency counting step in 2-5 of figure 3. To make this critical step fast, we pushed all special case checking, such as first sample, first row and first column, out of the frequency counting loop, and the entropy encoding loop. We also unrolled the loop that derives differences between a pixel value and its predictor and updates the frequency tables for all PSVs, eliminated all function calls from the inner loops, and used lookup tables to speed up the calculation of additional bits size. These optimizations increased the speed of the encoder by 266%.

We tried caching the difference values calculated in PSVSelection procedure. The cached values were then used in the Huffman encoding phase. But the extra buffering required for caching tripled the memory usage, causing page faults and introducing extra overhead in the frequency counting loop. The result was that this version of the encoder ran almost three times slower than the current version.

The decoder decompresses a lossless JPEG compressed image from either standard input or a file. The decoded image is output to either standard output or a file in PPM format. The decoder contains 2412 lines of C codes, including comments. Since the decoder buffers only two rows of image for predictor calculation, its memory usage is negligible.

4.0 Encoding and decoding speed

The section reports the results of designed experiments to characterize the speed of the lossless JPEG encoding and decoding programs. The first part compares the codec's speed with other popular lossless compression programs. The second part studies codec's speed as a function of the number of pixels to compress.

4.1 Comparison with other lossless compression programs

The goal of this experiment is to compare the lossless JPEG encoding and decoding speed with other popular lossless compression programs: *gzip* (using Lempel-Ziv and optimal Lempel-Ziv algorithm [2]), *compress* (using LZW algorithm [2]), and *pack* (using Huffman coding [2]). All programs in this experiment are compiled with “gcc -O2”, and are timed on a SUN SPARC 10 with 32MB of memory using the Unix *time* program. Both lossless JPEG encoding (LJ encoding) and optimal lossless JPEG encoding (OLJ encoding) use the PSVSelection procedure to select the best PSV among all seven PSVs.

Twelve 640x480 images of various scenes were used in the experiment. Every image was compressed and decompressed ten times by each program. This insures that our experiment covers various images and the experiment results are not greatly affected measurement noise. The means of CPU time (and their standard deviations) are listed in Table 2.

encoding program	**CPU time (sec)**	**decoding program**	**CPU time (sec)**
LJ encoder	5.2 (±0.4)	LJ decoder	1.9 (±0.3)
OLJ encoder	5.5 (±0.4)	LJ decoder	1.9 (±0.3)
compress	2.0 (±0.6)	uncompress	1.1 (±0.3)
gzip	4.9 (±1.5)	gunzip	0.5 (±0.1)
gzip -9	10.1 (±5.8)	gunzip	0.5 (±0.1)
pack	0.8 (±0.1)	unpack	1.8 (±0.4)

TABLE 2. Speed comparison of lossless coding programs

The above data shows that the speed of both the LJ encoder and the OLJ encoder is close to gzip, despite the incorporation of the computationally intensive automatic PSV selection procedure. The lossless JPEG decoder is one to four times slower than other decompress programs.

4.2 Breakdown of CPU time

The encoder spends about 58% of CPU time on frequency counting, 17% on Huffman coding (including emitting bitstream), 8% on image reading, and 7% on predictor calculation. The decoder spends about 31% of CPU time on Huffman decoding of the category symbols, 17% on predictor calculation, 15% on writing pixels, 13% on reading bitstream, and the rest CPU time on reading and writing header information, initialization, and command line argument parsing.

5.0 Compression ratio and compressed image data distribution

The compression ratio for lossless JPEG, defined as (original image size)/(compressed image size), is typically between 1.4:1 and 3:1. For "busy" scenes with many large changes in local pixel values, the compression ratio is around 1.5:1, but for scenes with smooth changes, the compression ratio can reach as high as 3:1.

5.1 Comparison with other lossless compression programs

We compared the compression ratio of Lossless JPEG with three popular compression programs, namely gzip, compress and pack. Four representative images were used in this experiment: a photo of human face ("Lena"), a sports photo ("football"), a flower scene ("flowers"), and a picture of an airplane ("F-18"). All images appear in appendix A. Table 3 shows the compression ratio obtained when each program was applied to the four images.

	compression ratio			
compression program	**Lena**	**football**	**F-18**	**flowers**
lossless JPEG	1.45	1.54	2.29	1.26
optimal lossless JPEG	1.49	1.67	2.71	1.33
compress (LZW)	0.86	1.24	2.21	0.87
gzip (Lempel-Ziv)	1.08	1.36	3.10	1.05
gzip -9 (optimal Lempel-Ziv)	1.08	1.36	3.13	1.05
pack (Huffman coding)	1.02	1.12	1.19	1.00

TABLE 3. Comparison of compression ratio for selected images

Both LJ and OLJ compress better than the other four programs for Lena, football and flowers. Gzip and gzip -9 compresses somewhat better for F-18 because of the image's constant background. Note that OLJ compresses 3% to 18% better than LJ.

5.2 Effects of PSV

This section studies the effect of different PSVs on compression ratio. This first subsection analyzes the compressed image data distribution and compression ratio in detail. The second subsection studies the relationship between PSVs and compression ratio using a larger set of images.

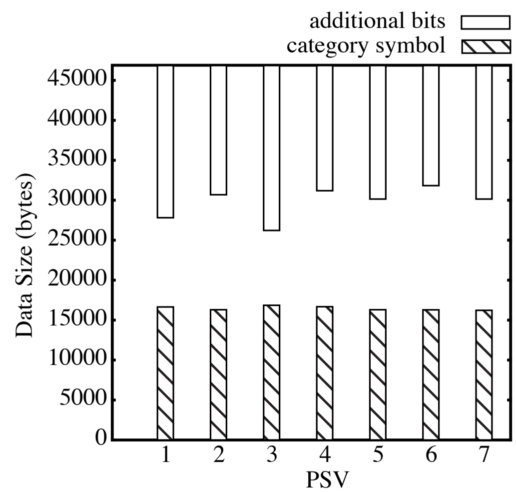
5.2.1 Compressed image data distribution and compression ratio for all PSV

Tables 4-7 show the compression ratio and bit allocation in the compressed bitstream for each PSV when the image *Lena*, *football*, *F-18* and *flowers* are compressed using OLencode. The table's columns list the PSV used, the percentage of bits in the compressed image used by *category symbols*, the percentage of bits used by the *additional bits* (described in section 2.1), and the *compression ratio*.

The figure beside each table gives a graphical view of the data listed in the table. Each bar corresponds to a PSV and has two parts, the lower bar for the category symbol size, and the upper bar for the additional bits size. The graph height is the original image size.

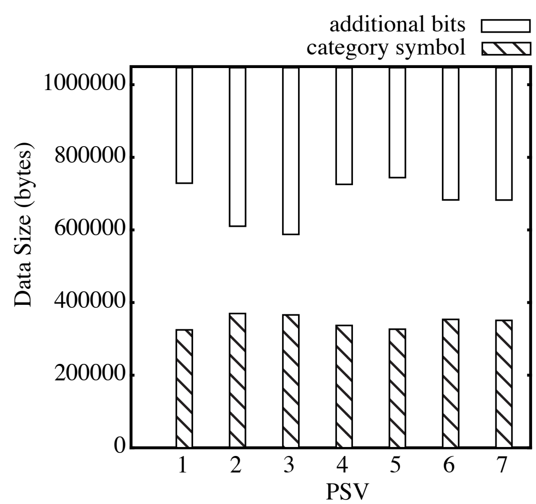
PSV	cat sym%	add bits%	comp ratio
1	46%	53%	1.31
2	50%	50%	1.44
3	45%	55%	1.25
4	51%	48%	1.44
5	49%	50%	1.41
6	52%	48%	1.49
7	49%	50%	1.42

TABLE 4. Lena - effects of PSV



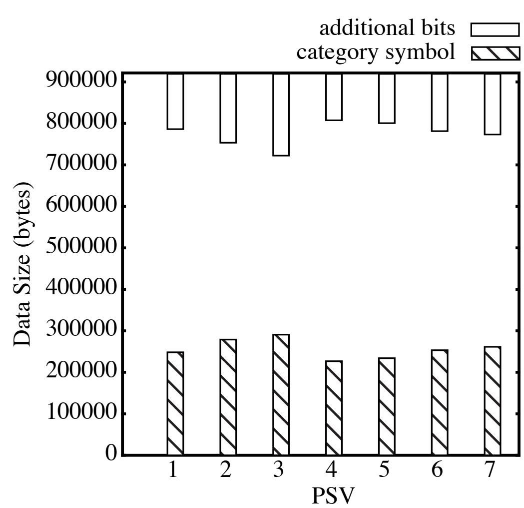
PSV	cat sym%	add bits%	comp ratio
1	50%	49%	1.63
2	46%	54%	1.30
3	44%	56%	1.27
4	51%	49%	1.59
5	52%	48%	1.67
6	49%	51%	1.46
7	49%	51%	1.47

TABLE 5. football - effects of PSV



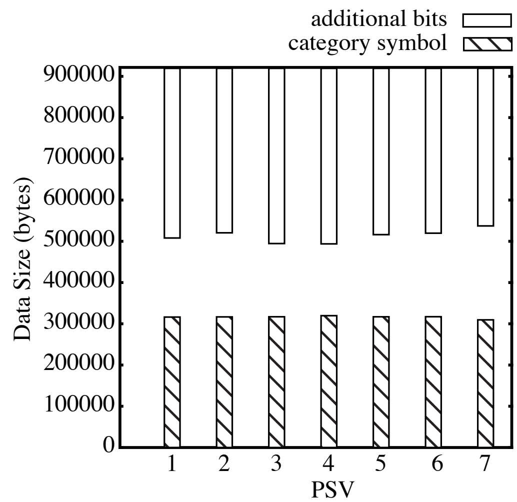
PSV	cat sym%	add bits%	comp ratio
1	65%	35%	2.40
2	62%	37%	2.06
3	59%	40%	1.88
4	67%	33%	2.71
5	66%	33%	2.59
6	64%	35%	2.34
7	64%	36%	2.24

TABLE 6. F-18 - effects of PSV



PSV	cat sym%	add bits%	comp ratio
1	44%	56%	1.27
2	44%	56%	1.29
3	43%	57%	1.24
4	43%	57%	1.24
5	44%	56%	1.28
6	44%	56%	1.28
7	45%	55%	1.33

TABLE 7. flowers - effects of PSV



In the 125x125 image Lena, PSV 6 gives the best compression ratio (1.49), and PSV 3 gives the worst compression ratio (1.25). The bitstream consists of category symbols and additional bits in roughly equal proportions. From the adjoining chart we see that the category symbols size does not vary much among PSVs. The variation in additional bits is the cause of the difference in compression ratio. This is because a good PSV accurately predicts a pixel's value, generating small difference values, and therefore less additional bits.

The second test image is a 720x486 scene of football game. Table 5 shows that PSV 5 gives the best compression ratio (1.67), and PSV 3 gives the worst (1.27). As before, the number of bits allocated to additional bits is much more variable than the number of bits allocated to category symbols.

Image 3 is a 640x480 picture of U.S. Navy Blue Angel - F18. From table 6, PSV 4 gives the best compression ratio (2.71) and, again, PSV 3 gives the worst (1.88). In this image, the category symbols comprise 60% to 70% of the bitstream, the additional bits comprise

30% to 40%. This picture contains a relative "simple" scene with large constant background, so almost all PSVs predict well and generate less additional bits.

Image 4 is a 640x480 colorful flower scene. Table 7 shows that PSV 7 gives the best compression ratio (1.33) and PSV 3 and PSV 4 tie for the worst compression ratio (1.24). The category symbol consists about 45% of the compressed image, the additional bits consists about 55%.

These experiments reveal the following properties of lossless JPEG. First, the header information consists at most 1% of the bitstream. Second, no single good PSV is best for all images. This observation led us to make automatic PSV determination the default behavior of our encoder. Third, the best PSV always generates the fewest additional bits. This is because the best PSV best predict a pixel's value, and the difference between a pixel and its predictor is small. So the compressed image comprise less additional bits. Fourth, the number of bits allocated to additional bits is much more variable than the number of bits allocated to category symbols. The total bits of additional bits is $\sum_{n=1}^{15} C(n) \cdot n$ and the total bits of category symbols is $\sum_{n=0}^{15} C(n) \cdot L(n)$, where $C(n)$ is the number of *difference values* in category n and $L(n)$ is the length of Huffman code for category symbol n. While both n and $L(n)$ may vary between 1 and 15, the Huffman encoder assigns a shorter code (smaller $L(n)$) to a category with more values falling in (larger $C(n)$. This makes the category symbol size less variable than the additional bits size. Finally, the best PSV is always one of the two-dimensional PSVs (i.e. PSV 4, 5, 6 or 7), and PSV 3 always compresses poorly. Is this a general pattern? Answering this question led us to study a larger set of images.

5.2.2 Average compression ratio for each PSV

We calculated the average compression ratio for each of the seven PSVs by apply the lossless JPEG algorithm with Huffman table optimization to twenty images and measuring the compression ratios. The images contain various scenes of people, nature, space craft, rooms, sports activities, scientific demonstrations, and graphs. Figure 5 shows the experiment results. For each PSV, the bar gives the mean compression ratio and the line at the top of each bar gives the standard deviation.

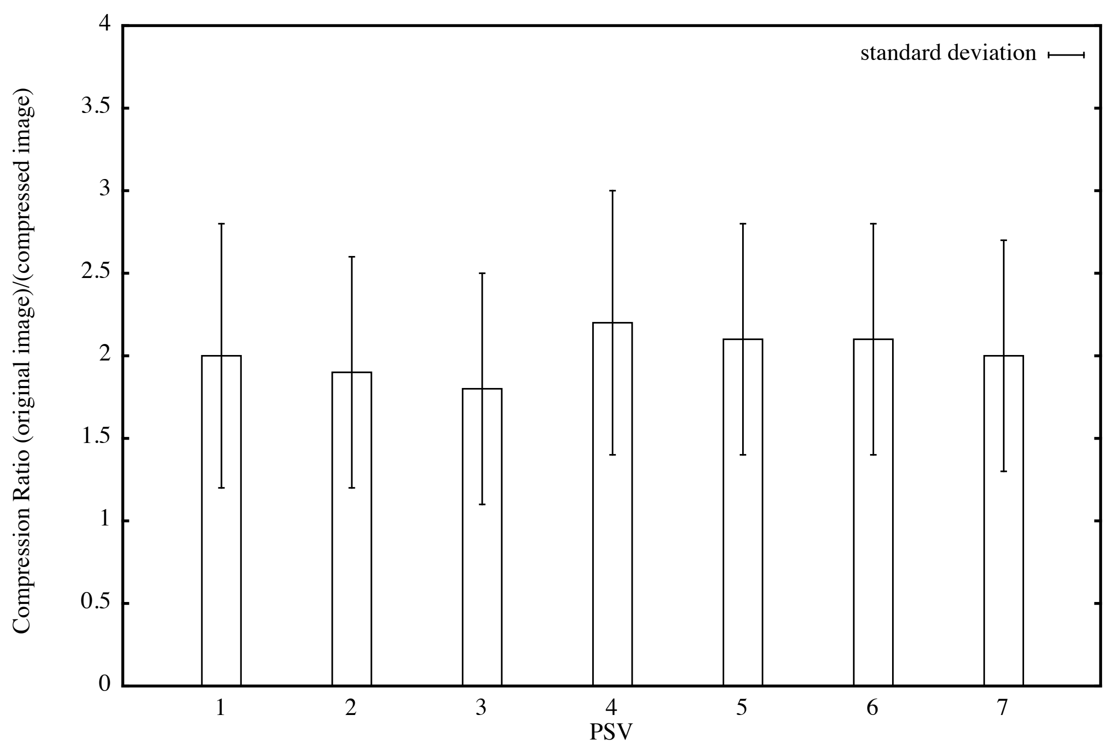


FIGURE 5. OLEncode Average Compression Ratio for the Seven PSVs

The results are consistent with our hypothesis: two-dimensional PSVs generally generate higher compression ratio than one-dimensional PSVs. PSV 4 compresses best on average, PSV 5 and 6 compress equally well and a little bit better than PSV 7. Among the three one-dimensional PSVs, PSV 3 compresses the worst. On average, the compression ratio of best PSV is about 0.5 bigger than worst.

Why should two dimensional PSVs compress better than one dimensional PSVs? PSV 4 ($P_{x, y-1} + P_{x-1, y} - P_{x-1, y-1}$), PSV 5 ($P_{x, y-1} - (P_{x-1, y} - P_{x-1, y-1}) / 2$), or PSV 6 ($P_{x-1, y} - (P_{x, y-1} - P_{x-1, y-1}) / 2$) uses three neighbor pixels in their predictor computation; PSV 7 ($(P_{x, y-1} + P_{x-1, y}) / 2$) uses two; the rest only use one ($P_{x, y-1}$, $P_{x-1, y}$ and $P_{x-1, y-1}$). So it seems that the more information that is used, the better the compression ratio. Among the three one-dimensional PSVs, PSV 3 uses the “farthest” neighbor, the upper left pixel, and compresses the worst. PSV 5 and 6 have similar prediction functions which lead to similar compression ratio. PSV 4 not only has the advantage of best average compression ratio, it contains less operation than PSV 5 and 6 in predictor calculation. From these observations we conclude that the best predictors are those which use as much locality as possible and average nearby pixels to reduce the effects of image noise.

To optimize the prediction for each image, a linear regression model can be used to derive an optimal prediction function $P_{x, y} = c_1 P_{x, y-1} + c_2 P_{x-1, y} + c_3 P_{x-1, y-1}$ such that $Min\{\sum_{x, y \in image} (I_{x, y} - P_{x, y})^2\}$, where $I_{x, y}$ is the pixel value at (x y). Then this function is used in OLencode to better predict each pixel and generate a higher compression ratio.

References

1. ISO/IEC Information technology - Digital compression and coding of continuous-tone still images. ISO/IEC DIS 10918-1
2. Mark Nelson. The Data Compression Book. M&T Publishing, Inc. (1991)
3. William B. Pennebaker, Joan L. Mitchell. JPEG Still Image Data Compression Standard. Van Nostrand Reinhold (1993)
4. Jef Poskanzer. Extended Portable Bitmap Toolkit (PBMPLUS), Distribution of 10 Dec. 1991.
5. Gregory K. Wallace. The JPEG Still Picture Compression Standard, Communications of ACM, April 1991, vol. 34 no. 4, pp. 30-44

Appendix A

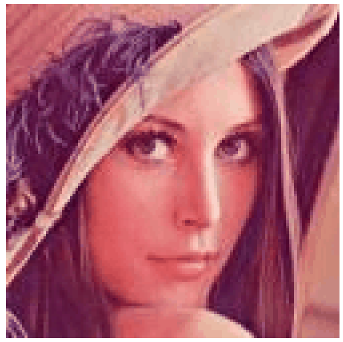


FIGURE 6. Image 1: Lena

FIGURE 7. Image 2: football

FIGURE 8. Image 3: F-18

FIGURE 9. Image 4: flowers