Fast and Efficient Lossless Image Compression

Paul G. Howard and *Jeffrey Scott Vitter*

appears in the proceedings of the
IEEE Computer Society/NASA/CESDIS Data Compression Conference,
Snowbird, Utah, March 30–April 1, 1993, pages 351–360.

Fast and Efficient Lossless Image Compression

Paul G. Howard[1]
Department of Computer Science
Brown University
Providence, R.I. 02912–1910

Jeffrey Scott Vitter[2]
Department of Computer Science
Duke University
Durham, N.C. 27706–0129

Abstract

We present a new method for lossless image compression that gives compression comparable to JPEG lossless mode with about five times the speed. Our method, called *FELICS*, is based on a novel use of two neighboring pixels for both prediction and error modeling. For coding we use single bits, adjusted binary codes, and Golomb or Rice codes. For the latter we present and analyze a provably good method for estimating the single coding parameter.

[1]Support was provided in part by NASA Graduate Student Researchers Program grant NGT–50420. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

[2]Work was performed while the author was at Brown University. Support was provided in part by a National Science Foundation Presidential Young Investigator Award with matching funds from IBM and by Air Force Office of Scientific Research grant number F49620–92–J–0515. Additional support was provided by a Universities Space Research Association/CESDIS associate membership.

1 Introduction

Most lossless image compression methods in the literature consist of four main components [6]: a selector to choose the next pixel to be encoded, a predictor to estimate the intensity of the pixel, an error modeler to estimate the distribution of the prediction error, and a statistical coder to code the prediction error using the error distribution. By using an appropriate pixel sequence we can obtain a progressive encoding, and by using sophisticated prediction and error modeling techniques in conjunction with arithmetic coding we can obtain state-of-the-art compression efficiency [6,7]. These techniques are computation intensive.

In this paper we present a simpler system for lossless image compression that runs very fast with only minimal loss of compression efficiency. We call this technique *FELICS*, for *Fast, Efficient, Lossless Image Compression System*. We use raster-scan order, and we use a pixel's two nearest neighbors to directly obtain an approximate probability distribution for its intensity, in effect combining the prediction and error modeling steps. We use a novel technique to select the closest of a set of error models, each corresponding to a simple prefix code. Finally we encode the intensity using the selected prefix code. The resulting compressor runs about five times as fast as an implementation of the lossless mode of the JPEG proposed standard while obtaining slightly better compression on many images.

2 Description of the method

Proceeding in raster-scan order, we code each new pixel P[3] using the intensities of the two nearest neighbors of P that have already been coded; except along the top and left edges, these are the pixel above and the pixel to the left of the new pixel (see Figure 1). We call the smaller neighboring value L and the larger value H, and we define Δ to be the difference $H - L$. We treat Δ as the prediction context of P, used for code parameter selection.

The idea of the coding algorithm is to use one bit to indicate whether P is in the range from L to H, an additional bit if necessary to indicate whether it is above or below the range, and a few bits, using a simple prefix code, to specify the exact value. This method leads to good compression for two reasons: the two nearest neighbors provide a good context for prediction, and the image model implied by the algorithm closely matches the distributions found in real images. In addition, the method is very fast, since it uses only single bits and simple prefix codes.

2.1 Intensity distributions

When we examine the distribution of an image's intensity values for each context Δ, we find that the intensities are generally distributed as shown in Figure 2. Typically

[3]With a slight abuse of notation, we use symbols P, H, L, N_1, and N_2 to refer both to pixels and to their intensity values.

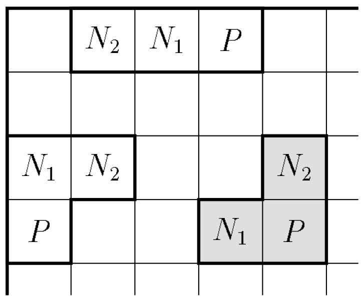

Figure 1: Nearest neighbors N_1 and N_2 used for coding the intensity of pixel P in FELICS. The "range" used in making the in-range/out-of-range decision is $[\min\{N_1, N_2\}, \max\{N_1, N_2\}]$, and the "context" used for modeling the probability distribution is $|N_1 - N_2|$. For points in the center of the image (shaded), the predicting pixels are the pixels immediately to the left of and above P. Along the edges adjustments must be made, but otherwise the calculations are the same.

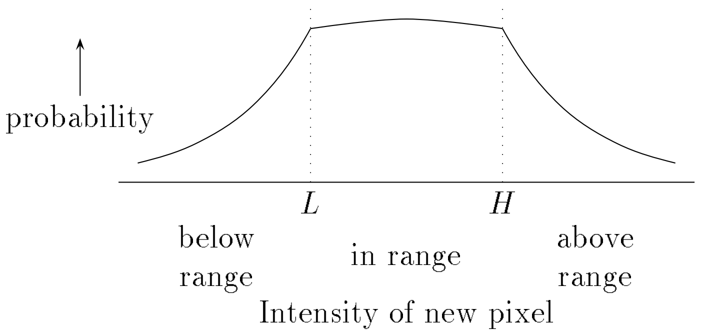

Figure 2: Schematic probability distribution of intensity values for a given context $\Delta = H - L$.

P lies in the range $[L, H]$ about half the time, requiring one bit to encode, and when P is out of range, the above-range/below-range decision is symmetrical, so another one-bit code is appropriate. In-range values of P are almost uniformly distributed, with a slight crest near the middle of the range, so an adjusted binary encoding gives nearly optimal compression when P is in range. The probability of out-of-range values falls off sharply, so when P is out of range it is reasonable to use exponential prefix codes, i.e., Golomb codes or the simpler Rice codes, to indicate how far out of range the value is. This distribution clearly differs from the Laplace distribution commonly assumed in predictive image coding, but it is consistent with the error modeling treatment that we presented in [7].

2.2 Adjusted binary codes

To encode an in-range pixel value P, we must encode $P - L$ in the range $[0, \Delta]$. If $\Delta + 1$ is a power of two we simply use a binary code with $\log_2(\Delta + 1)$ bits; this

Golomb Rice	$m = 1$ $k = 0$	$m = 2$ $k = 1$	$m = 3$	$m = 4$ $k = 2$	$\cdots$	$m = 6$	$\cdots$	$m = 8$ $k = 3$
$n = 0$	**0·**	**0·0**	**0·0**	**0·00**		**0·00**		**0·000**
1	**10·**	**0·1**	**0·10**	**0·01**		**0·01**		**0·001**
2	**110·**	**10·0**	**0·11**	**0·10**		**0·100**		**0·010**
3	**1110·**	**10·1**	**10·0**	**0·11**		**0·101**		**0·011**
4	**11110·**	**110·0**	**10·10**	**10·00**		**0·110**		**0·100**
5	**111110·**	**110·1**	**10·11**	**10·01**		**0·111**		**0·101**
6	**1111110·**	**1110·0**	**110·0**	**10·10**		**10·00**		**0·110**
7	**11111110·**	**1110·1**	**110·10**	**10·11**		**10·01**		**0·111**
8	**111111110·**	**11110·0**	**110·11**	**110·00**		**10·100**		**10·000**
9	**1111111110·**	**11110·1**	**1110·0**	**110·01**		**10·101**		**10·001**
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$		$\vdots$

Table 1: The beginnings of Golomb and Rice codes for a few parameter values. The codes can be extended to all non-negative values of n, and codes can be constructed for all $m > 0$ and all $k \geq 0$. In this table a midpoint ($\cdot$) separates the high-order (unary) part from the low-order (binary or adjusted binary) part of each codeword.

works well because the distribution of in-range values is nearly uniform. Otherwise we can adjust the code in the obvious way, assigning $\lfloor \log_2(\Delta + 1) \rfloor$ bits to some values and $\lceil \log_2(\Delta + 1) \rceil$ bits to others. Because the values near the middle of the range are slightly more probable, we assign shorter codewords to those values. For example, if $\Delta = 4$, we have to allow for coding the five values 0, 1, 2, 3, and 4. The adjusted binary codewords are **00**, **01**, **10**, **110**, and **111**. We start assigning the shorter codewords at the middle of the range, that is, at $P - L = 2$, and arrive at the following code:

$P - L$	0	1	2	3	4
codeword	**111**	**10**	**00**	**01**	**110**

In passing we note that the number of longer codewords is always even, so the codeword lengths within each of these codes are always symmetrical with respect to the range $[0, \Delta]$. Thus we never encounter the annoying problem present in many image compression schemes of having to assign different length codewords to values with theoretically equal probabilities.

2.3 Exponential prefix codes (Golomb and Rice codes)

When the distribution of values to be encoded is exponential, we can use an exponential prefix code, first discussed by Golomb [5]. The codes in this family are parameterized by a positive integer parameter m. Given the value of m, we encode non-negative integer n by encoding $\lfloor n/m \rfloor$ in unary, then encoding $n \bmod m$ using an adjusted binary code for the range $[0, m - 1]$, as in Section 2.2. It can be shown that the correct choice of the parameter m produces an optimal prefix code for a given exponential distribution [4].

Rice [10] independently discovered the special case of Golomb codes where $m = 2^k$ for some integer k. Restricting m to be a power of 2 leads to exceptionally simple codes. Given k, the value of the coding parameter, we encode n by first removing the k least significant bits of n and encoding the remaining bits as a unary number. Then we send the k least significant bits directly. Rice coding has been used as the basis for a lossless hardware compressor [14]. Its compression effectiveness is analyzed in [17].

Both Golomb coding and Rice coding require estimation of the coding parameter. We present an effective technique in Section 3.

Examples of some Golomb and Rice codes are shown in Table 1. Golomb codes give slightly better compression by providing finer control in choosing the model. On the other hand, Rice codes are somewhat simpler to implement and run slightly faster; parameter estimation is faster for Rice codes because there are fewer reasonable parameters from which to choose; and the analysis of Rice codes is more straightforward. In this paper we focus on Rice codes, but most of our work applies to Golomb codes as well. We use the term *Golomb-Rice code* when either may be used.

2.4 Formal description of algorithm

To encode an image, we output the first two pixels without coding, then repeat the following steps:

1. We select the next pixel P and find its two nearest neighbors N_1 and N_2.
2. We compute $L = \min(N_1, N_2)$, $H = \max(N_1, N_2)$, and $\Delta = H - L$.
3. (a) If $L \leq P \leq H$, we use one bit to encode IN-RANGE; then we use an adjusted binary code to encode $P - L$ in $[0, \Delta]$.

   (b) if $P < L$, we use one bit to encode OUT-OF-RANGE, and one bit to encode BELOW-RANGE. Then we use a Golomb-Rice code to encode the non-negative integer $L - P - 1$.

   (c) if $P > H$, we use one bit to encode OUT-OF-RANGE, and one bit to encode ABOVE-RANGE. Then we use a Golomb-Rice code to encode the non-negative integer $P - H - 1$.

The decoding algorithm involves simply reversing step 3, decoding the in-range/out-of-range and above-range/below-range decisions, branching accordingly, and adjusting the decoded numbers to obtain the value of P.

2.5 Example

In this example we use **1** to encode IN-RANGE, **0** to encode OUT-OF-RANGE, **0** to encode BELOW-RANGE, and **1** to encode ABOVE-RANGE. Suppose we are wish to

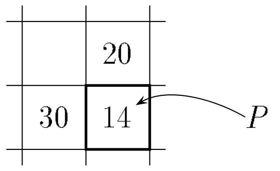

Figure 3: FELICS example.

encode the intensity of pixel P in Figure 3; its value is 14. The nearest neighbors have intensities $N_1 = 30$ and $N_2 = 20$, so $L = 20$ and $H = 30$. Since $P < L$, we output **0** for OUT-OF-RANGE and another **0** for BELOW-RANGE. The context for pixel P is the difference $\Delta = H - L = 10$. For the example we assume that we estimate the Rice coding parameter for context $\Delta = 10$ to be $k_{10} = 2$. Since Rice codes are used to encode non-negative integers, we encode $n = 0$ for a value just out of range (in this case, $P = 19$ would be encoded as $n = 0$), $n = 1$ for the next value, and so on. The value $P = 14$ is encoded as $n = 5$. To encode with the Rice code with $k = 2$, we divide $n = 5_{10} = 101_2$ into two parts, $1 \cdot 01$, the low-order part having $k = 2$ bits. Then we output the high-order part 1 in unary, giving **10**, and the k low-order bits, **01**, directly. Thus the output for this pixel is **001001**.

3 Selection of Rice-Golomb Coding Parameter

To complete the description of the algorithm in Section 2.4, we must specify the parameter of the prefix codes used in steps 3(b) and 3(c). In this section we describe an on-line algorithm for parameter estimation and prove a bound on its effectiveness.

3.1 Selection algorithm

A common method of obtaining and transmitting coding parameters in similar algorithms [11] is to divide the image into blocks and compute the code lengths that would be obtained using each of a set of reasonable parameter values. The problems with this approach are the delay at the start of each block, the need to send the block parameter value as side information, and the failure to use any pixel-specific context information.

Rather than assuming that the parameter is constant over a block of pixels, we make the more reasonable assumption that it is the same for pixels in regions of similar image activity levels; we take the image activity at a pixel to be simply the range Δ of its two predictors. It might be possible to adaptively estimate the mean and variance of the distribution of encoded values and then to select the best-fitting exponential distribution, but we have found a more straightforward method.

For each context Δ we maintain a cumulative total, for each reasonable Rice parameter value k, of the code length we would have if we had used parameter k to encode all values encountered so far in the context. Then we simply use the parameter with the smallest cumulative code length to encode the next value encountered in the context. Both in theory and in practice we quickly converge to good parameter estimates.

3.2 Analysis

In this section we analyze the code lengths obtained by using our parameter estimation method. We assume the use of Rice coding because it is easier to analyze and implement than Golomb coding; the extra code length involved is very small. We assume that the source probabilities follow a geometric distribution, given by $p_n = p_0(1-p_0)^n$, where p_0 is the probability that symbol $n = 0$ occurs. We assume that p_0 has a fixed but unknown value. For each non-negative integer value k of the coding parameter, we define

$$\beta_k = (1-p_0)^{2^k}.$$

We define random variable l_k to be the code length for one symbol encoded using parameter value k, and we define random variable d_k to be $l_{k+1} - l_k$. When we are using parameter k, the 2^k smallest values (0 to $2^k - 1$) require $k+1$ bits each to encode, the next 2^k values require $k+2$ bits, and so on, the number of bits increasing by 1 every 2^k values. From this we can find the values of $d_k(n)$, the code length difference for encoding symbol n. The value of $d_k(n)$ is 1 for $0 \leq n < 2^k$; it is 0 for the next 2^{k+1} values of n; then it decreases by 1 every 2^{k+1} values. The aggregate probability of the symbols for which $d_k = 1$ can be shown to be $1-\beta_k$; for $d_k = \Delta \leq 0$ the probability is $\beta_k^{-2\Delta+1}(1-\beta_k^2)$. Using these probabilities, we find that the expected value of d_k, denoted by $\overline{d_k}$, is given by

$$\overline{d_k} = \mathrm{E}(d_k) = 1 - \frac{\beta_k}{1-\beta_k^2},$$

and that the variance of d_k is given by

$$\mathrm{Var}(d_k) = \frac{\beta_k(1-\beta_k+\beta_k^2)}{(1-\beta_k^2)^2}.$$

For a given value of p_0, we find that $\mathrm{E}(l_k) = \mathrm{E}(l_{k+1})$ (that is, $\mathrm{E}(d_k) = 0$) when $\beta_k = (\sqrt{5}-1)/2 \approx 0.618$. (This value, $(\sqrt{5}-1)/2$, denoted by $\hat{\varphi}$, is $\varphi - 1 = 1/\varphi$, where φ is the golden ratio.) The critical values of p_0, where the best parameter choice changes from k to $k+1$, are those that make $\beta_k = \hat{\varphi}$. It is remarkable that this analysis holds for all values of k.

For now we restrict our choice to two parameter values, k and $k+1$, presumed to be the two best choices for the current value of p_0. We assume that k is the best choice; similar reasoning applies if $k+1$ is optimal. At any point our algorithm may choose the worse of the parameter values, but we now show that in coding a sequence of N symbols, we use on average at most $O(\sqrt{N})$ bits more than the average number used by the best parameter.

We note that since $\beta_{k+1} = \beta_k^2$ and $\beta_{k-1} = \sqrt{\beta_k}$, the value of β_k satisfies the relation $\hat{\varphi}^2 \leq \beta_k \leq \sqrt{\hat{\varphi}}$ when k is the optimal parameter value. Throughout this range, the standard deviation of d_k is bounded by a small constant $s \approx 2.117$, the value when $\beta_k = \sqrt{\varphi}$.

As we proceed in the coding, the average value of the cumulative difference D_t between the code lengths for the two candidate parameters will increase linearly. At first, when its expected value is not large (up to $O(\sqrt{N})$), the actual value will sometimes be negative, causing our algorithm to select the wrong parameter; but in this case the average excess code length is only $O(\sqrt{N})$. Eventually the average difference becomes large enough that choosing the wrong parameter becomes very unlikely, the low probability canceling the potentially higher cost of an incorrect choice. The net effect is an excess of $O(\sqrt{N})$ bits. We formalize this reasoning in the following theorem.

Theorem 1 *For a stationary source whose probability distribution is given by $p_n = p_0(1-p_0)^n$, using our parameter selection algorithm in conjunction with Rice coding gives an expected code length that exceeds the expected code length given by the optimal parameter by at most $O(\sqrt{N})$ bits.*

Proof: We define D_t to be the cumulative sum of random variable d_k up to time t. We let $\delta = s\sqrt{N}$, where s is the maximum standard deviation of d_k in the range of interest, and we let T be the number of symbols needed until $\mathrm{E}(D_T) = T\overline{d_k}$ becomes δ. Up to symbol T, even if we always choose the wrong parameter the total expected excess code length is only $s\sqrt{N}$, by definition.

We divide the remaining symbols into intervals of length T. In the interval from rT to $(r+1)T$, we will choose the wrong parameter only when $D_t < 0$. Since the expected difference is $r\delta$ at the beginning of the interval and increases within the interval, and the standard deviation of D_t throughout the interval is less than $s\sqrt{(r+1)T} \leq s\sqrt{N} = \delta$, we see that D_t becomes negative only if its value is more than $r\delta/\delta = r$ standard deviations away from its mean. By Chebyshev's inequality, the probability of this happening is at most $1/r^2$. Hence the expected number of times we choose the wrong parameter in the interval is at most T/r^2. The average excess code length when we choose the wrong parameter is $\overline{d_k} = \delta/T$, so the expected excess for the interval is at most δ/r^2. We sum this excess over all intervals, and find the total to be

$$\sum_{r=1}^{N/T} \frac{\delta}{r^2} < \delta \sum_{r=1}^{\infty} \frac{1}{r^2} = \delta \frac{\pi^2}{6} = \frac{\pi^2}{6} s\sqrt{N}.$$

Including the excess for the first interval, we see that the total expected number of excess bits over all N input symbols is less than

$$s\sqrt{N}\left(1 + \frac{\pi^2}{6}\right) = O(\sqrt{N}).$$

□

The constant factor on the bound can be improved by better tail estimates. In this shortened version of the paper we omit the proof that parameters other than the two closest to optimal contribute a negligible amount to the excess code length.

Our theorem bounds the average excess code length used by our algorithm. We expect that it can be extended by an introduction of randomness to arbitrary individual sequences, showing that with high probability our method gives a code length for a sequence within $O(\sqrt{N})$ bits of that produced by using the best value of the parameter on that sequence.

3.3 Extensions

The parameter estimation technique described here makes the use of Golomb-Rice coding feasible in the algorithm in Section 2.4. Its use can also be extended in two different directions.

First, it is now possible to use Golomb-Rice coding as an alternative to arithmetic or Huffman coding in almost any setting requiring adaptive modeling. All that is required is that the events to be encoded be arranged in approximately descending order of probability. In the image compression system described here, the ordering is the natural one due to the exponential distribution of prediction errors, but in other applications the ordering can be achieved by maintaining event frequency counts or by using heuristics such as move-to-front (move an event to the head of the list whenever it occurs) or transpose (move an event up one place in the list whenever it occurs). The only extra storage required is that needed for the cumulative counts for the possible parameter values. The lists and cumulative counts can be maintained for each of a number of contexts. Here the contexts are the intensity ranges, but we have also successfully applied the method to text compression using groups of preceding symbols as the contexts [8].

Golomb-Rice coding gives the fast, flexible modeling obtained with arithmetic coding without the time-consuming arithmetic. It gives faster coding even than Huffman coding because of the especially simple prefix codes involved, and adaptive modeling is possible without the complicated data structure manipulations required in dynamic Huffman coding [2,3,9,15,16]. The main drawback to Golomb-Rice coding is the limited class of distributions that can be modeled exactly, but even this is not a serious problem (unless one event's probability is close to 1) because the probabilities of the more probable events will be estimated fairly well. The idea of using a simple code in conjunction with ordered distributions is similar in spirit to the universal coding methods developed by Elias [1] and Rissanen [12], although their work involves finding a *single* universal code, not a family. Rissanen [13] presents a parameterized method for *finite* alphabets, the parameter being the reciprocal of the most probable event.

The second extension is to more general parameter estimation. Here we applied the technique to the estimation of the Golomb-Rice coding parameter m or k, but in fact we can adaptively estimate any modeling parameter. Most compression techniques can be improved by allowing tweaking of modeling parameters, but end users (and researchers) can be confused by a multiplicity of choices. Our technique permits hiding the tweaking within the program, and gives rapid convergence to good parameter values, at least for sources that can be modeled by stationary distributions.

4 Implementation and refinements

The basic algorithm of Section 2.4 is very easy to implement. As described it encodes and decodes very quickly and gives good compression. The implementation requires very little memory, only enough to store one scan line of input data and a few counts for each of the few hundred possible contexts.

In this section we describe several enhancements that can be made to improve speed and compression. One possibility is to freeze the parameter choice for a context after a number of symbols have been encoded. This saves the time needed to maintain the cumulative counts, and does not hurt compression much since in practice the parameter selection algorithm converges quickly to the best value.

A second enhancement is to adjust the range $[L, H]$ when $L = H$. In this special case, the in-range probability tends to be somewhat smaller than 1/2, and it makes sense to use the range $[L-1, H+1]$. We can do this either unconditionally or based on the number of times that the value to be encoded is equal to L (and H) when $\Delta = 0$. We choose the second possibility, adjusting the interval when the values encoded have fallen out of the one-value "range" more than 2/3 of the time. We might also consider *balance coding*, adjusting the range for each context to adaptively balance the in-range and out-of-range probabilities. We can use the parameter estimation algorithm of Section 3 to choose the amount of adjustment.

One final useful refinement is to assign a small initial penalty to the cumulative code lengths for small values of the parameter k in each context, to prevent their accidental use in contexts where the probability distribution is flat; such use can greatly increase the code length for a single pixel.

To further improve compression, we consider periodic count scaling to exploit locality of reference within an image. We tried applying it to the cumulative code lengths in each context. Halving all code lengths when the smallest one reaches 1024 can improve compression by up to 0.3 percent, but encoding time increases by about 10 percent, too much of a time penalty to pay for such a small increase in compression.

Finally, we note that Golomb coding gives only a marginal improvement over Rice coding despite the wider range of model parameters available. The extra compression is typically less than one percent, and the time required almost doubles. Therefore we use Rice coding in our implementation. In practice, it is seldom necessary to allow any value of the Rice coding parameter k above 3.

5 Experimental results

We compare FELICS with an implementation of the lossless mode of the JPEG proposed standard for image compression (using two-pixel prediction) and with the UNIX *compress* program. Our test files were 21 Landsat Thematic Mapper images and 7 other images widely used in compression studies. Three of the Landsat images (W6, D6, and R7) are highly compressible images with very little detail and few features. All runs were made on a Sun Sparcstation1GX.

	Compressed size				Encoding throughput			
	FELICS		JPEG	*compress*	FELICS		JPEG	*compress*
	Plain	Adjusted			Plain	Adjusted		
W1	2.10	2.10	2.07	1.70	78.5	87.7	16.3	93.6
W2	2.63	2.63	2.67	2.21	87.1	92.6	21.0	93.6
W3	2.31	2.31	2.28	1.92	82.2	90.1	17.8	77.1
W4	1.83	1.83	1.81	1.46	75.8	85.4	14.2	70.8
W5	1.70	1.70	1.68	1.34	73.6	83.5	13.2	70.8
W6	3.78	4.92	7.92	5.36	107.9	98.6	48.5	163.8
W7	2.11	2.12	2.10	1.70	79.0	87.7	16.5	63.9
D1	2.30	2.29	2.26	1.79	81.7	90.7	17.7	84.6
D2	2.84	2.84	3.07	2.36	92.6	93.0	23.6	100.8
D3	2.47	2.47	2.58	1.99	87.7	90.1	20.0	79.4
D4	1.86	1.86	1.85	1.34	76.9	84.6	14.6	81.9
D5	1.84	1.84	1.82	1.34	77.6	85.1	14.4	84.6
D6	3.84	5.32	9.25	6.14	107.0	98.2	53.5	154.2
D7	2.18	2.18	2.17	1.65	81.2	88.9	17.1	69.0
R1	2.34	2.34	2.28	1.79	80.9	89.2	17.9	101.3
R2	2.83	2.83	2.94	2.26	90.6	93.6	22.7	107.6
R3	2.47	2.47	2.45	1.86	86.1	90.6	19.4	95.7
R4	2.28	2.28	2.24	1.76	82.8	87.4	17.8	90.6
R5	1.84	1.84	1.78	1.34	76.2	80.9	14.0	71.8
R6	2.13	2.13	2.08	1.58	79.7	85.7	16.4	74.9
R7	3.82	5.01	7.43	5.50	105.0	97.9	46.5	143.5
couple	1.61	1.61	1.54	1.17	74.9	84.6	12.2	84.6
crowd	1.80	1.79	1.87	1.31	79.4	86.2	14.8	87.4
lax	1.35	1.35	1.31	1.04	68.4	79.9	10.4	61.0
lena	1.75	1.72	1.72	1.14	73.4	83.8	13.9	81.9
man	1.68	1.67	1.64	1.15	75.1	84.6	13.2	77.1
woman1	1.62	1.61	1.58	1.30	74.3	83.8	12.7	72.8
woman2	2.23	2.23	2.28	1.40	82.2	89.8	18.1	72.8

Table 2: Compression ratios and encoding throughput. The compression ratios are expressed as original size divided by compressed size. The encoding throughput is expressed as thousands of pixels encoded per second on a Sun SPARCstation1. Files W1–W7 are the Washington, D.C., Landsat Thematic Mapper images, files D1–D7 are the Donaldsonville, Louisiana, images, and files R1-R7 are the Ridgely, Maryland, images. All images are 512×512 pixels except the Ridgely images, which are 368×468.

In Table 2 we see that except for the three highly compressible images (which we shall henceforth ignore), FELICS compresses as well as JPEG lossless mode, with about five times the throughput. In fact, FELICS is about as fast as *compress*, and gives much better compression, not surprising since *compress* is designed for text, not images. The "plain" FELICS figures refer to a version with none of the refinements mentioned in Section 4; the "adjusted" FELICS figures include freezing when the smallest code length for a context reaches 1024, adjusting the range when $\Delta = 0$, and assigning small initial penalties to the cumulative counts for small parameter values. Note that the refinements seldom have much effect on compression, and they increase throughput by roughly 5 to 20 percent.

6 Conclusion

We have presented a very fast and very simple method for lossless image compression, called *FELICS*, based on prediction with a two-neighboring-pixel context and coding with single bits and prefix codes. Some of the prefix codes are Golomb-Rice codes, which are easy to implement and use and very fast in operation. Use of Golomb-Rice codes requires choosing a single integer parameter; we present a very general parameter estimation technique and prove bounds on its effectiveness. We have implemented the FELICS system, including some refinements that give even more speed, and we show experimentally that FELICS gives about the same compression as the JPEG lossless mode, while running about five times as fast.

FELICS is based on a raster-scan pixel sequence. We expect in the future to extend the FELICS system to a hierarchical pixel sequence as in [6,7] to allow progressive coding.

Acknowledgment. We wish to thank Allan R. Moser of E. I. duPont de Nemours and Company (Inc.) for his assistance in obtaining compression and timing data for JPEG lossless mode.

References

[1] P. Elias, "Universal Codeword Sets and Representations of Integers," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 194–203.

[2] N. Faller, "An Adaptive System for Data Compression," Record of the 7th Asilomar Conference on Circuits, Systems, and Computers, 1973.

[3] R. G. Gallager, "Variations on a Theme by Huffman," *IEEE Trans. Inform. Theory* IT–24 (Nov. 1978), 668–674.

[4] R. G. Gallager & D. C. Van Voorhis, "Optimal Source Codes for Geometrically Distributed Integer Alphabets," *IEEE Trans. Inform. Theory* IT–21 (Mar. 1975), 228–230.

[5] S. W. Golomb, “Run-Length Encodings,” *IEEE Trans. Inform. Theory* IT–12 (July 1966), 399–401.

[6] P. G. Howard & J. S. Vitter, “New Methods for Lossless Image Compression Using Arithmetic Coding,” *Information Processing and Management* 28 (1992), 765–779.

[7] P. G. Howard & J. S. Vitter, “Error Modeling for Hierarchical Lossless Image Compression,” in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 24-26, 1992, 269–278.

[8] P. G. Howard & J. S. Vitter, “Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding,” in *Proc. Data Compression Conference*, J. A. Storer & M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 98–107.

[9] D. E. Knuth, “Dynamic Huffman Coding,” *J. Algorithms* 6 (June 1985), 163–180.

[10] R. F. Rice, “Some Practical Universal Noiseless Coding Techniques,” Jet Propulsion Laboratory, JPL Publication 79–22, Pasadena, California, Mar. 1979.

[11] R. F. Rice, “Some Practical Universal Noiseless Coding Techniques—Part III, Module PSI14,K+,” Jet Propulsion Laboratory, JPL Publication 91–3, Pasadena, California, Nov. 1991.

[12] J. Rissanen, “A Universal Prior for Integers and Estimation by Minimum Description Length,” *Ann. Statist.* 11 (1983), 416–432.

[13] J. Rissanen, “Minimax Codes for Finite Alphabets,” *IEEE Trans. Inform. Theory* IT–24 (May 1978), 389–392.

[14] J. Venbrux, N. Liu, K. Liu, P. Vincent & R. Merrell, “A Very High Speed Lossless Compression/Decompression Chip Set,” Jet Propulsion Laboratory, JPL Publication 91–13, Pasadena, California, July 1991.

[15] J. S. Vitter, “Dynamic Huffman Coding,” *ACM Trans. Math. Software* 15 (June 1989), 158–167, also appears as Algorithm 673, Collected Algorithms of ACM, 1989.

[16] J. S. Vitter, “Design and Analysis of Dynamic Huffman Codes,” *Journal of the ACM* 34 (Oct. 1987), 825–845.

[17] P.-S. Yeh, R. F. Rice & W. Miller, “On the Optimality of Code Options for a Universal Noiseless Coder,” Jet Propulsion Laboratory, JPL Publication 91–2, Pasadena, California, Feb. 1991.