Punctured Elias Codes for variable-length coding of the integers

Peter Fenwick

Technical Report 137
ISSN 1173-3500
December 5, 1996

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
peter-f@cs.auckland.ac.nz

Abstract

The compact representation of integers is an important problem in areas such as data compression, especially where there is a nearly monotonic decrease in the likelihood of larger integers. While many different representations have been described, it is not always clear in which circumstances a particular code is to be preferred. This report introduces a variant of the Elias γ code which is shown to be better than other codes for some distributions.

1. Compact integer representations

The efficient representation of symbols of differing probabilities is one of the classical problems of information theory and coding theory, with efficient solutions known since the early 1950's (Shannon-Fano and Huffman codes[9]). In traditional, non-adaptive, coding we assume *a priori* probabilities of the input symbols and construct suitable codes to represent those symbols efficiently. There is no necessary or simple relation between a symbol and its representation.

Here we are concerned with a different problem, especially as the symbol alphabet (integers of arbitrary upper bound) may be so large as to preclude the formal construction of an efficient code. Given an arbitrary integer we wish to represent it as compactly as possibly, preferably by an algorithm which recognises only the magnitude and bit pattern of the integer (no table look-up or mapping needed). Equally, a simple algorithm should be able to recover an integer from an input bit stream, even if that particular integer has never been seen before. The binary representation of the integer is often visible within the representation and other information is appended to indicate the length or precision. Many variable-length representations have been described; here we concentrate on just a few, emphasising those which have a simple relation between code and value and are instantaneous or nearly so.

Following Elias[3], we first introduce two preliminary representations which are relatively unimportant *per se*, but are used in many other codes.

- $\alpha(n)$ is the unary representation, n 0's followed by a 1 (or 1's followed by a 0)
- $\beta(n)$, is the natural binary representation of n, from the most significant 1.

1.1 Levenstein and Elias γ Codes

These codes were first described by Levenstein[11], but the later description by Elias[3] is generally used in the English language literature. Elias describes a whole series of codes, with the α and β codes already described. His γ code writes the bits of the β code (the binary representation) in reverse order, with each preceded by a flag bit. All except the last flag bit are 0, with the last flag bit a 1 and implying the most-significant 1. Thus 13 is represented as $\underline{0}1\underline{0}0\underline{0}1\underline{1}$, with the flag bits underlined. The γ' code is permutation of the γ code, with the flag bits (an α code) preceding the data bits (a β code). With this code, 13 is written as 0001101 (the terminator of the α code doubles as the first bit of the β code). For most of this document the term "Elias γ code" will be used interchangeably for the two variants; often it will actually mean the γ' code.

An integer of N significant bits is represented in $2N+1$ bits, or an integer n is represented by $2\lceil \log n \rceil + 1$ bits[1] . (It is convenient to ignore the floor and ceiling operators in future discussions

[1] All logarithms will be to base 2, without stating an explicit base.

to simplify the mathematics. Most of the discussion involves only order of magnitude considerations, or averages over many symbols so that precise values are relatively unimportant. We therefore say that an Elias code represents an integer *n* in 2 log *n*+1 bits.)

n	code	*n*	code
1	1	11	0001011
2	010	12	0001100
3	011	13	0001101
4	00100	14	0001110
5	00101	15	0001111
6	00110	16	000010000
7	00111	17	000010001
8	0001000	18	000010010
9	0001001	19	000010011
10	0001010	20	000010100
100	0000001100100	250	000000011111010

Table 1. Example of Elias' γ' code.

The γ code can be extended to higher number bases where such granularity is appropriate. For example, numbers can be held in byte units, with each 8-bit byte containing 1 flag bit (last-byte/more-to-come) and 7 data bits, to give a base-128 code.

1.2 Elias ω and Even-Rodeh codes

All of the codes described here have a length part and a value part. In the γ code the length is given in unary; a natural progression is to specify the length itself in a variable-length code. Elias does this with his δ code, using a γ code for the length, but quickly proceeds to his ω codes. Some very similar codes were described by Even and Rodeh[4] and it is convenient to treat the two in parallel. Both of the codes have the value (as a β code) preceded by a series of length indications and followed by a 0 as a terminating comma.

Value	Elias ω code	Even-Rodeh code
0	—	000
1	0	001
2	10 0	010
3	11 0	011
4	10 100 0	100 0
7	10 111 0	111 0
8	11 1000 0	100 1000 0
15	11 1111 0	100 1111 0
16	10 100 10000 0	101 10000 0
32	10 101 100000 0	110 100000 0
100	10 110 1100100 0	111 1100100 0
1000	10 110 1100100 0	110 1100100 0

Table 2. Examples of Elias' ω and Even-Rodeh codes.

Some representative Elias ω codes are shown in Table 2, with the groups of bits separated by blanks. Each length is followed by the most-significant 1 of the next length or value; the final value is followed by a 0.

The codes are most easily described by giving the decoding process. For the Elias code, if a group is followed by a 0, its value is the value to be delivered. If a group is followed by a 1, its value is the number of bits to be read and placed *after* that following 1 to give the value of the next group. Thus 15 is read as the sequence {3, 15} and 16 as {2, 4, 16}.

The Even-Rodeh code is similar, but with the following differences

1. Each group gives the total number of bits in the following group, not the number of bits after the most significant 1.
2. A different starting procedure is used, with special considerations for the first 3 codes. The Elias code is special for only the first value.

Both codes are especially efficient just before a new length element is phased in and inefficient just after it is introduced, as for 15 and 16 in the Elias ω code. The codes alternate in relative efficiency as their extra length components phase in at different values.

Values	Elias	Even-Rodeh
1	1	3
2 – 3	3	3
4 – 7	6	4
8 – 15	7	8
16 – 31	11	9
32 – 63	12	10
64 – 127	13	11
128 – 255	14	17
256 – 512	21	18

Table 3. Lengths of Elias' ω and Even-Rodeh codes.

Bentley and Yao[2] develop a very similar code as a consequence of an optimal strategy for an unbounded search, recognising a correspondence between the tests of the search and the coding of index of the search target, but do not develop the code to the detail of either Elias or Even and Rodeh

1.3 Golomb and Rice codes

The Golomb codes[8] are designed for the coding of asymmetric binary events, where a probable event with probability p is interspersed with unlikely events of probability q ($q = 1-p$ and $p>>q$). The intention is to represent the sequence of events by encoding the lengths of the successive runs of the probable event. The Golomb codes have a parameter m, related to p by $p^m = 0.5$. A run of length $n+m$ is half as likely as a run of length n, indicating that the codeword for a run of length $n+m$ should be one bit longer than that for a run of length n.

If m is a power of 2, the codeword for n is a simple concatenation of $\alpha(^n/_m)$ as a prefix with the binary representation of n **mod** m to $\log m$ bits (ie $\alpha(^n/_m){:}\beta(n \textbf{ mod } m)$). For other values of m, let k be the smallest positive integer such that $2^k \geq 2m$. The dictionary contains exactly m codewords for every word length $\geq k$, plus $2^{k-1}-m$ words of length $k-1$; let $j = 2^{k-1}-m$. The first j codewords are represented in binary to $k-1$ bits and the next m codewords to k bits. Larger values of n are represented by a prefix of $\alpha\left(^{(n-p)}/_m\right)$, followed by $\beta\left((n-j) \textbf{ mod } m + 2j\right)$ to k bits. (The case power-of-2 code is a special case of the general rule.) The codeword values increment to preserve continuity across the boundaries between blocks and as the new block is formed by extending the low-order bits. The calculation of codeword bit patterns is most easily seen using Iverson's k-residue function[10], written in APL as $b|_j\ n$ (the j-residue of n modulo b). In C, the function is computed as `j+(n−j)%b`. With $j = 2^{k-1}-m$, The Golomb(m) code for n is given by the concatenation of the two codes $\alpha\left(^{(n-j)}/_m\right){:}\beta\left(^{(n-j)}/_m+2j\right)$.

m n	1	2	3	4	5
0	0	00	00	000	000
1	10	01	010	001	001
2	110	100	011	010	010
3	1110	101	100	011	0110
4	11110	1100	1010	1000	0111
5	111110	1101	1011	1001	1000
6	1111110	11100	1100	1010	1001
7	11111110	11101	11010	1011	1010
8	111111110	111100	11011	11000	10110
9	1111111110	111101	11100	11001	10111
10	11111111110	1111100	111010	11010	11000
11	111111111110	1111101	111011	11011	11001
12	1111111111110	11111100	111100	111000	11010
13	11111111111110	11111101	1111010	111001	110110
14	111111111111110	111111100	1111011	111010	110111
15	1111111111111110	111111101	1111100	111011	111000
16	11111111111111110	1111111100	11111010	1111000	111001
17	111111111111111110	1111111101	11111011	1111001	111010

Table 3. Golomb codes for the first few integers and parameter m.

Rice codes[12] have a parameter k. To encode the value n, we first form $m=2^k$ and then n **div** m and n **mod** m. The representation is the concatenation of (n **div** m) as a unary code and (n **mod** m) in binary. An integer n is represented by $^n/_{2^k} + k + 1$ bits, (in line with the approximations of the previous paragraph). A Rice(k) code is identical to the Golomb(2^k) code—the simplest case for coding. Representative Rice codes are shown in Table 4.

n \ k	1	2	3	4	5	6
0	0	000	0000	00000	000000	0000000
1	10	001	0001	00001	000001	0000001
2	110	010	0010	00010	000010	0000010
3	1110	011	0011	00011	000011	0000011
4	11110	1000	0100	00100	000100	0000100
5	111110	1001	0101	00101	000101	0000101
6	1111110	1010	0110	00110	000110	0000110
7	11111110	1011	0111	00111	000111	0000111
8	111111110	11000	10000	01000	001000	0001000
9	1111111110	11001	10001	01001	001001	0001001
10	11111111110	11010	10010	01010	001010	0001010
11	111111111110	11011	10011	01011	001011	0001011
12	1111111111110	111000	10100	01100	001100	0001100
13	11111111111110	111001	10101	01101	001101	0001101
14	111111111111110	111010	10110	01110	001110	0001110
15	1111111111111110	111011	10111	01111	001111	0001111

Table 4. Rice codes for the first few integers and parameter k.

The Golomb and Rice codes tend to be very efficient for moderate values, but large values are dominated by the long α code prefix, while small values are represented less efficiently than in the Elias γ codes.

1.4 Start-Step-Stop Codes

These codes[7] are defined by three parameters {i, j, k}. The representation may be less clearly related to the value than for Elias γ and Rice codes. The code defines a series of blocks of codewords (β code) of increasing length, the first block with a numeric part of i bits, the second with i+j bits, then i+2j bits and so on, up to a length of k bits. The groups of codewords have a unary prefix giving the group number. Thus a {3, 2, 9} code has codewords with numeric parts of 3, 5, 7 and 9 bits and prefixes of 0, 10, 110 and 111 (omitting the final 0 from the last prefix). It looks like

Codeword	Range
0*xxx*	0–7
10*xxxxx*	8–39
110*xxxxxxx*	40–167
111*xxxxxxxxx*	168–679

Table 5. Code values for a {3, 2, 9} *start-step-stop code.*

The start-step-codes can generate many of the other codes, or codes equivalent to them.

Parameters			generated code
k	1	k	a simple binary coding of the integers, and
0	1	∞	the Elias γ' code.
k	k	∞	the base 2^k Elias γ' code
k	0	∞	a code equivalent to the Rice(k) code

1.5 Ternary Comma codes

All codes so far have used binary coding. If we consider bit-pairs we can represent the values {0, 1, 2, comma}[5]. Table 6 shows the ternary comma code representation for the first few integers and some larger ones, with "c" representing the comma.

value	code	bits	value	code	bits
0	c	2	11	101c	8
1	0c	4	12	102c	8
2	1c	4	13	110c	8
3	2c	4	14	111c	8
4	10c	6	15	112c	8
5	11c	6	16	120c	8
6	12c	6	17	121c	8
7	20c	6	18	122c	8
8	21c	6	19	200c	8
9	22c	6	20	201c	8
64	2100c	10	1,000	1101000c	16
128	11201c	12	3,000	11010002c	18
256	100110c	14	10,000	111201100c	20
512	200221c	14	65,536	10022220020c	24

Table 6. Ternary codes for the various integers.

It will be seen later that the ternary code is one of the better ones for large values. It is also quite simple to encode and decode. The comma principle can be extended to larger number bases, but becomes increasingly inefficient for small values because the comma consumes a large amount of code space but conveys only 1 bit of information. The higher radix Elias γ codes would seem preferable.

2. The new "punctured" code

We start with the simplest of a family of new codes, called here P1. It is derived from the Elias γ codes, but with some major differences. Like those codes, it has two variants. In the γ code variant the data bits are written in reverse order with each 1 bit followed by a 0 for an "internal" 1 and a 1 for the most significant 1. Zeros are written "as is", with no following bit. The γ' variant has the data part written in reverse order (most-significant bit last) preceded by an α code to indicate the number of 1 bits. It is not possible to merge the last bit of the prefix with the numeric bit as is possible with the γ' code.

The name “punctured code” is chosen by analogy with error correcting codes. A systematic ECC codeword resembles an Elias γ' code in having a clearly identifiable natural representation of its data, with added check bits to provide the error correction facility. A punctured ECC has some of the check bits deleted to provide a shorter codeword, much as some of the unary length bits of the Elias code are removed do provide the new code. The code described (especially that corresponding to the γ' code) will be described as the P1 code.

For all cases except for a value of 0, the P1 codes start and stop with a 1 bit. If the represented value is biased by 1, encoding not *n* but (*n*+1), and the doubled bit replaced by a single bit, we obtain a variant of the punctured code, the “P2 code”.

Table 7 shows the representations for the first few integers, together with the Elias code for the same value. The digit which marks the end of the prefix is shown in boldface; for the Elias code it is also the most significant 1. As the Elias code has no representation for 0, it must often use a “biased” version as shown in the table. Finally, the last column shows the advantage in bits in using the new P2 code as compared with the biased Elias code.

Value	P1	P2	Elias	biased	advantage (bits)P2	
	Representation	Rep		Elias	P1	P2
0	**0**	**0**1	—	**1**	0	-1
1	1**0**1	**0**01	**1**	0**1**0	0	0
2	1**0**01	1**0**11	0**1**0	0**1**1	-1	-1
3	11**0**11	**0**001	0**1**1	00**1**00	0	1
4	1**0**001	1**0**101	00**1**00	00**1**01	0	0
5	11**0**101	1**0**011	00**1**01	00**1**10	-1	0
6	11**0**011	11**0**111	00**1**10	00**1**11	-1	-1
7	111**0**111	**0**0001	00**1**11	000**1**000	0	2
8	1**0**0001	1**0**1001	000**1**000	000**1**001	1	1
9	11**0**1001	1**0**0101	000**1**001	000**1**010	0	1
10	11**0**0101	11**0**1101	000**1**010	000**1**011	0	0
11	111**0**1101	1**0**0011	000**1**011	000**1**100	-1	1
12	11**0**0011	11**0**1011	000**1**100	000**1**101	0	0
13	111**0**1011	11**0**0111	000**1**101	000**1**110	-1	0
14	111**0**0111	111**0**1111	000**1**110	000**1**111	-1	–1
15	1111**0**1111	**0**00001	000**1**111	0000**1**0000	0	3
16	1**0**00001	1010001	0000**1**0000	0000**1**0001	2	2
…						
31	11111**0**11111	**0**000001	0000**1**1111	00000**1**00000	0	4
32	1**0**000001	1**0**100001	00000**1**00000	00000**1**00001	3	3
33	11**0**100001	1**0**010001	00000**1**00001	00000**1**00010	2	3

Table 7 Comparison of Punctured and Elias codes

For small values the punctured codes are often 1 bit longer than the biased Elias, but for large integers they average about 1.5 log *N* bits, in comparison with the 2 log *N* bits of the Elias codes.

3. Comparison of representations

There is no “best” variable-length coding of the integers, the choice depending on the probability distribution of the integers to be represented. A fundamental result of coding theory is that a symbol with probability P should be represented by $\log(^1/_P)$ bits. Equivalently a symbol represented by j bits should occur with a probability of 2^{-j}. The two simplest functions which produce a low value for large arguments n are a power function (n^{-x}) and an exponential function (x^{-n}). In both cases normalising factors are needed to provide true probabilities, but these do not affect the basic argument.

An Elias code represents a value n with about 2 log n+1 bits. The probability P_n of an integer n should therefore be $P_n \approx n^{-2}$, showing that an Elias code is well-suited to power law distributions and especially where the exponent is about –2.

A Rice code represents a value n with about $(^n/_{2^k} + k + 1)$ bits, so that $\log Pn \approx {}^n/_{2^k} + k + 1$. For a given k, we have that $Pn \propto 2^{-n}$, showing that the Rice code is suited to symbols following an exponential distribution.

A punctured code is similar to an Elias code but with a somewhat unpredictable length. If all the values with n significant bits are equiprobable, it represents those values with about 1.5 log n bits on the average. A skewed distribution will emphasise smaller values which tend to have fewer one bits and should reduce the average code length below that indicated.

Value	Binary	Elias	Elias	Golomb			Rice			Ternary	Punctured	
		γ	ω	2	3	4	2	3	4	std	P1	P2
1	1	1	1	2	3	3	3	4	5	4	1	2
2	2	3	3	2	3	3	3	4	5	4	3	3
3	2	3	3	3	3	3	3	4	5	6	4	4
4	3	5	6	4	4	4	4	4	5	6	5	4
5	3	5	6	4	4	4	4	4	5	6	5	5
10	4	7	7	7	6	5	5	5	5	8	7	6
20	5	9	11	12	9	8	8	6	6	8	8	7
50	6	11	12	27	19	15	15	10	8	10	10	9
100	7	13	13	52	36	28	28	16	11	12	12	11
200	8	15	14	102	69	53	53	29	17	12	13	12
500	9	17	16	252	169	128	128	66	36	14	17	16
1,000	10	19	17			253	253	129	67	16	18	17
2,000	11	21	18				503	254	130	16	20	19
5,000	13	25	19					629	317	18	20	19
10,000	14	27	20						630	20	22	21
20,000	15	29	22							22	23	22
50,000	16	31	27							22	25	24
100,000	17	33	28							24	26	25
200,000	18	35	29							26	28	27

Table 8. Comparison of various representations – codeword lengths.

Of particular importance to efficient coding is the codeword length for the most probable symbols. A power-law distribution is dominated by just a few symbols and particularly by the first. It is very skew and is best represented by a code such as the Elias code which is initially very short. An exponential distribution however has its smaller values nearly equiprobable and is best handled by a Rice code.

Table 8 compares codeword lengths for most of the codes for a range of values. Most of the results are exact, with Golomb and Rice lengths omitted for extreme cases. Results for the new punctured codes have been derived differently to allow for the irregular variation in length over quite small ranges of values. The P1 and P2 have exact lengths for values of 10 or less. All of the larger values are multiples of powers of 2, with fewer 1 bits and shorter codewords than most values in their neighbourhood. The exact length is therefore augmented by about half the number of least-significant zeros to approximate the average behaviour around the test value.

Important points from this table are

1. It is relatively easy to obtain good results for large values. The Elias ω, ternary and punctured codes show little difference.
2. It is much harder to obtain a code which is good for small values. In many situations the small values dominate and inefficiencies in their coding can dominate the performance. For example, the ternary code, which is very good at large values, is quite poor for small values.
3. It is difficult to have good performance at both large and small values. Of the traditional codes, the γ and ω codes are good at small values, and the ternary and ω at large values. The ω code though tends to be less efficient at intermediate values where new prefix elements are introduced.

4. The present requirement

The need for variable length codes arose from work with the “block sorting” text compressor of Burrows and Wheeler[1]. In that compressor the input file is first permuted and the permuted text then processed by a Move-To-Front transformation or recoding. The symbols from the MTF operation are then processed by a final statistical compressor. Burrows and Wheeler used a Huffman coder, while other work has used arithmetic coders or a complex of arithmetic coders for slightly better compression[6]. The question arises as to whether Elias or Rice codes might be suitable in the final encoding stage.The symbol frequencies (from the MTF output) are observed to follow an inverse power law, with the exponent usually about –2, indicating that an Elias code is probably preferable to a Rice code.

Most of the studies have been done apart from any actual compressor. A block sorting (or Burrows Wheeler Transform) compressor was modified to accumulate statistics of the values produced by the MTF operation for the files of the Calgary compression corpus, giving 256 code frequencies for each of the 14 files. These are then combined with the known representation length for the values with each code to predict the effect of encoding each file with each of the test coders. (The files have been preprocessed by a stage of run-encoding, which has minor effects on most files but reduces PIC to about 20% of its original size, with a corresponding increase in its stated entropy.) The results are shown in Table 9.

	Entropy	Elias γ	Rice-2	Rice-3	Rice-4	Punct	Punct-V2	SSS1.2.9
bib	2.30	2.44	2.50	2.53	2.72	2.50	3.01	3.00
book1	2.76	2.86	2.94	3.15	3.54	2.98	3.28	3.28
book2	2.40	2.49	2.60	2.72	3.00	2.58	3.03	3.01
geo	5.40	6.40	11.84	7.91	6.29	6.08	5.95	6.04
news	2.86	2.96	3.14	3.07	3.27	3.02	3.38	3.40
obj1	4.76	5.40	9.41	6.51	5.37	5.20	5.21	5.25
obj2	2.78	3.06	4.46	3.47	3.16	3.00	3.48	3.50
paper1	2.70	2.78	2.92	2.94	3.17	2.86	3.25	3.25
paper2	2.70	2.79	2.88	2.98	3.29	2.89	3.24	3.25
pic	3.58	3.79	4.11	3.72	3.80	3.81	3.99	4.05
progc	2.70	2.81	3.04	2.94	3.10	2.85	3.27	3.27
progl	1.98	2.17	2.28	2.29	2.45	2.22	2.82	2.81
progp	1.97	2.18	2.35	2.30	2.42	2.21	2.83	2.82
trans	1.68	1.98	2.10	2.05	2.14	2.00	2.69	2.68
Averages -	**2.90**	**3.15**	**4.04**	**3.47**	**3.41**	**3.16**	**3.53**	**3.54**

Table 9. Coding the Calgary Corpus files

There are several points to note

1. The Rice codes are definitely not optimal for this application
2. The Elias (gamma) code is generally well matched to the symbol distribution, with overall compression to within 10% of the entropy.
3. The newer, punctured, codes are best for the binary files (geo, obj1, obj2)

To see the reason, consider the two files Paper1 (text) and geo (binary), whose symbol distributions are shown in Figure 1.

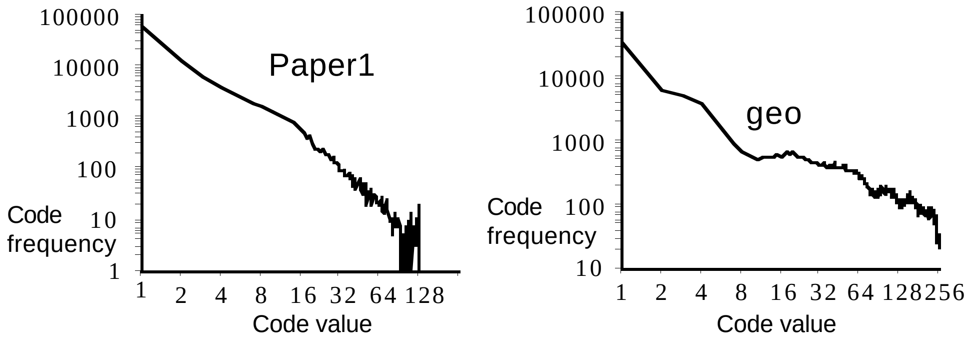

Figure 1 Code frequencies for PAPER1 and GEO

PAPER1 has a generally smooth variation in code frequency against code value, closely following *freq≈val*$^{-2}$, while GEO tends to much higher frequencies for larger values. More to the point, we can show for these files the number of bits to represent the codes according to their frequencies and compare that with the corresponding codeword lengths for each of the codes. These comparisons are shown in Figure 2. In this Figure the code lengths are all smoothed. Their true values are discontinuous and the multiplicity of stepped functions makes the graph hard to understand. The graphs are obtained by joining the length for value=1, the midpoints of the step "risers" for small values, and selected midpoints at large values (especially for the Rice code)

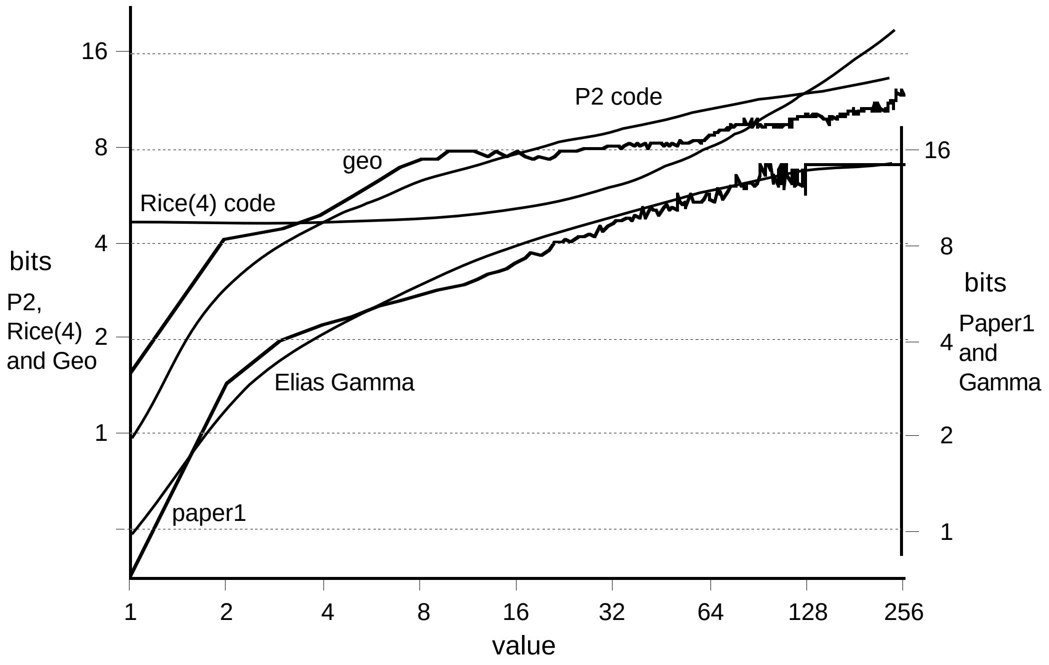

Figure 2. Comparison of Elias, Rice and Punctured codes for encoding Corpus files

Important points from Figure 2 are –

1. The Elias gamma code is a good match to the PAPER1 codes (both approximate an inverse square function)
2. The P2 code (modified punctured) is a good fit to the GEO data
3. The Rice(4) code is the best of all the codes for mid-range values (5 < value< 120), but is longer for both small values and for large values. The greater length for small values is especially important because of their very high frequency; inefficiencies in coding these values dominate the file encoding.

5. Conclusions

The Elias γ code is confirmed as a good overall code for compressing the data from the block sorting compressor. However better compression is achieved with P1 or P2 codes for binary files. The two can be combined by encoding all files with a γ code initially, but switching to a punctured code as soon as a symbol is found greater than 127. With this change the average compression should improve from 3.15 to about 3.10 bit/byte. In files where binary and text occur in alternating blocks there may be an advantage in switching back to the γ code if no non-text value has been seen for perhaps 8 symbols.

References

1. M. Burrows and D.J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm", SRC Research Report 124, Digital Systems Research Center, Palo Alto, May 1994
   `gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z`
2. J.L. Bentley, A.C. Yao, "An almost optimal solution for unbounded searching", *Info. Proc. Letters*, Vol 5, No 3, pp 82–87 Aug 1976
3. P. Elias, "Universal Codeword Sets and Representations of the Integers", *IEEE Trans. Info. Theory*, Vol IT 21, No 2, pp 194–203, Mar 1975
4. S. Even, M. Rodeh, "Economical Encoding of Commas Between Strings", *Comm ACM*, Vol 21, No 4, pp 315–317, April 1978.
5. P.M. Fenwick, "Ziv-Lempel encoding with multi-bit flags", *Proc. Data Compression Conference, DCC-93*, Snowbird, Utah, pp 138–147, Mar 1993
6. P.M. Fenwick, "Block sorting text compression", *Australasian Computer Science Conference, ACSC'96*, Melbourne, Australia, Feb 1996.
   `ftp.cs.auckland.ac.nz /out/peter-f/ACSC96.ps`
7. E.R. Fiala, D.H. Greene, "Data Compression with Finite Windows", *Comm ACM*, Vol 32, No 4, pp 490–505 , April 1989
8. S.W. Golomb, "Run-Length Encodings", *IEEE Trans Info. Theory*, Vol 12 pp 399–401 1966.
9. D.A. Huffman, "A method for the construction of minimum-redundancy codes", *Proc IRE*, Vol 40, pp 1098–1101, 1952
10. K.E. Iverson, "A Programming Language" Wiley, 1962
11. V.E. Levenstein, "On the redundancy and delay of separable codes for the natural numbers", *Problems of Cybernetics*, Vol 20, pp 173–179, 1968.
12. R.F. Rice, "Some Practical Universal Noiseless Coding Techniques", Jet Propulsion Laboratory, JPL Publication 79-22, Pasadena California Mar 1979

Appendix. Some new codes

Some new codes were discovered during the work for this report. They were really extra to the proposed intent of the report and it has been decided to leave their full investigation for a student project; for now they are just presented.

A.1 Modified Ternary Code

The ternary code suffers at small values from the added two comma bits. A modification allows it to handle these values more efficiently, at the cost of lower efficiency for large values. The first two bits are used to encode values and control the interpretation as

bits	Meaning
00	value 0
01	value 1
10	value 2, 3, 4, 5 in next digit 0, 1, 2, 3
11	use comma-delimited string

The modified and simple ternary codes are shown in Table A.1. The modified code is often better than the standard ternary code for values up to and including 14 (and is never worse), but is always 2 bits longer for larger values.

value	Ternary		modified		value	Ternary		modified	
	code	bits	code	bits		code	bits	code	bits
0	c	2	0	2	11	101c	8	312c	8
1	0c	4	1	2	12	102c	8	320c	8
2	1c	4	20	4	13	110c	8	321c	8
3	2c	4	21	4	14	111c	8	322c	8
4	10c	6	22	4	15	112c	8	3100c	10
5	11c	6	23	4	16	120c	8	3101c	10
6	12c	6	30c	6	17	121c	8	3102c	10
7	20c	6	31c	6	18	122c	8	3110c	10
8	21c	6	32c	6	19	200c	8	3111c	10
9	22c	6	310c	8	20	201c	8	3112c	10
64	2100c	10	32011c	12	1,000	1101000c	16	31100211c	18
128	11201c	12	311112c	14	3,000	11010002c	18	311002220c	20
256	100110c	14	3100021c	16	10,000	111201100c	20	3111201011c	22
512	200221c	14	3200202c	16	65,536	10022220020c	24	310022220001c	

Table A.1. Ternary codes, simple and modified, for various integers.

A.2 Variable radix γ codes

The Elias γ codes are excellent for representing small values, but larger values have representations which are dominated by the leading α code. The Elias ω, Even-Rodeh and P1 codes all attempt to reduce the effect of the prefix. This section introduces another approach. It was mentioned in Sections 1.1 and 1.4 that the γ code could be extended to higher radices. In general these high radix γ codes are very efficient indeed at representing large values, but are quite inefficient for small values. The suggestion is that the radix should vary across the number, starting at 2 (the standard γ code) and increasing as more digits are known.

To illustrate a possible code, we consider the value 1234_{10} = $100\ 1101\ 0010_2$ or, bit-reversed, 0100 1011 001. The bits are collected in groups of successively 1, 2, 3, … bits and each preceded by a leading more/last flag, giving the representation $\underline{0}0$ $\underline{0}10$ $\underline{0}010$ $\underline{0}1100$ $\underline{1}10000$. (The last group of bits must be padded out to the appropriate length.) An alternative grouping might be $\underline{0}0$ $\underline{0}1$ $\underline{0}00$ $\underline{0}101$ $\underline{0}100$ $\underline{1}100$, using groups of 1, 1, 2, 2, 3, 3, …. The decoding process starts reading very few bits at a time, but progressively more and more as the value grows.

The variable radix γ code can be described by a generalisation of the start-step-stop codes, where the step is a function of the internal block size, block number or bits read. For full generality we can replace the original code $\{i, j, k\}$ specification with $\{i, \mathrm{F}(\lambda), k : \lambda\}$, where λ is in the nature of a **var** parameter which returns the bits read. The number of codes is limited only by ingenuity in devising the function $\mathrm{F}(\lambda)$. A particularly simple function $\mathrm{F}(\lambda) = \lceil {}^{\lambda}/_{d} \rceil$ looks useful.