ISSN 2079-3316 **PROGRAM SYSTEMS: THEORY AND APPLICATIONS** **vol. 13**, No 2(53), pp. 35–63
Research Article HARDWARE, SOFTWARE AND DISTRIBUTED SUPERCOMPUTER SYSTEMS

UDC 004.627+004.272.45
10.25209/2079-3316-2022-13-2-35-63;

Memory-efficient sensor data compression

Yury Vladimirovich **Shevchuk**✉
Ailamazyan Program Systems Institute of RAS, Ves'kovo, Russia,
sizif@botik.ru✉ (*learn more about the author on p. 62*)

Abstract. We treat scalar data compression in sensor network nodes in streaming mode (compressing data points as they arrive, no pre-compression buffering). Several experimental algorithms based on linear predictive coding (LPC) combined with run length encoding (RLE) are considered. In entropy coding stage we evaluated (a) variable-length coding with dynamic prefixes generated with MTF-transform, (b) adaptive width binary coding, and (c) adaptive Golomb-Rice coding. We provide a comparison of known and experimental compression algorithms on 75 sensor data sources. Compression ratios achieved in the tests are about 1.5/4/1000000 (min/med/max), with compression context size about 10 bytes.

Key words and phrases: LPC, linear predictive coding, DTN, delay tolerant network, Laplace distribution, adaptive compression, bookstack, MTF transform, RLE, RLGR, prefix code, Elias Gamma coding, Golomb-Rice coding, vbinary coding

2020 *Mathematics Subject Classification*: 68P30; 94A45

Acknowledgments:
The research is a part of the Russian Academy of Sciences research project AAAA-A19-119020690043-9.

For citation: Shevchuk Y. V. *Memory-efficient sensor data compression* // Program Systems: Theory and Applications, 2022, **13**:2(53), pp. 35–63. http://psta.psiras.ru/read/psta2022_2_35-63.pdf

Introduction

While working on a project in the field of delay-tolerant networks (DTN [1]) we faced the task of compressing data on a sensor node with relatively large number of sensors ($n \sim 10 \div 100$). We need compression to increase the volume of data that fits in memory of the sensor node, thereby increasing the maximum time gap between communication sessions we can survive without data loss.

Эта статья по-русски: *http://psta.psiras.ru/read/psta2022_2_3-33.pdf*

TABLE 1. Sensor node RAM volumes

Year	Processor	RAM
2006	MSP430	10KB
2020	NRF52	256KB

TABLE 2. Input block sizes for selected compression algorithms

Algorithm	Input block (bytes)
`S-LZW` [2]	528
`FELACS` [4]	$16 \div 300$
`FLAC` [6]	$2048 \div 6144$
`Shorten` [7]	256

Data compression is popular in sensors networks as it reduces the volume of data transferred, and thus conserves (a) the battery resource, and (b) the network bandwidth. A number of compression algorithms with low memory (Table 1) and CPU requirements, targeted specifically toward sensor networks have been developed, e.g. [2–5]. The field appears well-established, but we could not find an algorithm fitting the task.

Most algorithms compress data blocks of fixed size (Table 2), the output block size varies with compression ratio actually achieved. What we need is an algorithm that receives a slow stream of data points and compresses into an *output buffer of fixed size*. The size may be chosen to be transferable atomically in one communication session, or to be convenient for memory management (fixed size buffer pool). We would want an algorithm with a small memory footprint, as it will be multiplied by the number of sensors n. We would also want good compression ratios for "runs" of $v = const$ which are often present in real world sensor data.

Sensor data are time series:

$$\{(t_0, v_0), (t_1, v_1), \dots\}.$$

Authors dealing with sensor data compression at database level additionally consider the issue of timestamp compression t_i [8, 9]. We avoid this issue by assuming the data are sampled in accordance with a schedule: $t_i = \mathbf{schedule}(t_0, i)$, with no points lost. In this case it is sufficient to send t_0 along with every compressed block.

1. Methodology

We will consider compression algorithms with low CPU usage and a small memory footprint, both existing and developed *ad hoc*. For every algorithm we implement a coder prototype to measure the compression ratio, and test the coders on real world sensor data. Test data come from our own sensor network (Table 3), and also from "ground truth" watt-meter

TABLE 3. Description of own datasets

Name	Bits	Frequency	Description
Pa	18	1/10Hz	electricity consumption data of a cottage: mean active power obtained by differentiating data from an energy counter based on *Analog Devices ADE7758* chip. Phase A
Pb	18	1/10Hz	ditto, Phase B
Pc	18	1/10Hz	ditto, Phase C
T1	12	1/60Hz	air temperature, room 1 (sensor *Sensirion SHTW2*)
T2	12	1/60Hz	air temperature, room 2
Tdp1	12	1/60Hz	air dew point in room 1
Tdp2	12	1/60Hz	air dew point in room 2

data of EMBED project [10]. For each source of our own data we consider three three-day periods to see how data distributions change in time.

All test results are shown in Table 4 and we refer to them while discussing algorithms. The algorithms are described informally, but encoder implementations in Perl5 are available under *https://github.com/yysizif/sencomp*.

Data in test datasets are real numbers, while the algorithms are intended for compressing integers (ADC samples). We convert test data to integer form by multiplying by 10^k, where k is the length of the fractional part in every particular dataset.

1.1. Compression ratio calculation

We define the compression ratio as $K = S_0/S_c$, where S_0 is uncompressed size in bits and S_c is compressed size in bits, i.e. larger K means better compression.

To compute the uncompressed size in bits, we need to know the ADC resolution. For sources T, T_{dp}, the ADC resolution is known to be 12 bits. For sources $P_{A/B/C}$ and EMBED project data, we don't know the ADC resolution and have calculated it based on measurement range and precision

$$m = \lceil \log_2(I_{\max} V_{\text{nom}} 10^k) \rceil,$$

where V_{nom} and $I_{\max}$ are nominal mains voltage and maximum measurable current ($I_{\max} = 100$A for $P_{A/B/C}$, $I_{\max} = 15$A for EMBED data), k is the length of fractional part ($k = 1$ in $P_{A/B/C}$ data, $k = 2$ in EMBED data). In both cases we obtained projected ADC resolution of $m = 18$ bits.

TABLE 4. Compression ratios for 75 data sources with 16 algorithms

Cell background: *hors concours* | best ratio in the row | -5% | -10% | -20% | -50% | -80% | -90%

№	Source	z22	z1	EB	F16	F256	RLGR	EG	EW	e1	e2	e3	e4	e5	DD	RLGR′	e10
1	period 1, Pa	5.21	3.12	2.43	1.16	1.18	1.49	1.14	1.23	1.47	1.62	1.61	1.59	1.51	1.48	1.56	1.61
2	period 1, Pa-noise	4.90	3.29	2.53	1.16	1.18	1.55	1.18	1.28	1.59	1.70	1.70	1.64	1.57	1.61	1.57	1.64
3	period 1, Pb	4.99	3.36	4.02	1.93	1.89	0.31	1.81	1.74	2.11	2.25	2.21	2.19	2.17	2.13	1.76	2.28
4	period 1, Pb-noise	5.96	4.24	4.95	1.96	1.90	0.67	2.20	2.15	2.72	2.78	2.77	2.70	2.74	2.68	2.19	2.78
5	period 1, Pc	5.43	3.67	4.85	2.39	2.52	1.92	2.13	1.94	2.58	2.58	2.57	2.60	2.54	2.48	2.39	2.71
6	period 1, Pc-noise	7.68	4.85	6.86	2.45	2.56	2.69	2.91	2.65	3.55	3.39	3.39	3.55	3.60	3.38	3.08	3.65
7	period 1, Tdp1	3.85	3.63	4.30	2.05	2.20	2.30	1.81	1.64	2.14	2.02	2.02	2.08	2.03	2.04	2.25	2.28
8	period 1, Tdp1-noise	6.65	5.03	9.52	2.57	2.57	5.00	3.99	3.57	4.78	4.46	4.46	4.78	4.86	4.54	4.90	5.20
9	period 1, T1	4.52	3.87	5.05	2.23	2.44	2.50	2.06	1.86	2.31	2.18	2.18	2.26	2.23	2.23	2.45	2.51
10	period 1, T1-noise	7.04	5.27	12.71	2.99	3.34	6.59	5.30	4.69	6.33	5.85	5.85	6.34	6.50	6.01	6.48	6.84
11	period 1, Tdp2	4.17	3.58	4.13	2.00	2.17	2.24	1.75	1.59	2.08	1.98	1.98	2.05	2.01	1.99	2.20	2.22
12	period 1, Tdp2-noise	6.34	4.89	7.71	2.38	2.51	4.06	3.26	2.94	4.02	3.73	3.73	3.97	4.04	3.82	4.00	4.23
13	period 1, T2	4.40	3.77	4.21	2.03	2.16	2.26	1.78	1.62	2.07	2.00	2.00	2.06	2.02	2.02	2.21	2.26
14	period 1, T2-noise	6.62	4.89	7.69	2.50	2.70	4.05	3.28	2.98	4.12	3.85	3.85	4.07	4.10	3.91	3.99	4.24
15	period 2, Pa	4.64	3.06	2.35	1.25	1.28	1.56	1.11	1.20	1.44	1.59	1.58	1.56	1.47	1.47	1.55	1.60
16	period 2, Pa-noise	4.25	3.17	2.42	1.25	1.28	1.55	1.13	1.23	1.52	1.64	1.64	1.58	1.51	1.55	1.55	1.61
17	period 2, Pb	4.95	3.33	3.82	1.89	1.90	0.39	1.73	1.68	2.07	2.19	2.16	2.15	2.12	2.08	1.91	2.23
18	period 2, Pb-noise	5.46	4.10	4.61	1.91	1.90	0.71	2.06	2.03	2.58	2.65	2.63	2.59	2.61	2.54	2.25	2.66
19	period 2, Pc	4.55	3.42	3.93	1.94	1.97	0.12	1.77	1.67	2.22	2.26	2.25	2.27	2.20	2.16	1.77	2.32
20	period 2, Pc-noise	5.31	3.98	4.71	1.96	1.98	0.24	2.07	1.98	2.68	2.61	2.60	2.64	2.63	2.55	1.87	2.61
21	period 2, Tdp1	4.58	3.72	4.24	2.04	2.23	2.29	1.79	1.60	2.17	2.02	2.02	2.08	2.05	2.02	2.25	2.26
22	period 2, Tdp1-noise	6.20	5.32	11.87	2.63	2.71	6.18	4.89	4.23	5.51	5.12	5.12	5.68	5.78	5.25	6.04	6.30
23	period 2, T1	4.40	3.64	3.74	1.90	2.02	2.12	1.61	1.43	2.01	1.89	1.89	1.94	1.91	1.88	2.08	2.11
24	period 2, T1-noise	6.40	4.24	7.23	2.11	2.23	3.77	2.98	2.62	3.52	3.26	3.26	3.58	3.67	3.34	3.71	3.87
25	period 2, Tdp2	4.61	4.04	6.53	2.53	2.84	2.91	2.52	2.32	2.65	2.42	2.42	2.61	2.58	2.57	2.83	2.91
26	period 2, Tdp2-noise	7.89	6.46	52.54	4.77	6.84	32.02	22.14	19.27	22.01	21.03	21.03	23.34	23.29	21.52	31.61	27.84
27	period 2, T2	5.24	4.20	6.21	2.49	2.81	2.86	2.43	2.23	2.62	2.38	2.38	2.56	2.52	2.52	2.78	2.84
28	period 2, T2-noise	8.87	5.82	92.51	5.42	8.02	48.58	39.23	34.38	40.60	38.58	38.58	43.21	43.10	39.47	47.45	48.81
29	period 3, Pa	4.46	3.07	2.30	1.07	1.08	1.55	1.09	1.19	1.43	1.57	1.56	1.53	1.45	1.46	1.55	1.58
30	period 3, Pa-noise	4.56	3.17	2.36	1.07	1.08	1.54	1.11	1.22	1.49	1.62	1.61	1.56	1.48	1.53	1.55	1.59
31	period 3, Pb	4.92	3.39	4.23	1.92	1.88	0.34	1.89	1.81	2.15	2.30	2.27	2.24	2.24	2.17	1.89	2.31
32	period 3, Pb-noise	6.08	4.43	5.51	1.96	1.89	0.62	2.43	2.36	2.91	3.02	3.00	2.92	3.00	2.86	2.46	2.98
33	period 3, Pc	4.44	3.53	4.32	2.14	2.22	0.23	1.93	1.79	2.37	2.40	2.39	2.42	2.35	2.29	1.95	2.49
34	period 3, Pc-noise	5.77	4.31	5.53	2.16	2.23	0.34	2.39	2.23	3.01	2.90	2.90	2.99	3.00	2.86	2.22	2.98
35	period 3, Tdp1	4.52	3.93	6.32	2.51	2.81	2.88	2.46	2.26	2.61	2.39	2.39	2.57	2.54	2.54	2.81	2.87
36	period 3, Tdp1-noise	7.72	6.52	55.66	4.77	6.85	33.80	23.51	20.50	23.65	22.58	22.58	25.31	25.28	23.08	33.26	29.43
37	period 3, T1	5.24	4.31	6.56	2.56	2.91	2.96	2.53	2.32	2.65	2.42	2.42	2.62	2.60	2.61	2.88	2.95
38	period 3, T1-noise	8.88	6.20	95.44	5.31	8.27	52.45	40.72	36.17	42.35	40.18	40.18	44.80	44.69	41.09	51.23	50.34
39	period 3, Tdp2	4.24	3.89	6.62	2.56	2.85	2.95	2.55	2.34	2.65	2.43	2.43	2.62	2.59	2.61	2.87	2.96
40	period 3, Tdp2-noise	7.67	6.64	52.87	4.74	6.67	32.65	22.32	19.46	22.22	21.19	21.19	23.51	23.51	21.73	32.17	27.73
41	period 3, T2	5.08	4.38	7.08	2.65	2.95	3.06	2.68	2.46	2.72	2.49	2.49	2.71	2.69	2.70	2.96	3.08
42	period 3, T2-noise	8.83	6.15	117.0	5.60	8.97	67.36	50.62	45.29	52.31	49.86	49.86	55.17	55.00	51.01	65.30	61.75

Continued on the next page

TABLE 4. *Continued from previous page*

№	Source	z22	z1	EB	F16	F256	RLGR	EG	EW	e1	e2	e3	e4	e5	DD	RLGR′	e10
43	1/Hair Dryer	11.47	7.03	18.60	5.02	7.01	0.01	6.22	5.85	5.11	4.92	4.92	5.81	5.80	6.05	6.73	7.27
44	1/Iron	2000	1428	3407	7.81	15.68	1.00	1468	1548	1492	1642	1618	1645	1663	1243	590.0	1165
45	1/Kettle	625.0	526.0	838.0	7.72	15.06	0.20	391.1	417.1	513.0	533.0	528.0	536.0	519.0	493.1	263.1	495.1
46	1/Laptop	3.61	3.04	3.25	1.92	2.03	1.65	1.49	1.49	1.92	2.02	2.01	2.02	1.95	1.97	2.02	2.13
47	1/Modem	8.64	5.15	12.25	3.92	4.97	4.88	4.46	4.13	4.25	3.95	3.95	4.32	4.34	4.37	4.71	5.00
48	1/Monitor	8.85	5.72	12.60	4.24	5.35	3.47	4.68	4.32	4.33	4.24	4.24	4.68	4.66	4.78	5.19	5.66
49	1/NILM LCD	6.05	4.26	6.12	2.72	3.10	3.14	2.60	2.33	3.05	2.96	2.96	2.90	2.87	2.86	3.09	3.18
50	1/Refrigerator	4.63	3.67	4.22	2.26	2.41	0.001	1.90	1.78	2.49	2.46	2.46	2.50	2.44	2.38	2.55	2.65
51	1/Router	5.73	3.92	4.84	2.57	2.91	3.01	2.13	1.83	2.85	2.63	2.63	2.85	2.81	2.69	2.96	2.96
52	1/TV	38.17	28.82	52.36	6.69	11.02	2.58	21.04	18.83	22.50	21.13	21.10	23.15	22.90	22.35	22.10	24.96
53	1/Toaster	909.0	769.0	1305	7.80	15.22	0.90	612.0	648.0	807.0	858.0	850.0	861.0	832.0	806.0	351.1	800.0
54	1/Washing Machine	17.09	10.42	25.77	5.49	8.35	0.42	7.97	7.77	5.83	5.78	5.78	7.16	7.14	7.69	8.64	9.56
55	2/AC	9.07	7.40	10.46	3.54	4.15	0.17	4.79	4.76	6.25	6.39	6.32	6.42	6.15	6.14	5.59	6.58
56	2/Cablebox	7.14	4.78	7.64	3.06	3.56	3.61	3.11	2.79	3.52	3.28	3.28	3.30	3.27	3.27	3.54	3.64
57	2/Coffeemaker	9.78	6.02	15.11	4.38	5.73	0.005	5.14	4.73	4.47	4.21	4.21	4.88	4.91	5.01	5.45	6.01
58	2/Floor Lamp	10000	10000	1.5e6	8.00	16.70	7.2e5	9.4e5	1.1e6	9.9e5	9.9e5	9.9e5	9.9e5	1.0e6	1.0e6	1.1e6	6.0e5	8.8e5
59	2/Kitchen Light	8.08	5.10	2.04	1.06	1.08	1.53	0.97	1.10	1.49	1.50	1.50	1.48	1.40	1.48	1.52	1.53
60	2/Laptop	8.10	6.00	9.80	3.94	4.98	3.14	3.91	3.86	4.04	3.98	3.98	4.45	4.41	4.54	4.95	5.28
61	2/Microwave	8.49	5.63	12.55	4.23	5.37	2.95	4.73	4.39	4.64	4.35	4.35	4.85	4.79	4.88	5.43	5.59
62	2/Refrigerator	8.95	7.02	10.01	3.42	3.38	0.009	4.54	4.30	5.93	5.92	5.88	6.04	5.87	5.70	5.05	6.24
63	2/Speaker	7.56	5.02	9.77	3.56	4.33	4.36	3.77	3.47	3.96	3.62	3.62	3.92	3.88	3.85	4.24	4.39
64	2/TV	14.10	8.70	14.44	4.81	6.33	3.10	5.21	4.61	5.02	4.60	4.60	5.40	5.40	5.23	5.67	5.99
65	2/Toaster	1428	1250	2202	7.85	15.69	0.79	1035	1096	1307	1426	1417	1425	1372	1338	549.0	1329
66	2/xbox	11.07	6.53	15.70	4.47	5.87	2.21	5.65	5.21	5.22	4.93	4.93	5.54	5.55	5.64	6.05	6.51
67	3/DishWasher	196.1	161.1	273.1	7.50	14.44	1.48	127.0	134.1	172.1	179.1	176.1	179.1	172.1	171.1	142.1	179.1
68	3/HairDryer	18.98	11.74	28.97	6.07	9.65	0.004	9.02	8.69	6.59	6.51	6.51	8.03	8.03	8.65	9.48	10.23
69	3/Kettle	256.1	227.1	357.1	7.41	13.68	0.07	163.1	168.1	196.1	199.1	198.1	207.1	202.1	189.1	104.0	196.1
70	3/Laptop	17.54	14.58	22.51	5.37	7.70	8.75	9.93	9.36	12.02	12.08	12.05	12.18	11.95	11.86	12.23	12.96
71	3/Microwave	8.73	5.33	14.95	4.39	5.69	0.24	5.15	4.78	4.51	4.26	4.26	4.91	4.92	5.04	5.51	5.99
72	3/Refrigerator	7.21	5.10	6.65	2.77	2.45	0.007	3.01	2.87	3.93	3.94	3.91	4.01	3.90	3.81	3.33	4.15
73	3/Surface	10.48	6.22	16.74	4.76	6.27	5.79	5.64	5.32	4.68	4.54	4.54	5.36	5.35	5.56	6.02	6.70
74	3/TV	10.38	6.60	17.21	4.83	6.40	2.53	5.81	5.44	4.87	4.69	4.69	5.51	5.51	5.72	6.32	6.93
75	3/Toaster	434.1	357.1	615.0	7.60	14.39	0.20	288.1	303.1	385.1	406.1	403.1	408.1	391.1	386.1	197.1	388.1

```
double hysteretic_filter(double sample)
{
    static double prev = 0;
    if (abs(sample - prev) > THRESHOLD) {
        prev = sample;
    }
    return prev;
}
```

FIGURE 1. Hysteretic noise reduction. `THRESHOLD=0.1` for sources T, T_{dp}, `THRESHOLD=1.0` for sources $P_{A/B/C}$

1.2. Compression mode

Most[1] algorithms are tested in streaming mode with output buffer size $B_{out} = 256$ bytes. When the buffer fills up, the algorithm state is re-initialized, all adaptation data gets lost. This reinitialization is necessary to be able to uncompress the blocks independently.

With every compressed block, the first data point from its respective uncompressed block (v_0) is transmitted uncompressed, taking m bits in binary coding. We take this into account when computing the compression ratio.

1.3. Redundant resolution and noise reduction

Temperature and humidity data (T, T_{dp}) change slowly and ought to compress well with RLE[2] technique. As we will see below, the actual compression ratio is not very good because of low amplitude higher frequency noise mixed in the slow signal. The sources T, T_{dp} are logged with fractional part of two decimal digits, which corresponds to the specified *resolution* of the sensor: 0.01°C [11]. But the specified *repeatability* of the sensor, limited by the noise level of the underlying physical sensor, is only 0.1°C. Consequently, storing data at maximum resolution is pointless. We will lose nothing if we apply noise reduction before compression. To test the effect of noise reduction on compression ratio, we include in the test routine the results of noise reduction with a simple hysteretic algorithm shown on Figure 1. The noise reduced sources bear the suffix `-noise` in Table 4.

[1] exceptions are noted in algorithm descriptions

[2] Run Length Encoding

Applying noise reduction before compression looks like switching from lossless to lossy compression, but it is not exactly the case. Lossy compression drops signal details that do not fit the compression schema. Noise reduction removes noise unrelated to measured physical parameter, or redundant resolution actually provided by the sensor but unnecessary in specific application, which results in improved compression with a lossless compression algorithm.

2. Compression technique selection

Sensor data are "analog signal" type sources [12] for which LPC[3] technique is a natural choice. This technique appears with variations in many algorithms intended for data with significant autocorrelation [3, 6, 7, 13–16]. Sensor data also fall into this category, if the sampling frequency is much higher than the higher frequency in the signal spectrum. The technique can be used in streaming mode and does not require considerable memory or CPU resources.

LPC employs a predictor which provides a prognosis of the next data value E_{v_i} based on k previous values:[4]

$$E_{v_i} = E(v_{i-k}, \dots v_{i-1}).$$

The algorithm calculates *residual errors* $r_i = v_i - E_{v_i}$, and replaces the original sequence $V = [v_0 \dots v_n]$ with $V' = [v_0, r_1, \dots r_n]$. With sufficiently good predictions, the average number of significant digits in r_i is lower than in v_i, so encoding V' in a variable-length coding yields a compression effect. For greater effect, the coding should use shorter codewords for values of r_i that occur more frequently.

The distribution of r is most often [7, 15, 16] modeled with Laplace distribution

$$p(r) = \mathcal{L}(0, b) = \frac{1}{2b} \mathrm{e}^{-\frac{|r|}{b}},$$

where $2b^2$ is the variation. For a dataset with a Laplace distribution (also called *double exponential distribution*), near optimal coding is achieved [18] by Golomb-Rice coding [19, 20], and indeed, the coding is often used in digital audio compression algorithms. The prefix coding provides short

[3] Linear Predictive Coding

[4] $k \in 0, 1, \dots$ is termed *prediction order* [6]. In the special case of $k = 1, E(v_i) = v_{i-1}$ the technique is called *delta-coding* or *differential coding* [17]

codewords for small r values, and also allows to encode large r values with no memory-consuming encoding table:

$$r \in [-2^m \dots 2^m], \tag{1}$$

where $m = \max_i \lceil \log_2(v_i) \rceil$is the ADC resolution.

Some authors use more powerful entropy coding techniques (namely, arithmetic coding) for residual coding [5, 16]. As the alphabet of all possible r values is rather large, arithmetic coding is used only for a limited range of most probable r values. The rest of r values are coded with Golomb-Rice coding [16], or with fixed length binary coding [5]. We cannot follow these examples as both arithmetic coding and optimal Golomb-Rice coding require knowledge of data distribution which is unavailable to a streaming coder.

A solution might be an adaptive algorithm where the algorithm state is adapted after compressing every data point r. Adaptation rules in the coder and decoder are identical, which removes the need to transfer a coding table along with compressed data. Examples of adaptive algorithms are adaptive Huffman coding [21], adaptive arithmetic coding [22], book stack compression [23], adaptive Golomb-Rice coding `RLGR` [24]. The latter is notable for its low memory footprint, so we will try to apply it to our task.

2.1. Prediction technique selection

We will use first-order linear prediction: $E(v_i) = v_{i-1}$ because of its low memory and CPU requirements. Approaches to improve prediction do exist: higher order linear prediction [25], context-based prediction [9, 14], neural networks [26, 27]. We can turn to them in cases where data characteristics are better known, and RAM and CPU resources are not so scarce. In this article we will pursue V' coding techniques which yield acceptable results even when prediction quality is not very good, and require minimal RAM and CPU resources.

3. Testing known algorithms

3.1. Algorithms `z22`, `z1`

We include results of the popular `zstd` [28] compressor in the test routine to get an idea of test data entropy. `z1` stands for `zstd -1` (the minimum compression level), `z22` stands for `zstd -ultra -22` (the maximum compression level). Data was fed to `zstd` in text form, one

sample per line, timestamp and decimal point removed. Compression ratio was found from `zstd` output in verbose mode (option `-v`). Results achieved by `zstd` are most often better than the results of streaming algorithms below, because `zstd` uses RAM at will to implement powerful compression algorithms, in particular LZ77[29] and ANS[30]. Besides, `zstd` compresses the input stream as a whole, with no limitation of output buffer size B_{out}.

3.2. Algorithms `F16`, `F256`

These two tests use block mode compression algorithm `FELACS` [4] with input block sizes of 16 and 256 bytes respectively. `FELACS` is closely based on the algorithm designed by NASA for a compression chip [31], adapted for software implementation by adding a heuristic method to choose [nearly] optimal Golomb-Rice coding parameter k.

`FELACS` is not a steaming algorithm and as such does not fit the task in hand. However, its results are interesting to compare with, as it is an algorithm intended for sensor data and using modest resources. To get the upper bound estimate of compression ratio, we test `FELACS` in the most favorable conditions with no limitation of output buffer size B_{out}.

Note that `FELACS` does not feature RLE encoding mode, so on test data with a large number of repetitions (rows №№ 44,45,53,58,65 in Table 4) its results are comparatively low.

3.3. Algorithm `RLGR`

Algorithm `RLGR` [24] combines run length encoding (RLE) and entropy coding with Golomb-Rice coding $\mathbf{GR}(k)$. Unlike `FELACS`, `RLGR` is a streaming algorithm that could fit our task. `RLGR` is an adaptive algorithm: the Golomb-Rice coding parameter k is changed after every codeword so as to reduce the average codeword length when encoding specific data stream V'.

After encoding several $r = 0$ in series, the algorithm switches to "run mode", where every codeword representing $r_i \neq 0$ is preceded by a counter of preceding zero r values. The run length counter is encoded in Golomb-Rice code $\mathbf{GR}(k_2)$, the parameter k_2 being adapted independently of k.

The test results for RLGR are contradictory. For many data sources the compression ratio is better than that of many other algorithms, but for a number of sources the compression ratio is lower than 1, i.e. applying RLGR results in *expansion* instead of compression. Using the coder

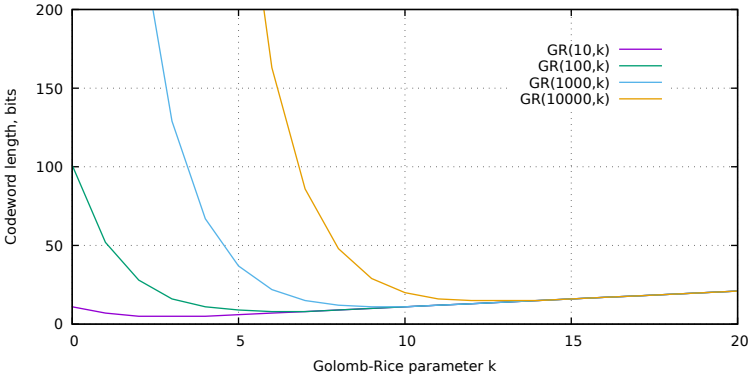


FIGURE 2. Golomb-Rice **GR**(k) effectiveness

prototype *rlgr.pl* (URL) we can study[5] the stream of codewords produced by the coder and explain the effect. When processing a long sequence of small r the adaptation rule reduces k to make $\mathbf{GR}(k)$ efficient for encoding small numbers. If the sequence is followed by a large r, $\mathbf{GR}(k)$ with a small k turns out very inefficient. For example, when encoding data source № 3, at step $i = 44$ the number 1530 is encoded with $\mathbf{GR}(2)$, which results in a "superlong" codeword of 385 bits:

$$\mathbf{bitlength}(\mathbf{GR}(x,k)) = \lceil \frac{x}{2^k} \rceil + k$$
$$\mathbf{bitlength}(\mathbf{GR}(1530,2)) = 385.$$

There are three reasons for this behavior.

(*1*) The algorithm works on the principle of "backward adaptation": the adaptation is based on *already processed* values in the hopes that more similar values are ahead. There is no way to change k explicitly when needed; the only available signaling from coder to decoder to change k is by transmitting suboptimal codewords.

(*2*) Golomb-Rice is a *linear* coding (with $k = const$ the codeword length is proportional to the value encoded):

$$x > k : \mathbf{bitlength}(\mathbf{GR}(x,k)) \sim x$$

i.e. the cost of encoding large numbers with insufficiently large k is very high (Figure 2). By contrast, in *exponential* codings the number or representable values grows in geometric progression (cf.

[5] `shell command: ./rlgr.pl -18 data/1/aem1-BWATT`

Elias γ [32]), so encoding large values does not produce superlong codewords:

$$\mathbf{bitlength}(\gamma(x)) \sim \log_2(x),$$
$$\mathbf{bitlength}(\gamma(1530)) = 21.$$

(3) The rule of "upwards" adaptation is $k_{i+1} = k_i + (p+1)/4$, where $p > 1$ is the length of the unary prefix in the suboptimal codeword $\mathbf{GR}(k)$ just produced. In the example being studied, the rule yields $k = 97$ (!) at step $i = 45$. Then the adaptation algorithm starts lowering k at the rate of 0.5/step, which is rather slow, so optimal $k = 8$ is only reached at step $i = 222$. The average codeword length in the interval $i \in \{44, \ldots, 222\}$ equals 56, which is more than the bit length of encoded numbers (Equation 1). Consequently, the compression ratio in the interval is lower than 1: $K = S_0/S_c = (18 + 1)/56 = 0.34$.

We see that on real sensor data the algorithm `RLGR` sometimes gets into "tight places" and needs some adjustments to be usable for the task in hand. Let us put it aside for a while and try to use exponential codings instead of Golomb-Rice coding. We will return to `RLGR` in subsection 4.11.

This preliminary test is done without output buffer size limit (B_{out}).

4. Experimental algorithms design and testing

4.1. Algorithms `EG` and `EW`

Algorithm `EG` encodes values r in Elias γ coding (Appendix 1). After encoding $r = 0$ (which corresponds to a perfect prediction), the algorithm follows it with a repetition counter, also in Elias γ coding. This way, the algorithm implements RLE mode.

Algorithm `EW` is similar to `EG`, except the coding employed is Elias ω.

Test results of the algorithms `EG` and `EW` (Table 4) are also similar, neither is a definite leader. The results are much more consistent than those shown by `RLGR`, but for some sources the compression ratio is low. As we will see below, these sources have heavy distribution tails — a considerable share of $r : |r| \geqslant 128$.

4.2. Exponential binary coding (expbinary)

To describe the following algorithms, we will need a coding that we will call expbinary for lack of a better name. It is a variable length coding without the prefix property[6], so it is never used alone — always in

[6] $\nexists A, B : Ax = B$ (no codeword A is a prefix of another codeword B)

combination with some prefix code, and is seldom mentioned in literature. In [18] it is mentioned under the name of $\overline{B}(n)$. It can be seen in the suffix part of Elias γ or Elias ω which are exponential codings with prefix property. The expbinary coding as well as Elias γ and Elias ω coding examples are shown in Appendix 1.

In binary coding an n-bit word represents numbers in

$$[0 \dots 2^n - 1], n \in 1, 2, \dots. \tag{2}$$

In expbinary coding an n-bit word represents numbers in

$$[2^n \dots 2^{n+1} - 1], n \in 0, 1, 2, \dots. \tag{3}$$

One can note that

- codeword length in expbinary is one bit less than in binary:

$$\mathbf{bitlength}(C_{expbinary}(N)) = \mathbf{bitlength}(C_{binary}(N)) - 1$$

- expbinary coding cannot encode zero ($N = 0$);
- empty (0-bit) codeword is in use: $C_{expbinary}(1) = empty$; this is possible because expbinary codewords are always accompanied by a kind of prefix.

To decode $C = C_{expbinary}(N)$ one needs to use the prefix to determine the length of expbinary codeword $n = \mathbf{bitlength}(C)$, then find the start of the value range for codeword length n: $S = 2^n$, and finally calculate N:

$$N = \begin{cases} S + C, & n > 0, \\ 0, & n = 0, \end{cases}$$

where C is interpreted as an n-bit number in binary coding.

4.3. Algorithm EB

The test is intended to obtain an upper-bound compression ratio estimate when coding V' with a variable length coding based on expbinary. The algorithm encodes r_i in expbinary and outputs without prefixes that would be necessary for the decoder to split the stream into codewords, i.e. decoding is impossible. Results of the test are painted gray in Table 4 to show they are *hors concours*.

The results are better than the results of EG and EW by a factor of ~ 2. Hence we can hope to improve the results of EG and EW if we find a more suitable prefix coding schema than that used in Elias γ and Elias ω codings.

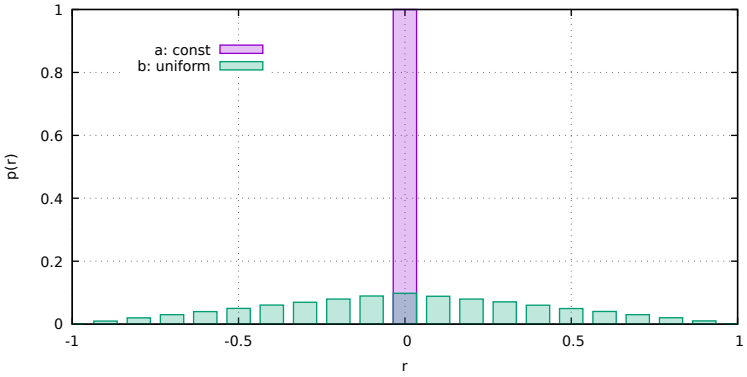


Figure 3. Residual distribution (a) for constant data $v_i = \text{const}$, (b) for uniform random data $v_i = \mathbf{uniform}(0, 1)$

4.4. Distribution of delta-coded data

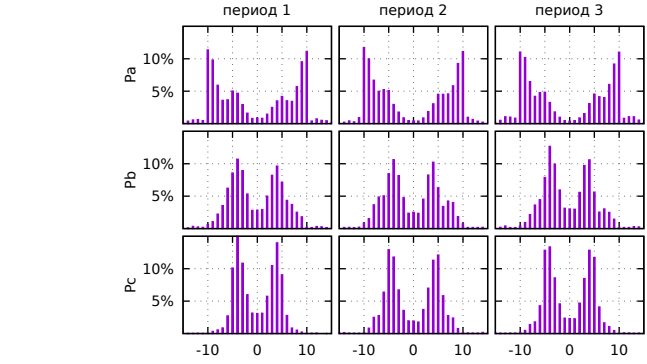
The residual distribution $p(r)$ is usually modeled with the Laplace distribution but does not necessarily fit the model well. Figure 3 shows both the best case (a) $r_i \equiv 0$ with the distribution $p(0) = 1$ and the worst case (b) in which v_i ie random data with uniform distribution. The best case occurs when the predictor works perfectly; for the simplest delta-predictor, this happens only on constant data $v_i \equiv \text{const}$. In the worst case $p(r)$ has the form of an equicrural triangle.

Consider the distributions of the datasets used in the tests. For the purpose of coding technique selection it is convenient to consider not the probability density function $p(r)$ but the discrete distribution of an auxiliary function $p(W(r))$, where

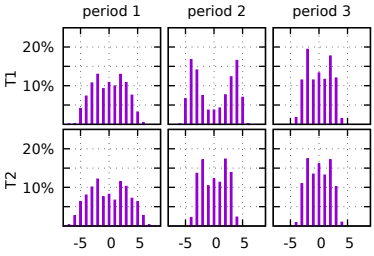
$$W(r) = \begin{cases} \mathbf{bitlength}(C_{expbinary}(r)) + 1, & r > 0 \\ 0, & r = 0 \\ -\mathbf{bitlength}(C_{expbinary}(-r)) - 1, & r < 0 \end{cases}$$

and $w_i = |W(r_i)| - 1$ is the number of bits necessary to represent r_i in expbinary. $W(r)$ is negative for negative r to make the distribution asymmetry observable. $W(0) = 0$ appears when $v_i = v_{i-1}$ and indicates an opportunity for RLE.

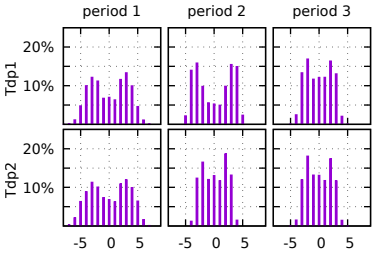
By comparing the distributions on Figure 4, we can observe that:



(*a*) electrical power



(*b*) air temperature



(*c*) air dew point

FIGURE 4. Distribution of $W(r)$ for different sources

- delta coding does cope with the task of bit width reduction for all datasets considered, despite prediction quality being far from perfect. With a perfect predictor, the distribution would be a single peak: $p(W(0)) = 1$;
- the source `Pa`, for which `EG` and `EW` show the worst results, has a considerable (up to 40%) share of large values requiring 8–9 bits to code in expbinary, apparently because delta-coding is not a good predictor for the dataset. Encoding the values in Elias γ results in codewors with 9–10 bit prefix, the total codeword length being 18–20 bits, hence the low compression ratio;
- range of $W(r)$ and distribution modes vary from source to source as well as from period to period. Hence, neither Elias γ nor any other fixed coding schema will be good for all datasets; the coding schema has to be adaptive;
- there is a slight asymmetry in distributions for positive and negative

r which reflects a difference of rise and fall rates in the original sequence V.

4.5. `e1`: dynamic prefixes with "book stack" algorithm

The alphabet of prefixes needed to support encoding V' in expbinary is rather small: $[-m, m]$ at worst (Equation 1), much less actually (Figure 4*b*). For a small alphabet, we can implement dynamic encoding with little resources using the "book stack" technique[7] [23].

Algorithm `e1` uses $W(r)$ for prefixes, i.e. a prefix specifies the bit length and the sign of the value encoded in expbinary that follows the prefix. To encode prefixes dynamically, the algorithm uses a data structure similar to a book stack; book titles correspond to encoded prefix values ($W(r)$). Initially the book stack is empty, its height $h = 0$. Encoding a residual r_i goes as follows:

- $w = W(r_i)$ is computed and looked up in the stack with downward linear search: $j \in [0 \dots h-1]$;
  - if the book w is found in the stack at depth j, the prefix is encoded as $C_p(j)$, and the book w is moved to the top of the stack, its index becoming $j = 0$;
  - otherwise (the book w is not found in the stack), the prefix is encoded in two codewords: $C_p(h), C_n(w)$, and the book w is added at the top of the stack. The stack height h is thus increased by 1.
- if $r_i \neq 0$, the value codeword is sent: $C_{expbinary}(r_i)$, the length of the codeword is $|w|$;
- otherwise ($\{r_i, \dots r_{i+R-1}\} = \{0 \dots 0\}$), the codeword $C_r(R)$ is sent, where $R \in 0, 1, \dots$ is the run length.

Here $C_p(j)$ is a variable length coding used for prefix encoding, such as unary or Golomb-Rice codings. The best results have been obtained with vbinary2x1x [34] coding, where the first three codewords have the same length of 2 bits (Appendix 1). A variable length coding $C_r(x)$ is used for run length encoding; the best results have been obtained with Elias ω coding.

The results shown by `e1` are indeed better for data sources where `EW` and `EG` were weak, especially for № 59. However, for a number of sources, `e1` is weaker than `F256` and `RLGR`.

[7] also known as Move-to-front transform [33]

The possible reason for the weakness is the decision to move r_i sign info to prefixes in the hopes of benefiting from distribution asymmetry. This way, we encode $|r_i|$, not r_i in expbinary, which reduces the length of all expbinary codewords by 1 bit. This could be beneficial on monotonic ranges of V'. But the number of prefixes in use (the book stack height h) is increased by a factor of two, so we need high locality (the use of prefixes near the top of the stack) to achieve the benefit. Otherwise, the losses from increased prefix codeword lengths will outweigh the gains.

4.6. Algorithm e2: sign in expbinary part

Algorithm e2 differs from e1 only by moving the sign from prefixes back to the expbinary part. Now prefixes are computed as $w = |W(r_i)|$, the number of different prefixes in use is reduced by factor of two. In tests without B_{out} limitation (not included in Table 4) e2 outperforms e1 for all data sources, but with $B_{out} = 256$ limitation, e2 is often weaker than e1.

4.7. Algorithm e3: limiting the stack height

In the naive book stack implementation, the "move-to-front" operation is relatively expensive. The operation is executed once for every r_i, i.e. frequently. With 10–20 different prefixes in use, the stack has to be implemented in memory, e.g. as an array; moving a book from position j to the top requires a shift (sequential copying) of array elements from 0 to j to free the place at the top.

If we limit the stack height by a small value, like $h_{max} = 5$, we can implement the stack as a 32-bit integer. The operations on the stack on 32/64-bit processors can be implemented very efficiently in the spirit of SWAR algorithms [35]. Shifting the book stack becomes a bitwise shift instead of memory copying. Naturally, limiting the height implies more "missed" cases which will have to be handled as a "new prefix". We will minimize the codeword length of the "new prefix" case by coding it as $j = 0$ instead of $j = h$ used in the original book stack technique.

It is also likely that a book found deep in a high book stack will be not better but even worse than a book from the shelf, as for large j the codeword $C_p(j)$ might be longer than the "new prefix" codeword pair $C_p(0), C_n(w)$.

Test results in Table 4 of e3 as compared to e2 have a slightly worse compression ratio for 31 sources of 75, with no change for 44 sources.

4.8. Algorithm e4: slower adaptation

Let us try to improve compression by slowing down the adaptation. Do not move the book to the top after use (giving it the shortest prefix at once, taking it from the current favorite), move the book up only by one position, similar to what bubble sort does.

Test results of e4 in Table 4 as compared to e3 are better for 59 sources out of 75, and worse for 16 sources.

Test results of e4 as compared to e1 are better for 50 sources, worse for 21 sources, no change for 4 sources.

4.9. Algorithm e5: multiple book stacks

Limiting the book stack height has increased the number of "new prefix" cases because when adding a new book to the stack we have to remove a book to make space (subsection 4.7). We can try to decrease the number of "new prefix" cases using multiple low-height book stacks. Organize the coder as a Moore finite state machine with the number of states equal to the number of used prefixes N_w. After handling r_i, the coder enters the state $w = w_i$ and stays there while handling r_{i+1}. There is a separate book stack in each state; the coder uses the current state book stack to map $w_{i+1} \to j_{i+1}$, then goes to state w_{i+1}. If the input data is such that not all state transitions are equiprobable, there is a hope that the book stack of every state will contain the books needed in that state.

Algorithm e5 requires slightly more RAM than its predecessors as it needs to store multiple book stacks

$$M \sim \lceil \log_2(N_w) \rceil h_{max}.$$

E.g. if the number of states $N_w = 16$ and every book stack is represented by a 32-bit number ($h_{max} \leq 8$), $M = 64$ bytes of RAM will be needed. Halfway implementations are also possible, with the number of book stacks $N_s < N_w$, where the state number is hashed to choose the book stack to use in the state.

In a preliminary test without B_{out} limitation which is not included in Table 4, algorithm e5 was better than e4 for 26 sources of 75, insignificantly worse for 9 sources (№ 26, 28, 36, 38, 42, 45, 50, 69, 75), and the same for 1 source. However results of the test with $B_{out} = 256$ bytes, shown in Table 4, are very different: e5 is better than e4 for 16 sources, worse for 58 sources, and no change for 1 source. Overall, the effect from multiple book stacks was lower than expected, and even negative in $B_{out} = 256$ mode that is needed for our task.

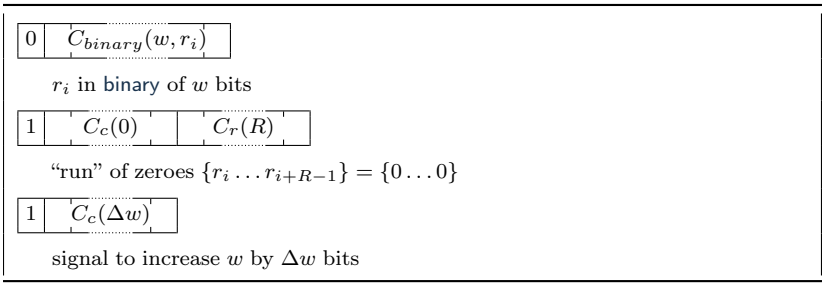


FIGURE 5. Codewords used by algorithm DD. C_c — control coding (vbinary2x), C_r — coding for RLE counter R (Elias ω)

4.10. Algorithm DD: "dynamic delta"

Let us try a different approach based on the idea from [3]: encode r_i not with a variable-length prefix coding, but with binary coding of fixed length w which can be changed when necessary by transmitting an escape codeword. The algorithm in [3] is not specified exactly; we will implement the idea in adaptive style, without accumulating blocks of uncompressed data to choose w per block.

The coder converts the sequence V' into a sequence of codewords shown at Figure 5. No signal to decrease w is provided; instead, the decrease is performed automatically after each step:

$$w_{i+1} = \begin{cases} w_i - d & w_i > 1, \\ w_i & w_i = 1, \end{cases}$$

where d and w_0 are the algorithm parameters. The best results in the tests are obtained with $d = 0.5$, $w_0 = 7$.

Algorithm DD loses to e1 ...e5 in binary part coding efficiency because of insignificant leading zeroes it has to transfer when encoding $r < 2^{w-1}$. But it gains in efficiency of prefix encoding when a significant part of data is transferred with a minimal one-bit prefix.

Test results of algorithm DD shown in Table 4 are mostly worse than those of predecessors, but they are consistent and unexpectedly good for an algorithm so simple.

4.11. Algorithm RLGR′: RLGR with $B_{out} = 256$

Let us return to the algorithm RLGR (subsection 3.3) and test it with output buffer limitation: $B_{out} = 256$ bytes. For the test to pass, a small

parameter correction was needed as the original RLGR starts with $k = 2$ and for 10 sources (№№ 2, 3, 4, 18, 19, 20, 30, 32, 34, 43 in Table 4) crashes, unable to handle a large r soon after the start:

$$\lceil \mathbf{bitlength}(\mathbf{GR}(r, k))/8 \rceil > B_{out}.$$

The correction is to start with $k = 10$; then the test RLGR′ does pass and (surprise!) the results show no inconsistencies that we saw in the test RLGR. Why?

The algorithm still gets into "tight places", but the consequences of that are limited by the space available in the current output buffer. Once in a "tight place", the algorithm quickly uses up the space in the current output block and switches to a new block — which involves algorithm re-initialization and thereby automatic escape from the "tight place". This pop-up solution is hard to call elegant, and its efficiency decreases for larger B_{out}. Nevertheless, the test has passed and demonstrated an improvement for 42 sources as compared to e4. This means we can learn from RLGR.

(*1*) Adaptive Golomb-Rice coding turns out more efficient than exponential codings (Elias γ, Elias ω) which contain full suffix length information in the prefix. Dynamic prefix coding in algorithms e1...e5 is akin to adapting k in RLGR, and yields a comparable effect.

(*2*) "run"—"no run" mode switching improves compression for data with few runs, which are much less compressible (sources №№ 1, 2, 15, 16, 29, 30 in Table 4). At the same time, for sources with sparse values among runs (№№ 36, 38, 40, 42) RLGR' also outperforms the nearest competitors by 14–36%.

(*3*) Run length encoding in RLGR is defined algorithmically [**24**]; essentially the length is also encoded in adaptive Golomb-Rice coding GR($k2$), but with an interesting quirk: the adaptation procedure for $k2$ is called not after the codeword generation, but after outputting every bit of the prefix. This way, when encoding large R with small $k2$, the adaptation of $k2$ occurs in the middle of codeword generation. This improves efficiency of encoding large R with small $k2$ at the cost of extra CPU usage, but does not help in the case of small R with large $k2$. A large R after a long silence will be encoded efficiently, but then the algorithm will needlessly stay in "run mode" for a long time while $k2$ slowly adapts down to zero.

(*4*) The "backward adaptation" principle works well for fine k adjustments, but cannot prevent generating "superlong" codewords for large r when k is small. It could be solved by introducing a signaling

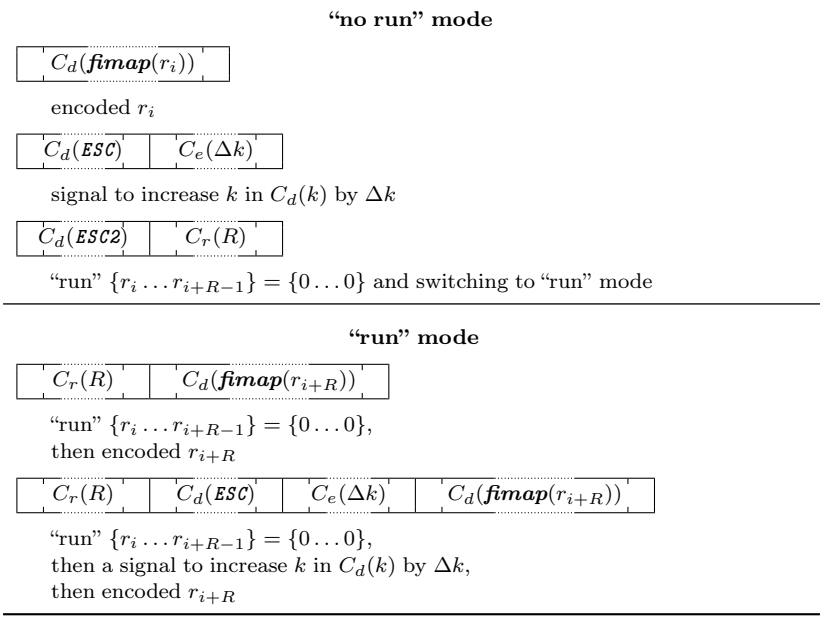


FIGURE 6. Codewords used by algorithm `e10`. C_d — data coding (**GR**(k)), C_r — coding for RLE counter R (Elias ω), C_e — escape increment coding (vbinary2x(1,2,3x))

method for increasing k instantly before encoding a sudden large r. With such a method in hand, we can dispose of the doubtful rule of quick upward adaptation which is the source of the "tight place" problem in `RLGR`.

4.12. Algorithm `e10`

Algorithm `e10` is the final in a series of experimental algorithms derived from `RLGR`. Like `RLGR`, `e10` has "run" and "no run" modes, and switches between them using a backward adaptation rule. The list of codewords used by `e10` is shown at Figure 6.

Residuals r are encoded in Golomb-Rice coding

$$C_d(r) = \mathbf{GR}(\mathbf{fimap}(r), k).$$

Parameter k adapts slowly, synchronously in coder and decoder. The function **fimap** maps the set of r (integers) to the domain of GR (non-

negative integers), and also reserves two ESC-values for signaling. Signaling is not expected to be used frequently, therefore the codewords allocated for ESC-values are not the shortest codewords available.

When r is *too large* to be [nearly] optimally encoded with current k, the coder sends the signal `ESC` to increase k to $k_{opt} = \lceil \log_2(\mathbf{fimap}(r)/2) \rceil$, followed by optimally encoded r: $C_d(r) = GR(\mathbf{fimap}(r), k_{opt})$. The algorithm continues with $k = k_{opt}$, following the usual backward adaptation rule. The condition *"too large"* is examined only in the coder and can be varied for implementation convenience without changes to the decoder.

When a "run" $\{r_i \ldots r_{i+R-1}\} = \{0 \ldots 0\}$ is encountered by the coder, the coder can do either of:

(*1*) encode the run verbatim $\{C_d(r_i) \ldots C_d(r_{i+R-1})\}$. This variant can be better when R is small;

(*2*) send the mode switch signal `ESC2`, then the encoded run length $C_r(R)$, where C_r is a coding suitable to encode arbitrary non-negative integers. Then the algorithm switches to "run" mode.

In "run" mode, every $C_d(r_i, k)$ is preceded by a counter of zero r values that precede r_i, encoded in C_r (subsubsection 4.12.2). The counter can be 0 if there was no preceding run. The algorithm returns to "no run" mode (synchronously in both coder and decoder) if there were no preceding runs for $T = 4$ subsequent r values.

For the coding C_e used for encoding Δk, the best results were obtained with vbinary2x(1,2,3x) (Appendix 1).

4.12.1. *Adaptation in `e10`*

The condition of Golomb-Rice coding optimality [**36**] can be rewritten as $\lfloor 2^{k-1} \rfloor \leqslant r < 2^{k+1} + 2^k$. When the condition is true, the length of the codeword $\mathbf{GR}(r, k)$ is minimal.

The adaptation algorithm for C_d coding changes k as follows:[8]

$$k_{i+1} = \begin{cases} k_i, & \lfloor 2^k \rfloor \leqslant r < 2^{k+1} + 2^k \\ k_i + 0.75, & r \geqslant 2^{k+1} + 2^k \\ k_i - 0.25, & r < \lfloor 2^k \rfloor \end{cases}$$

As we already have seen on Figure 2, the rate of upward adaptation is higher than the rate of downward adaptation because the efficiency of Golomb-Rice coding with $k > k_{opt}$ does not suffer as much as it does with $k < k_{opt}$.

[8] calculations yield fractional values that will be rounded down before use: $\mathbf{GR}(r, \lfloor k \rfloor)$

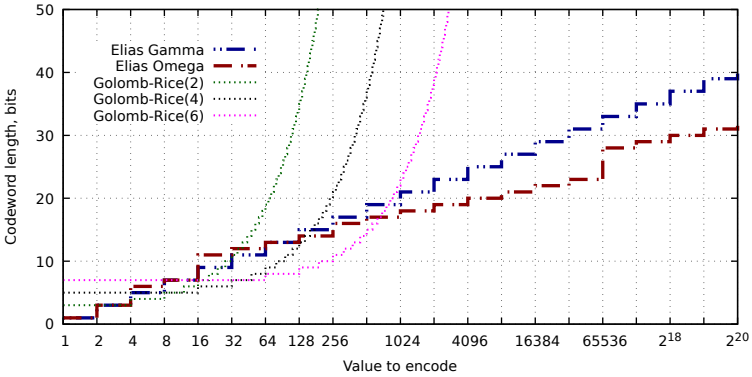


FIGURE 7. Comparative efficiency of RLE counter candidate codings for algorithm `e10`

4.12.2. *Choosing coding for RLE counter in `e10`*

We will look for C_r such that $C_r(R = 0) = \min = 1$ to minimize the overhead of being in "run" mode for r_i not preceded by runs.

Another requirement for C_r is to encode a large range of values efficiently, say $[0, 10^6]$, so Golomb-Rice codes are not suitable. The candidates considered were exponential codings Elias γ and Elias ω. The choice falls on Elias γ because it has better efficiency for lower R: $R \in \{4, \ldots, 7\} \cup \{16, \ldots, 63\}$). Elias ω coding is better than Elias γ for $R \geqslant 128$ (Figure 7), but it does not matter as RLE with big length yields compression by orders of magnitude better than is in principle possible in entropy encoding mode, with either of the codings. Thus, Elias γ gives slightly better result consistency.

4.12.3. *Final notes on algorithm `e10`*

Algorithm `e10` is safe from the "tight places" of `RLGR` thanks to the method of instant upward adaptation with ESC signaling. There is no method for instant downward adaptation because coding unexpected small r with large k does not result in superlong codewords: for $r < 2^k$ $\mathbf{bitlength}(\mathbf{GR}(r, k)) = k + 1$.

Using exponential coding for run lengths and the short timeout for "run" mode in the absence of runs provide for efficient coding in the case of infrequent long runs. Encoding short runs with exponential C_r could be less efficient than with adaptive Golomb-Rice used by `RLGR`, but this effect did not show up for test data. The compression ratio for runs in "run"

mode will not be worse than

$$K = S_0/S_c = mR/(G + \mathbf{bitlength}(\gamma(R+1))),$$

where $G < T$ is the number of steps with $R = 0$ between steps with $R > 0$. In the worst case of short runs ($R = 1$) with small input data width $m = 8$ and $G = T - 1 = 3$ steps we will get $K = 8/(3 + 3) = 1.25$.

Test results of `e10` as compared to `RLGR′` are better for 69 of 75 sources, worse for 5 sources, and the same for 1 source.

Test results of `e10` as compared to `e4` are better for 64 sources, worse for 10 sources, and the same for 1 source.

4.13. Algorithm comparison criterion

Comparing the algorithms directly by data from Table 4 is awkward as every algorithm's performance is represented by a vector of 75 values. A term-by-term vector comparison is only applicable in rare cases where algorithm A is better than algorithm B for every source. To range algorithms by merit, we need a scalar metric calculated from the vector.

Table 5 shows common distribution metrics calculated from vectors for each algorithm: minimum $\min K$, maximum $\max K$, arithmetic mean $\operatorname{avg} K$, and median $\operatorname{med} K$. The need to estimate the maximum communication delay that a sensor node can tolerate given the number of data sources and compression ratio for each source requires each algorithm to be attributed with its mean compression ratio. However, arithmetic mean it not good for this role as it is strongly affected by large maximum values present in current experimental data. The median is insensitive to large maximum values but can be far from the mean for skewed distributions.

The desired mean is given by

$$\overline{K} = \frac{\sum_{i=1}^{n} S_{0i}}{\sum_{i=1}^{n} S_{ci}},$$

where S_{0i} is a size of uncompressed data received from source i S_{ci} is a size of data received from source i after compression, n is the number of sources in the mix. The vector $(S_{01}, \ldots, S_{0n})$ gives relative source intensity (size of data sample and sampling frequency).

By compression ratio definition,

$$S_{ci} = S_{0i}/K_{ij},$$

TABLE 5. Algorithm comparison metrics. Rows sorted by harmonic mean $H(K)$ descending

Background color: *hors concours* | best in column | worst in column

Algorithm	min K	max K	avg K	med K	$H(K)$
z22	3.61	10000	218.1	6.85	6.98
z1	3.04	10000	201.1	4.96	5.15
e10	1.53	8.8e5	11817	4.19	3.79
e4	1.48	1.0e6	13748	3.95	3.54
RLGR′	1.52	6.0e5	8036	3.85	3.48
e5	1.40	1.0e6	13747	3.89	3.48
e1	1.43	9.9e5	13263	3.95	3.46
DD	1.46	1.1e6	14256	3.82	3.45
e2	1.50	9.9e5	13268	3.68	3.43
e3	1.50	9.9e5	13267	3.68	3.42
EG	0.97	9.4e5	12593	3.19	2.92
F256	1.08	16.70	4.100	2.91	2.92
EW	1.10	1.1e6	14807	2.91	2.82
F16	1.06	8.00	3.55	2.64	2.64
RLGR	0.001	7.2e5	9646	2.25	0.04

where K_{ij}—compression ration for source i by algorithm j (data from Table 4). Consequently,

$$\overline{K_j} = \frac{\sum_{i=1}^{n} S_{0i}}{\sum_{i=1}^{n} S_{0i}/K_{ij}}.$$

Source intensity data are only available when considering specific applications; for general algorithm comparison we will assume equal intensity (equal data volume from each source)

$$S_{01} = S_{02} = \cdots = S_{0n}.$$

Then

$$\overline{K_j} = \frac{\sum_{i=1}^{n} S_{01}}{\sum_{i=1}^{n} S_{01}/K_{ij}} = \frac{\sum_{i=1}^{n} 1}{\sum_{i=1}^{n} 1/K_{ij}},$$

which matches the well-known harmonic mean formula:

$$H(x_1, \ldots, x_n) = \frac{n}{1/x_1 + \cdots + 1/x_n}.$$

Harmonic mean values for data mix of all 75 test data sources are shown in the column $H(K)$ of Table 5. We adopt harmonic mean as the main algorithm comparison criterion.

5. Conclusion

We have empirically treated a number of known and experimental scalar sensor data compression algorithms, using small amounts of RAM and working in streaming mode with no uncompressed data buffering. All algorithms are based on differential coding, which is the simplest form of classical linear predictive coding (LPC); the article deals with techniques for efficient residual coding. We evaluated encoding residuals using variable length codes with dynamic prefixes obtained by MTF transform, adaptive width binary coding, adaptive Golomb-Rice coding. The evaluation used datasets from 75 real-world sensor data sources. The best compression ratios achieved are about 1.5/4/1000000 (min/median/max) with RAM usage of 256 bytes for the output compressed data buffer, and about 10 bytes for the algorithm state. The large maximum compression ratios are achieved for constant data sources thanks to the streaming compression mode.

We have observed that the compression ratio for slowly changing data can suffer from small amplitude random noise that inhibits the use of the efficient RLE compression mode. Applying a simple hysteretic noise reduction algorithm prior to compression improved the compression ratio for these sources by a factor of 11–20. Applying noise reduction to quickly changing data did not improve compression.

By compression ratio comparison, the best candidates for practical use are the algorithms `e10` (best compression) and `e4` (simpler to implement). Algorithms are primarily intended for use in sensor network nodes, but can also be useful in other cases where a compression algorithm with small RAM usage is needed, e.g. on a server receiving sensor data from a very large number of sources.

Test datasets used in the article, coder prototypes and the test framework are freely available (*https://github.com/yysizif/sencomp*[URL]).

Author is grateful to editorial staff of “Program Systems: Theory and Applications” and personally to Prof. S. V. Znamensky for insightful comments that enabled significant improvement of section 4.13.

References

[1] K. Fall. "A delay tolerant network architecture for challenged internets", *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (August 25–29, 2003, Karlsruhe, Germany), ACM, New York, 2003, ISBN 978-1-58113-735-4, pp. 27–34. ↑

[2] C. M. Sadler, M. Martonosi. "Data compression algorithms for energy-constrained devices in delay tolerant networks", *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems* (31 October 2006–3 November 2006, Boulder, Colorado, USA), ACM, New York, 2006, ISBN 978-1-59593-343-0, pp. 265–278. ↑

[3] S. Arrabi, J. Lach. "Adaptive lossless compression in wireless body sensor networks", *BodyNets '09: Proceedings of the Fourth International Conference on Body Area Networks* (April 1–3, 2009, Los Angeles, CA, USA), ICST, Brussels, 2010, ISBN 978-963-9799-41-7. ↑

[4] J. G. Kolo, S. A. Shanmugam, D. W. G. Lim, L. M. Ang. "Fast and efficient lossless adaptive compression scheme for wireless sensor networks", *Computers & Electrical Engineering*, **41** (2015), pp. 275–287. ↑

[5] F. Huang, Y. Liang. "Towards energy optimization in environmental wireless sensor networks for lossless and reliable data gathering", IEEE International Conference on Mobile Adhoc and Sensor Systems (08–11 October 2007, Pisa, Italy), 2007, pp. 1–6. ↑

[6] J. Coalson, Description of the FLAC format. ↑

[7] T. Robinson. *SHORTEN: Simple lossless and near-lossless waveform compression*, Technical report CUED/F-INFENG/TR.156, 1994, 16 pp. ↑

[8] T. Pelkonen, P. Cavallaro, Q. Huang, S. Franklin, J. Meza, J. Teller, K. Veeraraghavan. "Gorilla: a fast, scalable, in-memory time series database", *Proceedings of the VLDB Endowment*, **8**:12 (2015), pp. 1816–1827. ↑

[9] E. Lazin. *Compression algorithms in Akumuli.* ↑

[10] F. Jazizadeh, M. Afzalan, B. Becerik-Gerber, L. Soibelman. "EMBED: A dataset for energy monitoring through building electricity disaggregation", *e-Energy '18: Proceedings of the Ninth International Conference on Future Energy Systems* (June 12–15, 2018, Karlsruhe, Germany), ACM, New York, 2018, ISBN 978-1-4503-5767-8, pp. 230–235. ↑

[11] *Digital Humidity Sensor SHTW2 (RH/T)*, Datasheet, Sensirion AG, Switzerland, 14 pp. ↑

[12] D. Vatolin, A. Ratushnyak, M. Smirnov, V. Yukin. *Data compression techniques. Archiver internals, compressing images and video*, DIALOG-MIFI, M., 2002, ISBN 5-86404-170-X (in Russian), 384 pp. ↑

[13] P. Ratanaworabhan, J. Ke, M. Burtscher. "Fast lossless compression of scientific floating-point data", *Data Compression Conference*, DCC'06 (28–30 March 2006, Snowbird, UT, USA), IEEE Computer Society, 2006, ISBN 0-7695-2545-8, pp. 133–142. ↑

[14] M. Burtscher, P. Ratanaworabhan. "High throughput compression of double-precision floating-point data", *Data Compression Conference*, DCC'07 (27–29 March, 2007, Snowbird, UT, USA), IEEE Computer Society, 2007, ISBN 0-7695-2791-4, pp. 293–302. ↑

[15] M. Hans, R. W. Schafer. *Lossless compression of digital audio*, HPL-1999-144, Hewlett-Packard Company, 1999, 37 pp. ↑

[16] Yu. A. Reznik. "Coding of prediction residual in MPEG-4 standard for lossless audio coding (MPEG-4 ALS)", *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing* (17–21 May 2004, Montreal, QC, Canada), 2004, ISBN 0-7803-8484-9. ↑

[17] C. C. Cutler. *Differential quantization of communication signals*, U.S. patent 2605361, 1950. ↑

[18] D. Salomon. *Data Compression. The Complete Reference*, 4th ed., Springer-Verlag, London, 2007, ISBN 978-1-84628-602-5, xxviii+1092 pp. ↑

[19] S. W. Golomb. "Run-length encodings", *IEEE Transactions on Information Theory*, **12**:3 (1966), pp. 399–401. ↑

[20] R. F. Rice, J. R. Plaunt. "Adaptive variable-length coding for efficient compression of spacecraft television data", *IEEE Transactions on Communication Technology*, **19**:6 (1971), pp. 889–897. ↑

[21] N. Faller. "An Adaptive System for Data Compression", 7th Asilomar Conference on Circuits, Systems, and Computers, 1973, pp. 593–597. ↑

[22] D. Marpe, H. Schwarz, T. Wiegand. "Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard", *IEEE Transactions on Circuits and Systems for Video Technology*, **13**:7 (2003), pp. 620–636. ↑

[23] B. Ya. Ryabko. "Szhatiye dannykh s pomoshch'yu stopki knig", *Problemy peredachi informormatsii*, **XVI**:4 (1980), pp. 16–20 (in Russian); B. Ya. Ryabko. "Data compression by means of a "book stack"", *Problems of Information Transmission*, **16**:4 (1980), pp. 265–269. ↑

[24] H. M. Malvar. "Adaptive run-length/Golomb-Rice encoding of quantized generalized Gaussian sources with unknown statistics", *Data Compression Conference*, DCC'06 (28–30 March 2006, Snowbird, UT, USA), IEEE Computer Society, 2006, ISBN 0-7695-2545-8, pp. 23–32. ↑

[25] J. Durbin. "The fitting of time-series models", *JSTOR: Revue de l'Institut International de Statistique*, **28**:3 (1960), pp. 233–344. ↑

[26] I. Schiopu, A. Munteanu. "Deep-learning-based lossless image coding", *IEEE Transactions on Circuits and Systems for Video Technology*, **30**:7 (2020), pp. 1829–1842. ↑

[27] M. Goyal, K. Tatwawadi, S. Chandak, I. Ochoa. "DZip: improved general-purpose loss less compression based on novel neural network modeling", *2021 Data Compression Conference*, DCC (23–26 March 2021, Snowbird, UT, USA), IEEE, 2021, pp. 153–162. ↑

[28] Y. Collet, Kucherawy M. (eds.). *Zstandard compression and the 'application/zstd' media type*, RFC 8878, 2018, 54 pp. ↑

[29] J. Ziv, A. Lempel. "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, **23**:3 (1977), pp. 337–343. ↑

[30] J. Duda. *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*, 2014, 24 pp. arXiv 1311.2540 ↑

[31] R. F. Rice, P. S. Yeh, W. Miller. *Algorithms for a very high speed universal noiseless coding module*, JPL Publication 91-1, Jet Propulsion Laboratory, Pasadena, 1991, 30 pp. ↑

[32] P. Elias. "Universal codeword sets and representations of the integers", *IEEE Transactions on Information Theory*, **21**:2 (1975), pp. 194–203. ↑

[33] J. L. Bentley, D. D. Sleator, R. E. Tarjan, V. K. Wei. "A locally adaptive data compression scheme", *Communications of the ACM*, **29**:4 (1986), pp. 320–330. ↑

[34] Yu. Shevchuk. "Vbinary: variable length integer coding revisited", *Program Systems: Theory and Applications*, **9**:4(39) (2018), pp. 239–252. ↑

[35] H. G. Dietz. *The Aggregate Magic Algorithms*, Aggregate.Org online technical report, University of Kentucky, 2021. ↑

[36] A. Kiely. *Selecting the Golomb parameter in Rice coding*, IPN Progress Report 42-159, 2004, 18 pp. ↑

Received 18.01.2022;
approved after reviewing 28.02.2022;
accepted for publication 04.04.2022.

Recommended by *Ph.D. S. A. Romanenko*

Information about the author:

Yury Vladimirovich Shevchuk

Researcher at Multiprocessor Systems Center of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Ph.D. Interest areas: sensor networks, operational monitoring, distributed data storage, high availability systems, distributed programming

0000-0002-2327-0869
e-mail: sizif@botik.ru

The author declare no conflicts of interests.

Appendix 1. Codings used in the article

dec	binary	expbinary	Elias γ	Elias ω	GR(3)	vbinary2x	vbinary2x1x	vbinary2x(1,2,3x)
0	0	—	—	—	0000	00	00	00
1	1	пусто	1	0	0001	01	01	010
2	10	0	010	10 0	0010	10	10	011
3	11	1	011	11 0	0011	1100	110	1000
4	100	00	00100	101 00 0	0100	1101	1110	1001
5	101	01	00101	101 01 0	0101	1110	11110	1010
6	110	10	00110	101 10 0	0110	111100	111110	1011
7	111	11	00111	101 11 0	0111	111101	1111110	11000
8	1000	000	0001000	111 000 0	10000	111110	11111110	11001
9	1001	001	0001001	111 001 0	10001	11111100	111111110	11010
10	1010	010	0001010	111 010 0	10010	11111101	<10 bits>	11011
11	1011	011	0001011	111 011 0	10011	11111110	<11 bits>	11100
12	1100	100	0001100	111 100 0	10100	<10 bits>	<12 bits>	111010
13	1101	101	0001101	111 101 0	10101	<10 bits>	<13 bits>	111011
14	1110	110	0001110	111 110 0	10110	<10 bits>	<14 bits>	1111000
15	1111	111	0001111	111 111 0	10111	<12 bits>	<15 bits>	1111001
16	10000	0000	000010000	101001 0000 0	110000	<12 bits>	<16 bits>	1111010
17	10001	0001	000010001	101001 0001 0	110001	<12 bits>	<17 bits>	1111011
18	10010	0010	000010010	101001 0010 0	110010	<14 bits>	<18 bits>	11111000
19	10011	0011	000010011	101001 0011 0	110011	<14 bits>	<19 bits>	11111001
20	10100	0100	000010100	101001 0100 0	110100	<14 bits>	<20 bits>	11111010
21	10101	0101	000010101	101001 0101 0	110101	<16 bits>	<21 bits>	11111011
22	10110	0110	000010110	101001 0110 0	110110	<16 bits>	<22 bits>	11111100
23	10111	0111	000010111	101001 0111 0	110111	<16 bits>	<23 bits>	111111010
24	11000	1000	000011000	101001 1000 0	1110000	<18 bits>	<24 bits>	111111011
25	11001	1001	000011001	101001 1001 0	1110001	<18 bits>	<25 bits>	1111111000
...	...	...	...	...	...	...	...	...
100	1100100	100101	0000001100100	10110 1100100 0	<16 bits>	<68 bits>	<100 bits>	<30 bits>
101	1100101	100110	0000001100101	10110 1100101 0	<16 bits>	<68 bits>	<101 bits>	<30 bits>
102	1100110	100111	0000001100110	10110 1100110 0	<16 bits>	<70 bits>	<101 bits>	<31 bits>
...	...	...	...	...	...	...	...	...
1000	1111101000	111101001	0000000001111101000	111001 1111101000 0	<129 bits>	<668 bits>	<1001 bits>	<275 bits>
1001	1111101001	111101010	0000000001111101001	111001 1111101001 0	<129 bits>	<668 bits>	<1002 bits>	<275 bits>
1002	1111101010	111101011	0000000001111101010	111001 1111101010 0	<129 bits>	<670 bits>	<1003 bits>	<276 bits>
...	...	...	...	...	...	...	...	...