# LABORATORY DATA COMPRESSION

M. Emoto, M. Shoji, S. Yamaguchi, NIFS, Toki, 509-5292 Japan
J. Kariya, Yamaguchi University, Ube, 755-8611 Japan
H. Okumura,* Matsusaka University, Matsusaka, 515-8511 Japan
M. Tamura, Nihon Sun Microsystems, Tokyo, 158-8633 Japan
Y. Teramachi, University of Industrial Technology, Sagamihara, 229-1196 Japan

*Abstract*

Most of the existing tools for lossless data compression, including *LHA*, *Zip*, *gzip*, and *bzip2*, are based on either textual substitution (LZ77 or LZ78) or block sorting, followed by entropy coding. These tools assume that the data have clear 8-bit boundaries and contain many repetitive substrings. Laboratory data such as A/D converter outputs, however, does not in general satisfy these conditions. To compress such data, we developed a general-purpose real-time compression library suitable for quantized (up to 16-bit) time-series data of unlimited number of channels. The first part of the algorithm adaptively chooses a prediction model among a family of polynomials, and estimates the variance of the prediction residuals. The second part of the algorithm encodes the residuals by length-limited minimum-redundancy coding, assuming either Gaussian or Laplace distributions. The library is used by our Java-based data management system developed for the National Institute for Fusion Science (NIFS). It can also be used as a standalone compression tool. Typical compression ratio is around 4 : 1, and compression/decompression throughputs are around 2-million 16-bit samples per second on a 400MHz Pentium-II PC running Linux.

## 1 INTRODUCTION

The NIFS collaboration on "workstation-based data acquisition, analysis, and control systems" was started in 1993 [1], and in 1996–1998 culminated in the construction of a Java-based data management system for the Large Helical Device (LHD) at NIFS.

A short description of the monitoring subsystem is in order[1]: Sensors attached to the reactor and the superconducting coils measure quantities such as temperatures, pressures, strains, voltages, and currents. Outputs from these sensors are amplified, low-pass-filtered, digitized by "oversampling" A/D converters, and fed into workstations, where the software averages the oversampled data down to the specified rates and eliminates random noise. The averaged data are stored locally and sent on the network to clients. The client software consists of Java applets that run within a Web browser.

The aim of the compression library is to save local storage and (hopefully) reduce network latency and traffic. The design requirements are low complexity (high throughput) and delayless transmission of compressed data. This latter requirement precludes block-oriented tools such as *Zip*, *gzip*, *LHA*, and *bzip2*.

## 2 ALGORITHM

The algorithm is based on a simplified length-limited minimum-redundancy (Huffman) coding of adaptive prediction residuals. Since at each sampled time we just loop over the channel index, henceforth we suppress channel indices and pretend as if there were only one channel, and let $x_t$ represent the quantized (integer) datum for the discrete (integer) time $t$.

At each time $t$, we predict the value $x_t$ on the basis of past few samples by one of the three extrapolations

| | |
|---|---|
| $\hat{x}_t^{(0)} = x_{t-1}$ | previous value |
| $\hat{x}_t^{(1)} = 2x_{t-1} - x_{t-2}$ | linear extrapolation |
| $\hat{x}_t^{(2)} = 3x_{t-1} - 3x_{t-2} + x_{t-3}$ | quadratic extrapolation |

that best fits the local nature of the time series, as will be explained below. The prediction error

$$e_t = x_t - \hat{x}_t$$

is assumed to obey discretized versions of either the Gaussian (normal) or the Laplace (two-sided exponential) distributions with zero mean and slowly changing variance. More precisely, $e_t$ is assumed to be distributed as $\lfloor Y + 0.5 \rfloor - \lfloor X + 0.5 \rfloor$, where $X$ and $Y$ are two random (undiscretized) variables such that $X - \lfloor X \rfloor$ is uniformly distributed over $[0, 1)$ and $Y - X$ is either Gaussian or Laplace with zero mean.

Typical laboratory time-series data are not stationary; it may move wildly, then calm down for an extended time interval. For such data it is necessary to estimate variance on the basis of a small number of recent sample points. We use the quantity

$$z = |e_{t-16}| + |e_{t-15}| + \cdots + |e_{t-2}| + |e_{t-1}|$$

On the basis of this value, we construct 16 canonical Huffman codewords, corresponding to 16 intervals of $e_t$ shown

* E-mail: okumura@matsusaka-u.ac.jp

[1] The overall system and Java 3D visualization are discussed elsewhere in this Conference [2, 3].

Table 1: 16 groups for prediction errors

| Group Number | $e_t$ | Number of bits that follow |
|---|---|---|
| 0 | 0 | 0 |
| 1 | $\pm 1$ | 1 |
| 2 | $\pm 2, \pm 3$ | 2 |
| 3 | $\pm 4, \ldots, \pm 7$ | 3 |
| 4 | $\pm 8, \ldots, \pm 15$ | 4 |
| 5 | $\pm 16, \ldots, \pm 31$ | 5 |
| $\vdots$ | $\vdots$ | $\vdots$ |
| 14 | $\pm 8192, \ldots, \pm 16383$ | 14 |
| 15 | $\pm 16384, \ldots$ | 15 (16) |

Table 2: Exceptions to Table 1.

| Value | codeword |
|---|---|
| −32767 | 1111111111111110 |
| −32768 | 1111111111111111 |
| +32767 | 0111111111111110 |
| End-Of-Data | 0111111111111111 |

in Table 1, with lengths given by either Table 3 or Table 4. Given $e_t$, we output one of these codewords that corresponds to the group to which $e_t$ belongs (by looking up Table 1, with some exceptions given by Table 2), then output a fixed number of bits that specifies the position of $e_t$ among the values within the same group.

The variable-length minimum redundancy codes for the 16 groups are carefully determined by numeical calculation assuming Gaussian (Table 3) and Laplace (Table 4) distributions.

For example, if $z = 400$ and $e_t = 27$, we construct the canonical Huffman code with codeword lengths given by the 14th row of Table 3 (or Table 4). Since $e_t = 27$ belongs to group 5 of Table 1, we output the variable-length codeword whose length is $\ell_5 = 2$ bits. Next, we output the 5-bit position of the number 27 within this group. To be concrete, the bit pattern of 27 is '11011', but since every number between 16 and 31 are 5-bit numbers with the leftmost bit '1', we can omit the leftmost bit and instead insert the sign bit. That is, the positive number 27 will be encoded as '01011' whereas the negative number −27 would be '11011'.

A more precise description of the overall compression algorithm is as follows. As above, we suppress the obvious indices for the channel number over which we loop. Each time ($t = 0, 1, 2, \ldots$) the encoder receives a new datum $x$, we calculate three prediction errors:[2]

$$e^{(0)} = x - x_{\mathrm{prev}}$$
$$e^{(1)} = e^{(0)} - e^{(0)}_{\mathrm{prev}}$$
$$e^{(2)} = e^{(1)} - e^{(1)}_{\mathrm{prev}}$$

that correspond to the aforementioned three extrapolations,

[2] Unused variables are initialized to zero.

Table 3: Length-limited minimum redundancy code for Gaussian distribution

| $\lvert e_{t-1}\rvert + \cdots + \lvert e_{t-16}\rvert$ | $\ell_0, \ldots, \ell_{15}$ |
|---|---|
| 0–9 | 1 2 3 4 7 7 7 7 8 8 8 8 8 8 8 8 |
| 10–17 | 2 1 3 4 6 7 8 8 8 8 8 8 8 8 8 8 |
| 18–23 | 3 1 2 4 6 7 8 8 8 8 8 8 8 8 8 8 |
| 24–31 | 3 2 1 4 6 7 8 8 8 8 8 8 8 8 8 8 |
| 32–37 | 4 2 1 3 6 7 8 8 8 8 8 8 8 8 8 8 |
| 38–56 | 3 2 2 2 4 6 7 7 8 8 8 8 8 8 8 8 |
| 57–66 | 4 2 2 2 3 6 7 7 8 8 8 8 8 8 8 8 |
| 67–100 | 4 3 2 2 2 6 7 7 8 8 8 8 8 8 8 8 |
| 101–114 | 4 4 2 2 2 4 6 6 8 8 8 8 8 8 8 8 |
| 115–138 | 4 3 3 2 2 3 6 6 8 8 8 8 8 8 8 8 |
| 139–190 | 6 4 3 2 2 2 7 7 8 8 8 8 8 8 8 8 |
| 191–230 | 6 4 4 2 2 2 4 6 8 8 8 8 8 8 8 8 |
| 231–310 | 6 4 3 3 2 2 3 6 8 8 8 8 8 8 8 8 |
| 311–438 | 6 6 5 3 2 2 2 5 8 8 8 8 8 8 8 8 |
| 439–623 | 6 6 4 3 3 2 2 3 8 8 8 8 8 8 8 8 |
| 624–879 | 8 6 6 5 3 2 2 2 5 8 8 8 8 8 8 8 |
| 880–1249 | 8 6 6 4 3 3 2 2 3 8 8 8 8 8 8 8 |
| 1250–1762 | 8 8 6 6 5 3 2 2 2 5 8 8 8 8 8 8 |
| 1763–2502 | 8 8 6 6 4 3 3 2 2 3 8 8 8 8 8 8 |
| 2503–3526 | 8 8 8 6 6 5 3 2 2 2 5 8 8 8 8 8 |
| 3527–5007 | 8 8 8 6 6 4 3 3 2 2 3 8 8 8 8 8 |
| 5008–7055 | 8 8 8 8 6 6 5 3 2 2 2 5 8 8 8 8 |
| 7056–10018 | 8 8 8 8 6 6 4 3 3 2 2 3 8 8 8 8 |
| 10019–14113 | 8 8 8 8 8 6 6 5 3 2 2 2 5 8 8 8 |
| 14114–20040 | 8 8 8 8 8 6 6 4 3 3 2 2 3 8 8 8 |
| 20041–28229 | 8 8 8 8 8 8 6 6 5 3 2 2 2 5 8 8 |
| 28230–40084 | 8 8 8 8 8 8 6 6 4 3 3 2 2 3 8 8 |
| 40085–56460 | 8 8 8 8 8 8 8 6 6 5 3 2 2 2 5 8 |
| 56461–80172 | 8 8 8 8 8 8 8 6 6 4 3 3 2 2 3 8 |
| 80173–112829 | 8 8 8 8 8 8 8 8 6 6 5 3 2 2 2 5 |
| 112830–149277 | 8 8 8 8 8 8 8 8 6 6 4 3 3 2 2 3 |
| 149278–205656 | 8 8 8 8 8 8 8 8 7 7 6 4 3 2 2 2 |
| 205657– | Output raw 16-bit value |

Table 4: Length-limited minimum redundancy code for Laplace distribution

| $\lvert e_{t-1}\rvert + \cdots + \lvert e_{t-16}\rvert$ | $\ell_0, \ldots, \ell_{15}$ |
|---|---|
| 0–13 | 1 2 3 4 6 7 8 8 8 8 8 8 8 8 8 8 |
| 14–22 | 2 1 3 4 6 7 8 8 8 8 8 8 8 8 8 8 |
| 23–37 | 2 2 2 3 4 6 7 7 8 8 8 8 8 8 8 8 |
| 38–60 | 3 2 2 2 4 6 7 7 8 8 8 8 8 8 8 8 |
| 61–75 | 4 2 2 2 3 6 7 7 8 8 8 8 8 8 8 8 |
| 76–99 | 3 3 2 2 3 4 6 6 8 8 8 8 8 8 8 8 |
| 100–163 | 4 3 3 2 2 3 6 6 8 8 8 8 8 8 8 8 |
| 164–203 | 4 4 3 2 2 3 4 5 8 8 8 8 8 8 8 8 |
| 204–301 | 6 4 3 3 2 2 3 6 8 8 8 8 8 8 8 8 |
| 302–397 | 5 4 4 3 2 2 3 4 8 8 8 8 8 8 8 8 |
| 398–451 | 5 4 4 3 3 2 2 4 8 8 8 8 8 8 8 8 |
| 452–608 | 6 5 5 3 3 2 2 3 6 7 8 8 8 8 8 8 |
| 609–794 | 6 6 4 4 3 2 2 3 4 7 8 8 8 8 8 8 |
| 795–910 | 6 6 4 4 3 3 2 2 4 7 8 8 8 8 8 8 |
| 911–1216 | 7 6 5 5 3 3 2 2 3 6 8 8 8 8 8 8 |
| 1217–1584 | 7 6 6 4 4 3 2 2 3 4 8 8 8 8 8 8 |
| 1585–1820 | 8 6 6 4 4 3 3 2 2 4 7 8 8 8 8 8 |
| 1821–2432 | 8 7 6 5 5 3 3 2 2 3 6 8 8 8 8 8 |
| 2433–3169 | 8 7 6 6 4 4 3 2 2 3 4 8 8 8 8 8 |
| 3170–3641 | 8 8 6 6 4 4 3 3 2 2 4 7 8 8 8 8 |
| 3642–4865 | 8 8 7 6 5 5 3 3 2 2 3 6 8 8 8 8 |
| 4866–6816 | 8 8 7 6 6 4 4 3 2 2 3 4 8 8 8 8 |
| 6817–9730 | 8 8 8 7 6 5 5 3 3 2 2 3 6 8 8 8 |
| 9731–13633 | 8 8 8 7 6 6 4 4 3 2 2 3 4 8 8 8 |
| 13634–19461 | 8 8 8 8 7 6 5 5 3 3 2 2 3 6 8 8 |
| 19462–27266 | 8 8 8 8 7 6 6 4 4 3 2 2 3 4 8 8 |
| 27267–38921 | 8 8 8 8 8 7 6 5 5 3 3 2 2 3 6 8 |
| 38922–54498 | 8 8 8 8 8 7 6 6 4 4 3 2 2 3 4 8 |
| 54499–77220 | 8 8 8 8 8 8 7 6 5 5 3 3 2 2 3 6 |
| 77221–103705 | 8 8 8 8 8 8 7 6 6 4 4 3 2 2 3 4 |
| 103706–154498 | 8 8 8 8 8 8 8 8 6 6 4 3 3 2 2 3 |
| 154499–207725 | 8 8 8 8 8 8 8 8 7 7 6 4 3 2 2 2 |
| 207726– | Output raw 16-bit value |

and determine $d$ according to

$$d = \begin{cases} e^{(0)} & (\text{if } s^{(0)} \le s^{(1)}) \\ e^{(1)} & (\text{if } s^{(0)} > s^{(1)} \le s^{(2)}) \\ e^{(2)} & (\text{otherwise}) \end{cases}$$

We also determine $z$ according to

$$z = \begin{cases} s^{(0)} & (\text{if } s^{(0)} \le s^{(1)}) \\ s^{(1)} & (\text{if } s^{(0)} > s^{(1)} \le s^{(2)}) \\ s^{(2)} & (\text{otherwise}) \end{cases}$$

We then look up the code table corresponding to $z$, and encode $d$.

Finally, with $p = t \bmod 16$, we update the variables by

$$s^{(0)} \leftarrow s^{(0)} - d_p^{(0)} + |e^{(0)}|$$
$$s^{(1)} \leftarrow s^{(1)} - d_p^{(1)} + |e^{(1)}|$$
$$s^{(2)} \leftarrow s^{(2)} - d_p^{(2)} + |e^{(2)}|$$

and

$$d_p^{(0)} \leftarrow |e^{(0)}|, \quad d_p^{(1)} \leftarrow |e^{(1)}|, \quad d_p^{(2)} \leftarrow |e^{(2)}|$$
$$e_{\text{prev}}^{(0)} \leftarrow e^{(0)}, \quad e_{\text{prev}}^{(1)} \leftarrow e^{(1)}$$

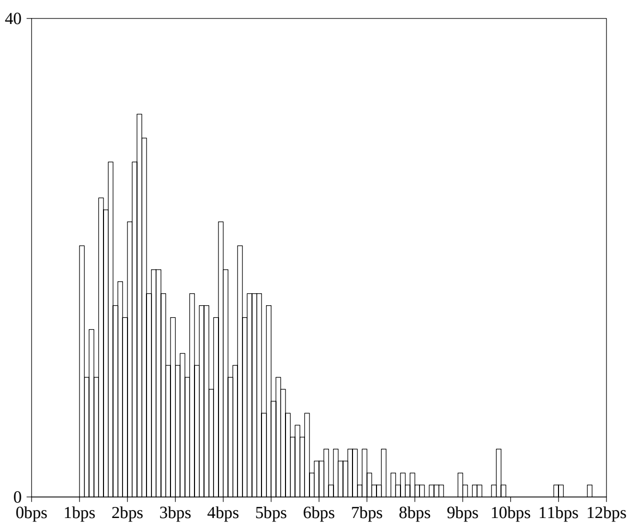


Figure 1: Histogram of compression performances (bits/sample) for 1620 net channels (405 channels $\times$ 4 files), assuming Gaussian distribution. (The histogram for Laplace distribution is almost identical.) Ordinate: compression (bits/sample), Abscissa: number of net channels.

## 3 PERFORMANCE AND CONCLUSION

Figure 1 shows the histogram of compressed sizes (bits/sample) of 1620 net channels for randomly chosen four laboratory files each containing 405 channels of raw 16-bit A/D converter outputs. It can be seen that almost all of the channels are compressed to 1/16–1/2 of the original size.

Table 5 shows the compressed sizes and execution speeds of our standalone compression tool *nifsq* and two popular tools *Zip* and *LHA* for two representative laboratory files (405-channel 16-bit data as described above), on a 400MHz Pentium-II PC running Linux.

Table 5: Comparison of compressed sizes and compression/decompression wall-clock times of *nifsq* and two popular compression tools.

| | | Size (bytes) | Comp. (secs) | Decomp. (secs) |
|---|---|---|---|---|
| File-263 | | 8743652 | | |
| *Zip* | | 4794261 | 6.31 | 1.29 |
| *LHA* | | 4769586 | 9.19 | 1.86 |
| *nifsq* | Gaussian | 2003136 | 2.25 | 2.04 |
| | Laplace | 2000022 | | |
| File-318 | | 18566522 | | |
| *Zip* | | 9822477 | 14.56 | 2.70 |
| *LHA* | | 9807894 | 19.94 | 3.92 |
| *nifsq* | Gaussian | 3956084 | 4.94 | 4.38 |
| | Laplace | 3958382 | | |

Although the current version of *nifsq* (and its library version *nifsqlib*) is not sufficiently optimized for speed,[3] it is sufficiently fast, and compresses better.

We conclude that we succeeded in constructing a compression tool/library suitable for online compression of laboratory data (raw A/D converter outputs, to be more exact). Its compression is tighter and faster than currently-available popular tools.

The source code is available at http://www.matsusaka-u.ac.jp/˜okumura/nifsq/.

## REFERENCES

[1] H. Okumura *et al*, “A Scalable Data Acquisition System for Superconducting Coil Experiment”, K. Herschbach, W. Maurer, J.E. Vetter (editors), *Fusion Technology 1994: Proceedings of the 18th Symposium on Fusion Technology, Karlsruhe, Germany, 22–26 August 1994*, Elsevier Science, 1995. pp. 835–838.

[2] J. Kariya *et al*, “Java Based Data Monitoring and Management System for LHD”, PCaPAC’99, KEK, Jan. 1999.

[3] M. Emoto *et al*, “Interactive Data Visualization with Java 3D”, PCaPAC’99, KEK, Jan. 1999.

[3] Our code is entirely written in C, whereas *Zip* and *gzip* use assembly-language code for *x86* platforms.