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: 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.

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

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

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

\[
\hat{x}_t^{(0)} = x_{t-1}, \quad \text{previous value}
\]
\[
\hat{x}_t^{(1)} = 2x_{t-1} - x_{t-2}, \quad \text{linear extrapolation}
\]
\[
\hat{x}_t^{(2)} = 3x_{t-1} - 3x_{t-2} + x_{t-3}, \quad \text{quadratic extrapolation}
\]

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 \( |Y + 0.5| - |X + 0.5| \), where \( X \) and \( Y \) are two random (undiscretized) variables such that \( X - |X| \) 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
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 numerical calculation assuming Gaussian (Table 3) and Laplace (Table 4) distributions.

For example, if \( \varepsilon = 400 \) 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 ‘01101’ 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:

\[
\begin{align*}
    e^{(0)} &= x - x_{\text{prev}} \\
    e^{(1)} &= e^{(0)} - e_{\text{prev}} \\
    e^{(2)} &= e^{(1)} - e_{\text{prev}}
\end{align*}
\]

that correspond to the aforementioned three extrapolations,

\(^2\)Unused variables are initialized to zero.
and determine \( d \) according to
\[
d = \begin{cases} 
  e^{(0)} & \text{(if } s^{(0)} \leq s^{(1)} \text{)} \\
  e^{(1)} & \text{(if } s^{(0)} > s^{(1)} \leq s^{(2)} \text{)} \\
  e^{(2)} & \text{(otherwise)}
\end{cases}
\]
We also determine \( z \) according to
\[
z = \begin{cases} 
  s^{(0)} & \text{(if } s^{(0)} \leq s^{(1)} \text{)} \\
  s^{(1)} & \text{(if } s^{(0)} > s^{(1)} \leq s^{(2)} \text{)} \\
  s^{(2)} & \text{(otherwise)}
\end{cases}
\]
We then look up the code table corresponding to \( z \), and encode \( d \).

Finally, with \( p = t \mod 16 \), we update the variables by
\[
\begin{align*}
  s^{(0)} & \leftarrow s^{(0)} - d_p^{(0)} + |e^{(0)}| \\
  s^{(1)} & \leftarrow s^{(1)} - d_p^{(1)} + |e^{(1)}| \\
  s^{(2)} & \leftarrow s^{(2)} - d_p^{(2)} + |e^{(2)}|
\end{align*}
\]
and
\[
\begin{align*}
  d_p^{(0)} & \leftarrow |e^{(0)}|, \\
  d_p^{(1)} & \leftarrow |e^{(1)}|, \\
  d_p^{(2)} & \leftarrow |e^{(2)}| \\
  e^{(0)}_{\text{prev}} & \leftarrow e^{(0)}, \\
  e^{(1)}_{\text{prev}} & \leftarrow e^{(1)}
\end{align*}
\]

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


\(^3\)Our code is entirely written in C, whereas Zip and gzip use assembly-language code for x86 platforms.