高変動時系列整数データの圧縮手法

柴田　秀哉†　　高山　茂伸†

† 三菱電機株式会社 情報技術総合研究所　〒 247-8501 神奈川県鎌倉市大船 5-1-1
E-mail: †{Shibata.Hideya@cb,Takayama.Shigenobu@db}.MitsubishiElectric.co.jp

あらまし　本稿では，センサデータのように時系列として生成される整数データ列に特化した圧縮手法を提案する．時系列整数データはある程度の連続性を持つ傾向にあるため，その連続性を利用してデータを圧縮するのが一般的である．しかしながら，高周波ノイズを含む信号データなど，変動が大きいデータに関しては，圧縮が困難という課題があった．提案手法では，ビット分割の手法を様々な符号化方式と併用することで，変動が大きい整数データ列に対し，従来手法より高い圧縮率が得られる．提案手法を実装した圧縮ライブラリと汎用圧縮ライブラリである zlib とを比較した結果，圧縮，伸張速度を同程度に保ちつつ，最大で zlib の 66%程度に圧縮することができた．

キーワード　データ圧縮，整数列圧縮，センサデータ，時系列データ，高変動

1. はじめに

近年，センシング技術の高度化を背景とし，多様な機器から大量のセンサデータが生成され，収集される傾向にある．工場や商業ビル等の各種設備からデータを収集することで，機器，設備の予防保全，遠隔監視等に活用する，といったセンサデータ活用事例が急速に増加している．

大量のデータを活用するに当たっては，通信データ，蓄積データの増大が大きな課題となる．例えば，通信データ量増加に伴う通信遅延の増大は，実時間的な処理を要求するシステムに支障をきたす可能性がある．また，通信回線費用に制約が掛けられているような状況においては，限られた回線の範囲内で必要なデータが通信可能なことが求められる．蓄積に関しても同様であり，データ量増加に伴うストレージコストの増大や処理速度の低下が課題となる．

このような問題を回避する方法の 1 つとして，データ圧縮がある．データ圧縮により通信データや蓄積データの増大を抑制することができ，データ活用の一層の促進が期待される．

データ圧縮に関しては，これまで様々な研究結果が報告されているが，数値データ列の圧縮については，いくつか課題が残されている．その中の 1 つに，変動が大きい数値データ列の圧縮がある．センサデータでは，数値データの占める割合が大きい．例えば，温度，湿度，消費電力量，電圧，電流，圧力，加速度といった情報は全て数値データである．このような数値データの中には，機器の状態変化に応じて，値が大きく変動するものが多く含まれる．また，高周波ノイズや測定誤差等の要因から高変動となる例も多い．一般に，時系列数値データを圧縮する際は，データ列にある程度の連続性を仮定し，連続性を利用した圧縮を行う．しかしながら，高変動な数値データ列については，連続性の低さから圧縮が困難となる．

このような背景から，本稿では，変動が大きい整数列に対する可逆圧縮手法を提案する．提案手法では，図 1 に示すような，固定長バイナリで表現される符号付き整数型から成る可変長データ列を対象とする．実用的には，多くのセンサデータが実数値データであり，整数のみで表現できることは少ないが，固定小数点数であれば，データ列全体に一律で固定の値を乗じることで，実数列を整数列として扱うことができる．浮動小数点数に対しては，整数とは異なる仕組みが必要となるが，本稿では対象外とする．

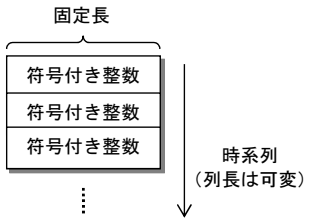


図 1　圧縮対象とする整数列

本稿の構成は以下のとおりである．2 節で関連研究について述べる．3 節では，時系列整数データ圧縮について整理した上で，高変動な整数列に対する圧縮手法を提案する．4 節では，提案手法を実装した圧縮ライブラリについて説明する．5 節では，提案手法に対する評価結果を報告する．評価においては，OSS の汎用圧縮ライブラリである zlib, libbzip2, lzma との性能比較を実施する．最後に，6 節で本稿のまとめを行う．

2. 関連研究

本節では，整数列圧縮，時系列データ圧縮に関する関連研究を紹介する．

2.1　整数列圧縮

整数データ列の圧縮は，整数列圧縮，整数圧縮などと呼ばれ，汎用データ圧縮とは異なる手法が多数提案されている．以下，整数列圧縮の既存研究を概観する．

最も単純な符号化方式は単進符号（アルファ符号）である．実用的かつ最も古くから知られている符号化方式は，単進符号の応用であるガンマ符号，デルタ符号 [1]，ゴロム符号 [2]，ライス符号 [3] である．ライス符号は，ゴロム符号の特別な場合であり，ゴロム・ライス符号と呼ばれることもある．これらの

符号化方式は，1 つの整数を 1 つの符号に変換する方式であり，その出力はビット単位となる．圧縮，伸張処理において，ビット単位のデータ処理を行うことは処理速度の低下を招くとして，近年では，符号化結果の出力単位がバイト単位，ワード単位となるような方式が複数提案されている．

符号化結果をバイト単位に生成する方式として，Variable-Byte [4]，Varint-G8IU [5] がある．Variable-Byte は，整数を 1 つずつ符号化する方式であり，ガンマ符号等と比較し圧縮率が低くなる傾向にある．Varint-G8IU は Variable-Byte を改良方式であり，複数の整数をまとめて符号化することで，圧縮率，速度の両面で性能向上を図っている．

符号化結果をワード単位に生成する方式としては，Simple9 [6]，Simple16 [7]，Simple-8b [8] 等の Simple 系符号や PFOR [9] などが知られている．PFOR については，種々の改良が報告されている [7], [10]．これらの符号化方式に見られるように，近年の整数列圧縮では，CPU 特性を活かして高速化を追求する傾向が顕著である．

2.2 時系列データ圧縮

前項で述べた各種符号化方式は，正整数の列のみを対象としている．このような背景として，整数列圧縮の主な適用先がソート済み整数列と考えられている点が挙げられる．ソート済み整数列の場合，前回値との差分を取った残差列を生成することにより，正整数のみから成る列が得られる．更に，元のデータ列がソート済みであるため，得られる残差列は，比較的小さい値から成る傾向にあり，圧縮が容易なケースが多い．

しかしながら，センサデータのような時系列においては，データの発生順序が重要であるため，データの順序を入れ替えて符号化することはできない．このように，ソートが許されない時系列整数データに対する符号化方式の先行事例として，データベース向けに開発された RID 符号化 [11]，DGR 符号化 [12] がある．RID 符号，DGR 符号の考え方は，提案手法の動機付けとなっているため，それぞれの方式について説明する．

2.2.1 RID 符号化

RID 符号化の名称は，連長符号化（Run-length encoding），インデックス符号化（Index encoding），差分符号化（Difference encoding）のそれぞれの頭文字を取ったものであり，文字通り，それぞれの方式の組合せにより圧縮を実現する符号化方式である．RID 符号化は，元々データウェアハウス向けに開発された方式であり，整数列のみを対象とした手法ではないが，整数列への応用が可能である．以下では，符号付き整数列 $\{d_i\}_{i=0}^{n}$ に RID 符号化を適用する場合を例に説明する．

RID 符号化では，元データ列 $\{d_i\}_{i=0}^{n}$，及び元データ列において前回値との差分を取った残差列 $\{r_i\}_{i=0}^{n}$ を対象に，それぞれの符号表を作成する．ここで，

$$r_i = \begin{cases} d_0 & (i = 0 \text{ の場合}) \\ d_i - d_{i-1} & (i > 0 \text{ の場合}) \end{cases} \tag{1}$$

である．符号表には，重複除去した一意の整数のみが含まれ，各整数は 0 で始まる添字で表現される．符号表の大きさは，それぞれのデータ列に現れる整数の種類の数に等しい．RID 符号化では，各整数を符号表の添字として固定長ビット幅で表現する．添字を表現可能なビット幅は符号表の大きさに依存するため，データ列に現れる整数の種類が少ないほど，添字を表現するビット幅が小さくなり圧縮率が高まる．更に，連続する値については連長符号化を適用する．RID 符号化では，全ての連長を固定長で表現する．この際，圧縮後サイズが最小となるような最適なビット幅を選択する．特に，ビット幅 0 を選択した場合は，連長圧縮を適用しない．最後に，元データ列と残差列とで，圧縮率が高い方を選択する．

RID 符号化は 2 パス符号である．すなわち，1 パス目で最も圧縮可能なパラメータを選択し，2 パス目に符号化を行う．具体的には，1 パス目で符号表を作成し，圧縮パラメータとして連長を表現するビット幅，及び差分符号化の適用有無を選択する．2 パス目では，選択したパラメータ，作成した符号表に基づき符号化を実施する．

RID 符号化は，符号化対象データ列に現れる整数の種類が少ないような場合に有効な方式である．差分符号化は，データ列に現れる整数の種類を減少させる効果を期待したものである．実際，整数列に差分符号化を適用した場合，元データ列の変動がそれほど大きくなければ，残差列はより 0 に近い値に集中する．これにより，元データ列よりも整数の種類を少なくできる場合が多い．

RID 符号化により圧縮されたデータは，図 2 に示すような構造を取る．ヘッダには，圧縮方式を決定するパラメータが格納される．

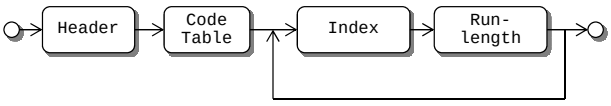


図 2　RID 符号方式による圧縮データの構造

2.2.2 DGR 符号化

DGR 符号化の名称は，差分符号化（Difference encoding），ガンマ符号化（Elias Gamma encoding），連長符号化（Run-length encoding）のそれぞれの頭文字を取ったものであり，それぞれの方式の組合せにより圧縮を実現する符号化方式である．以下では，符号付き整数列 $\{d_i\}_{i=0}^{n}$ に DGR 符号化を適用する場合を例に説明する．

DGR 符号化では，式 (1) で定義される残差列 $\{r_i\}_{i=0}^{n}$ に対してガンマ符号を適用する．但し，$\{r_i\}_{i=0}^{n}$ は符号付きの整数列であるため，各残差 r_i を正整数へ写す．写像は

$$f : \mathbb{Z} \to \mathbb{N}^{+}, \quad r \mapsto \begin{cases} 2r & (r > 0 \text{ の場合}) \\ 2|r| + 1 & (r < 0 \text{ の場合}) \\ 1 & (r = 0 \text{ の場合}) \end{cases} \tag{2}$$

で定義される．写像 f は，絶対値が小さい整数ほど，より小さい正整数へと割り当てる．

更に，DGR 符号化では連長符号化を併用するが，この連長

符号化の適用方法に特徴がある．比較的変動が小さいデータ列については，連長符号化は有効な場合が多いが，変動が大きいデータ列になると，連長が 1 であるケースが多くなり，却って圧縮率が低下する傾向にある．このような問題を回避するため，DGR 符号化では，連長符号化の有無を残差の値によって切り替える．また，連長についても，小さい値の出現頻度が高いためガンマ符号を適用する．具体的には，残差 r が m 回連続して現れる場合，DGR 符号化は，次の 3 種類の方式から，最も圧縮率が高い方式を選択する．

- 方式 G: 連長符号化を使用せず，残差 r の符号化結果 $Gamma \circ f(r)$ を m 回出力する．
- 方式 G0R: $r = 0$ の場合は連長符号化を使用し，$Gamma \circ f(r)$ と $Gamma(m)$ を出力する．$r \neq 0$ の場合は，連長符号化を使用せず，残差 r の符号化結果 $Gamma \circ f(r)$ を m 回出力する．
- 方式 G01R: $|r| \leq 1$ の場合は連長符号化を使用し，$Gamma \circ f(r)$ と $Gamma(m)$ を出力する．$|r| > 1$ の場合は，連長符号化を使用せず，残差 r の符号化結果 $Gamma \circ f(r)$ を m 回出力する．

ここで，$Gamma()$ はガンマ符号による写像である．

DGR 符号化は 2 パス符号である．1 パス目で連長符号化の適用方式を決定し，2 パス目で符号化を実施する．DGR 符号化は，緩やかに変動するようなデータ列に対して有効な方式であるが，変動が大きくなると，急激に圧縮率が低下する傾向がある．

DGR 符号化により圧縮されたデータは，図 3 に示すような構造を取る．ヘッダには，圧縮方式を決定するパラメータが格納される．DGR 符号化では，RID 符号化と異なり符号表を必要としない．

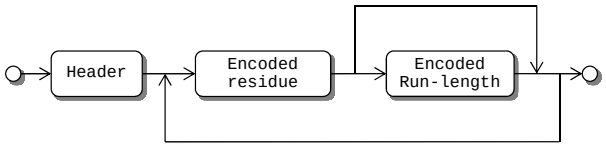


図 3　DGR 符号方式による圧縮データの構造

3. 提 案 手 法

本節では，前節で説明した RID 符号化や DGR 符号化の考え方を元にしつつ，より高度な整数列圧縮の手法を適用することで，変動が大きい整数列へも適用可能な圧縮手法を提案する．

3.1 問題の定式化

本稿で扱う問題は，以下の条件を満たす．

- 符号化対象データは，固定長バイナリで表現される符号付き整数型から成る可変長データ列である（図 1 参照）．
- 圧縮手法は可逆である．データ伸張時はデータの順序も含めて再現可能である．

3.2 符号化方式の整理

提案手法の説明の前に，RID 符号化や DGR 符号化の考え方を元に，時系列整数データの符号化方式について整理する．多くの符号化方式は，図 4 に示すように，予測，残差生成，残差符号化，連長符号化，といった手順で一般化することが可能である．以下，それぞれの手順について説明する．

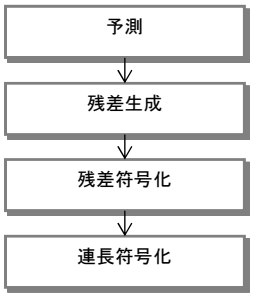


図 4　時系列整数データ符号化方式の一般化

3.2.1 予　　測

符号化対象のデータ列 $\{d_i\}_{i=0}^{n}$ を考える．このとき，d_i を予測するとは，d_i 以前の履歴データ $\{d_0, \ldots, d_{i-1}\}$ からある値 p_i を導くことである．差分符号化は，予測値として前回値を採用しているので，

$$p_i = \begin{cases} 0 & (i = 0 \text{ の場合}) \\ d_{i-1} & (i > 0 \text{ の場合}) \end{cases} \tag{3}$$

である．差分符号化を行わずに，元データ d_i を直接符号化する場合は，$p_i = 0$ と言い換えることできる．一般には，予測値 p_i を決定する方法として，線形予測など様々な方法を取り得る．

3.2.2 残 差 生 成

通常，d_i に対する残差 r_i は，d_i に対する予測値 p_i を用いて

$$r_i = d_i - p_i \tag{4}$$

である．d_i が整数の場合は，単純に計算機上で整数減算を行えばよいが，d_i が浮動小数点数の場合は，単純な減算では情報落ちが発生し非可逆な符号化方式となってしまうため，減算を行うための工夫が必要となる．

3.2.3 残差符号化

生成された残差は一般に正整数とは限らないため，正整数を対象とするような符号化方式を適用する場合，残差を正整数へ変換する前処理が必要となる．前処理の最も単純な方法は，DGR 符号化のように，式 (2) を利用することである．他の例としては，RID 符号化のようにインデックス符号化を利用する場合，残差を符号表の添字へ変換する作業を正整数への写像と見做すことができる．

正整数へ変換された残差を符号化する方式として，2.1 節で紹介したような種々の方式を適用することが可能である．例えば，RID 符号化では，残差に対応する添字を固定長のまま出力しているが，符号表を整数の出現頻度によりソートすることで，出現頻度が高い整数により短い符号を割り当てる，という改良が有り得る．DGR 符号化方式についても，符号化方式はガンマ符号のみでなく，他の符号化方式を取り得る．

但し，Simple9 や PFOR のように，複数の整数をまとめて符号化するような方式については，連長符号化との併用方法が自明ではないため，何らかの工夫が必要となる．

3.2.4 連長符号化

同一の残差が連続する場合は，連長符号化を適用することが可能である．この際，DGR 符号化のように，残差の値によって，連長符号化の適用有無を動的に切り替えることが可能である．連長自体についても，小さい値が高頻度で現れることが期待されるので，ガンマ符号のような符号化方式により，符号化することができる．

3.3 高変動整数列への対応

高変動整数列へ対応するためには，以下の 2 つアプローチが考えられる．

- 予測を工夫することにより，変動が大きい整数列に対しても，残差が小さくなるようにする．
- 残差符号化を工夫することにより，残差が大きい値であっても，高い圧縮率となるようにする．

本稿では，残差符号化を工夫するアプローチを取る．予測の改良については，今後の課題である．

変動が大きい整数列では，整数の下位ビットはランダムなノイズと見做すことができる．すなわち，下位ビットだけを見ると圧縮が原理的に不可能なため，固定長バイナリとして出力するのが最も効率的である．一方，上位ビットについては，ある程度の連続性を期待することができる．従って，高変動な整数列の圧縮率を高めるためには，上位ビットと下位ビットを分割して，個別に符号化することが有効である．この考え方は，ゴロム・ライス符号に酷似している．ゴロム・ライス符号では，上位ビットを単進符号で符号化しているが，この部分を任意の整数符号化方式に置き換えることで，より効率的な圧縮が可能と期待できる．以下，上位ビットと下位ビットを分割して符号化する方式を総称して，ビット分割方式と呼ぶ．ビット分割のイメージを図 5 に示す．

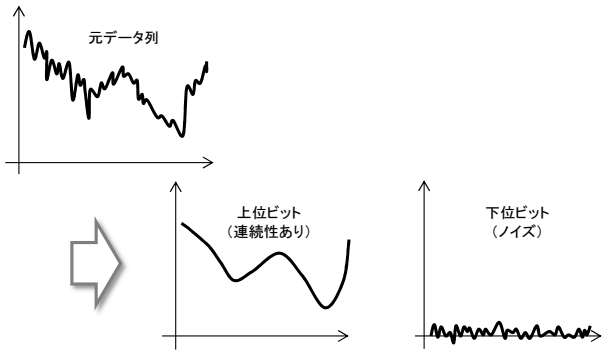


図 5　ビット分割のイメージ

提案手法では，ビット分割方式を，インデックス符号化を利用する方式と，インデックス符号化を利用せずに直接残差を符号化する方式のそれぞれに適用する．インデックス符号化にビット分割を適用する場合は，ビット分割により得られた上位ビットに対して符号表を作成する．

ビット分割方式では，ビットの分割位置がパラメータとなる．2 パス符号の中でビット分割方式を採用すれば，最適なビット分割位置を選択することが可能となる．但し，2 パス符号では圧縮処理の速度性能が低下するため，どの程度パラメータに自由度を持たせるかにより，圧縮速度と圧縮率のトレードオフが発生する．

提案手法において，圧縮データは図 6 に示す構造となる．

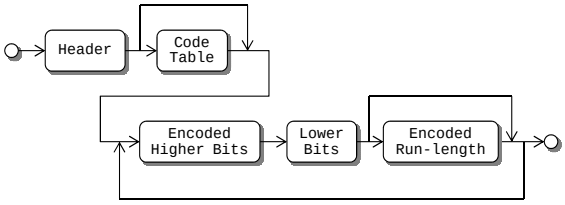


図 6　提案手法による圧縮データの構造

4. 圧縮ライブラリの実装

前節で説明した符号化方式を 32 ビット整数列の圧縮ライブラリとして実装したので，その内容について説明する．

4.1 圧縮 API

圧縮処理時の API 呼出し手順を図 7 に示す．

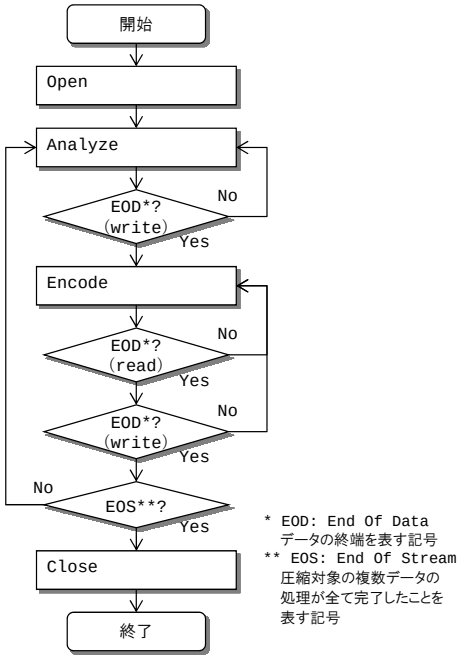


図 7　圧縮 API 呼出し手順

本圧縮ライブラリは，2 パス符号を採用しており，Analyze 関数が 1 パス目，Encode 関数が 2 パス目にそれぞれ対応している．インデックス符号化を利用する方式では，Analyze 関数にて符号表を作成する．扱うデータサイズが可変長であるため，Analyze 関数，Encode 関数ではそれぞれデータの分割入力，分割出力が可能である．また，Open 関数での指定により，Analyze 関数にて試行する符号化方式を制限することが可能である．

搭載している符号化方式については，4.3 節で説明する．

4.2 伸張 API

伸張処理時の API 呼出し手順を図 8 に示す．伸張処理についても扱うデータサイズが可変長であるため，Decode 関数ではデータの分割入力，分割出力が可能である．

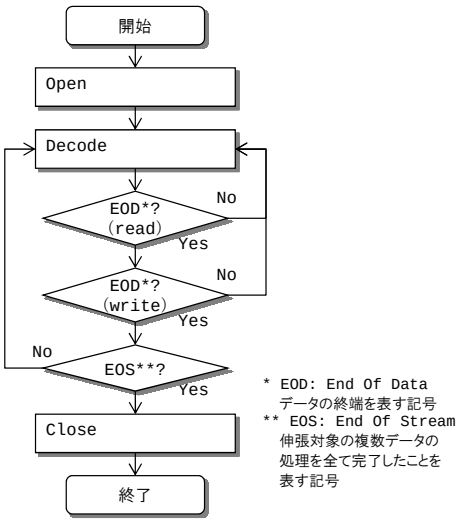


図 8　伸張 API 呼出し手順

4.3 符号化方式

本圧縮ライブラリに実装した符号化方式は次の通りである．

- **予測，残差生成**

単純な差分予測（式 (3)）と，予測を用いない方式（$p_i = 0$）の両方から選択する．残差生成は式 (4) により行う．

- **残差符号化**

インデックス符号化の適用有無により 2 種類に大別される．インデックス符号化を適用しない方式では，残差を式 (2) により正整数へ変換し，ビット分割を行った後，上位ビットをガンマ符号，デルタ符号のいずれかで符号化する．下位ビットは固定長符号として出力する．

インデックス符号化を適用する方式では，ビット分割を行った後，上位ビットの符号表を作成し，符号表の添字をガンマ符号，デルタ符号，固定長符号のいずれかで符号化する．下位ビットは固定長符号として出力する．

いずれの方式についても，ビット分割位置は下位ビットから数えて 0 ビットから 30 ビットを試行する．

- **連長符号化**

連長符号適用無，常に適用，残差が 0 のときのみ適用（DGR 符号化方式の G0R 方式に対応）の 3 種類から選択する．

5. 評　　価

本節では，提案手法を実装した圧縮ライブラリと，汎用圧縮ライブラリである zlib, libbzip2, lzma との性能比較を実施した結果を報告する．

5.1 評 価 方 針

高変動整数列となるようなセンサデータを模擬した人工データにより評価を行う．整数列 $\{d_i\}_{i=0}^{n}$ を，

$$d_i = \lfloor m + a_1 \sin(2\pi f_1 i) + a_2 \sin(2\pi f_2 i) + a_3 X \rfloor \qquad (5)$$

により生成する．ここで，m, a_1, a_2, a_3, f_1, f_2 は定数パラメータであり，X は正規分布 $\mathcal{N}(0,\ v)$ に従う確率変数である．分散 v もまた定数パラメータである．図 9 に $m = 0$, $a_1 = 100$, $a_2 = 10$, $a_3 = 100$, $f_1 = 0.002$, $f_2 = 0.2$, $v = 0.2$, $n = 1000$ としたときの例を示す．

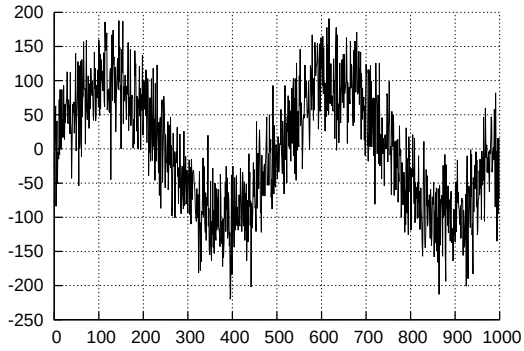


図 9　評価データ例

式 (5) により生成されるデータ列に対する圧縮率，圧縮速度，伸張速度を評価する．評価は，いくつかのパラメータの組に対して実施する．なお，本評価では，圧縮処理，伸張処理の CPU 時間のみを計測する．ディスク I/O 等，その他の処理は測定対象に含めない．

評価では，圧縮率と圧縮速度とのトレードオフを考慮し，4.3 節で述べた残差符号化のうち，インデックス符号化を適用しない方式のみを試行する．

5.2 評 価 条 件

評価条件を以下に記す．

- **評価指標**

圧縮率，圧縮速度，伸張速度をそれぞれ以下で定義する．

$$\text{圧縮率} = 1 - \text{圧縮後サイズ}/\text{圧縮前サイズ} \qquad (6)$$

$$\text{圧縮速度} = \text{圧縮前サイズ}/\text{圧縮 CPU 処理時間} \qquad (7)$$

$$\text{伸張速度} = \text{圧縮前サイズ}/\text{伸張 CPU 処理時間} \qquad (8)$$

- **評価パラメータ**

評価では，$m = 0$, $f_1 = 0.0005$, $f_2 = 0.05$, $v = 1$, $n = 1000000$ を固定し，パラメータ a_1, a_2, a_3 を変化させる．パラメータ a_1, a_2, a_3 の組合せを表 1 に示す．

表 1　評価パラメータ

	a_1	a_2	a_3
パターン 1	1,000	100	0
パターン 2	1,000	100	10
パターン 3	1,000	100	100
パターン 4	1,000	100	1,000
パターン 5	10,000	1,000	100
パターン 6	100,000	10,000	1,000
パターン 7	1,000,000	100,000	10,000

パターン 1 からパターン 4 では，ノイズの振幅であるパラ

メータ a_3 のみが変化する．ノイズの大きさが性能に与える影響を調査することが目的である．一方，パターン 2 と，パターン 5 からパターン 7 では，データ全体の振幅が変化する．すなわち，パラメータ a_1，a_2，a_3 の比率は一定であるため，純粋に，データの変動幅が圧縮率に与える影響を調べることができる．

- **評価環境**

評価で使用するマシンの詳細を表 2 に示す．

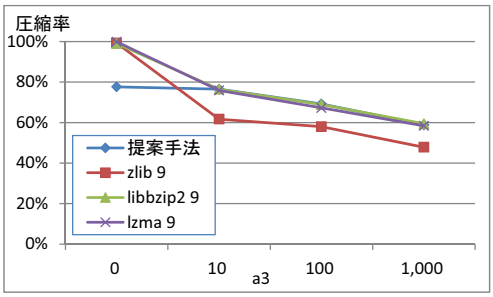
表 2　評 価 環 境

OS	Windows 7 Professional（x64）
CPU	Intel Core i5-4690 3.50GHz（4 コア） （1 コアのみ使用）
メモリ	16GB

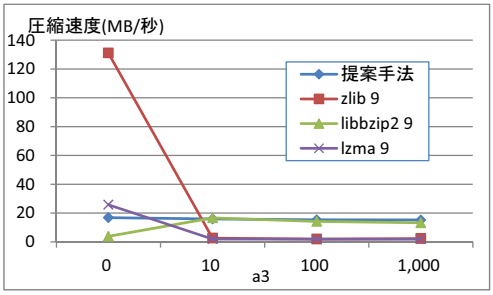
5.3　評 価 結 果

5.3.1　評　価　1

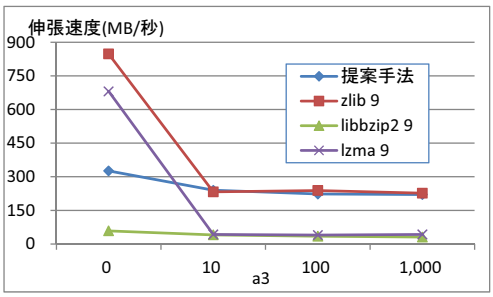
パターン 1 からパターン 4 までの評価結果を図 10 と表 3 に示す．横軸はパラメータ a_3 の値である．zlib, libbzip2, lzma については，レベル 1（最高速），レベル 6（標準），レベル 9（最圧縮）の 3 通りの性能をそれぞれ採取した．



(a) 圧縮率



(b) 圧縮速度



(c) 伸張速度

図 10　評価結果 1

提案手法において，パターン 1（$a_3 = 0$）の場合のみ，例外的に他の圧縮ライブラリと比較して圧縮率が低い結果となっている．圧縮速度，伸張速度についても，zlib, lzma に大きく劣っている．$a_3 = 0$ はノイズが無いという意味であるので，式 (5) は周期関数となる．これらの汎用圧縮ライブラリは，繰り返し現れるバイナリ列を高く圧縮する傾向があるため，この結果は妥当と言える．

一方で，ノイズが存在する場合，提案手法は zlib と比較して，圧縮率において 10%から 20%程度高い．libbzip2, lzma に対しては，同程度の圧縮率となっている．圧縮速度については，zlib のレベル 1 には劣っているが，他の方式に対しては，同程度かそれ以上の性能が得られている．伸張速度については，zlib と同程度であり，その他のライブラリに対しては 5 倍以上の性能

表 3　評価結果 1

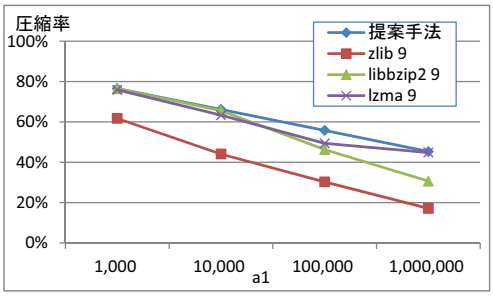
	パラメータ	圧縮率	圧縮速度	伸張速度
提案手法	パターン 1	77.6%	16.8MB/秒	326.0MB/秒
	パターン 2	76.5%	15.9MB/秒	240.0MB/秒
	パターン 3	69.2%	15.3MB/秒	223.1MB/秒
	パターン 4	58.9%	15.2MB/秒	220.5MB/秒
zlib	パターン 1	94.9%	254.3MB/秒	829.3MB/秒
（レベル 1）	パターン 2	56.8%	56.6MB/秒	231.2MB/秒
	パターン 3	53.0%	53.0MB/秒	235.5MB/秒
	パターン 4	42.6%	48.3MB/秒	227.1MB/秒
zlib	パターン 1	99.2%	158.3MB/秒	811.6MB/秒
（レベル 6）	パターン 2	56.4%	8.3MB/秒	229.8MB/秒
	パターン 3	53.7%	6.5MB/秒	239.9MB/秒
	パターン 4	46.6%	8.0MB/秒	227.1MB/秒
zlib	パターン 1	99.3%	131.1MB/秒	847.7MB/秒
（レベル 6）	パターン 2	61.7%	2.6MB/秒	232.6MB/秒
	パターン 3	58.0%	2.0MB/秒	238.4MB/秒
	パターン 4	47.8%	2.3MB/秒	227.1MB/秒
libbzip2	パターン 1	93.9%	13.3MB/秒	64.4MB/秒
（レベル 1）	パターン 2	73.2%	15.9MB/秒	45.1MB/秒
	パターン 3	66.7%	13.4MB/秒	41.6MB/秒
	パターン 4	58.6%	13.2MB/秒	34.5MB/秒
libbzip2	パターン 1	98.6%	4.7MB/秒	53.5MB/秒
（レベル 6）	パターン 2	76.2%	17.6MB/秒	41.0MB/秒
	パターン 3	68.7%	14.4MB/秒	35.3MB/秒
	パターン 4	59.4%	13.4MB/秒	30.9MB/秒
libbzip2	パターン 1	99.0%	3.8MB/秒	58.4MB/秒
（レベル 9）	パターン 2	76.5%	16.7MB/秒	40.3MB/秒
	パターン 3	68.9%	14.2MB/秒	34.9MB/秒
	パターン 4	59.5%	13.3MB/秒	30.7MB/秒
lzma	パターン 1	99.9%	114.9MB/秒	748.0MB/秒
（レベル 1）	パターン 2	68.3%	13.3MB/秒	49.5MB/秒
	パターン 3	64.7%	11.7MB/秒	46.1MB/秒
	パターン 4	55.7%	8.5MB/秒	37.8MB/秒
lzma	パターン 1	99.9%	27.8MB/秒	657.7MB/秒
（レベル 6）	パターン 2	75.9%	1.9MB/秒	42.9MB/秒
	パターン 3	67.3%	1.8MB/秒	39.3MB/秒
	パターン 4	58.4%	1.9MB/秒	42.8MB/秒
lzma	パターン 1	99.9%	25.8MB/秒	681.2MB/秒
（レベル 9）	パターン 2	75.9%	1.9MB/秒	42.6MB/秒
	パターン 3	67.3%	1.8MB/秒	39.8MB/秒
	パターン 4	58.4%	1.8MB/秒	42.7MB/秒

となっている．まとめると，提案手法においてはノイズが存在する場合に，zlib の高速性と，libbzip2 や lzma の高圧縮性を両立するような性能が実現されていると言える．これは，高変動な整数列に対する提案手法の有効性を表している．

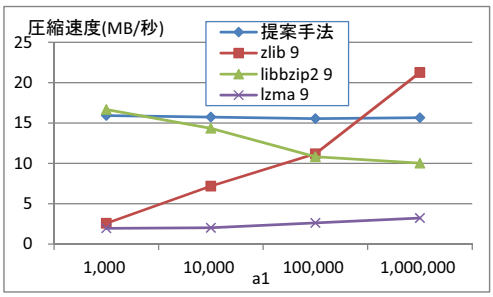
パラメータ $a_3 = 0$ の変動に対する性能への影響について，提案手法においては，圧縮率はノイズが大きくなるにつれ減少する傾向にあるが，速度性能についてはほとんど変化が見られない．

5.3.2 評　価　2

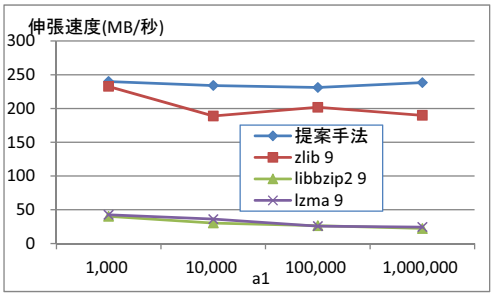
続いて，パタン—2，及びパターン 5 からパターン 7 までの評価結果を図 11 と表 4 に示す．横軸はパラメータ a_1 の値である．



(a) 圧縮率



(b) 圧縮速度



(c) 伸張速度

図 11　評価結果 2

提案手法では，zlib と比較してやはり圧縮率が 10%から 20%程度高い結果となっている．特に，パターン 7 においては，zlib の圧縮率が 20%程度にまで低下しており，提案手法は，zlib の 66%にデータを圧縮できていることが分かる．また，評価 1 においては，提案手法と圧縮率の差がほとんど見られなかった libbzip2 に対しては，データ変動幅が大きくなるにつれ，提案手法の優位性が現れている．lzma に対しては，圧縮率はほぼ同等であるが，僅かながら提案手法の方が優位である．いずれの方式に対しても，データ変動幅が大きくなるにつれ，圧縮率は低下する．

表 4　評価結果 2

	パラメータ	圧縮率	圧縮速度	伸張速度
提案手法	パターン 2	76.5%	15.9MB/秒	239.9MB/秒
	パターン 5	66.2%	15.7MB/秒	234.0MB/秒
	パターン 6	55.8%	15.5MB/秒	231.2MB/秒
	パターン 7	45.3%	15.7MB/秒	238.4MB/秒
zlib (レベル 1)	パターン 2	56.8%	56.6MB/秒	231.2MB/秒
	パターン 5	44.9%	42.3MB/秒	182.5MB/秒
	パターン 6	32.1%	39.5MB/秒	200.8MB/秒
	パターン 7	18.9%	33.4MB/秒	187.0MB/秒
zlib (レベル 6)	パターン 2	56.4%	8.3MB/秒	229.8MB/秒
	パターン 5	43.6%	9.4MB/秒	198.7MB/秒
	パターン 6	30.3%	11.2MB/秒	207.3MB/秒
	パターン 7	17.1%	21.5MB/秒	186.1MB/秒
zlib (レベル 9)	パターン 2	61.7%	2.6MB/秒	232.6MB/秒
	パターン 5	44.0%	7.2MB/秒	188.8MB/秒
	パターン 6	30.3%	11.2MB/秒	201.8MB/秒
	パターン 7	17.1%	21.3MB/秒	189.8MB/秒
libbzip2 (レベル 1)	パターン 1	73.2%	15.9MB/秒	45.1MB/秒
	パターン 5	59.6%	13.9MB/秒	35.8MB/秒
	パターン 6	39.9%	10.6MB/秒	27.1MB/秒
	パターン 7	26.5%	9.9MB/秒	24.4MB/秒
libbzip2 (レベル 6)	パターン 1	76.2%	17.6MB/秒	41.0MB/秒
	パターン 5	65.2%	14.6MB/秒	31.6MB/秒
	パターン 6	45.1%	10.8MB/秒	25.0MB/秒
	パターン 7	29.9%	10.0MB/秒	22.0MB/秒
libbzip2 (レベル 9)	パターン 1	76.5%	16.7MB/秒	40.3MB/秒
	パターン 5	65.6%	14.3MB/秒	30.3MB/秒
	パターン 6	46.3%	10.8MB/秒	26.4MB/秒
	パターン 7	30.6%	10.0MB/秒	22.3MB/秒
lzma (レベル 1)	パターン 1	68.3%	13.3MB/秒	49.5MB/秒
	パターン 5	57.9%	7.8MB/秒	42.5MB/秒
	パターン 6	44.2%	5.9MB/秒	30.3MB/秒
	パターン 7	40.6%	6.0MB/秒	25.3MB/秒
lzma (レベル 6)	パターン 1	75.9%	1.9MB/秒	42.9MB/秒
	パターン 5	63.3%	2.1MB/秒	35.1MB/秒
	パターン 6	49.4%	2.6MB/秒	25.3MB/秒
	パターン 7	44.8%	3.1MB/秒	24.3MB/秒
lzma (レベル 9)	パターン 1	75.9%	1.9MB/秒	42.6MB/秒
	パターン 5	63.3%	2.0MB/秒	36.2MB/秒
	パターン 6	49.4%	2.6MB/秒	25.7MB/秒
	パターン 7	44.8%	3.2MB/秒	24.4MB/秒

圧縮速度について，zlib の一部の条件に対しては，提案手法の圧縮速度が劣っているが，他の方式に対しては，同程度かそれ以上の性能である．伸張速度についても，他の方式以上の性能が得られており，特に，libbzip2 や lzma と比較すると，5 倍以上の性能となっている．

評価 2 では，評価 1 の結果と比較し，圧縮率に関して libbzip2 や lzam に対する優位性が現れている．データの変動幅が大きいほど一般に圧縮は困難となるが，提案手法では，データ変動の影響を他方式よりも受けにくいと言える．

6. おわりに

本稿では，高変動であり，かつ順序変更が許されないような時系列整数データの圧縮手法を提案した．提案手法では，ビット分割方式を各種符号化方式と併用することで，高変動な整数列に対して高い圧縮率を得ることができる．提案手法を圧縮ライブラリとして実装し，汎用圧縮ライブラリである zlib, libbzip2, lzma との比較評価により，提案手法が有効であることを実証した．

残された課題の 1 つとして，浮動小数点数への対応がある．センサデータの中には，浮動小数点数でデータを表現するものも多く含まれるため，浮動小数点数データを情報落ちを発生させることなく圧縮する方法が必要である．また，別の課題として，提案手法の高速化がある．近年の整数列圧縮の研究により，高速化が急激に進んでいるため，これらの手法を上手く取り入れることで，高い圧縮率を維持しつつ，高速化が実現可能と期待できる．今後は，これらの課題に取り組んでいく予定である．

文　献

[1] P. Elias, “Universal codeword sets and representations of integers,” IEEE Trans. Inform. Theory 21 (2) (1975) 194–203.

[2] S.W. Golomb, “Run-length encodings,” IEEE Transactions on Information Theory, IT–12(3):399–401.

[3] R. Rice and J. Plaunt, “Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data,” IEEE Transactions on Communications, vol. 16(9), pp.889–897, Dec., 1971.

[4] F. Scholer, H. Williams, J. Yiannis, and J. Zobel, “Compression of inverted indexes for fast query evaluation, ” In Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval, August 2002.

[5] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst and P. S. Oberoi, “SIMD-based decoding of posting lists,” In Proc. of CIKM’ 11, pp.317–326, 2011.

[6] V. N. Anh, A. Moffat, “Inverted Index Compression using Word-Aligned Binary Codes, ” Information Retrieval, 8(1), pp.151–166, 2005.

[7] H. Yan, S. Ding, T. Suel, “Inverted index compression and query processing with optimized document ordering,” Proc. of WWW’ 09, pp.401–410, 2009.

[8] V. N. Anh, A. Moffat, “Index compression using 64-bit words,” Software Pract. Exp., 40(2), pp.131–147, 2010.

[9] M. Zukowski, S. Heman, N. Nes, and P. Boncz, “Super-scalar RAM-CPU cache compression. ” In Proc. of the Int. Conf. on Data Engineering, 2006.

[10] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization,” Journal of CoRR, 2012.

[11] 郡 光則, “データウェアハウス向け高性能データ圧縮方式,” 情報処理学会論文誌, Vol.47, No.SIG13, pp.58–73(2006).

[12] 加藤 守, 谷垣 宏一, 郡 光則, “環境情報データベース向け高性能センサデータ圧縮方式,” 情報処理学会第 73 回全国大会, 2C-5(2011).