高速・可逆な実時間データ圧縮アルゴリズムの開発研究

奥村 晴彦

博士（学術）

総合研究大学院大学
数物科学研究科
核融合科学専攻

平成 10 年度
（1998）

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. Moreover, since these tools are batch-oriented, they cannot be used to distribute real-time data on the network.

To compress such data (specifically, raw A/D converter outputs) in both batch and real-time modes, we conducted a thorough research on lossless data compression for laboratory data. The research resulted in a new general-purpose real-time compression algorithm 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 extrapolation polynomials, and estimates the variance of the prediction residuals, for each channel of the input. The second part of the algorithm encodes the residuals by a simplified length-limited minimum-redundancy code, assuming either a Gaussian or Laplace distribution.

Based on this algorithm, we implemented a standalone compression tool *NIFSq*, and a compression library *NIFSqlib*, which is used by our Java-based data management system developed for the Large Helical Device (LHD) experiments at the National Institute for Fusion Science (NIFS). Typical compression ratios are around 4 : 1.

KEYWORDS: data compression, laboratory data, A/D converter, LHD.

目次

第1章

序

核融合科学研究所で稼働中の Java ベースの制御データ処理装置は，核融合科学研究所で平成5年から著者たちが共同研究として進めてきた「ワークステーションを用いたデータ収集・解析・制御システムの研究」の成果として開発されたものである [16, 10, 40, 11]。

このシステムは，データ収集・監視システムとしての機能と，実験支援データベースシステムとしての機能とを併せ持つ。データ収集・監視については，多チャンネルのセンサ出力等をノイズ低減処理した後，データベースに登録するとともに，ネットワークで実時間配送し，WWW ブラウザと Java アプレットからなるクライアントプログラムを用いて，複数のコンピュータで実時間監視することができる。高サンプリングレートでショットごとにデータを取り込むほか，低サンプリングレートで全チャンネルにわたって無休でデータを取り込み続け，すべてのデータを保存する。したがって，データ量は膨大になり，データの保管・転送のコストが無視できない。

そこで重要になるのが計測データ圧縮の技術である。圧縮によってデータ量を数分の1にできたならば，データ保存用のハードディスクあるいは MSS（Mass Storage System）の費用は数分の1になり，ネットワークのトラフィックも数分の1にとどめることができる。

データ圧縮の技術は，情報量の損失のない可逆（lossless）な圧縮と，損失のある不可逆（lossy）な圧縮とに大別できる。特に不可逆な圧縮は近年マルチメディアへの応用のため盛んに研究が行われているが，計測データに用いることができるのは可逆な圧縮である。

また，処理形態として，実時間圧縮とバッチ圧縮とに分けることができる。継続的に収集するデータでは，実時間での圧縮が適当である。これに対して，短時間に多量のデータを収集する場合は，ファイルに落とした後にバッチ処理的に圧縮することが可能である。なお，後者の場合でも，ディスクへの書き込みの手前で実時間で圧縮することにより，ディ

スクの書き込み速度を超えた速度でのデータ収集が可能になる。

汎用の可逆圧縮ツールとしては，バッチ方式の圧縮に限れば，著者がアルゴリズムの基本部分を開発したLHAを始めとして，これを高速化したZipおよびgzipとそれらをライブラリ化したzlibなど，多数のものがある。LHA，Zip，gzip，zlibはいずれも前段にLZ77法，後段にHuffman法を使ったものである。また，最近ではBurrows-Wheeler法（ブロック整列法）を使ったbzip2などの新しい圧縮ツールも使われ始めている。しかし，これらはいずれもテキストファイルの圧縮には向くが，計測データに対しては，たとえバッチ処理でもそのままでは十分な圧縮率が得られず，実時間圧縮に至っては，速度や遅延などの問題があり，不向きである。

特に遅延の問題は本質的である。ほとんどの圧縮ツールは，一定の量のデータを読み込んでコンピュータのメモリに蓄えた後に，そのデータの統計的性質を解析し，圧縮パラメータの決定や符号表の構築をし，それに基づいて，蓄えたデータを圧縮し出力する。したがって，1サンプル分のデータが入力されても，1サンプル分の圧縮データの出力が可能なわけではない。たとえば999サンプルの入力があっても何一つ出力できず，1000サンプル目の入力があって始めて1000サンプル分の圧縮データの出力がまとめて可能になるといった具合である。これではデータの実時間配送には使えない。

さらに，多チャンネルデータでは，チャンネル間の相関は一般に低いため圧縮には役立たず，専ら個々のチャンネル内の時系列的な相関を利用して圧縮することになる。したがって，たとえば1000チャンネルの入力を実時間圧縮するには，1000個分の圧縮ルーチンを独立に動作させなければ十分な圧縮率が得られない。LHAやgzipなどでは，実時間性の問題を度外視しても，チャンネルあたり数百キロバイトのメモリを消費するので，チャンネル数の多いデータに対してチャンネルごとに独立に動作させることは困難である。これに対して，バッチ圧縮の場合は，時間ごとのデータをチャンネルごとに再編成してから圧縮することにより，単一の圧縮ルーチンの動作で圧縮が可能である。

このような計測データに固有な問題を考慮して，多チャンネル時系列データの実時間圧縮アルゴリズムの研究を進め，一定の成果を得ることができた。特に，チャンネルあたりの所要メモリ量を100バイト前後に押さえることができたので，同時に実時間圧縮できるチャンネル数は事実上無制限である。圧縮率についても，LHD制御データ処理装置のデータについては，比較した他のどの汎用圧縮ツールよりも良好な結果を得ている。

本論文では，Shannonの情報理論に始まる現在までのデータ圧縮の分野での諸研究を，可逆な場合を中心に総説するとともに，計測データ圧縮アルゴリズムの研究成果[22]の詳細を述べる。

第2章

計測システムにおけるデータ圧縮の必要性

2.1 LHD制御データ処理装置の概要

序でも述べたように，核融合科学研究所のLHD制御データ処理装置は，核融合科学研究所で平成5年から著者たちが共同研究として進めてきた「ワークステーションを用いたデータ収集・解析・制御システムの研究」の成果として開発されたものである[16, 10, 40, 11]。

このシステム（図2.1）は，まず，任意チャンネル数のセンサ出力を，ローパスフィルタ付きの絶縁アンプで増幅した後，A/D変換器（ADC）でディジタル化（サンプリングおよび量子化），マルチプレックス（多重化）し，UNIXワークステーションで取り込む。データの取り込みは，ショットごとに高サンプリングレートで行うほか，無休で全チャンネルにわたって低サンプリングレートで取り込みを続ける。これらをバイナリファイルとして保存し，データベースに登録するとともに，ネットワークで実時間配送する。

利用者側（クライアント）は，パーソナルコンピュータまたはUNIXワークステーションで，WWWブラウザ（Netscape NavigatorやHotJavaなど）を用いて，複数のコンピュータで実時間に監視できる。この機能の実現のためにJavaを用いている。実時間監視だけでなく，実験データベースの照会，共有実験メモの記入など，さまざまな機能がある。

このシステムの特徴は次の通りである。

2.1.1 UNIXの利用

システムのサーバ側はすべてUNIXワークステーションで構成されている。

UNIXはマルチユーザ・マルチタスクのオペレーティングシステム（基本ソフト）であ

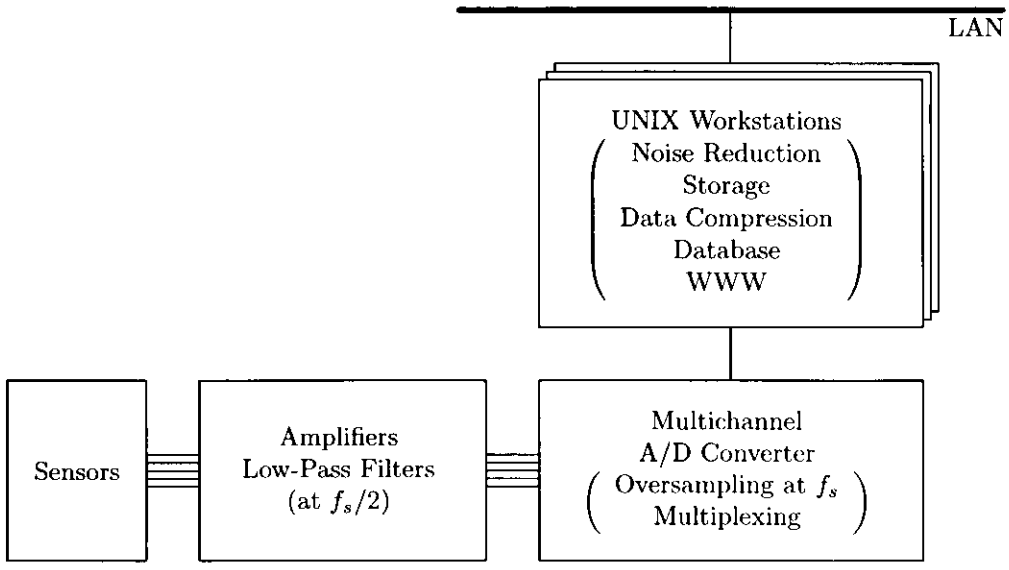


図 2.1: LHD 制御データ処理装置の概念図。センサ出力から $f_s/2$ 以下の周波数成分を取り出し，A/D 変換器（ADC）で f_s のレートでサンプリングして，ワークステーションで必要なサンプリングレートに落とし，ネットワーク経由で配信する。

る[35]。1970 年ごろ開発され[注1]，その後，長い時間をかけて多くの人の手を経て改良されてきたので，動作が十分安定しており，長期間（実質的に無制限）の連続稼働が可能である[注2]。

標準的な UNIX では，処理時間の平均について最適化されており，処理時間の上限については保証がない。割込み処理などで必ず一定時間内に処理を完了しなければならない場合は，**リアルタイムオペレーティングシステム（real-time operationg system）**を使う必要がある。しかし，一般の計測では，A/D 変換器のバッファをあふれさせないだけの平均的な速さが要求されるだけであるので，標準的な UNIX で十分である。

現在の本システムでは，UNIX ワークステーションとして Sun Microsystems 社の SPARCstation を使っている。保守サポート体制を含めて考えれば，現状ではこのような専用ワークステーションを利用するのがよいが，性能対価格比だけを見れば最近では市販

[注1] UNIX の開発は 1969 年に始まり，*UNIX Programmer's Manual* の第 1 版が出たのは 1971 年である。UNIX についての最初の論文[33] は 1974 年に出版された。

[注2] UNIX は，アプリケーションがメモリを解放せずに異常終了した場合でも，オペレーティングシステムがそのメモリを解放し，再利用する。一方，現状の Windows NT では，オペレーティングシステムによるメモリ解放の機能がおそらくうまく働いていないため，メモリが次第に不足し，最終的にはオペレーティングシステムが落ちてしまうという現象がしばしば見られる。これを防ぐには，ときどきリブート（再起動）するとよい。

のパーソナルコンピュータの方がUNIXワークステーションより優れるので，今後はパーソナルコンピュータをLinux[注3]などのUNIX系のオペレーティングシステムで稼働させることも検討すべきであろう。実際，KEKなどで開発されているUNIXベースのデータ収集システムUNIDAQ[注4]にはLinux上で動作する版もある[注5]。

Linuxにはハードリアルタイム性をサポートした拡張もある[注6] [5]。

2.1.2 WWW，Javaへの対応

LHD制御データ処理装置は，全面的にWWWおよびJavaの利点を活用している。

WWW（World Wide Webの略。単にWebともいう）はインターネット上での情報共有のための仕組みである。1989年にCERNのTim Berners-Leeによって考案された。WWWの情報を見るためのソフト（WWWブラウザ）としては，古くは1993年にMarc Andreessenらにより開発され無償で配布されたMosaicがあり，これは1994年を境としてインターネットが爆発的に普及する原動力の一つとなった。現在ではAndreessenらによるNetscape社のNavigator（Communicator）と，Microsoft社のInternet Explorerとが，機能を競い合いながら無償配布されている。他にSun Microsystems社のHotJavaなどがある。

Java（ジャバ）は，Sun Microsystems社のJames Goslingらによって開発され，1995年に発表されたばかりの，新しい汎用プログラミング言語である。C++に似たオブジェクト指向言語であるが，バグの原因となりやすい機能を思いきって削り，代わりにインターネット環境との親和性を高めるための種々の工夫を盛り込んでいる。特に，Javaプログラムの一形態であるアプレット（**applet**，小さなapplicationの意）は，WWWブラウザの中で動作させることができるので，ブラウザの機能を拡張するために広く使われている。LHD制御データ処理装置のように，WWWブラウザの中で計測結果を実時間でグラフ化するためには，現在ではJavaを利用するのが最も妥当な手段である。

言語としてのJavaは，複雑になりすぎたC++のエレガントな部分だけを取り出したような，たいへん美しい言語である。ネットワークサーバプログラムの作成，あるいはWWWブラウザの機能強化には，Javaは最適の言語である。しかし，大規模なクライア

[注3] Linux（リナックスあるいはリヌクスと読む）はフリーソフトウェアのUNIX系オペレーティングシステムである。

[注4] `http://www-online.kek.jp/~online/Unidaq/default.htm`

[注5] 長谷野雅哉, 奥野義雄, 田中義人, 能町正治, 田中万博, 藤井啓文, 安芳次, LinuxとCAMAC ACCを用いたデータ収集システムの開発, `http://www.elc.nias.ac.jp/~daq/haseno/linux_daq/`

[注6] Real-Time Linux, `http://rtlinux.cs.nmt.edu/~rtlinux/`

ントプログラムの作成にはまだまだ課題が多い。Java のサポートされるオペレーティングシステムも現状では Solaris（Sun Microsystems 社の UNIX）と Windows 95/98/NT が中心であり[注7]，“Write once, run anywhere”（一度書いたらどこでも動く）という当初の公約はいまだ達成されたとは言いがたい。

Java をデータ収集に利用する試みは他所でもいくつか行われている [41]。

2.1.3 ノイズ対策

LHD 制御データ処理装置はノイズ対策に特に注意を払って作られている。

ノイズ対策としては，センサからアンプまでの結線に撚り線やシールド線を用い，接地に留意する，絶縁アンプを用いるなどの工夫は必須であるが，アンプ・A/D 変換器・コンピュータからなるシステムでも，ノイズ低減のための工夫が可能である。

まず，第 4.1 節で述べるように，Nyquist 周波数（サンプリングレートの 1/2）以上の周波数成分は，単にノイズ（折返し歪）となるだけである。これは絶縁アンプにローパスフィルタを組み込むことにより対処している。

また，やはり第 4.1 節で述べるオーバーサンプリングという手法を用いて，必要以上のレートでサンプルし平均化することによって，ノイズを低減している。この方式のもう一つの利点として，サンプリングレートの変更が，ローパスフィルタや A/D 変換器のサンプリングレートの設定をまったく変えることなく，コンピュータ内のソフトウェアの設定だけで容易にできることがあげられる。

2.2 データ圧縮の必要性

LHD に限らず，近年の大形物理実験は，膨大な量の計測データを生産し，データの保管・転送のコストが無視できない。それだけではなく，データの生産量が記憶装置の I/O（入出力）速度をも上回ることがある。

そこで重要になるのが計測データ圧縮の技術である。

圧縮によってデータ量を数分の 1 にできたならば，データ保存用のハードディスクや MSS（Mass Storage System）の費用は数分の 1 になり，ネットワークのトラフィックも

[注7] Java 開発キットは各メーカの手によって AIX, DIGITAL OpenVMS, DIGITAL Unix, HP-UX, IRIX, MacOS, NetWare, OS/2, OS/390, OS/400, SCO, UnixWare, VxWorks, NT Alpha 版が開発されている。また，有志によって Linux, FreeBSD, NetBSD, Reliant Unix にも移植されている。Linux については Sun Microsystems 社自身がサポートを開始する予定である。

減らすことができる。

データ圧縮の技術については次章で詳しく述べるが，情報量の損失を伴わない可逆（lossless）な圧縮と，情報量の損失を伴う不可逆（lossy）な圧縮とに大別できる。不可逆な圧縮は近年マルチメディアへの応用のため盛んに研究が行われているが，計測データに安心して用いることができるのは可逆な圧縮である。

以下で物理計測にデータ圧縮を利用している例をいくつか挙げておく。

2.2.1 KLOE でのデータ圧縮

KLOE[注8] はイタリアの Frascati 国立研究所[注9] の DAFNE 加速器[注10] での CP 対称性の破れの実験である。数ヵ月間にわたって毎秒 50MB のデータを生成する。加速器のデータは LHD 制御データ処理装置の連続データ取得とはかなり性格を異にする。Schindler[注11] によれば，可逆な場合の圧縮比（ここでは圧縮後のサイズ÷圧縮前のサイズ）は熱量計（アドレス，ADC，TDC），ドリフトチェンバー（アドレス，TDC）に分けて，表 2.1 のように見積もられている。

Calorimeter	Addresses	11%
	ADC's	4%
	TDC's	4%
Drift Chamber	Addresses	45%
	TDC's	32%

表 2.1: KLOE におけるデータ圧縮の見積もり。数値は圧縮後のサイズ÷圧縮前のサイズ。

どの場合も，一般の圧縮ツールではほとんど圧縮できないが，データの性質をよく見極めた上で正確にモデル化し，それを Huffman 符号化あるいは算術符号化するという正統的な方法により圧縮される。

注8 http://www.lnf.infn.it/kloe/
注9 LNF, Laboratori Nazionali di Frascati, http://www.lnf.infn.it/
注10 http://www.lnf.infn.it/acceleratori/dafne.html
注11 http://eiunix.tuwien.ac.at/~michael/kloe/

2.2.2 科学衛星「ようこう」のデータ圧縮

科学衛星「ようこう」では，12ビットのCCDカメラ出力 x を8ビット値 y に圧縮するために次の方法を使っている[注12]（この項では round(·) は最も近い整数に丸める関数である）。

$$y = \begin{cases} x & (x \leq 64) \\ \mathrm{round}\left(59.249 + \sqrt{3510.39 - 9.50(431.14 - x)}\right) & (x > 64) \end{cases}$$

伸張は上の逆関数である。

$$x = \begin{cases} y & (y \leq 64) \\ \mathrm{round}(0.10526y^2 - 12.473y + 431.14) & (y > 64) \end{cases}$$

この方法は非常に単純であり，可逆ではない。

2.2.3 ACISデータの圧縮

MITのCenter for Space ResearchのACIS（AXAF[注13] CCD Imaging Spectrometer）では，12ビット単位のピクセルデータの圧縮にTruncated Huffman First-Differenceと名づけられた方法を用いている[注14]。

この方法は，まず各ピクセル値と隣りのピクセル値との差分を求める（この演算により12ビット値が13ビット値になると説明されているが，これは後で述べるように12ビットの範囲に収めることが可能である）。この差分は正規分布に従うと仮定され，あらかじめ生成された符号表に従って，各差分を符号化する。メモリの制約から，符号表全体をロードすることができないので，範囲外の差分値は特別な符号を冠して，元の12ビットデータをそのまま出力する。

2.2.4 SGFのデータ圧縮オプション

データフォーマットHDF（Hierarchical Data Format）[注15]は，NCSA（The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign）で開発され，科学技術分野でよく使われているものである。

[注12] http://umbra.nascom.nasa.gov/yohkoh/docs/yag/iguide/node70.html

[注13] Advanced X-ray Astrophysics Facility

[注14] Massachusetts Institute of Technology, Center for Space Research, "Huffman Coding of ACIS Pixel Data," http://acis.mit.edu/huff/, 1996.

[注15] http://hdf.ncsa.uiuc.edu/

HDF はいくつかの圧縮オプション（ランレングス，Huffman，JPEG など）をサポートしているが，特に zlib（Zip や gzip の圧縮アルゴリズムをライブラリ化したもの）をそのまま使うオプションもある。

NCSA はまた Java による HDF インターフェースや Java ベースの HDF ビューア（視覚化ツール）も開発している。

zlib は，テキストファイルなどのように，8 ビット単位で見たときに同じパターンの反復が多いファイルを念頭に置いて設計されており，反復パターンの探索に時間を要するため，圧縮速度は遅い。人為的な反復パターンのない科学技術データの圧縮には不適である。しかし，たとえば 12 ビットのセンサ出力を一定の関数で浮動小数点の物理値に変換したファイルなどでは，意外に縮むことがある。このようなデータファイルは，浮動小数点に変換したため 32 ビット/サンプルのサイズを持つが，たかだか 12 ビット/サンプルの情報しかなく，同じ 32 ビットのパターンが繰り返し現れるためであろう。

2.2.5 Java での圧縮サポート

Java 開発キット（Java Development Kit，JDK）1.1 以降では JAR という圧縮アーカイブ形式をサポートしている。これはやはり zlib を圧縮エンジンとして使っている。

第3章

データ圧縮技術の概要

3.1 可逆な圧縮と不可逆な圧縮

データ圧縮は，情報量の損失を伴わない**可逆な圧縮**（**lossless compression**，**無歪み圧縮**，**歪みを許さない圧縮**などともいう）と，情報量の損失を伴う**不可逆な圧縮**（**非可逆な圧縮**，**lossy compression**，**有歪み圧縮**，**歪みを許す圧縮**などともいう）に大別できる。テキストファイルやコンピュータの実行ファイルなど，1ビットでも変化してはならないデータの圧縮では，当然ながら可逆な圧縮しか使うことができない。これに対し，画像や音声など，最終的に人間の五感に訴えるためのデータの圧縮では，人間が細部の情報の変化に鈍感であることから，情報が失われても圧縮率を高くできる不可逆な圧縮が広く使われている。

そもそも画像や音声，センサ出力などの計測データでは，アナログ量をディジタル化（サンプリング，量子化）する時点で情報量が失われているので，ディジタル化する際の情報量の損失とデータ圧縮による情報量の損失とを合わせて考えなければならない。たとえば44.1 kHzでサンプリングした16ビットの音声データと，22.05 kHzでサンプリングした8ビットの音声データとでは，サイズは前者が後者の4倍である。仮に前者に不可逆な圧縮をし，後者に可逆な圧縮をしたところ，同じサイズになったとする。これらを伸張して人間が音質を評価したとすると，確実に，前者の不可逆な圧縮をした方が高音質と評価されるであろう。実際には両者の情報量は同じであっても，不可逆な圧縮の場合は人間の耳の特性を勘案して，耳の敏感な領域で情報量をできるだけ保ち，鈍感な領域で情報量を落とすことができるからである。

これと同じことが計測データでも原理的には可能である。たとえば温度センサの場合，センサの種類および温度領域によって精度が異なる上に，実験の目的に応じて，たとえば

ヘリウム温度での情報は詳しく，室温付近の情報はおおまかでよいというように，所要精度も状況により変化する。これらを勘案して不要な情報量を落とせば，高圧縮が可能である。しかし，センサごと・実験ごとにこのような勘案をするのはたいへんであるし，たとえ精度的に意味がなくても A/D 変換器の出力は念のため末位ビットまで保存したいという要望が根強いので，ほとんどの場合，計測データについては可逆な圧縮が使われる。

この章では，主に可逆な圧縮に限って，現在までに知られている技術を概観する。

3.2 圧縮率，圧縮比

まず基本的な用語について説明を加えておく。

圧縮率または**圧縮比**（**compression ratio**）という用語には種々の定義がある。たとえば汎用圧縮ツールの一つ LHA では

$$r = \frac{\text{圧縮後のサイズ}}{\text{圧縮前のサイズ}}$$

を圧縮比と呼び，この値が小さいことを漠然と「圧縮率が高い」と言っている。Nelson たち [15] は $100(1-r)$ を compression ratio と呼び，Hankerson たち [7] は $1/r$ を compression ratio と呼んでいる。Williams [39] は，r を proportion remaining，$100r$ を percentage remaining，$1-r$ を proportion removed，$100(1-r)$ を percentage removed，$1/r$ を compression gain と呼ぶことを提案している。Salomon [28] は，r を compression ratio，$1/r$ を compression factor，$100\log_e(1/r)$ を compression gain と呼んでいる[注1]。

さらにいえば，何をもって「圧縮前のサイズ」とするかもはっきりしない。たとえば英語のプレーンテキストファイルの多くは ASCII（American Standard Code for Information Interchange）という 7 ビット符号を使うが，現在のコンピュータは 8 ビット単位の方が扱いやすいので，1 ビット余らせて 8 ビットのスペースに保存する。また，計測でよく使われる 12 ビットの A/D 変換器出力も，4 ビット余らせて 16 ビットのスペースに保存することが多い。このようなとき，余らせたビットも「圧縮前のサイズ」に含めるかどうかで，圧縮比が異なってしまう。さらに，計測データがバイナリ形式でなくアスキー形式で格納されていれば，同じ情報量でもファイルサイズはずっと大きくなる。このようなことを考えれば，元ファイルのデータ格納法が関係する量を使う代わりに，テキストファイルなら 1 文字あたりの圧縮後の平均ビット数（bits/character，bpc），画像ファイルなら 1 ピクセ

[注1] より正確には，$100\log_e$(基準サイズ/圧縮後のサイズ) を compression gain と呼んでいる。基準サイズとは，文脈によって圧縮前のサイズのこともあり，基準となる圧縮法による圧縮後のサイズのこともある。この用語は Howard and Vitter [8] に因むようである。

ルあたりの圧縮後の平均ビット数（bits/pixel，bpp），計測データなら1サンプル点あたりの圧縮後の平均ビット数（bits/sample）等を使う方がよい。

また，ある圧縮アルゴリズムの圧縮率はどれだけかと問うのも，あまり意味がない。圧縮率はデータによって大きく変わるからである。単純な文書ファイル圧縮でも，新聞の文章とJames Joyceの文章とで圧縮率は異なるであろう。同様に，計測データ圧縮でも，緩やかに変化する正弦波とホワイトノイズとでは圧縮率は全く異なるし，全く同じ波形でもアンプのゲインを変えただけで圧縮率は異なる。そもそも，必ず圧縮できるとは限らない。当然ながら，ランダムノイズは全く圧縮できない。

不可逆な圧縮の場合は，圧縮率はさらに無意味である。よく使われる騙しのテクニックとして，画像をたとえば600 dpi，1677万色といった必要以上の解像度でスキャンし，それをたとえば1/1000に圧縮したものと並べて示し，画質の劣化が小さいことを誇示することがある。しかし，同じ画像を75 dpi，256色でスキャンしただけでもファイルサイズは1/192になり，しかも印刷上の画質はそれほど変わらないであろう。このような場合には，圧縮率は使わず，従来の技術と新しい技術で同じ画像を同じサイズに圧縮し，それを並べて画質の差を示すのがよい。動画の場合も，同じビットレートに圧縮した場合の画質の差を云々すべきである。

3.3 標準データ

前節で「あるアルゴリズムの典型的な圧縮率」という概念が無意味であることを述べたが，「典型的な圧縮率」という指標への要望は大きい。典型的な圧縮率を求めるには，「典型的なデータ」があればよい。このような理由で，標準データを策定する試みがいくつか行われた。

可逆なテキスト主体の圧縮を評価するための標準データ集（コーパス）としては，1987年に集成されたCalgary Corpus [3]が最近までよく使われてきたが，これに取って代わるものとしてCanterbury Corpus [1]が1997年に提案された。これは次の11個のファイルから成る。

> 英語テキスト，Fax画像，Cソースコード，表計算ソフトのファイル，SPARCの実行ファイル，技術文書，英詩，HTMLファイル，LISPソースコード，GNUマニュアルページ，演劇

現在ではテキスト主体の圧縮ではこのCanterbury Corpusを使うのが最も一般的であるが，日本人から見れば英語に片寄っているのが欠点である。

画像圧縮では “Lena” または “Lenna” と呼ばれる帽子をかぶった女性の写真を始めとするいくつかの準標準的なテストパターンが存在する。

しかし，標準的な計測データというものはまだ提案されていないようである。これは，計測データが多様を極め，何が典型的かという合意が得られそうにないためであろう。

3.4 符号化

データはシンボル（**symbol**，**文字**，**記号**）の並びからなる。何をシンボルと見るかは自由である。たとえばテキストデータの場合，通常の 1 文字を 1 シンボルとすることが多いが，1 単語を 1 シンボルとすることも可能であるし，極端なことを言えば 1 ビットを 1 シンボルとすることも可能である。

同様に，画像の場合はピクセル（画素），測定値の系列の場合は一つ一つの測定値をシンボルとすることが多い。

考えているシンボルの集合を**アルファベット**（**alphabet**）と呼ぶ。たとえば通常の英文（ASCII 文字）のプレーンテキストの場合，1 文字を 1 シンボルと考えるならば，アルファベットは 95 個の図形文字と数個の制御文字（改行，タブ等）を合わせた約 100 個の文字を要素とする集合である。しかし，同じプレーンテキストでも，1 単語を 1 シンボルと考えるならば，アルファベットは数万通りの単語を要素とする集合である。12 ビット量子化した測定値では，1 測定値を 1 シンボルと考えるならば，アルファベットは $2^{12} = 4096$ 通りの値からなる。4 ビット（16 階調）のグレイスケール画像では，1 ピクセルを 1 シンボルと考えるならば，アルファベットは 16 通りの階調からなる。

アルファベットを S で表し，その要素すなわちシンボルを s_1，s_2 等々で表せば，

$$S = \{s_1, s_2, \ldots, s_m\} \tag{3.1}$$

となる。$m = |S|$ をアルファベットの大きさという。

シンボルの列をコンピュータ等のディジタル装置で扱うためには，最終的には ‘0’ と ‘1’ からなるビット列に置き換えなければならない[注2]。換言すれば，入力されたシンボルの列を，アルファベットが $\{\text{‘0’}, \text{‘1’}\}$ であるようなシンボルの列に置き換えなければならない。このように，あるアルファベットのシンボル列を別のアルファベットのシンボル列で置き換えることを**符号化**（**encoding**，**coding**）といい，その置き換えのルールを**符号**

[注2] 電信の時代なら ‘.’ と ‘–’ と空白とからなる**モールス符号**（**Morse code**）を考えなければならなかったであろう。

(**code**) という。符号化の逆の操作を**復号**[注3] (**decoding**) という。

符号化で最も単純なものは，各シンボル s_i を一定のビット列 w_i に対応させること

$$s_i \rightarrow w_i \tag{3.2}$$

である。各 w_i を**符号語** (**code word** または **codeword**) という。

符号語 w_i の長さ (ビット数) を $\ell(w_i)$ で表すことにする。たとえば ASCII 符号を使った通常のプレーンテキストファイルでは，

$$\text{'A'} \rightarrow 01000001 \tag{3.3}$$

$$\text{'B'} \rightarrow 01000010 \tag{3.4}$$

$$\text{'C'} \rightarrow 01000011 \tag{3.5}$$

のような 8 ビット固定長の符号で表されるので，

$$\ell(\text{'A'}) = \ell(\text{'B'}) = \ell(\text{'C'}) = 8$$

である[注4]。これに対して，たとえば

$$\text{'A'} \rightarrow 1$$

$$\text{'B'} \rightarrow 01$$

$$\text{'C'} \rightarrow 001$$

のような可変長の符号も考えられる。このとき

$$\ell(\text{'A'}) = 1 \qquad \ell(\text{'B'}) = 2 \qquad \ell(\text{'C'}) = 3$$

である。

シンボル s_i が現れる確率を p_i とすれば，このような可変長符号化による 1 シンボルあたりのビット数の期待値は $\sum_i p_i \ell(w_i)$ で与えられる。1 シンボルあたりのビット数 (bits per symbol) は，符号化の効率の良さを表す尺度としてよく使われる。

注3 「復号する」という代わりに「復号化する」ということもあるが，本来は「復号する」が正しいであろう。

注4 もっとも，ASCII 符号は本質的に 7 ビットであり，8 ビットで表した場合の最上位ビットはつねに 0 である。

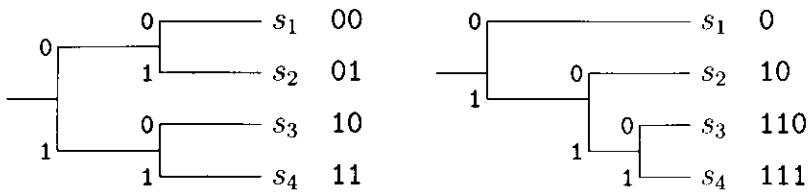


図 3.1: 4 個のシンボル s_1, s_2, s_3, s_4 についての冗長性のないプレフィックス符号は本質的にこの 2 通りしかない。

3.5 プレフィックス符号

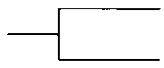
符号を構成する際に，どの符号語も他の符号語の頭の部分でないという条件を課すと便利である。たとえば，もし 01011 という符号語があれば，0101，010，01，0 という符号語は存在しないということである。この条件を課すことによって，先読みすることなしに，一意的に復号することができる。このような条件のことを**プレフィックス条件（接頭条件，語頭条件，prefix condition）**という。プレフィックス条件を満たす符号を**プレフィックス符号（接頭符号，語頭符号，prefix code，prefix-free code）**と呼ぶ。

プレフィックス符号は図 3.1 のような木（あるいはトーナメントの試合表のような図）で表すことができる。大きさ 4 のアルファベットでは，プレフィックス符号は本質的には図 3.1 の 2 通りしかない。

このような葉ノードの数がアルファベットの大きさに一致する木で表されるプレフィックス符号は，次の条件を満たす。

$$\sum_{i=1}^{m} 2^{-\ell(w_i)} = 1 \tag{3.6}$$

これは，葉ノードが 2 個の場合



に $2^{-1} + 2^{-1} = 1$ が成り立つことから始めて，数学的帰納法を適用すれば，簡単に証明できる。

プレフィックス符号の定義からは，上の符号の末尾に任意のビット列を付加したもの，たとえば

A: 00

B: 1010

C: 110110

D: 111111

等もプレフィックス符号である。このような冗長性のある場合も含めて，プレフィックス符号は次の式を満たす。

$$\sum_{i=1}^{m} 2^{-\ell_i} \leq 1 \tag{3.7}$$

逆に，与えられた n 個の非負の整数 $\ell_1, \ldots, \ell_n$ について，不等式 (3.7) が成り立つならば，各符号語の長さが $\ell_1, \ldots, \ell_n$ のプレフィックス符号が存在する。不等式 (3.7) は **Kraft の不等式（Kraft's inequality）**として知られるものである。

プレフィックス符号は，ビット列を読みながら木を根からたどれば復号できるという点で，たいへん便利である。しかし，プレフィックス符号でなくても一意的に復号できる符号は存在する。たとえば

A: 0

B: 01

はプレフィックス符号ではないが明らかに一意的に復号できる。

プレフィックス符号でない一般の符号を使っても，プレフィックス符号より効率を良くすることはできない。これは，上述の Kraft の不等式 (3.7) がプレフィックス符号でない場合にも成り立つことから証明できる。プレフィックス符号という条件をはずしたときの不等式 (3.7) を **McMillan の不等式（McMillan's inequality）**と呼ぶ。

McMillan の不等式の証明　$\ell_1, \ldots, \ell_m$ の最大の値を $\ell_{\max}$ とする。任意の正の整数 k について

$$\left(\sum_{i=1}^{m} 2^{-\ell_i}\right)^k = \left(\sum_{i_1=1}^{m} 2^{-\ell_{i_1}}\right) \cdots \left(\sum_{i_k=1}^{m} 2^{-\ell_{i_k}}\right) \tag{3.8}$$

$$= \sum_{i_1=1}^{m} \sum_{i_2=1}^{m} \cdots \sum_{i_k=1}^{m} 2^{-(\ell_{i_1}+\cdots+\ell_{i_k})} \tag{3.9}$$

この右辺の和で $\ell_{i_1} + \cdots + \ell_{i_k}$ が r に等しくなる回数を $n(r)$ とすれば

$$= \sum_{r=1}^{k\ell_{\max}} n(r) 2^{-r} \tag{3.10}$$

と書ける。ここで，$\ell_{i_1}+\cdots+\ell_{i_k}=r$ ということは，$i_1 i_2 \ldots i_k$ というメッセージは r ビットの長さを持つという意味であるので，メッセージが一意的に復号できるためには，$\ell_{i_1}+\cdots+\ell_{i_k}=r$ を満たすメッセージの数は 2^r を超えることはできない。したがって $n(r) \leq 2^r$ であり，

$$\left(\sum_{i=1}^{m} 2^{-\ell_i}\right)^k = \sum_{r=1}^{k\ell_{\max}} n(r)2^{-r} \leq \sum_{r=1}^{k\ell_{\max}} 1 = k\ell_m \tag{3.11}$$

両辺を $1/k$ 乗すれば

$$\sum_{i=1}^{m} 2^{-\ell_i} \leq k^{1/k}\ell_{\max}^{1/k} \tag{3.12}$$

ここで $k \to \infty$ とすると $k^{1/k}\ell_{\max}^{1/k} \to 1$ であるので，

$$\sum_{i=1}^{m} 2^{-\ell_i} \leq 1 \tag{3.13}$$

を得る。 □

3.6 Shannon のエントロピー

無限にシンボル（文字）を送出し続ける**情報源**（**information source** または単に **source**）を考える。特に，今までにどういうシンボルが送出されたかにかかわらず，シンボル s_i $(i=1,\ldots,m)$ が送出される確率がつねに p_i であるような情報源を，**無記憶情報源**（**memoryless source**）と呼ぶ。このような情報源の出力を '0', '1' の並びで符号化する方法を考える[注5]。無雑音とは，符号化が可逆，すなわち一意的に復号できるという意味である。

各符号語 w_k の長さを ℓ_k とすると，その平均 $\bar{\ell}=\sum_{i=1}^{m} p_k\ell_k$ は

$$\bar{\ell} \geq \sum_{i=1}^{m} p_i \log_2 p_i^{-1}$$

を満たす。この右辺の量を，この情報源の**エントロピー**（**entropy**）[注6] と呼ぶ。この不等式は次のようにして証明できる。

[注5] 本稿では符号のアルファベットを {'0','1'} に限るが，任意の大きさのアルファベットでの符号化に一般化することは簡単である。

[注6] エントロピーの概念はもちろん物理学（熱力学および統計力学）に端を発するものであるが，これに基づいて情報理論を定式化したのは Shannon [30] である。

証明　$\log_e x \le x-1\ (x>0)$ を用いると

$$\begin{aligned}\sum_{i=1}^{m} p_i \log_2 p_i^{-1} - \sum_{i=1}^{m} p_i \ell_i &= \sum_{i=1}^{m} p_i \log_2 p_i^{-1} - \sum_{i=1}^{m} p_i \log_2 2^{\ell_i} \\ &= \sum_{i=1}^{m} p_i \log_2 (p_i^{-1}/2^{\ell_i}) \\ &= (\log_2 e) \sum_{i=1}^{m} p_i \log_e (p_i^{-1}/2^{\ell_i}) \\ &\le (\log_2 e) \sum_{i=1}^{m} p_i (p_i^{-1}/2^{\ell_i} - 1) \\ &= (\log_2 e) \sum_{i=1}^{m} (2^{-\ell_i} - p_i) \\ &= (\log_2 e) \left(\sum_{i=1}^{m} 2^{-\ell_i} - 1 \right) \le 0\end{aligned}$$

最後で Kraft-McMillan の不等式 $\sum_{i=1}^{m} 2^{-\ell_i} \le 1$ を用いた。 □

3.7 Shannon のアルゴリズム

Shannon の符号化定理 $\overline{\ell} \ge \sum_{i=1}^{m} p_i \log_2 p_i^{-1}$ から，確率 p_i で生じるシンボルは，およそ $\log_2 p_i^{-1}$ ビットの符号語に対応させればよいことが示唆される。これが整数でない場合には，大きめの整数値を取って，$\lceil \log_2 p_i^{-1} \rceil$ ビットを用いれば十分であろう。ここで $\lceil x \rceil$ は x を下回らない最小の整数（要するに切上げ）である（たとえば $\lceil 3 \rceil = 3$, $\lceil 3.1 \rceil = 4$ である）。この考え方に基づいて符号を構成するのが次に述べる **Shannon のアルゴリズム**である。

m 通りのシンボル $s_1, s_2, \ldots, s_m$ について，出現確率 p_i が与えられており，しかも $p_1 \ge p_2 \ge \cdots \ge p_m$ の順に整列しているとする（そのように番号を付け直す）。このとき，Shannon のアルゴリズムは，シンボル k には $\sum_{i=1}^{k-1} p_i$ を 2 進数で表したものの小数第 $\ell_k = \lceil \log_2 p_k^{-1} \rceil$ 位までのビット列を対応させる。

2 進法の小数表現について少し注意をしておく。たとえば $\frac{1}{2}$ は 2 進法で表すと 0.1 である。このことを $\frac{1}{2} = (0.1)_2$ と記す。ただ，$\frac{1}{2} = \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$ であるので，$\frac{1}{2} = (0.01111\cdots)_2$ でもある。このように無限小数と有限小数の二通りの表現が可能な数については，有限小数の方をとると定める。

Shannon の符号の例として，

$$p_1 = 1/2, \qquad p_2 = 1/4, \qquad p_3 = p_4 = 1/8$$

の場合を考えよう。このとき，各符号語の長さは

$$\ell_1 = \lceil \log_2 2 \rceil = 1, \qquad \ell_2 = \lceil \log_2 4 \rceil = 2,$$
$$\ell_3 = \ell_4 = \lceil \log_2 8 \rceil = 3$$

であり，実際の符号語は

$$\begin{aligned}
w_1 &= 0 \\
w_2 &= 10 && p_1 = (0.1)_2 \\
w_3 &= 110 && p_1 + p_2 = (0.1)_2 + (0.01)_2 = (0.11)_2 \\
w_4 &= 111 && p_1 + p_2 + p_3 = (0.1)_2 + (0.01)_2 + (0.001)_2 = (0.111)_2
\end{aligned}$$

となる。

Shannon の符号がプレフィックス符号であることは次のようにして証明できる。

証明　$1 \le k < r \le m$ を満たす任意の整数 k, r について，$p_k \ge p_r$ したがって $\ell_k \le \ell_r$ であるので，符号語 w_k が w_r のプレフィックスでないことを証明すればいい。w_k は $p_1 + \cdots + p_{k-1}$ の 2 進展開の $\lceil \log_2 p_k^{-1} \rceil$ 桁，w_r は $p_1 + \cdots + p_{r-1}$ の 2 進展開の $\lceil \log_2 p_r^{-1} \rceil$ 桁であるが，$p_1 + \cdots + p_{k-1}$ と $p_1 + \cdots + p_{r-1}$ の差は少なくとも p_k であり，p_k は小数第 $\lceil \log_2 p_k^{-1} \rceil = \ell_k$ 桁目に 0 でない数字が現れるので，w_r は w_k のプレフィックスではありえない。 □

また，Shannon の符号の平均長の上限は次のようにして求められる。

$$\begin{aligned}
\bar{\ell} = \sum_{i=1}^{m} p_i \ell_i &= \sum_{i=1}^{m} p_i \lceil \log_2 p_i^{-1} \rceil && (3.14) \\
&< \sum_{i=1}^{m} p_i (\log_2 p_i^{-1} + 1) && (3.15) \\
&= \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + \sum_{i=1}^{m} p_i && (3.16) \\
&= \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + 1 && (3.17)
\end{aligned}$$

この $\sum_{i=1}^{m} p_i \log_2 p_i^{-1}$ は情報源のエントロピーにほかならない。そこで，符号化定理と合わせて，

$$\sum_{i=1}^{m} p_i \log_2 p_i^{-1} \leq \bar{\ell} < \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + 1$$

と書くことができる。

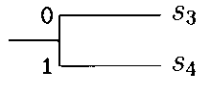
3.8 Huffman のアルゴリズム

上述の Shannon のアルゴリズムは，簡単ではあるが，必ずしも最適な（$\bar{\ell}$ を最小とする）符号を構成しない。$\bar{\ell}$ が最小になるようなプレフィックス符号（**最小冗長符号，minimum-redundancy code**）を構成するために 1952 年に考えられたのが **Huffman のアルゴリズム**である [9]。

話を具体的にするために，4 個のシンボル s_1，s_2，s_3，s_4 からなるアルファベットを考えよう。出現確率は $p_1 \geq p_2 \geq p_3 \geq p_4$，符号語の長さは $\ell_1 \leq \ell_2 \leq \ell_3 \leq \ell_4$ を満たすと仮定しよう。

図 3.1 からもわかるように，プレフィックス符号の木では，最も根から遠くにある葉が，最も長い符号語に相当する。冗長でないプレフィックス符号では，最も長い符号語はペアになって現れる。これらは同じ長さで，同じ親ノードを共有する。

そこで，最も出現確率の小さい二つのシンボル s_3，s_4 は，これらペアをなす二つの最も長い符号語を当てることにしても一般性を損なわない。こうすれば，次のような部分木が構成される。



これらの符号語は最後の 1 ビットだけで区別されることになる。そこで，この最後の 1 ビットを外し，これら二つのシンボルをとりあえず区別しないことにして，両者を合わせたものを s_{34} と表すことにする。すると，s_{34} の出現確率は

$$p_{34} = p_3 + p_4 \tag{3.18}$$

で求められる。

この時点でシンボルは s_1，s_2，s_{34} の 3 種類に減った。この 3 種類のシンボルに対して，p_1，p_2，p_{34} の中から最小の二つを選ぶ。仮に $p_2 \leq p_1$，$p_{34} \leq p_1$ であったならば，シン

ボル s_2 および s_{34} をまとめて1シンボル扱いにする。この時点でシンボルは s_1 と s_{234} の二つに減るので，これらをまとめて木を作れば，図 3.1 の右側の図の木が得られる。逆に，$p_1 \leq p_{34}$，$p_2 \leq p_{34}$ であったならば，シンボル s_1 および s_2 をまとめて1シンボル扱いにする。この時点でシンボルは s_{12} と s_{34} の二つに減るので，これらをまとめて木を作れば，図 3.1 の左側の図の木が得られる。

このようにして，確率最小の2シンボルをまとめるという操作を繰り返せば，どんな場合にも最終的にはアルファベットは一つの木で表される。このようにして構成した木が長さの期待値が最小になることは，構成の仕方から明らかであろう。

Huffman の符号は Shannon の符号より短い $\bar{\ell}$ を与えるが，符号化定理による下限を破ることは当然ながらできない。したがって，Shannon の符号について導いた不等式

$$\sum_{i=1}^{m} p_i \log_2 p_i^{-1} \leq \bar{\ell} < \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + 1$$

はそのまま Huffman の符号にも成り立つ。

3.9 アルファベットの拡大

Huffman の符号化は最適であるが，そのままではまったく役に立たないこともある。たとえばモノクロ2値画像では各ピクセルは黒・白の2値しかとりえず，各ピクセルをビット列で符号化するには，

$$w_{\text{black}} = 0, \qquad w_{\text{white}} = 1$$

のような1ビットでの符号化しかできない。

しかし，たとえば

$$p_{\text{black}} = 0.1, \qquad p_{\text{white}} = 0.9$$

であったとすれば，Shannon の符号化定理からは

$$\bar{\ell} \geq \sum_{i=1}^{2} p_i \log_2 p_i^{-1} = 0.1 \log_2 0.1^{-1} + 0.9 \log_2 0.9^{-1} \approx 0.469$$

が得られる。この違いを埋めることはできないであろうか。

このようなときに便利な一つの方法がアルファベットの拡大（**alphabet extension**）である。上の画像の例では，複数のピクセルをまとめて考えることがこれに当たる。たとえば2ピクセルをまとめて考えれば，

{ 黒黒, 黒白, 白黒, 白白 }

が拡大されたアルファベットになる。こうすれば，シンボルの種類は増えるが，符号化の効率は良くなる。

出現確率が $p_1, \ldots, p_m$ で与えられるシンボルを n 個ずつまとめて符号化することにすれば，符号化定理から

$$\sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} p_{i_1} \ldots p_{i_n} \log_2 (p_{i_1} \ldots p_{i_n})^{-1} \leq n\bar{\ell}$$
$$< \sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} p_{i_1} \ldots p_{i_n} \log_2 (p_{i_1} \ldots p_{i_n})^{-1} + 1$$

ところが

$$\begin{aligned}
&\sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} p_{i_1} \ldots p_{i_n} \log_2 (p_{i_1} \ldots p_{i_n})^{-1} \\
&= \sum_{i_1=1}^{m} \cdots \sum_{i_n=1}^{m} p_{i_1} \ldots p_{i_n} (\log_2 p_{i_1}^{-1} + \cdots + \log_2 p_{i_n}^{-1}) \\
&= \sum_{i_1=1}^{m} p_{i_1} \log_2 p_{i_1}^{-1} \left(\sum_{i_2=1}^{m} \cdots \sum_{i_n=1}^{m} p_{i_2} \ldots p_{i_n} \right) \\
&\quad + \cdots + \sum_{i_n=1}^{m} p_{i_n} \log_2 p_{i_n}^{-1} \left(\sum_{i_1=1}^{m} \cdots \sum_{i_{n-1}=1}^{m} p_{i_1} \ldots p_{i_{n-1}} \right) \\
&= \sum_{i_1=1}^{m} p_{i_1} \log_2 p_{i_1}^{-1} + \cdots + \sum_{i_n=1}^{m} p_{i_n} \log_2 p_{i_n}^{-1} \\
&= n \sum_{i=1}^{m} p_i \log_2 p_i^{-1}
\end{aligned}$$

となるので，

$$n \sum_{i=1}^{m} p_i \log_2 p_i^{-1} \leq n\bar{\ell} < n \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + 1$$

したがって

$$\sum_{i=1}^{m} p_i \log_2 p_i^{-1} \leq \bar{\ell} < \sum_{i=1}^{m} p_i \log_2 p_i^{-1} + 1/n$$

となり，n を大きくすれば 1 シンボルあたりのビット数の範囲をいくらでも狭めることができる。

3.10 ランレングス符号化

アルファベットの拡張の一変種として，**ランレングス符号化**（**run-length encoding**，**RLE**，**連長符号化**）がよく用いられる。特にモノクロ 2 値画像の圧縮に向く。

たとえばピクセルが

$$\underbrace{白白白}_{3}\underbrace{黒黒}_{2}\underbrace{白白白白白}_{5}\underbrace{黒黒}_{3}\ldots$$

のように続くとき，{'黒','白'} を基本的なアルファベットと見ずに，その連（run）の長さを基本的なアルファベットと見て，

$$3, 2, 5, 3, \ldots$$

のような長さの列を出力する。この方法は，G3 ファクシミリや，組版システム TEX で使われる PK フォントの圧縮など，様々なところでよく使われている。

3.11 Golomb-Rice 符号

Golomb 符号 [6] は指数分布に従う整数 $n \geq 0$ を最小冗長符号化するための簡単な方法である。結果的には Huffman のアルゴリズムと等価であるが，符号の構成が簡単なため，よく使われる。

Golomb 符号は一つの整数パラメータ $m > 0$ を持つ。パラメータ m の Golomb 符号で整数 n を符号化するには次のようにする。

- まず '1' を $\lfloor n/m \rfloor$ 個出力する
- 続いて '0' を 1 個出力する
- 最後に $n \bmod m$ をたかだか $\lceil \log_2 m \rceil$ 桁で出力する

整数 n の確率分布が指数分布 $p_n = (1-\rho)\rho^n$ で与えられるとき，最適なパラメータ m の値は

$$m = \left\lfloor \log(1+\rho) / \log(\rho^{-1}) \right\rfloor$$

である。

$m = 2^k$ $(k = 0, 1, 2, \ldots)$ の場合に限ったときの Golomb 符号を Rice 符号ともいう [25]。この場合，上の Golomb のアルゴリズムの最後の部分が

- $n \bmod 2^k$ を 2 進法 k 桁で出力する

言い換えれば

- n を 2 進法で表したときの下 k 桁を出力する

のように簡単になる。

Golomb 符号は $n \geq 0$ の分布が（片側）指数分布のときに最小冗長符号になるのであり，n が Laplace 分布（両側指数分布）のときには Golomb 符号に符号ビットを付けても最小冗長符号にならない。

3.12 算術符号化

話を具体的にするために，白のピクセルが確率 $\frac{3}{4}$，黒のピクセルが確率 $\frac{1}{4}$ で出現するとしよう。

始めに，区間 $[0,1)$ すなわち不等式 $0 \leq x < 1$ で表される区間を考える。白のピクセルが現れたなら，区間の下側 $\frac{3}{4}$ を残して後は捨て，黒のピクセルが現れたなら，区間の下側 $\frac{1}{4}$ を残して後は捨てることにする。

たとえば最初に白が現れたとしよう。区間は $[0,1)$ の下側 $\frac{3}{4}$ の $[0,\frac{3}{4})$ に縮む。次に黒が現れたとしよう。区間は $[0,\frac{3}{4})$ の上側 $\frac{1}{4}$ の $[\frac{9}{16},\frac{3}{4})$ に縮む。このように，区間を次第に狭くしていく。最後に得られた狭い区間の幅を h とすると，その区間内に 2 進法で小数第 $\lceil \log_2(1/h) \rceil$ 位までで表せる点があるので，その小数点以下をビット列と見て出力する。こうすれば，1 シンボルあたりのビット列の長さは，Shannon のエントロピーにいくらでも近づく。

このような区間を使った符号化のアイデアは Shannon [30] にすでに萌芽が見られるが，これを**算術符号化**（**arithmetic coding**）の名前で定式化したのは Rissanen [26] である[注7]。しかし，算術符号化が実際的なアルゴリズムとして使われるようになるまでには，1970–80 年代の多くの研究をまたねばならなかった。

現在でも算術符号化は，基本特許の多くを IBM 等が押さえていること，処理が Huffman 法より複雑でかなり遅くなること，および Huffman 法と他の方法をうまく組み合わせれば算術符号化に近い性能が得られることが多いことから，あまりよく使われているとは言

[注7] Elias が 1960 年代の初めにこのような形のアイデアを考えたが，実現可能なアルゴリズムが思い付かなかったので発表しなかったという伝説がある。しかし，Elias 自身は 1998 年 4 月 22 日に Verdú に宛てた私信でこれを否定している [36]。

いがたい。たとえばファクシミリ等の2値画像のための比較的新しいデータ圧縮標準であるJBIGは，算術符号化を使って高い圧縮率を実現しているが，すでに普及したG3ファクシミリに取って代わるには至っていない。1998年現在，最新のJBIG2のドラフトが発表されている[注8]が，上記のような事情も考慮して，算術符号化を使わないオプションが含まれている。

静止画圧縮のJPEG標準にも，算術符号化を使った可逆圧縮のオプションが入っているが，IBM特許の問題もあり，JPEG対応ソフトでも実際にこのオプションを実装していないものも多い。Independent JPEG Group[注9]が無償配布している広く使われているJPEGの実装にも，算術符号化は含まれていない。

Michael Schindlerは算術符号化の特許を回避するために，G. N. N. Martinの提案[注10]に基づく算術符号化の一変形range encoder（range coder）を提案している[注11]。これは本来の算術符号化より若干圧縮率が落ちる（多くの場合0.01%以内）が，速度は約2倍である[29]。

3.13 高次Markovモデル

英語の文章で最も出現確率の高い文字は“e”である。しかし，もし直前の文字が“q”であれば，最も出現確率の高い文字は“u”に変わる。さらに，もし直前の4文字が“Iraq”であれば，今度は“i”または空白または句読点しか現れない。

このように，出現確率は直前の文字によって大きく変わる。直前の文字を考えないHuffman圧縮では高い圧縮率は期待できない。より高い圧縮率を得るには，直前の文字が“a”であった場合の各文字の出現確率，直前の文字が“b”であった場合の各文字の出現確率，等々を調べておき，それに基づいて符号化すればよい。さらに進めて，直前のn文字のパターン（**文脈**，**context**）ごとに各文字の出現確率を調べ，それに基づいて符号化すれば，nを十分大きくすることによって，圧縮限界にいくらでも近づけることができると考えられる。

このような高次Markovモデルの考え方を実装した圧縮ツールも存在し，確かに高い圧

[注8] `http://www.jpeg.org/public/jbigpt2.htm`

[注9] `http://www.ijg.org/`

[注10] G. N. N. Martin. “Range Encoding: An Algorithm for Removing Redundancy from a Digitised Message.” Presented in March 1979 at the Video & Data Recording Conference, Southampton, July 24-27 1979.

[注11] `http://www.compressconsult.com/rangecoder/`

縮率を達成している。しかし，直前の 2 文字まで考えるだけでも $256^2 = 65536$ 通りの文脈を覚えておかなければならず，膨大な記憶域を必要とするので，敬遠されがちである。

富士通で最近開発された FDLC（Fujitsu Lossless Data Compression）は，単純化した高次 Markov モデルを使って，かなり実用的な水準の使い勝手を達成している [42]。FDLC は富士通が特許を保有している[注12]。

3.14 適応型の圧縮

圧縮しようとするデータの性質がわかっていれば，あらかじめ符号を定め，圧縮側と伸張側に組み込んでおけばよい。しかし，現実にはあらかじめ符号を完全に定めることは不可能な場合が多い。文書ファイルの圧縮に限っても，利用者は同一ソフトで英語の文章も日本語の文章も圧縮したいであろうし，開発者の知らない未知の言語に適用されるかもしれない。計測データの圧縮でも，途中でランダムノイズの標準偏差が大幅に変わるかもしれない。

データ全体を 2 度走査することができるならば，最初の走査でデータの性質を調べて符号を構成して，その符号化のパラメータあるいは符号表そのものを出力し，2 度目の走査でその符号を使って実際に圧縮した結果を出力することができる。しかし，高次モデルを適用する場合には，モデルのパラメータが膨大であり，符号を出力するためのオーバヘッドが非常に大きい。

しかし，データの性質が時間的に変化する場合は，たとえデータ全体について 2 度走査することができても，固定した符号で圧縮するよりも，時々刻々と変化するデータの性質に適応するように符号を変化させながら圧縮するほうがよい。このような方法を**適応型**の（**adaptive** な）圧縮と称する。

適応型の圧縮には，過去のデータだけから符号を決定する方式と，ある程度未来を先読みして符号を決定する方式とがある。

たとえば時系列のデータ

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_{n-1},\ x_n,\ \ldots$$

がある場合，過去のデータだけから符号を決定する方式では，x_n を符号化する方法は $x_0, x_1, x_2, \ldots, x_{n-1}$ あるいはその一部（たとえば最近の k 個 $x_{n-k}, x_{n-k+1}, \ldots, x_{n-1}$）だけに基づいて定める。符号化側も復号側も，これらの過去のデータは既知であるので，

[注12] 公開番号 特許 H08-030432 H08.02.02 および公開番号 特許 H08-223054 H08.08.30

符号化側が符号化の方式を出力しなくても，復号側は復号することができる。これに対して，ある程度未来を先読みして符号を決定する方式では，符号化側は1ブロック分のデータをバッファに読み込んでその性質を調べ，その符号化のパラメータあるいは符号表を出力してから，続いて各データを符号化して出力する。LHA，Zip，gzip など多くの圧縮ツールはこのようなブロックを使っている。

3.15 LZ77 系のアルゴリズム

LZ77は，繰り返し現れる文字列をポインタで置き換える方法である。Ziv と Lempel [43] が 1977 年に発表した論文で取り上げられたのでこのように呼ばれるようになった。

LZ77 の具体的なインプリメント法はいろいろ考えられるが，特に Storer と Szymanski [32] による LZSS という方法とその変形がよく使われている。この方法では，ファイルから過去に読み込んだデータ 4K～32K バイト程度をヒストリーバッファに溜めておき，ファイルの現在位置から 16～256 文字程度先読みして，その文字列と最長一致する文字列をヒストリーバッファから探索する。長さ2バイト以上の最長一致文字列が見つかれば，その文字列のヒストリーバッファ中の位置オフセット（何文字戻ったところと一致したか）と一致長（何文字一致したか）を出力する。

p 文字戻ったところと l 文字一致したことを (p,l) で表すと，たとえば

SWIFT␣AND␣NIFTY␣NIFS

は

SWIFT␣AND␣N$(9,3)$Y␣$(6,3)$S

となる。この変換結果を実際に符号化するには，たとえば文字コードが8ビットで p に 12 ビット，l に4ビット使うとすれば

- 文字そのものは1ビットの '0' と8ビットの文字コード
- (n,m) は1ビットの '1' と 12 ビットの n と4ビットの m

のようにする。

一致文字列がまったくない場合は，8ビットが9ビットに増えるだけであるが，人間の書く文章は非常に反復が多く，このような方法で約半分に縮む。

3.16 LZ78 系のアルゴリズム

LZ77 では，過去に読み込んだデータをそのままヒストリーバッファに溜めるが，LZ78 は，これをやや簡略化して，その中の主な文字列だけ辞書に登録し，次に同じ文字列が現れたときは辞書中の登録番号を出力するようにしたものである。Ziv と Lempel [44] が 1978 年の論文で解析したので，LZ78 と呼ばれる。ただし，Ziv と Lempel の主な目的は，この方法が漸近的に Shannon のエントロピー限界に近づく圧縮をすることの証明であり，実際のアルゴリズムは必ずしも実装向きではなく，見落としもあった。

これを Welch [38] が実装向きの完全なアルゴリズムの形に仕上げたのがいわゆる LZW 法である。圧縮率は LZ77 よりやや劣ることが多いが，LZ77 に比べて高速化が容易であり，Ziv と Lempel の論文に比べて Welch の論文がたいへんわかりやすかったことも手伝って，LZW 法は，UNIX の圧縮ツール compress，MS-DOS の圧縮ツール ARC や PKARC，CompuServe の画像ファイルフォーマット GIF の圧縮形式などとして，広く使われるようになった[注13]。

たとえば

```
NIFTY␣NIFS
```

を LZW では次のような流れで圧縮する。

- LZW は初期化時にすべてのシンボルを辞書に登録する。ここでは 8 ビットで表すことのできる 256 個のシンボルを辞書の 0 から 255 までの項目に登録する。たとえば A は 65，B は 66，C は 67 等々のように ASCII 符号順に登録したとしよう。
- 圧縮すべき文字列を先頭から読む。NI や NIF 等々はまだ辞書に登録されていないので，辞書にある N を見つけ，その登録番号 78 を出力する。この時点で NI という文字列を辞書に登録する。登録番号は 256 になる。
- 次に I の登録番号 73 を出力し，IF を辞書の 257 番の位置に登録する。
- 次に F の登録番号 70 を出力し，FT を辞書の 258 番の位置に登録する。
- 次に T の登録番号 84 を出力し，TY を辞書の 259 番の位置に登録する。

[注13] LZW のアルゴリズムは Sperry Corporation（現在の Unisys Corporation）が，論文が出る 1 年前の 1983 年 6 月 20 日に特許を申請し，1985 年 12 月 10 日に米国特許 4,558,302 として成立していた。Unisys 社は 1995 年に，LZW 特許を使うソフトウェアで 1995 年以降に開発もしくは更新されたものはたとえフリーソフトウェアといえども同社のライセンスが必要であるという方針を明らかにしたため，その後の圧縮ソフトは，すでに広く普及してしまった GIF 画像をサポートする目的以外では，LZW を使わないことが多い。

- 次に Y の登録番号 89 を出力し，Y␣ を辞書の 260 番の位置に登録する。
- 次に ␣ の登録番号 32 を出力し，␣N を辞書の 261 番の位置に登録する。
- 次に NI の登録番号 256 を出力し，NIF を辞書の 262 番の位置に登録する。

要するに，辞書に登録された文字列が見つかればその登録番号を出力し，それより 1 文字だけ長い文字列を登録するというわけである。

辞書に登録される文字列の長さは非常にゆっくりと成長する。理論的には十分長い入力についてはエントロピー的に許される極限まで圧縮できることが証明されるが，現実的な長さの入力に対しては，漸近理論の結果はまったくといっていいほど成り立たない。

3.17 LHA，Zip，gzip 等のアルゴリズム

1988 年から 1991 年にかけて，著者はさまざまな圧縮アルゴリズムを評価し，LZW と比べて当時ほとんど使われていなかった LZSS の高圧縮率に着目し，園田豊英，三木和彦，益山健，吉崎栄泰らと共同で LZSS の改良に着手した [17, 19]。その成果は，圧縮ツール LArc（三木和彦，1988 年），LHarc（吉崎栄泰，1988 年），LHA（吉崎栄泰，1991 年）等に結実した。ここでは現在も広く使われている LHA について，そのアルゴリズムの概要を記す。LHA のアルゴリズムの詳細は付録 A に詳述した。なお，Zip，gzip，zlib，PKWARE 社の PKZIP，Microsoft の CAB[注14] などもほぼ同様なアルゴリズムを踏襲している。

LZ77 で挙げた例

SWIFT␣AND␣NIFTY␣NIFS

を再度考えてみよう。p 文字戻ったところと l 文字一致したことを，さきほどと順序を変えて (l, p) で表すと，

SWIFT␣AND␣N$(3,9)$Y␣$(3,6)$S

となる。これを符号化する際に，l の範囲を $3 \leq l \leq 258$ に限れば，l は 256 通りの値をとる。また，文字そのものが現れたときは，1 文字 8 ビットであるから，やはり 256 通りの値をとる。これらを合わせれば 512 通りの値となる。これを大きさ 512 のアルファベッ

[注14] Microsoft の CAB（Cabinet）は圧縮法というよりはむしろファイル形式で，実際には複数の圧縮形式をサポートしている。そのうち MSZIP は Zip とまったく同じ形式であり，LZX は LZ77 法の環状バッファのサイズを最大 2^{21} バイトまで増し，さらに Intel x86 プロセッサの実行形式を圧縮しやすい形に変換するためのプリプロセッサを付けたものである。

トとして，著者の開発した長さ制限のある近似的な Huffman 法で符号化する。l を符号化した場合は，続いて必ず $p\ (=1,2,3,\ldots)$ が来るが，p は $(\lfloor \log_2 p \rfloor, p - 2^{\lfloor \log_2 p \rfloor})$ という対に変換し，$\lfloor \log_2 p \rfloor$ をやはり長さ制限のある近似的な Huffman 法で符号化し，$p - 2^{\lfloor \log_2 p \rfloor}$ を $\lfloor \log_2 p \rfloor$ ビットの固定長符号で符号化する。

3.18 ブロック整列法

Burrows と Wheeler によるブロック整列法（**block sorting**）は，それ自体は圧縮法ではないが，文字列をより圧縮しやすい形に変換する。英国 Cambridge 大学の Wheeler が AT&T Bell 研究所にいた 1983 年に考案したとされているが，Burrows と共著の技報 [4] が発表されたのは 1994 年である[注15]。

たとえば "NIFTYNIFS" という文字列を考える。まずこれを左に 1 文字ずつ巡回させる：

```
NIFTYNIFS
IFTYNIFSN
FTYNIFSNI
TYNIFSNIF
YNIFSNIFT
NIFSNIFTY
IFSNIFTYN
FSNIFTYNI
SNIFTYNIF
```

これを辞書式順序に並べ換える：

```
FSNIFTYNI
FTYNIFSNI
IFSNIFTYN
IFTYNIFSN
NIFSNIFTY
NIFTYNIFS     ←元の文字列
SNIFTYNIF
TYNIFSNIF
YNIFSNIFT
```

[注15] 著者は 1994～1995 年に書いた総説記事 [20, 21] でこの方法をおそらく初めて日本語で紹介し，それに触発されて日本でもいくつかの実装が試みられたが，第 2 の LHA として広く使われるに至るツールは残念ながら日本では開発されなかった。

この各列の最後の文字を並べると “IINNYSFFT” になる。また，元の文字列は上の表の 6 番目にある。この

IINNYSFFT, 6 番目

という情報だけから元の “NIFTYNIFS” が復元できる。

この “IINNYSFFT” は元の文字列 “NIFTYNIFS” に比べて同じ文字が近くに集まっている。これは一般的に言える傾向である。圧縮法を工夫すれば，元のファイルより圧縮しやすい。

ブロック整列法を併用した圧縮ツールでは Julian Seward の bzip2[注16] と Michael Schindler の szip[注17] が有名である。LHA や Zip，gzip 等と比べて，圧縮・伸張の速度はやや遅いが，ある程度の大きさのファイルでは圧縮率はより高い[注18]。

3.19 音声圧縮

音声圧縮は気圧を数 kHz〜数十 kHz 程度の周波数でサンプリングした時系列計測データの圧縮と考えることができる。不可逆な音声圧縮は，人間の耳の特性をうまく利用して聴覚上の音質劣化を小さくとどめるが，実際にはかなりの情報量の損失を伴うことが多いので，その技術を計測データ圧縮に利用できるわけではない。しかし，可逆な音声圧縮は，計測データの圧縮と非常に近いものである。

まず音声の分野では A/D 変換器から出力されるディジタル信号の類を指し示すために **PCM**（Pulse Code Modulation）という言葉を使う。たとえば現在の音楽 CD（Compact Disc）は，サンプリング周波数 44.1 kHz，16 ビット量子化，2 チャンネル（左・右）の PCM データを無圧縮で記録している。

このような PCM データを圧縮する一つの方法は，直前の値（より一般には直前の数個の値の線形結合）と現在の値との差分（予測誤差）を符号化して送出する **DPCM**（Differential PCM）とその変形である。特に，差分の符号化を状況によって適応的に変化させるような DPCM を **ADPCM**（Adaptive Differential PCM）という。不可逆な方法も可逆な方法も可能であるが，一般には音声の場合は多少の音質劣化は許されるので，不可逆な方法が使われることが多い。

注16 http://www.muraroa.demon.co.uk/

注17 http://www.compressconsult.com/szip/

注18 近年ブロック整列法による圧縮がよく使われる背景の一つとして，LZ77，LZ78 関連で多く成立した特許を避けたいという事情もある。

予測誤差の代わりに予測係数を送る**線形予測符号化**（**Linear Predictive Coding, LPC**）もよく使われている。具体的には，サンプリングされた音声信号 x_t について

$$x_t = a_1 x_{t-1} + a_2 x_{t-2} + \cdots + a_n x_{t-n} + G\varepsilon_t$$

のような線形モデルを仮定し，10 ミリ秒のオーダーのフレームごとに，実データに基づいて最適な係数ベクトル $(a_1, \ldots, a_n)$ と G を求め，それらを送出する。ここで ε_k は白色ノイズまたはパルスである。圧縮というよりは音声合成に近いものであり，元の音声の代わりに合成用のパラメータを送る。

さらに高圧縮が必要な場合は，音声を離散 Fourier 変換や離散余弦変換により周波数領域に変換し，各周波数ごとに人間の聴覚の特性を考慮して情報量を減らす不可逆な方法が用いられる。MPEG Audio や ATRAC（Sony の開発した MD の圧縮法）がこれにあたる。

3.20 SHORTEN

音声圧縮の具体例として，"shorten" [27] という可逆および準可逆（ほとんど可逆）な音声圧縮ソフトを紹介しておく。これは主に 16 kHz サンプリングの 16 ビット線形量子化 PCM データからなる音声認識研究用のコーパスを圧縮して CD-ROM に収める目的で開発されたが，一般の PCM 音声圧縮に適用できる。

これはまず標準で 256 点のサンプル（16 kHz では 16 ミリ秒に相当）ごとにデータを区切り，0 次から 3 次までの多項式予測

$$\hat{x}_t = \begin{cases} 0 \\ x_{t-1} \\ 2x_{t-1} - x_{t-2} \\ 3x_{t-1} - 3x_{t-2} + x_{t-3} \end{cases}$$

のうち最もよくあてはまるものを選び，残差を Golomb-Rice 符号で符号化する。オプションとして，次数 p を指定して線形予測符号化

$$\hat{x}_t = a_1 x_{t-1} + a_2 x_{t-2} + \cdots + a_p x_{t-p}$$

を行うこともできる。データを区切る操作（blocking）のため，実時間圧縮とはいえないが，サンプリングレートが 16 kHz で遅延が 16 ミリ秒といった程度なので，実用上問題はない。

3.21 LOCO-I

JPEG（Joint Photographic Experts Group）は静止画圧縮の国際規格を検討するための専門家グループの名称であるが，その規格自体も JPEG と呼ばれている [24]。JPEG には不可逆な圧縮法と可逆な圧縮法が含まれているが，可逆なものは算術符号化を使っているため，特許の問題で敬遠されることが多く，ほとんどの場合 JPEG といえば不可逆なものを指す。

そこで，従来の JPEG の可逆モードに代わる可逆な静止画圧縮法がいろいろ検討されており，**LOCO-I**（LOw COmplexity LOssless COmpression for Images）がその候補とされている [37]。

LOCO-I は本稿で提案する予測誤差符号化と非常に似た方法を使っているので，ここでやや詳しく解説する。

2 次元静止画の圧縮は，2 次元の行列の形で与えられた数値の集まり

$$\begin{pmatrix} x_{00} & x_{01} & x_{02} & x_{03} & \cdots \\ x_{10} & x_{11} & x_{12} & x_{13} & \cdots \\ x_{20} & x_{21} & x_{22} & x_{23} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

の圧縮にほかならない。符号化器はこれを通常の横書きの文章と同じように左から右に，上から下に読んでいく。LOCO-I は，$x_{i,j-1}$ まで読んだところで，すでに読んだデータだけに基づいて $x_{i,j}$ の値を予測する。具体的には，

$$\hat{x}_{i,j} = \begin{cases} \min(x_{i-1,j}, x_{i,j-1}) & (x_{i-1,j-1} \geq \max(x_{i-1,j}, x_{i,j-1})) \\ \max(x_{i-1,j}, x_{i,j-1}) & (x_{i-1,j-1} \leq \min(x_{i-1,j}, x_{i,j-1})) \\ x_{i-1,j} + x_{i,j-1} - x_{i-1,j-1} & (\text{その他の場合}) \end{cases}$$

を予測値とする。予測誤差 $x_{i,j} - \hat{x}_{i,j}$ は Laplace 分布と仮定し，そのパラメータは過去の 5 値

$$x_{i-1,j-1},\ x_{i-1,j},\ x_{i-1,j+1},\ x_{i,j-2},\ x_{i,j-1}$$

の文脈から推定する。予測誤差は正・負・零の値をとりうるが，

$$(0, 1, -1, 2, -2, 3, -3, \ldots) \rightarrow (0, 1, 2, 3, 4, 5, 6, \ldots)$$

のようにして非負の整数に写像し，Golomb-Rice 符号で符号化する。ただしこの不自然な写像のために予測誤差の期待値が 0 にならないので，予測誤差の分散だけでなく平均値も過去のデータ列から推定している。

3.22 可逆な圧縮の分類

以上で総説した可逆な圧縮法と，そのそれぞれを使った主な圧縮ツールを，一覧の形で示す。

- エントロピー符号化（適応型／非適応型）
  - Huffman 符号化（長さ制限有／無）
    - Golomb-Rice 符号化（指数分布に従う非負の整数の Huffman 符号化）
  - 算術符号化 …… 可逆 JPEG，JBIG
- ランレングス符号化 …… G3 ファクシミリ（Huffman との組合せ）
- Ziv-Lempel（テキスト置換）
  - LZ77 …… LHA，Zip，gzip，zlib，CAB（Huffman との組合せ）
  - LZ78 …… compress，GIF
- Burrows-Wheeler（ブロック整列）…… bzip2（Huffman との組合せ）
  szip（range encoder との組合せ）
- 予測誤差符号化 …… LOCO-I，shorten（Golomb-Rice との組合せ）
  NIFSq（長さ制限のある Huffman との組合せ）

上で NIFSq と書いたものが次章以降で提案する方法である。

なお，ここでは便宜上エントロピー符号化として Huffman 符号化と算術符号化をまとめたが，エントロピー符号化という呼称は単に慣習的なものである。また，エントロピー符号化には Shannon（-Fano）の符号化も入るが，これは Huffman 符号化によって凌駕されたという観点から，省略した。

また，たとえば LHA が LZ77 と Huffman 法の組合せと書いた場合に，多くの重要な点を捨象していることに注意されたい。

第 4 章

新しい圧縮アルゴリズム

4.1 サンプリング・量子化・ノイズ低減

ここでは，**アナログ・ディジタル変換器**（**A/D 変換器**, **analog-to-digital converter**, **ADC**）と呼ばれる装置を使って，時刻 t の関数として与えられる物理量 $f(t)$ をコンピュータに取り込む過程を考察する。なお，実際の多チャンネル計測ではベクトル量 $\big(f_1(t), f_2(t), \ldots, f_m(t)\big)$ を扱うが，ここでは説明を簡単にするため，特定のチャンネルだけに注目して，スカラー量のように扱うことにする。

A/D 変換器の動作はサンプリングと量子化に分けられる。サンプリングとは，離散的な時刻

$$t = t_0,\ t_1,\ t_2,\ \ldots$$

における値

$$\xi_0 = f(t_0),\ \xi_1 = f(t_1),\ \xi_2 = f(t_2),\ \ldots$$

を取り出すことであり，量子化とはこれらの実数値 ξ_k を何らかの方法で離散値に丸めることである。

一般に，A/D 変換器は

$$t = t_0,\ t_0 + \Delta t,\ t_0 + 2\Delta t,\ t_0 + 3\Delta t,\ \ldots$$

のように一定の時間間隔 Δt でサンプリングする。

サンプリング定理によれば，A/D 変換器に入力されるさまざまな周波数成分のうち，**Nyquist 周波数** $1/(2\Delta t)$ までの周波数成分の情報しかとらえることができない。それ以

上の周波数成分を A/D 変換器に入力しても，単にノイズが増すだけである。このノイズのことを**折返し歪**（**aliasing**）という。

そこで，A/D 変換器より前の段階で**ローパスフィルタ**（**low-pass filter**）を通し，Nyquist 周波数以上の成分を除去することにより，ノイズが低減できる（図 4.1）。

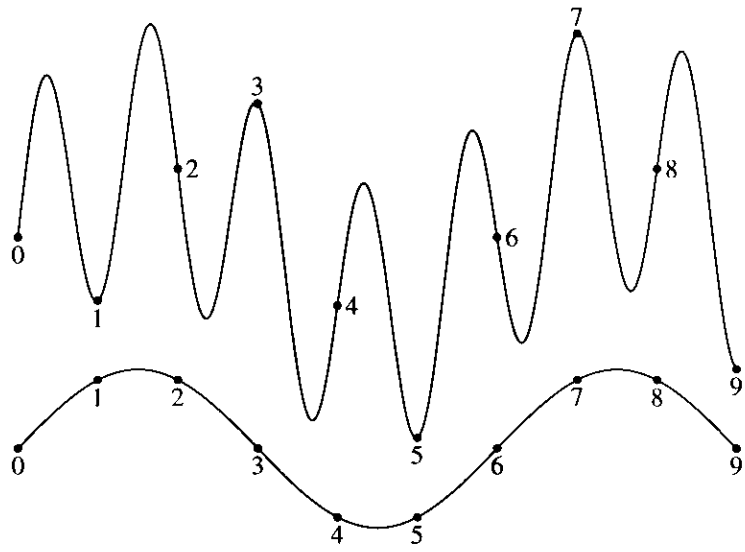


図 4.1: ローパスフィルタによるノイズ低減の概念図。上は元データ（Nyquist 周波数の 1/3 倍および 1.5 倍の正弦波を合成した人工データ）。サンプリングによって折返し歪が混入している。下はローパスフィルタで Nyquist 周波数以上の成分を除去したもの。

さらにノイズを低減するためには，**移動平均**（**moving average**）などのディジタルフィルタが用いられることがある。これは，時系列のデータをたとえば

$$x'_t = \frac{x_{t-1} + 2x_t + x_{t+1}}{4}$$

あるいは

$$x''_t = \frac{x_{t-2} + 4x_{t-1} + 6x_t + 4x_{t+1} + x_{t+2}}{16}$$

のような系列で置き換えるといった手法である。この方法は，確かにノイズを削減できるが，ノイズでない情報まで鈍らせてしまうという弊害がある。

情報を失わずにノイズを除去するには，**オーバーサンプリング**（**oversampling**）に基づく方法がある。これは，たとえば 1 Hz のレートで測定したい物理量を，1000 Hz といった高めのレートでサンプリングし，コンピュータのソフトウェア側で毎秒 1000 点の値の

代表値（たとえば平均値）を求めて，毎秒1点に還元する。これにより，A/D変換器も含めた系のノイズを大幅に低減できる。実際，もしA/D変換器の各サンプル点における誤差が独立であるとすれば，1000点の平均値を求めることによりノイズは

$$1/\sqrt{1000} \approx 1/32 = 2^{-5}$$

に低減でき，5ビット分のランダムノイズが除去できることになる。

あるいは平均値（相加平均）の代わりに**中央値（median）**すなわち大きさの順に並べたときの中央の値を用いることがある。数値の個数が偶数のときは，中央値は中央の2値の平均をとる。たとえば1000点の数値を次のように大きさの順に並べたとする。

$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_{1000}$$

このときの中央値は

$$x_{\text{median}} = \frac{x_{500} + x_{501}}{2}$$

である。

平均値の代わりに中央値を使うと，測定したい量とはまったく関係のないパルス性のノイズの重畳や，A/D変換器またはコンピュータでのデータの取りこぼしのような，いわゆる**外れ値（outliers）**の影響を受けにくくなる。

さらに，平均値と中央値の両方の利点を生かした方法として，大きい方と小さい方から同じ個数の値を取り除いてから平均値を求める**トリムド平均（trimmed mean）**という手法も提案されている。たとえば毎秒1000点のデータの中にパルス性ノイズが仮に最大200点含まれているとすれば，トリムド平均として

$$x_{\text{trimmed mean}} = \frac{x_{101} + x_{102} + \cdots + x_{900}}{800}$$

を採ることにより，ノイズが大幅に低減できる。

中央値やトリムド平均のような技術は，**ロバスト（robust）**な方法と総称され，最近広く使われるようになってきた[34]。ただし現在の本データ処理システムでは専ら平均値を用いている。

これらの代表値による方法は，時間領域での波形が重要な場合には適当であり，イベント検出などに広く用いられているが，音声信号のディジタル化のように周波数領域での特性が重要な場合には，より周波数特性の良いディジタルフィルタにより高域をカットするのが良い[23]。

最近ではオーバーサンプリングとディジタルフィルタの利点をフルに活用する代わりにA/D変換部分を簡略化して1ビット量子化ですませるデルタ・シグマ変調（シグマ・デルタ変調）方式が，特にオーディオ周波数領域のA/D変換によく用いられている[2]。

オーバーサンプリング方式のもう一つの利点として，ノイズ低減の効果だけでなく，サンプリングレートの変更が，Nyquist周波数以上をカットするローパスフィルタの設定や，A/D変換器のサンプリングレートの設定をまったく変えることなく，ソフトウェアの設定だけで容易にできることが挙げられる。

なお，以下ではこのような平均化操作によりダウンサンプリングされたデータ列だけを考え，そのデータ列の1サンプリング周期 Δt を時間の単位に取り，測定開始時刻を時刻の原点とすることによって，離散化された時刻を $t = 0, 1, 2, \ldots$ のように整数で書くことにする。したがって，$t = 0$ における物理量が ξ_0，$t = 1$ における物理量が ξ_1 等々である。

こうして得られた実数値 ξ_t の量子化については，多くの物理計測では，出力をある一定量 $\Delta\xi$ の整数倍に制限する一様（線形）な量子化を用いる。この場合，物理量の単位を量子化レベル $\Delta\xi$ に等しく取れば，量子化の操作は実数値を整数値に丸める操作に帰着できる。整数への丸めは，切捨て $x = \lfloor\xi\rfloor$，切上げ $x = \lceil\xi\rceil$，四捨五入 $x = \lfloor\xi + 0.5\rfloor$ などが考えられる[注1]が，本質的な違いはないので，以下ではこれらの操作を $x = \mathrm{round}(\xi)$ のように書くことにする[注2]。

4.2 一般化されたランダムウォークモデル

前節で見たように，時刻の関数 $f(t)$ で与えられる物理量は，時間と物理量の単位を適当に取れば，一様な時間間隔のサンプリングによって

$$\xi_0 = f(0),\ \xi_1 = f(1),\ \xi_2 = f(2),\ \ldots$$

のような実数値の列になり，さらに一様な量子化によって整数値の列

$$x_0 = \mathrm{round}(\xi_0),\ x_1 = \mathrm{round}(\xi_1),\ x_2 = \mathrm{round}(\xi_2),\ \ldots$$

になる。各整数値 x_t の範囲は，12ビットA/D変換器なら $0 \le x_t \le 4095$（または $-2048 \le x_t \le 2047$），16ビットA/D変換器なら $0 \le x_t \le 65535$（または $-32768 \le x_t \le 32767$）

注1 ここで $\lfloor x\rfloor$ は x を超えない最大の整数，$\lceil x\rceil$ は x を下回らない最小の整数である。

注2 第2.2.2節（8ページ）では $\mathrm{round}(x)$ を最も近い整数に丸める関数 $\lfloor x + 0.5\rfloor$ の意味で用いたが，以下では特にこの意味に限定しない。

の範囲である[注3]。

物理学や工学で扱う時系列の計測データは，大きな変動に小さなランダムノイズが重なった形であることが多いので，ここでは量子化される前の量 ξ_t を，次のような一般化されたランダムウォークでモデル化することにしよう。

$$\xi_t = f(\xi_{t-1}, \xi_{t-2}, \dots) + \varepsilon_t \tag{4.1}$$

ここで ε_t は何らかの分布に従う独立な確率変数である。

特に $f(\xi_{t-1}, \dots) = c$（定数）の場合

$$\xi_t = c + \varepsilon_t$$

はホワイトノイズになり，$f(\xi_{t-1}, \dots) = \xi_{t-1}$ の場合

$$\xi_t = \xi_{t-1} + \varepsilon_t$$

は通常のランダムウォークになる。

なお，式 (4.1) は，時刻 t における値 ξ_t を，それ以前の時刻における値だけに依存する関数 $f(\xi_{t-1}, \xi_{t-2}, \dots)$ で予測したときの予測誤差が ε_t であると解釈することもできる。ξ_t の予測値を $\hat{\xi}_t$ と書くと，

$$\begin{aligned} \xi_t &= f(\xi_{t-1}, \xi_{t-2}, \dots) + \varepsilon_t \\ &= \hat{\xi}_t + \varepsilon_t \end{aligned}$$

のように書くことができる。

このような予測式・予測誤差の解釈をすれば，予測式としては

$\hat{\xi}_t =$ 定数（たとえば 0）	無信号	(4.2)
$\hat{\xi}_t = \xi_{t-1}$	直前の値	(4.3)
$\hat{\xi}_t = 2\xi_{t-1} - \xi_{t-2}$	直前の 2 値による 1 次補外	(4.4)
$\hat{\xi}_t = 3\xi_{t-1} - 3\xi_{t-2} + \xi_{t-3}$	直前の 3 値による 2 次補外	(4.5)

などが考えられる。

[注3] 前節で述べたようなオーバーサンプリングにより，量子化レベルより小さな値も再現できる可能性がある。

4.3 離散化した確率分布

具体的なランダムウォークモデルの一つ

$$\xi_t = \xi_{t-1} + \varepsilon_t \tag{4.6}$$

について，量子化された後の整数値

$$x_t = \mathrm{round}(\xi_t), \quad x_{t-1} = \mathrm{round}(\xi_{t-1})$$

を考え，$e_t = x_t - x_{t-1}$ と置けば，式 (4.6) と同じ形の

$$x_t = x_{t-1} + e_t \tag{4.7}$$

に書くことができるが，一般に $e_t = \mathrm{round}(\varepsilon_t)$ ではない。実際，四捨五入 $\mathrm{round}(x) = \lfloor x + 0.5 \rfloor$ を仮定すれば，$|\xi_t - x_t| \leq 0.5$，$|\xi_{t-1} - x_{t-1}| \leq 0.5$ より，$|\varepsilon_t - e_t| \leq 1$ になり，$|\varepsilon_t - \mathrm{round}(\varepsilon_t)| \leq 0.5$ とは明らかに範囲が異なる。

確率変数 ε_t の分布の密度関数を $p(\varepsilon)$ としよう。すなわち，ε_t が $a \leq \varepsilon_t \leq b$ の範囲に入る確率がつねに $\int_a^b p(\varepsilon)d\varepsilon$ で与えられ，特に $\int_{-\infty}^{\infty} p(\varepsilon)d\varepsilon = 1$ であるとする。

丸め round(·) による誤差の分布は幅 1 の一様分布であり，二つの幅 1 の一様分布の差の分布は三角分布

$$p_{\mathrm{triangular}}(x) = \begin{cases} 1 - |x| & (-1 \leq x \leq 1) \\ 0 & (\text{それ以外}) \end{cases}$$

である（これは丸めの関数 round(·) が切上げ・切捨て・四捨五入のいずれでも同じである）。したがって，整数値 $n = e_t$ の確率分布 p_n は，元の ε_t の分布 $p(x)$ と三角分布 $p_{\mathrm{triangular}}(x)$ との畳み込み

$$p_n = \int_{n-1}^{n+1} (1 - |x - n|)p(x)dx \tag{4.8}$$

で与えられる。

一般のランダムウォークモデル

$$\xi_t = f(\xi_{t-1}, \xi_{t-2}, \dots) + \varepsilon_t$$

についても同様な式を導くことができるが，いずれにしても測定できるのは連続量 ε_t ではなく離散量 e_t であり，両者の分布の対応関係を厳密に調べてもあまり意味がないので，

ここでは式 (4.8) を，ランダムウォークモデルの詳細によらず，正規分布や Laplace 分布（両側指数分布）のようなよく知られた連続分布の密度関数 $p(x)$ から離散量の確率分布 p_n を導くための単なる便法と解釈して扱うことにする。

なお，式 (4.8) によって関連づけられる離散分布の分散 $\sigma^2_{\rm disc}$ と連続分布の分散 $\sigma^2_{\rm cont}$ との間には，各積分区間 $\int_{n-1}^{n+1}$ 内で関数 $p(x)$ が急激な変化をしないという仮定の下に，近似的に次の関係がある。

$$\sigma^2_{\rm disc} \approx \sigma^2_{\rm cont} + \int_{-1}^{1} x^2(1-|x|)dx = \sigma^2_{\rm cont} + \frac{1}{6} \tag{4.9}$$

4.4 正規分布モデル

式 (4.8) に現れる密度関数 $p(x)$ の候補としては，平均 0，分散 $\int_{-\infty}^{\infty} x^2 p(x)dx = \sigma^2$ の正規分布

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{4.10}$$

がまず考えられる。

この密度関数を式 (4.8) に代入し，離散量 e_t の確率分布 p_n を導くと，次のようになる。後で数値計算に便利なように，いくつかの形を挙げておく。

$$\begin{aligned} p_n = \frac{1}{\sqrt{2\pi}\sigma} \Bigg\{ & \sigma^2 \left(e^{-\frac{(n-1)^2}{2\sigma^2}} - 2e^{-\frac{n^2}{2\sigma^2}} + e^{-\frac{(n+1)^2}{2\sigma^2}} \right) \\ & -(n-1)\int_{n-1}^{n} e^{-\frac{x^2}{2\sigma^2}}dx + (n+1)\int_{n}^{n+1} e^{-\frac{x^2}{2\sigma^2}}dx \Bigg\} \end{aligned} \tag{4.11}$$

$$\begin{aligned} \sum_{i=n}^{m} p_i = \frac{1}{\sqrt{2\pi}\sigma} \Bigg\{ & \sigma^2 \left(e^{-\frac{(n-1)^2}{2\sigma^2}} - e^{-\frac{n^2}{2\sigma^2}} - e^{-\frac{m^2}{2\sigma^2}} + e^{-\frac{(m+1)^2}{2\sigma^2}} \right) \\ & -(n-1)\int_{n-1}^{n} e^{-\frac{x^2}{2\sigma^2}}dx + \int_{n}^{m} e^{-\frac{x^2}{2\sigma^2}}dx + (m+1)\int_{m}^{m+1} e^{-\frac{x^2}{2\sigma^2}}dx \Bigg\} \end{aligned} \tag{4.12}$$

$$\begin{aligned} \sum_{i=n+1}^{\infty} p_i = & \frac{\sigma}{\sqrt{2\pi}} e^{-\frac{n^2}{2\sigma^2}} \left(1 - e^{-\frac{2n+1}{2\sigma^2}}\right) - \frac{n}{\sqrt{2\pi}\sigma}\int_{n}^{\infty} e^{-\frac{x^2}{2\sigma^2}}dx + \frac{n+1}{\sqrt{2\pi}\sigma}\int_{n+1}^{\infty} e^{-\frac{x^2}{2\sigma^2}}dx \\ & (n = 0, \pm 1, \pm 2, \pm 3, \dots) \end{aligned} \tag{4.13}$$

正規分布の場合のエントロピーは，離散分布を連続分布で近似することによって次のよ

うに簡単に求められる。

$$\begin{aligned} H &\approx -\int_{-\infty}^{\infty} p(x)\log_2 p(x)dx \\ &= -\frac{1}{\sqrt{2\pi\sigma^2}}\int_{-\infty}^{\infty} e^{-x^2/2\sigma^2}\left(\log_2\frac{1}{\sqrt{2\pi\sigma^2}} - \frac{x^2}{2\sigma^2}\log_2 e\right)dx \\ &= \log_2\sigma + \frac{1}{2}\log_2(2\pi e) \approx \log_2\sigma + 2.047\,\text{bits} \end{aligned} \tag{4.14}$$

また，正規分布に従う確率変数の絶対値の期待値は

$$\frac{1}{\sqrt{2\pi\sigma^2}}\int_{-\infty}^{\infty}|x|e^{-x^2/2\sigma^2}dx = \sqrt{\frac{2}{\pi}}\sigma \tag{4.15}$$

である。

4.5 Laplace 分布モデル

式 (4.8) に現れる密度関数 $p(x)$ のもう一つの候補としては，平均 0，絶対値の期待値 $\int_{-\infty}^{\infty}|x|p(x)dx = a$ の Laplace 分布（両側指数分布）

$$p(x) = \frac{1}{2a}\exp\left(-\frac{|x|}{a}\right) \tag{4.16}$$

が考えられる。

予測誤差が Laplace 分布をする先験的な理由は特にない。むしろ先験的な誤差分布としては正規分布が自然である。しかし，特に音声の PCM データについて，予測誤差が正規分布よりむしろ Laplace 分布に近いことが経験的に知られている。おそらく，分散の異なるいくつかの正規分布の確率変数が重畳して，結果的に Laplace 分布の様相を呈するのではないかと考えられる。

Laplace 分布の場合の離散量 e_t の確率分布は

$$p_0 = 1 - a + ae^{-1/a} \tag{4.17}$$

$$p_n = a\left(\frac{e^{1/a}+e^{-1/a}}{2} - 1\right)e^{-|n|/a} \qquad (n = \pm1, \pm2, \pm3, \dots) \tag{4.18}$$

$$\sum_{i=n+1}^{\infty} p_i = \frac{a}{2}\left(1 - e^{-1/a}\right)e^{-n/a} \qquad (n = 0, 1, 2, 3, \dots) \tag{4.19}$$

となる。

Laplace 分布の場合のエントロピーは，離散分布を連続分布で近似することによって次のように簡単に求められる。

$$\begin{aligned} H &\approx -\int_{-\infty}^{\infty} p(x)\log_2 p(x)dx \\ &= -\frac{1}{2a}\int_{-\infty}^{\infty} e^{-|x|/a}\left(\log_2\frac{1}{2a} - \frac{|x|}{a}\log_2 e\right)dx \\ &= \log_2 a + 1 + \log_2 e \approx \log_2 a + 2.443\,\text{bits} \end{aligned} \tag{4.20}$$

また，Laplace 分布に従う確率変数の分散は

$$\frac{1}{2a}\int_{-\infty}^{\infty} x^2 e^{-|x|/a}dx = 2a^2 \tag{4.21}$$

である。

なお，同じ絶対平均偏差 a を持つ正規分布のエントロピーは，式 (4.14)，(4.15) より，

$$\log_2\left(\sqrt{\frac{\pi}{2}}a\right) + \frac{1}{2}\log_2(2\pi e) \approx \log_2 a + 2.373$$

であり，Laplace 分布より小さい。

4.6 具体的な圧縮法の考え方

前節までで見たように，一般のランダムウォークモデル

$$\xi_t = f(\xi_{t-1}, \xi_{t-2}, \dots) + \varepsilon_t$$

を離散化した式

$$x_t = f(x_{t-1}, x_{t-2}, \dots) + e_t = \hat{x}_t + e_t$$

は，過去のデータ x_{t-1}，x_{t-2}，... から予測式

$$\hat{x}_t = f(x_{t-1}, x_{t-2}, \dots)$$

で時刻 t のデータをできるだけ正確に予測したとき，その予測誤差が e_t であることを表すと解釈することができる。

そこで，もし情報の送り手と受け手が同じ予測式 $\hat{x}_t = f(x_{t-1}, \dots)$ を使うならば，送り手は予測誤差 e_t を送るだけで，過去の測定値を知っている受け手は，新しい測定値を $x_t = f(x_{t-1}, \dots) + e_t$ で再現することができる。予測式 $f(\cdot)$ をうまく選んで予測誤差 e_t

をなるべく 0 に近くできれば，0 に近い値ほど短い符号語をあてる可変長符号化で圧縮が可能である。これが予測誤差符号化の基本的な考え方である。

この考え方による圧縮の手順をさらに具体的に説明する。

まず最初に，情報の送り手（符号化器）と受け手（復号器）は，データについて何の予備知識もない。そこで，両者とも最初の値 x_0 の予測値を $\hat{x}_0 = 0$ と定め，予測誤差 e_0 の確率分布を，両者間であらかじめ合意した分布（おそらく一様分布）と仮定し，その分布に応じた符号表（おそらく単純な固定長符号）を作成する。

時刻 $t = 0$ に，最初のサンプル値 x_0 が送り手に与えられる。その値は，おそらく予測値 $\hat{x}_0 = 0$ とは異なるであろう。送り手は，予測誤差 $e_0 = x_0 - \hat{x}_0 = x_0$ を，さきほど定めた符号表により符号化し，受け手に送る。送り手は同じ符号表により復号し，x_0 の値を得る。

この時点で，最初の値 x_0（したがって予測誤差 e_0）が送り手・受け手双方の知るところとなるので，両者は，この知識に基づいて，次の値 x_1 の予測値 $\hat{x}_1$ を既知の値 x_0 に基づいて定める（おそらく $\hat{x}_1 = x_0$ と定めるであろう）。また，その予測誤差 $e_1 = x_1 - \hat{x}_1$ の分布を e_0 に基づいて推定する（たとえば分散 $\sigma^2 = e_0^2$ の正規分布）。この分布を推定する手順も両者間で合意しておけば，e_1 を符号化するための同じ符号表を両者が用意することができる。

時刻 $t = 1$ に，実際に x_1 の値が送り手に与えられる。その値は，おそらく予測値 $\hat{x}_1$ とは異なるであろう。送り手は，予測誤差 $e_1 = x_1 - \hat{x}_1$ を，あらかじめ作っておいた符号表により符号化し，受け手に送る。送り手は同じ符号表により復号し，e_1 および $x_1 = \hat{x}_1 + e_1$ の値を得る。両者は，次の推定値 $\hat{x}_2 = f(x_1, x_0)$ を定め，今までの推定誤差 e_0, e_1 だけに基づいて，次の予測誤差 e_2 の分布を推定し，符号表を作成する。

一般に，時刻 $t-1$ の処理が完了した時点で，送り側も受け側も，その時点までの測定値 $x_0, x_1, \ldots, x_{t-1}$ と，予測誤差 $e_0, e_1, \ldots, e_{t-1}$ を知っている。そこで，送り側も受け側も，あらかじめ定めた予測式 $\hat{x}_t = f(x_{t-1}, x_{t-2}, \ldots)$ によって次の予測値 $\hat{x}_t$ を求め，現在までの予測誤差 $e_0, e_1, \ldots, e_{t-1}$ に従って次の予測誤差 e_t の分布を推定し，その分布に適した符号表を作成する。送り側は，時刻 t における測定値 x_t が得られるとすぐに，予測誤差の実現値 $e_t = x_t - \hat{x}_t$ を計算し，それをすでに構成した符号表で送る。受け側は，同じ符号表でこれを復号して e_t を求め，$x_t = \hat{x}_t + e_t$ により実際の測定値 x_t を得る。

このように続けることにより，送り手・受け手は協調しながら情報を受け渡すことができる。

このような方式では，符号器・復号器で合意が必要なものは，予測式 $\hat{x}_t = f(x_{t-1}, \ldots)$ と，現在までの予測誤差 $e_{t-1}, e_{t-2}, \ldots$ に基づいて次の予測誤差 e_t を符号化するための

符号表を作成する手順である。

4.7 パラメータの推定

この節では，過去における予測誤差の列

$$e_0,\ e_1,\ e_2,\ \ldots,\ e_{t-3},\ e_{t-2},\ e_{t-1}$$

に基づいて e_t の分布を推定し，さらにそれに基づいて e_t を符号化するための符号を作成する問題を考える。

もし測定値の時系列が定常であるならば，できるだけ遠い過去まで遡って，多数の予測誤差を調べ，それに基づいて予測誤差の分布を詳しく調べることができる。

しかし，実際の測定値の時系列は，ほぼ無信号の状態が長く続くこともあれば，急激な動きが続くこともあり，時間とともに統計的性質が大きく変動する。

このような状況では，長い時系列を調べれば調べるほど，かえって現時点での分布からかけ離れてしまう可能性がある。ごく最近の少数個のデータだけに基づく方が正確な判断ができる。このように，データの統計的性質が時間とともに変化してもそれに追随することのできるデータ圧縮法を指すために，26 ページで述べたように，適応型（adaptive）ということばを使う。

ここでは，適応型の圧縮を実現するために，過去の少数個（m 個とする）の予測誤差

$$e_{t-m},\ e_{t-m+1},\ \ldots,\ e_{t-3},\ e_{t-2},\ e_{t-1}$$

だけに基づいて e_t の分布を推測することにする。

個数 m を少なくするほど分布の急激な変化に追随できるようになるが，あまり少ないと分布の推定が困難になる。統計学でよく知られているように，パラメータが 1 個，2 個，3 個，……のモデル系列があるとき，与えられたデータセットから各モデルのパラメータを推定し，それらのモデルを使って新しいデータを予測する場合，パラメータの個数が多すぎるとかえって予測からの外れが大きくなる。データセットが小さい場合，安定して推定できるパラメータの個数は意外に少ないものである。

そこで，たとえば e_t の分布（正確にいえば式 (4.8) で離散分布と関連づけられる連続分布）を平均 0 の正規分布の族 $N(0,\sigma^2)$ だけに限れば，推定すべきパラメータの数をわずか 1 個（σ^2 だけ）にすることができる。

系列 $e_{t-m}, e_{t-m+1}, \ldots, e_{t-1}$ を平均 0，分散 σ^2 の正規分布 $N(0,\sigma^2)$ に従う確率変数の

m 個の実現値とすれば，統計学で良く知られているように，σ^2 の不偏推定値は

$$s^2 = \frac{1}{m}(e_{t-m}^2 + e_{t-m+1}^2 + \cdots + e_{t-1}^2)$$

で与えられる（平均値は推定せず 0 に固定するので，分母は $m-1$ ではなく m になる）。こうして求めた不偏分散 s^2 を使って e_t の分布を $N(0, s^2)$ とするのが一つの方法であろう。なお，厳密にいえば，e_t は整数値しかとらないので，正規分布に従うと仮定すべきものは式 (4.8) などによって関連づけられる連続変数のほうであるが，後述のように，σ^2 の値自体は圧縮には必要ないので，ここでは厳密に区別しない。

ちなみに，時間とともに緩やかに変化する分散 s_t^2 の推定は，このように過去の m 点の 2 乗和の平均を使う方法以外に，$0 < r < 1$ の定数 r を使って，

$$\begin{aligned} s_t^2 &= \frac{e_{t-1}^2 + re_{t-2}^2 + r^2e_{t-3}^2 + r^3e_{t-4}^2 + \cdots}{1 + r + r^2 + r^3 + \cdots} \\ &= (1-r)(e_{t-1}^2 + re_{t-2}^2 + r^2e_{t-3}^2 + r^3e_{t-4}^2 + \cdots) \end{aligned}$$

のように減衰する重みを付けて平均を求める方法がある。この方法は，過去の m 点のデータを覚えておかなくても，簡単な漸化式

$$s_t^2 = rs_{t-1}^2 + (1-r)e_{t-1}^2$$

で計算できるので便利である。

さらに，実際的な問題として，浮動小数点演算はどうしても避けなければならない。この理由は，浮動小数点演算が整数演算より遅いというだけではなく[注4]，計算機によって浮動小数点演算は必ずしも同じ結果を与えないということにある。前節でも述べたように，この種のデータ圧縮は，符号化する側と復号する側とが完全に協調して作業をしなければならない。浮動小数点演算の誤差のために協調作業に狂いが生じると，正しい復号ができなくなる。整数演算であれば，どの計算機でも同じ結果を生じることが保証される。

さらに実際的な問題として，整数演算を使う場合でも，測定値（したがって予測誤差も）が 16 ビットの場合，予測誤差の 2 乗は 32 ビットになり，その和は 32 ビットを超えてしまう。現在普及しているほとんどの計算機は 32 ビットの整数演算について最適化されているので，高速化のために 32 ビットを超える値は扱いたくない。

そこで考えられるのが，2 乗平均の代わりに絶対値の平均

$$a = \frac{1}{m}(|e_{t-m}| + |e_{t-m+1}| + \cdots + |e_{t-2}| + |e_{t-1}|)$$

[注4] 最近の計算機では浮動小数点演算が整数演算より目に見えて遅いという昔からの構図は成り立たなくなっている。

を使う方法である。個数が十分多いならば，絶対値の平均 a は 2 乗平均 s と

$$a = \sqrt{\frac{2}{\pi}} s$$

の関係で結ばれる。したがって，正規分布の分散を推定するには，2 乗平均を使って $\sigma^2 \approx s^2$ とすることも，絶対平均を使って $\sigma^2 \approx \pi a^2/2$ とすることもできる。予測誤差 e_t が厳密に正規分布に従うならば，2 乗和を使う方法の方がやや正確であるが，絶対平均を使う方がロバスト（robust）である（外れ値に影響されにくい）ことが統計学で知られている。

絶対平均を使う方法についても，

$$\begin{aligned} a_t &= \frac{|e_{t-1}| + r|e_{t-2}| + r^2|e_{t-3}| + r^3|e_{t-4}| + \cdots}{1 + r + r^2 + r^3 + \cdots} \\ &= (1-r)(|e_{t-1}| + r|e_{t-2}| + r^2|e_{t-3}| + r^3|e_{t-4}| + \cdots) \end{aligned}$$

あるいは漸化式

$$a_t = r a_{t-1} + (1-r)|e_{t-1}|$$

による方法が考えられる。

4.8 実際のデータによる例

図 4.2, 4.3, 4.4 は，LHD 制御データ処理装置の一つのデータファイル[注5] をグラフ化したものである[注6]。各サンプル値は 0 から 65535 までの整数値である。

時刻 t におけるサンプル値 x_t を，直前の値 x_{t-1}，直前の 2 値の 1 次補外 $2x_{t-1} - x_{t-2}$，直前の 3 値の 2 次補外 $3x_{t-1} - 3x_{t-2} + x_{t-3}$ で予測したときの予測誤差

$$\begin{aligned} e_t^{(0)} &= x_t - x_{t-1} \\ e_t^{(1)} &= x_t - 2x_{t-1} + x_{t-2} \\ e_t^{(2)} &= x_t - 3x_{t-1} + 3x_{t-2} - x_{t-3} \end{aligned}$$

の度数分布をプロットすれば図 4.5 のそれぞれ d0, d1, d2 の折れ線のようになる。このデータに限っては，直前の値だけによる予測誤差 $e_t^{(0)} = x_t - x_{t-1}$ を使うのが最も誤差分散が小さいことがわかる。

この $e_t^{(0)} = x_t - x_{t-1}$ に限って，同じ分散 $E((e_t^{(0)})^2)$ の正規分布と，同じ絶対平均偏差 $E(|e_t^{(0)}|)$ の Laplace 分布を重ねて描いたのが図 4.6 である。

注5 ファイル名は 00010_00001_00000_A_00263_19981122.rsl である。

注6 具体的には，1998 年 11 月 22 日の 1A_LB_U 温度センサ出力を 16 ビット A/D 変換器でディジタイズした生データである。

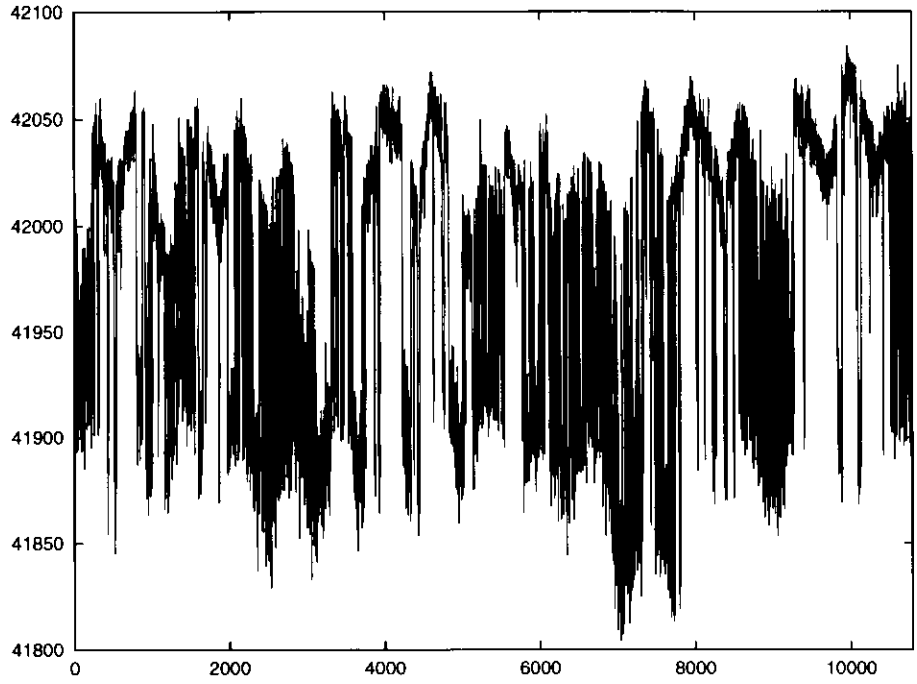


図 4.2: 1998 年 11 月 22 日 0 時 30 分 36 秒から 4 秒ごとの 10794 点（約 12 時間）の温度データ（チャンネル番号 001，測定場所 1A_LB_U)。

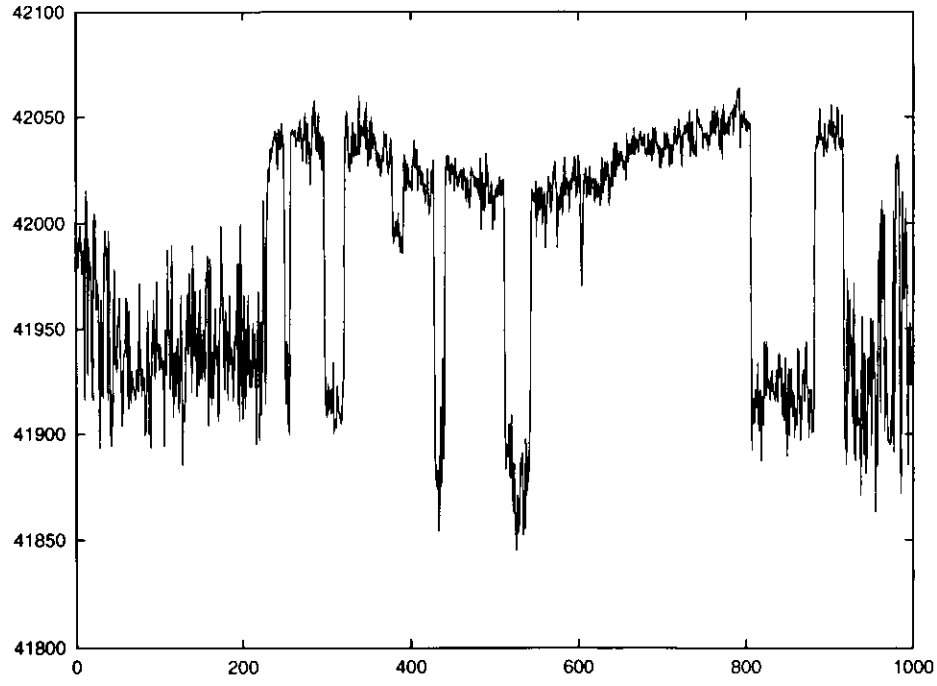


図 4.3: 図 4.2 の先頭 1000 点。

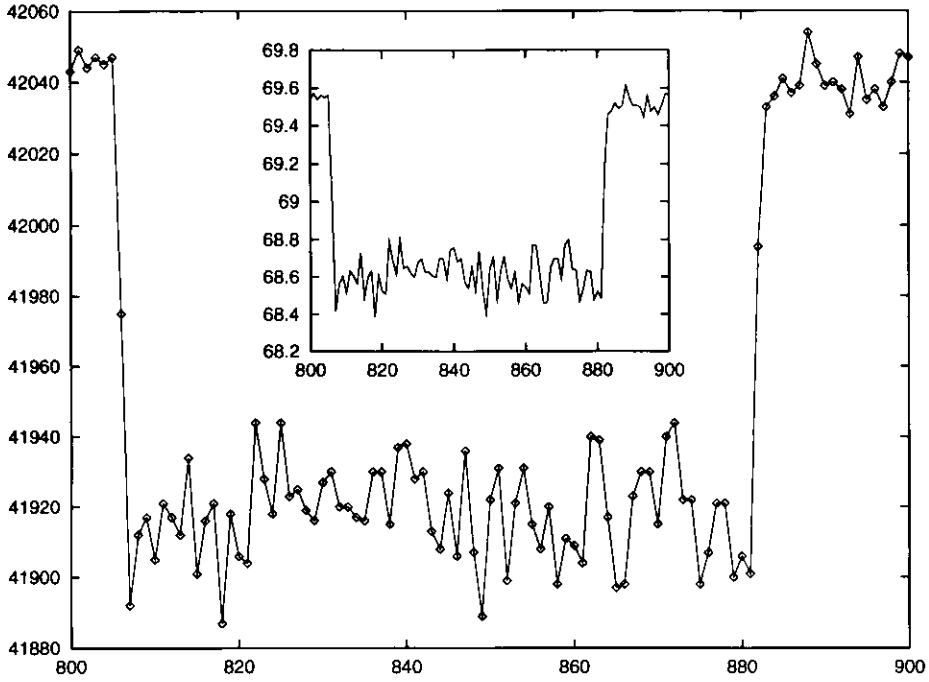


図 4.4: 図 4.2 の 800～900 の 101 点。小さいグラフは物理量（K）に換算したもの。

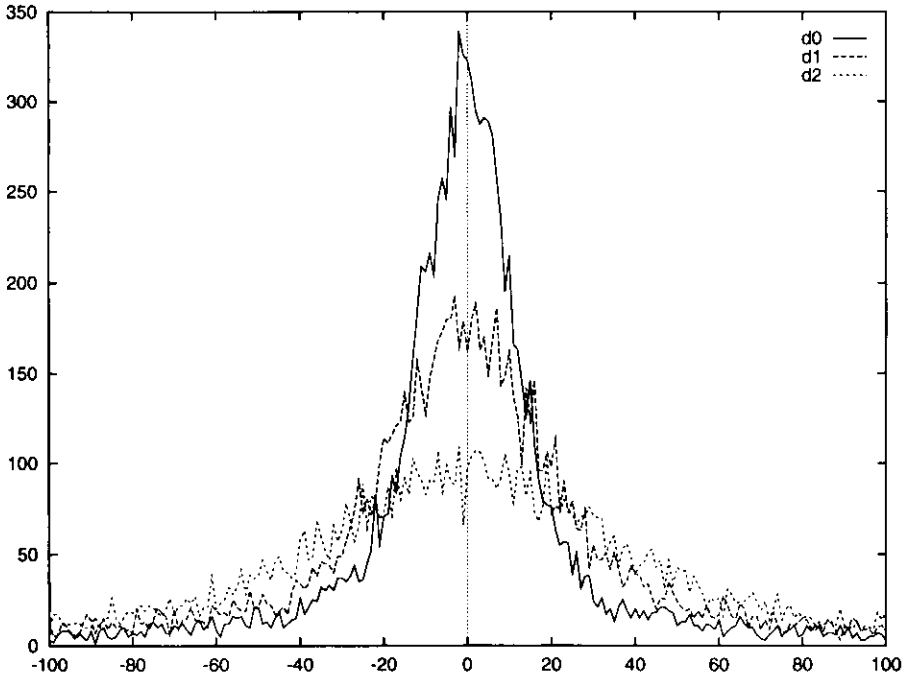


図 4.5: 図 4.2 の各値を過去の数点で予測したときの予測誤差の分布。

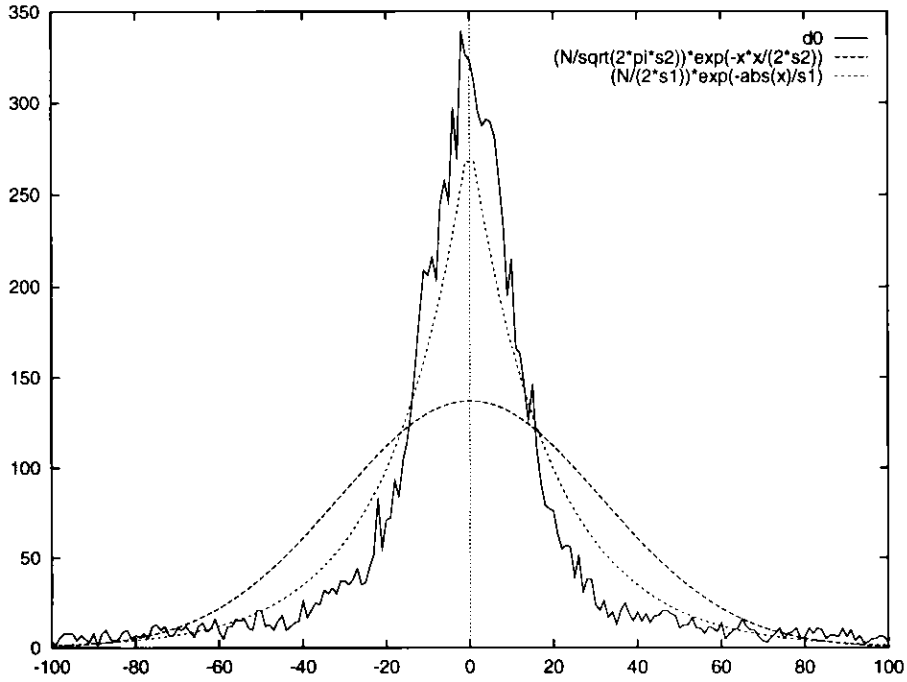


図 4.6: 図 4.2 の各値を直前の 1 点で予測したときの予測誤差の分布と，正規分布，Laplace 分布。

この図 4.6 は正規分布から大きく隔たり，Laplace 分布よりさらに中央の尖った裾の広い分布をする。このことから次のような考察ができる（一部すでに述べたことと重複するが，グラフからの考察であるので重複を厭わず繰り返す）。

- 図 4.4 からは単純なランダムウォークのように見えるが，図 4.2，4.3 をよく見れば，時おり過去の動きからは予測のできない急峻な動きをする。すなわち，いわゆる外れ値が多い。したがって，予測誤差の 2 乗和の平均で分散を推定すると，分布の中央部の度数分布から推定した分散より大きめの値を得る。これが図 4.6 の正規分布が実際の分布と大きく外れる原因と考えられる。対策として，より外れ値に対してロバストな推定量を使うべきである。
- 概してこのような時系列測定データは，小さい変動が続くことや，大きい変動が続くことがある。つまり，分布の分散が緩やかに変化する。これが正規分布や Laplace 分布より裾の広い分布をする理由の一つである。より正確に予測誤差の分布を反映するには，局所的な分布の分散をたとえば

$$z_t = \sqrt{e_{t-1}^2 + e_{t-2}^2 + e_{t-3}^2 + \cdots + e_{t-m}^2}$$

あるいは

$$z_t = |e_{t-1}| + |e_{t-2}| + |e_{t-3}| + \cdots + |e_{t-m}|$$

のような量で推定し，これによって規格化した予測誤差 e_t/z_t を考えるとよいであろう。ロバストな推定量という見地からは，後者の絶対和に基づくものの方が良い。

- 予測誤差 e_t としては，このデータに限っては，直前の値で予測したときの誤差 $e_t^{(0)}$ が最も分散が小さかったが，一般にはこれは必ずしも言えない。そこで，直前の値で予測したときの誤差 $e_t^{(0)}$，直前の 2 値で予測したときの誤差 $e_t^{(1)}$，直前の 3 値で予測したときの誤差 $e_t^{(2)}$ のうちで過去において最も分散が小さいものを e_t として選び，それを用いるとよいであろう。

上の考察に基づき $e_t/z_t = e_t/(|e_{t-1}| + |e_{t-2}| + \cdots + |e_{t-16}|)$ の分布を調べたところ，図 4.7 のようになり，より Laplace 分布に近い分布形が得られる。また，正規分布との隔たりも小さくなる。

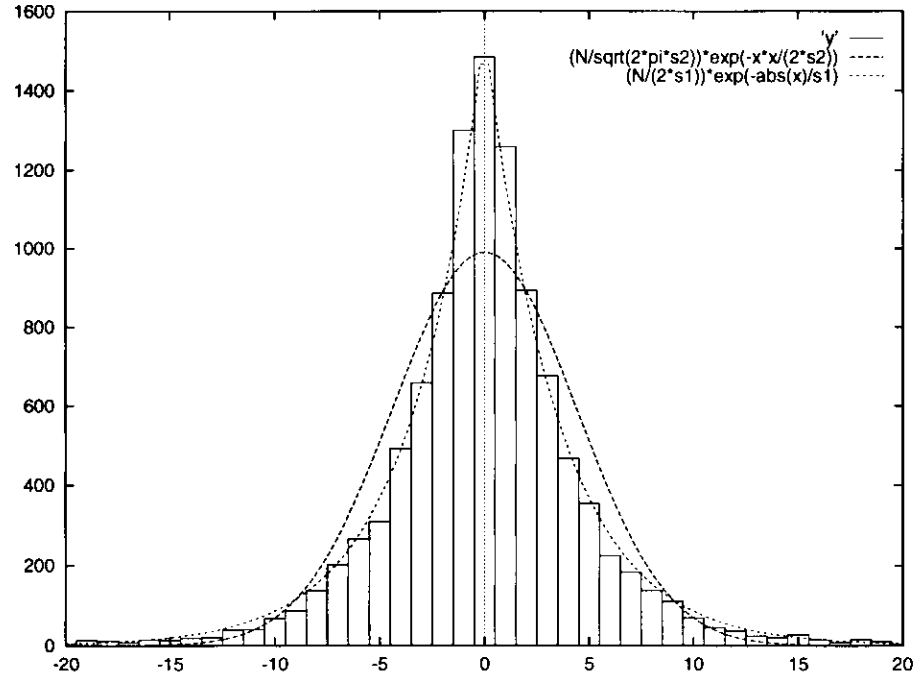


図 4.7: 図 4.6 と同じデータについて，$50e_t/z_t$ の度数分布と，$|e_t|$ から求めた正規分布，Laplace 分布。

以下では，ここまでで述べたことに基づき，実際に符号を構成する手順をより具体的に述べる。

分割の番号	e_t	この分割内で各値を区別するための固定長符号のビット数
0	0	0
1	±1	1
2	±2, ±3	2
3	±4, ..., ±7	3
4	±8, ..., ±15	4
5	±16, ..., ±31	5
...	...	...
14	±8192, ..., ±16383	14
15	±16384, ...	15 (16)

表 4.1: 予測残差の 16 分割。

4.9 符号の構成

まず，測定値 x_t とその予測値 $\hat{x}_t = f(\cdot)$ がいずれも 16 ビット整数であるとしよう。16 ビット整数は，符号付きならば $-32768 \leq x_t \leq 32767$ の範囲，符号なしならば $0 \leq x_t \leq 65535$ の範囲になるが，本質的な違いはないので，以下では話を具体的にするために，符号なしと解釈することにする。

いずれにしても，予測誤差 $e_t = x_t - \hat{x}_t$ は $-65535 \leq e_t \leq 65535$ の範囲になり，これを表すには 17 ビットが必要である。これを 16 ビットの範囲に収めるため，65536 を法とする剰余演算

$$e_t \equiv x_t - \hat{x}_t \pmod{65536}$$

を使うことにする。このようにしても情報量が失われるわけではない。実際，元データはやはり剰余演算

$$x_t \equiv \hat{x}_t + e_t \pmod{65536}$$

で完全に復元できる。予測誤差 e_t は任意の連続した 65536 個の整数値に限ることができるが，以下では $-32768 \leq e_t \leq 32767$ の範囲と定める。

たとえこのようにしても，最大 65536 通りの e_t の値を符号化するための表を用意するのは大掛かりになるので，ここでは次の工夫をすることにした。

まず，表 4.1 のように，e_t の可能な値の範囲を 16 分割し，このうちどれに属するかを Huffman 符号で出力し，次にその分割中での値を区別するための固定長符号を出力することにする。

値	固定長符号
−32767	1111111111111110
−32768	1111111111111111
+32767	0111111111111110
データ終端	0111111111111111

表 4.2: 表 4.1 の例外。分割 15 に便宜上含める。

たとえば $e_t = 27$ を符号化するとしよう。まず 27 は ±16, ... , ±31 の範囲にあるので，分割の番号 5 を可変長符号で出力し，次にこの分割内での 27 の位置を 5 ビットで出力する。具体的には，27 は 2 進法で表すと '11011' となるが，16 から 31 までの整数はすべて 5 ビットで表され，その左端のビットは必ず '1' である。したがって，左端ビットは省略し，代わりに正負の符号を表すために使うことにする。具体的には，正の値 27 では左端が 0 の '01011'，負の値 −27 では左端が 1 の '11011' を出力することにする。

ただし，−32768 とデータ終端だけは上記の規則で符号化することができない。そこで，表 4.2 のものだけ，上記の符号化の例外とする。

次に，上記の 0〜15 の 16 通りの分割のうち，どれに属するかを表す可変長符号の構成法を考えよう。

本節では，確率分布がわかっていると仮定するので，最適な可変長符号（最小冗長符号）の構成は Huffman 法で可能である。しかし，特に復号を高速にする必要から，表引きで復号できることが望ましい。そのため，任意長の符号語を許すのではなく，長さ制限のある Huffman 符号を使うことにし，最大の長さを 8 ビットと定める。8 ビットと定めた理由は，8 ビット符号なら大きさ 256 の表に収まり，実装が簡単になるからである。

次節で，この長さ制限のある Huffman 符号の構成法を示す。

4.10 長さ制限のある Huffman 符号

長さ制限のある Huffman 符号の実現可能な構成法を初めて示したのは Larmore and Hirschberg [13] である。また，最近 Moffat のグループが Huffman 法のアルゴリズムを見直している [12] が，長さ制限のある Huffman 符号についても，比較的高速かつメモリ効率の良い方法が提案されている [14]。

しかし，Moffat たちの方法といえども，長さ制限のある Huffman 符号を構成するのは，通常の Huffman 法に比べてかなりの計算コストを要する。もともと長さ制限を破るシンボルが出現する確率は非常に小さいので，そのような場合は近似的に最適な符号でよしと

1	4444444444444444		399	1235677777777788
2	3444444444444455		400	1235667777778888
3	3344444444445555		401	1235666777888888
4	3344444444444566		402	1235666688888888
5	3334444444555555	...	403	1235577777888888
6	3334444444455566		404	1235567788888888
7	3334444444446666		405	1234777788888888
8	3334444444445677		406	1234678888888888

表 4.3: アルファベットのサイズが 16 のとき，符号語長を 8 ビットに制約した Huffman 木は本質的に 406 通りしかない。その一部を挙げておく。

するならば，もっと簡単なアルゴリズムが存在する。このアルゴリズムの一つ（付録 A 参照）は著者によって LHA のために開発され，Zip や gzip，zlib の Huffman 符号生成部分でも採用されている[注7]。

ここではこれらの方法によらず，次の方法で符号を構成した。

話を具体的にするために，s_0 から s_{15} までの 16 通りのシンボルを符号化し，符号語の最大長を 8 ビットに制限するとしよう。まず各シンボルの出現確率は

$$p_0 \geq p_1 \geq p_2 \geq \cdots \geq p_{15}$$

を満たすと仮定し，各符号語の長さは

$$1 \leq \ell_0 \leq \ell_1 \leq \ell_2 \leq \cdots \leq \ell_{15} \leq 8$$

を満たすとしても一般性を失わない。このとき，最小冗長符号の条件から，Kraft の不等式の不等号を等号で置き換えた式

$$2^{-\ell_0} + 2^{-\ell_1} + \cdots + 2^{-\ell_{15}} = 1$$

を満たさなければならない。このような組合せは 406 通りある。その一部を表 4.3 に挙げておく。

もし長さ制限がなければ，最長の符号語長を与えるのは

$$\ell_0 = 1,\ \ell_1 = 2,\ \ell_2 = 3,\ \ldots,\ \ell_{13} = 14,\ \ell_{14} = 15,\ \ell_{15} = 15$$

[注7] たとえば gzip-1.2.4 の `trees.c` に "This idea is taken from 'ar' written by Haruhiko Okumura." と紹介されている。'ar' は LHA のアルゴリズムをテストするために当時私が C 言語で書いた簡単なアーカイバの名前である。

の場合である。この符号が最適になるような確率分布は $p_i = 2^{-l_i}$ であり，その場合の長さの期待値（ビット）は

$$1 \cdot 2^{-1} + 2 \cdot 2^{-2} + 3 \cdot 2^{-3} + \cdots + 15 \cdot 2^{-15} + 15 \cdot 2^{-15} = 2 - 2^{-14} \approx 1.99994$$

になる。一方，長さ制限のある場合のビット数の期待値は，表 4.3 の 406 番目の符号を使って，

$$1 \cdot 2^{-1} + 2 \cdot 2^{-2} + 3 \cdot 2^{-3} + \cdots + 8 \cdot 2^{-15} + 8 \cdot 2^{-15} \approx 2.04688$$

になり，1 シンボルあたり約 0.047 ビットの損である。しかし，これは最悪の場合であり，通常は 8 ビットに長さ制限しても，ほとんど影響はない。

この 406 通りの符号語長の表をあらかじめ求めておけば，実際に確率分布 $(p_0, \ldots, p_{15})$ が与えられたときに各符号長を求めるのは次のように非常に簡単である。

まず，確率の組 $(p_0, \ldots, p_{15})$ を降順に並べ替え，

$$p'_0 \geq p'_1 \geq p'_2 \geq \cdots \geq p'_{15}$$

が成り立つようにする。次に，この並べ替えた確率と，406 通りの符号語長との内積をおのおの求め，その 406 通りの内積のうちで最小の値を与える符号を採用する。この方法なら，確率 $p_n = 0$ が含まれていても，不自然な操作をしないで符号を定めることができる。

4.11 可変長符号の生成

実際には次のアルゴリズムで可変長符号を生成した。

1. 実数 σ の値を，非常に小さな実数値に設定する。具体的には $\sigma = 0.1$ とした。
2. 連続分布と離散分布を関連づける式 (4.8) を使って，連続分布として正規分布 $N(0, \sigma^2)$ を仮定し，e_t の値として可能なすべての整数値 n について，離散確率分布 p_n を，式 (4.11) または (4.17), (4.18) の数値計算により求める。
3. この p_n を使って最適な可変長符号を構成する。
4. こうして構成した符号語長の期待値が 16 以上になれば，それ以上の S の値については，すべての値を 16 ビットで表す固定長符号を採用することにして，ループを脱出する。
5. σ の値をごくわずか増やして，ステップ 2 に戻る。具体的には，σ を 1.0001 倍した。

このような方法で符号を生成したところ，表 4.3 の 406 通りのうち，正規分布の場合は，実際に使われたのは表 4.8 の 11 通りに過ぎないことがわかった。参考までに，Laplace 分布では 10 通りであった。

正規分布
2233346777888888
2233346688888888
2224446688888888
2223556688888888
2223467788888888
1255666666666666
1246666666666677
1236667777777777
1235777777777777
1234777788888888
1234678888888888

Laplace 分布
2233444667888888
2233444588888888
2233355667888888
2233355588888888
2233346777888888
2233346688888888
2224446688888888
2223467788888888
1234777788888888
1234678888888888

図 4.8: 予測誤差が正規分布と Laplace 分布に従う場合の，長さを 8 ビットに制限した Huffman 符号の実質的に異なる種類。

このことを使えば，実際の符号化が高速に行える。すなわち，出現確率 p_i を大きい順に並べ替え，表 4.8 の 11 個のベクトルとの内積をとり，それが最小になるものを選べばよい。実際の符号を得るには，p_i を並べ替えた際の置換の逆に並べ替える必要がある。

具体的に $\sigma = 4$ の場合に符号生成のアルゴリズムを追っていこう。

まず，正規分布 $N(0, \sigma^2)$ の密度関数 (4.10) に $\sigma = 4$ を代入し，第 4.3 節で導いた諸式を使って，離散値 e_t の分布を計算する。その際，正規分布の密度関数の数値積分が必要になるが，奥村 [18, pp. 227–230] に挙げた不完全ガンマ関数による方法を用いた。表 4.1 の 16 分割のそれぞれに入る確率を求めると，表 4.4 の p_i の欄のようになる。この確率分布から最適な Huffman 符号を適当なプログラム（たとえば奥村 [18, pp. 364–370]）で求めると，各符号語の長さは表 4.4 の ℓ'_i のようになる。

実際には，分割番号 i に入る場合はさらに長さ i の固定長符号が続くので，その長さも含めた符号語長の期待値は

$$\sum_{i=0}^{15} p_i(\ell'_i + i) = 4.276$$

である。

分割番号 i	確率 p_i	Huffman 符号の長さ ℓ'_i	長さを 8 ビット以下に制限した Huffman 符号の長さ ℓ_i	符号語
0	0.0992	3	3	110
1	0.1924	2	2	00
2	0.3256	2	2	01
3	0.3213	2	2	10
4	0.0614	4	4	1110
5	0.0001	5	6	111100
6	$\sim 10^{-15}$	6	7	1111010
7	$\sim 10^{-56}$	7	7	1111011
8	$\sim 10^{-222}$	7	8	11111000
9	~ 0	–	8	11111001
10	~ 0	–	8	11111010
11	~ 0	–	8	11111011
12	~ 0	–	8	11111100
13	~ 0	–	8	11111101
14	~ 0	–	8	11111110
15	~ 0	–	8	11111111

表 4.4: $\sigma^2 = 16$ の場合の計算例。

表 4.4 で，分割番号 9 以降は p_i の値はこの計算機の精度では 0 になり，通常の Huffman 法では符号は定まらない。しかし，これでは万一これらの分類に入る値が出現したときに符号化できなくなる[注8]。

第 4.10 節で示した長さ制限のある Huffman 符号なら，このような不都合はない。最大符号長を 8 ビットに制限した場合の符号語長を ℓ_i の欄に挙げた。この場合の符号語長の期待値は

$$\sum_{i=0}^{15} p_i(\ell_i + i) = 4.276$$

となり，この精度では符号語長の期待値は変わらない。なお，この表の右欄に示した実際の符号は，こうして求めた長さ制限のある符号語長から第 4.12 節のアルゴリズムで定めたものである。

表 4.4 は $\sigma = 4$ の場合に最適な符号である。σ を小刻みに変化させれば，上のような符号はたくさんできる。図 4.9 のたくさん（81 通り）の曲線（実線，破線）は，いろいろな符号について，その符号が最適な σ の値だけでなく，広い範囲の σ の値（横軸）について，それらの符号による符号語長の期待値（縦軸）をグラフにしたものである（この図の

[注8] 通常の Huffman 法で確率 $p_n = 0$ のシンボルに強いて符号語を与えるには，非常に小さい値を各 p_n に加えればよい。

下端近くの直線の破線は式 (4.14) による下限を示す)。

このグラフを見れば，少数の符号だけでも十分全範囲ををカバーすることができることがわかる。そこで，この多数の符号から破線で示した符号を省き，実線で示した 32 通りの符号だけに限ることにした。

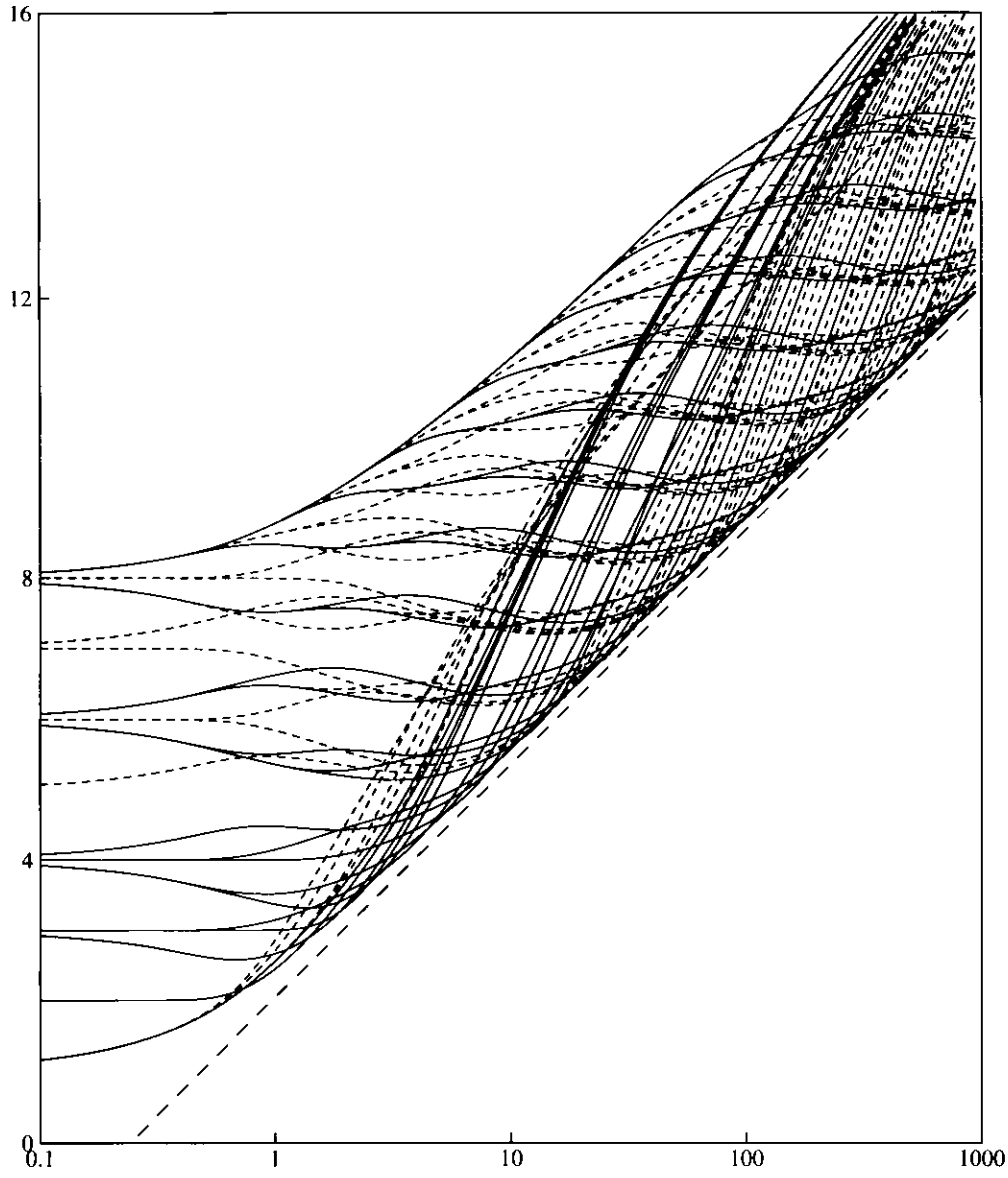


図 4.9: 予測誤差の標準偏差 σ（横軸）を変化させたときの 1 サンプルあたりのビット数の期待値（縦軸)。

81 通りから 32 通りの代表的な符号を選ぶために用いたアルゴリズムは次の通りである。

1. $\sigma = 0.1$ と置く。

2. σ をパラメータとする確率分布を求める。
3. 81 通りの候補それぞれについて，符号語のビット長の期待値 $\bar{\ell}_1$, $\bar{\ell}_2$, ... , $\bar{\ell}_{81}$ を求める。
4. これら 81 個のビット長の中で最小のもの $\bar{\ell}_j$, 2 番目に小さいもの $\bar{\ell}_k$ を求める。
5. 両者の比 $r_j = \bar{\ell}_k/\bar{\ell}_j$ を求める。
6. $\sigma \leftarrow 1.0001\sigma$ と置き換える。
7. $\sigma \leq 32768$ ならステップ 2 に戻る。
8. この時点で r_1, r_2, ... , r_{81} が求められたはずである。これら 81 個の比の中で最小のものが r_m であったとすると，81 個の符号のうち m 番のものは，たとえそれを外して次善のもので置き換えたとしても，損害は僅少であるので，m 番のものを外すことにする。

このようにして 81 個の符号のうち外しても最も影響の少ない 1 個が外され，残りは 80 個となった。これと同様なことを次々に行い，外しても影響の少ない符号を 1 個ずつ外していき，残りが 32 個になったところで停止する。

次に，もう一度 σ の値を 0.1 から始めて少しずつ増していき，それぞれの σ についてこの 32 個の符号のどれが最適かを調べ直す。それと同時に，それぞれの σ について $|e_{t-1}| + |e_{t-2}| + \cdots + |e_{t-16}|$ の期待値を求め，$|e_{t-1}| + |e_{t-2}| + \cdots + |e_{t-16}|$ と最適な符号との対応関係を表にする（過去 16 点にした理由は 76 ページの表 5.10 を参照されたい）。結果は，確率分布が正規分布ならば表 4.5 のようになり，確率分布が Laplace 分布ならば表 4.6 のようになった。

表 4.5 は，たとえば過去の 16 個の絶対値の和が 38 から 56 までなら，可変長符号部分の長さは

$$(\ell_0, \ell_1, \ldots, \ell_{15}) = (3, 2, 2, 2, 4, 6, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8)$$

となることを示す。さらに，たとえば $e_t = 27$ なら，27 は表 4.1 より $i = 5$ の範囲に入るので，$\ell_5 = 6$ ビットを可変長符号部分にあてることになる。具体的にこの 5 ビットの符号がどのようなビット列であるかは，各範囲について表 4.4 のような表を用意して，表引きで出力する（この場合は '111100' である）。

なお，予測誤差の絶対値の和が 205657（正規分布の場合）または 207726（Laplace 分布の場合）以上になると，上述のような形式の符号では期待値が 16 ビットを超えるので，可変長符号，固定長に分けずに，e_t の値をそのまま 16 ビット固定長で出力する。

$\lvert e_{t-1}\rvert + \cdots + \lvert e_{t-16}\rvert$	$\ell_0, \ldots, \ell_{15}$
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–	16 ビット固定長で出力

表 4.5: 正規分布の場合の可変長符号の長さ。

4.12 実際の符号表の生成

前節の表 4.5 は Kraft の不等式を満たす最小冗長符号の各符号語の長さを与えるものであった。これに基づいて実際の符号を構成する方法を述べる。以下では具体例として表 4.5 の

$$38 \leq |e_{t-1}| + \cdots + |e_{t-16}| \leq 56$$

$\lvert e_{t-1}\rvert + \cdots + \lvert e_{t-16}\rvert$	$\ell_0, \ldots, \ell_{15}$
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–	16 ビット固定長で出力

表 4.6: Laplace 分布の場合の可変長符号の長さ。

の場合に相当する符号語長

$$(\ell_0, \ell_1, \ldots, \ell_{15}) = (3, 2, 2, 2, 4, 6, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8)$$

を取り上げる。

まず符号語長を短い順に並べ替える。同じ符号語長については元の順序関係を崩さない

ようにする（安定な整列アルゴリズムを使う）。上の例では次のようになる。

$$(\ell_1, \ell_2, \ell_3, \ell_0, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10}, \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}, \ell_{15})$$
$$= (2, 2, 2, 3, 4, 6, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8)$$

こうしておいてから，次にすべて 0 のビットから始めて，最下位ビットに 1 を加えていく。長さが増したときは，直前の符号の最下位ビットに 1 を加えた上で，右側に新しく追加されたビットはすべて 0 とする。上の例では表 4.7 のようになる。

$\ell_1 = 2$	00	
		+ 01
$\ell_2 = 2$	01	
		+ 01
$\ell_3 = 2$	10	
		+ 01
$\ell_0 = 3$	110	
		+ 001
$\ell_4 = 4$	1110	
		+ 0001
$\ell_5 = 6$	111100	
		+ 000001
$\ell_6 = 7$	1111010	
		+ 0000001
$\ell_7 = 7$	1111011	
		+ 0000001
$\ell_8 = 8$	11111000	
		+ 00000001
$\ell_9 = 8$	11111001	
		+ 00000001
$\ell_{10} = 8$	11111010	
		+ 00000001
$\ell_{11} = 8$	11111011	
		+ 00000001
$\ell_{12} = 8$	11111100	
		+ 00000001
$\ell_{13} = 8$	11111101	
		+ 00000001
$\ell_{14} = 8$	11111110	
		+ 00000001
$\ell_{15} = 8$	11111111	

表 4.7: 符号生成の例。

この方法がうまくいくことは Kraft-McMillan の不等式

$$2^{-\ell_0} + 2^{-\ell_1} + 2^{-\ell_2} + \cdots + 2^{-\ell_{15}} \leq 1$$

から明らかであろう。

4.13 実際の圧縮アルゴリズム（1）

以上で述べた方針にしたがって実際に M チャンネルのデータを圧縮するには次のような手順を踏む。

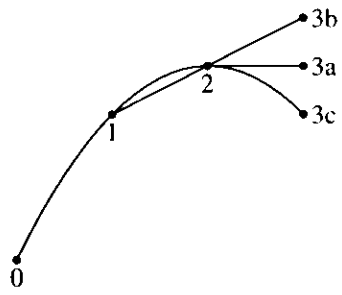


図 4.10: 点 0，1，2 による点 3 の予測の方法：直前の値（3a），1 次補外（3b），2 次補外（3c）。

まず，各時刻 $t = 0, 1, 2, \ldots$ ごとに，M チャンネル分のデータを受け取る。これを

$$x_t[0], x_t[1], x_t[2], \ldots, x_t[M-1] \tag{4.22}$$

としよう。各 $x_t[i]$ はたかだか 16 ビットの整数である。この 16 ビットのビットパターンは連続した 65536 通りの整数値にマッピングされていればよい。0 から 65535 でも -32768 から 32767 でもかまわない。

なお，以下の処理はすべて各チャンネルについて独立に行う[注9] ので，チャンネル番号を表す添字 $[i]$ は多くの場合省略する。

特定のチャンネル i のデータの時系列を，上述のように添字 $[i]$ を省いて

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_t,\ \ldots$$

とする。x_0 以前のデータは存在しないが，便宜上

$$x_{-1} = x_{-2} = \cdots = 0$$

と定める[注10]。

時刻 $t-1$ のデータについての処理を終え，時刻 t のデータ待ちの時点から話を始めよう。

この時点で行うべきことは，各チャンネル i について，現在までの情報に基づいて，次の時刻 t のデータ x_t の最も確からしい値 $\hat{x}_t$ を予測し，さらに予測誤差を符号化するための符号表を選ぶことである。

注9 このため並列演算が可能である。

注10 データについて何らかの先験的な知識があれば，それに基づいて最も確からしい値に置けばよい。

予測値 $\hat{x}_t$ については，過去の数個の値に基づいて，

$$\begin{aligned}
\hat{x}_t^{(0)} &= x_{t-1} && \text{直前の値} \\
\hat{x}_t^{(1)} &= 2x_{t-1} - x_{t-2} && 1\text{次補外} \\
\hat{x}_t^{(2)} &= 3x_{t-1} - 3x_{t-2} + x_{t-3} && 2\text{次補外}
\end{aligned}$$

の中から選ぶ（図 4.10)。

このうち実際にどれを選ぶかについては，過去 16 点（$t-1, \ldots, t-16$）における次の 3 個の和

- 直前の値を選んだときの予測誤差の絶対値の和 $s^{(0)}$
- 1 次補外値を選んだときの予測誤差の絶対値の和 $s^{(1)}$
- 2 次補外値を選んだときの予測誤差の絶対値の和 $s^{(2)}$

を比較し，

もし $s^{(0)} \leq s^{(1)}$ ならば直前の値
そうではなくて，もし $s^{(1)} \leq s^{(2)}$ ならば 1 次補外値
このどちらでもなければ 2 次補外値

というアルゴリズムで選ぶ。なお，これらの 3 個の和 $s^{(0)}$, $s^{(1)}$, $s^{(2)}$ の具体的な計算法については後述するが，いずれも初期値 0 から出発するので，$t = 0$ においては $s^{(0)} = s^{(1)} = s^{(2)} = 0$ であり，予測値 $\hat{x}_0$ としては直前の値 $x_{-1} = 0$ が選ばれる。

また，やはり過去 16 点における予測誤差の絶対値の和

$$s_t = \begin{cases} |e_{t-16}| + |e_{t-15}| + \cdots + |e_{t-1}| & (t \geq 16) \\ 16(|e_0| + |e_1| + \cdots + |e_{t-1}|)/t & (1 \leq t < 16) \\ 2^{31} - 1 & (t = 0) \end{cases} \tag{4.23}$$

の値に従って，表引きにより符号表を選ぶ。$t = 0$ での値 $s_0 = 2^{31} - 1$ は 32 ビット整数の最大値であり，これに対応する符号表はすべての予測誤差を 16 ビット固定長で表す。

このようにして予測値と符号表を準備しておき，実際に時刻 t における測定値 x_t が得られた時点で，予測誤差

$$e_t = x_t - \hat{x}_t \tag{4.24}$$

を符号化して出力する。多チャンネル（M チャンネル）の場合は

$$e_t[0],\ e_t[1],\ e_t[2],\ \ldots,\ e_t[M-1]$$

の順で出力する。ここで1単位時間の遅延も許されない実時間伝達の場合は，出力ビット数が最小出力単位（8ビットまたは16ビット）の整数倍になるようにパッド（詰め物）ビットを出力する。遅延の許されるバッチ処理ではパッドビットは省略する。

4.14 実際の圧縮アルゴリズム（2）

前節で述べたことを効率よく行うため，実際には次のようにしている。

あらかじめ全チャンネル $i = 0, \ldots, M-1$ について

$$e^{(0)}_{\text{prev}} = e^{(1)}_{\text{prev}} = 0$$
$$d^{(0)}_t = d^{(1)}_t = 0 \qquad (t = 0, \ldots, 15)$$
$$s^{(0)} = s^{(1)} = s^{(2)} = 0$$
$$x_{\text{prev}} = 0$$

と置いておく。実際にはこれらの量はすべて添字 $[i]$ を持つが，省略した。

新しいデータ x が得られるごとに，全チャンネル $i = 0, \ldots, M-1$ について次のことを行う。

まず3通りの予測誤差を次のような漸化式で計算する。

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

そして，次のようにして d の値を定める。

$$d = \begin{cases} e^{(0)} & (\text{もし } s^{(0)} \le s^{(1)} \text{ ならば}) \\ e^{(1)} & (\text{上以外でもし } s^{(1)} \le s^{(2)} \text{ ならば}) \\ e^{(2)} & (\text{上のいずれでもないならば}) \end{cases} \tag{4.25}$$

同様に，次のようにして z の値を定める。

$$z = \begin{cases} s^{(0)} & (\text{もし } s^{(0)} \le s^{(1)} \text{ ならば}) \\ s^{(1)} & (\text{もし } s^{(0)} > s^{(1)} \le s^{(2)} \text{ ならば}) \\ s^{(2)} & (\text{上のいずれでもないならば}) \end{cases} \tag{4.26}$$

z の値により符号表を定め，それにより d の値を出力する。

最後に，$p = t \bmod 16$ として

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

および

$$
\begin{aligned}
d_p^{(0)} &\leftarrow |e^{(0)}| \\
d_p^{(1)} &\leftarrow |e^{(1)}| \\
d_p^{(2)} &\leftarrow |e^{(2)}| \\
e_{\mathrm{prev}}^{(0)} &\leftarrow e^{(0)} \\
e_{\mathrm{prev}}^{(1)} &\leftarrow e^{(1)}
\end{aligned}
$$

という置き換えをする。ここで $x \leftarrow y$ は値の置き換え（x の値を y の値にすること）を表す。

これらのデータ構造を保存するための記憶域は 100 バイト/チャンネル程度である。

第5章

性能試験

5.1 人工データによる試験

本稿で提案したアルゴリズムを設計する際に想定したデータは，現在の値が直前の数個の値のたかだか2次式で予測でき，予測誤差が正規分布またはLaplace分布によってモデル化できるような時系列データである。しかし，高速化とオンライン性の確保のため，Huffman符号の長さ制約や簡素化を含むさまざまな工夫がされている。

このアルゴリズムで16ビット/サンプルのデータを圧縮した場合，まったく一定値のデータ，あるいは1次または2次の導関数が一定のデータであれば，1ビット/サンプルまで縮む。逆に，まったくのランダムな16ビットデータであれば，圧縮してもほぼ16ビット/サンプルのままである。現実のデータはこの間に位置する。

ここではまず，意図したデータおよび意図しない分布のデータについて，実際にどれくらい圧縮がなされるかの試験結果を示す。

σ	正規分布	Laplace分布	一様分布	三角分布
1	2.49	2.41	2.51	2.53
4	4.33	4.21	4.37	4.36
16	6.30	6.20	6.33	6.31
256	10.27	10.18	10.30	10.27
4096	14.26	14.17	14.30	14.26

表5.1: ランダムウォークの移動量を種々の乱数にした場合の本アルゴリズムによる圧縮率（ビット/サンプル）。予測誤差は正規分布を仮定した。

σ	正規分布	Laplace 分布	一様分布	三角分布
1	2.53	2.37	2.61	2.58
4	4.32	4.15	4.35	4.34
16	6.36	6.16	6.45	6.39
256	10.30	10.11	10.39	10.32
4096	14.31	14.11	14.44	14.35

表 5.2: ランダムウォークの移動量を種々の乱数にした場合の本アルゴリズムによる圧縮率（ビット/サンプル）。予測誤差は Laplace 分布を仮定した。

表 5.1, 5.2 は，さまざまな連続分布の乱数 ε_t について，ランダムウォークモデル

$$\xi_0 = 32768, \quad \xi_{t+1} = \xi_t + \varepsilon$$

で実数の乱数列を生成し，これを

$$x_t = \lfloor \xi_t + 0.5 \rfloor \bmod 65536$$

のように整数化した長さ 100,000 の疑似データ列を生成し，実際に本稿のアルゴリズムで圧縮した結果を示したものである。最左欄は乱数 ε_t の RMS 値 $\sigma = \sqrt{E(\varepsilon_t^2)}$，他の欄は圧縮後のサイズ（ビット/サンプル）を表す。各分布の密度関数は次の通りである。

正規分布 $$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2}$$

Laplace 分布 $$p(x) = \frac{1}{\sqrt{2}\sigma} e^{-\sqrt{2}|x|/\sigma}$$

一様分布 $$p(x) = \frac{1}{\sqrt{12}\sigma} \qquad (-\sqrt{3}\sigma < x < \sqrt{3}\sigma)$$

3 角分布 $$p(x) = \frac{1 - |x|/\sqrt{6}\sigma}{\sqrt{6}\sigma} \qquad (-\sqrt{6}\sigma < x < \sqrt{6}\sigma)$$

この結果から，このアルゴリズムは，予測誤差分布が正規分布（または Laplace 分布）から外れた場合にも，十分に効果を発揮することがわかる。

5.2 実際の計測データによる試験

表 5.3, 5.4, 5.5, 5.6 は，実際に核融合科学研究所で収集された表 5.7 の 4 個のデータファイルを圧縮した結果である。これらはいずれも 405 チャンネルで，512 バイトのヘッダ部を持つ（詳細なファイル構造は付録 B を参照されたい）。

チャンネル	物理量名	センサ位置	正規分布の場合（ビット/サンプル）				Laplace 分布の場合（ビット/サンプル）			
001	Temp	1A_LB_U	6.673	6.582	5.244	6.010	6.642	6.569	5.292	6.030
002	Temp	1B_LB_UI45	6.895	6.796	5.363	6.193	6.851	6.767	5.416	6.217
003	Temp	1C_RB_OEQ	6.799	6.703	5.278	6.163	6.762	6.687	5.341	6.195
004	Temp	1D_RB_IEQ	6.605	6.514	5.103	5.777	6.579	6.504	5.163	5.788
005	Temp	1E_RSD_U	7.279	7.131	5.576	6.303	7.247	7.112	5.596	6.342
006	Temp	1F_LSD_L	6.905	6.789	5.225	5.836	6.871	6.766	5.301	5.852
007	Temp	1G_TR_OU45	5.155	5.026	3.677	4.266	5.119	4.999	3.740	4.261
008	Temp	1H_TR_UIEQ	5.763	5.629	4.339	4.284	5.709	5.594	4.343	4.289
009	Temp	1I_TR_RIEQ	4.927	4.810	3.546	3.596	4.912	4.809	3.631	3.656
010	Temp	1J_LB_OU45	5.399	5.263	3.910	4.346	5.344	5.219	3.942	4.349
011	Temp	2A_LB_U	5.337	5.215	3.964	3.972	5.288	5.187	3.994	4.006
012	Temp	2B_LB_UI45	5.468	5.331	4.041	3.977	5.413	5.296	4.067	4.009
013	Temp	2C_RB_OEQ	4.884	4.788	3.458	3.806	4.887	4.781	3.539	3.855
014	Temp	2D_RB_IEQ	5.169	5.051	3.560	3.606	5.142	5.021	3.641	3.665
015	Temp	2E_RSD_U	4.483	4.368	3.448	3.993	4.480	4.359	3.506	4.014
016	Temp	2F_LSD_L	4.321	4.107	3.837	4.751	4.340	4.134	3.870	4.760
017	Temp	2G_TR_OL45	4.658	4.494	4.287	4.660	4.681	4.522	4.289	4.655
018	Temp	2H_TR_LIEQ	4.992	4.825	4.356	4.216	4.972	4.822	4.360	4.235
019	Temp	2I_TR_LIEQ	5.219	5.042	4.134	3.652	5.197	5.021	4.140	3.712
020	Temp	2J_RB_OL45	6.006	5.839	4.914	4.869	5.958	5.796	4.983	4.890
021	Temp	2K_RB_L	5.097	4.939	4.139	4.061	5.080	4.928	4.139	4.094
022	Temp	2L_RB_IL45	5.414	5.287	4.529	4.818	5.381	5.244	4.550	4.831
023	Temp	2M_RSD_IEQ	5.254	5.104	4.606	5.119	5.232	5.088	4.632	5.180
024	Temp	2N_LSD_IEQ	5.938	5.796	5.139	5.143	5.923	5.796	5.178	5.200
025	Temp	3A_LB_U	6.628	6.487	5.565	4.658	6.613	6.472	5.604	4.651
026	Temp	3B_LB_UI45	6.637	6.474	5.636	4.826	6.629	6.460	5.662	4.821
027	Temp	3C_RB_OEQ	6.325	6.235	5.060	4.096	6.302	6.229	5.105	4.134
028	Temp	3D_RB_IEQ	6.525	6.397	5.336	4.359	6.514	6.382	5.367	4.370
029	Temp	3E_RSD_U	6.910	6.771	5.588	4.616	6.884	6.742	5.615	4.616
030	Temp	3F_LSD_L	5.788	5.605	3.654	4.612	5.741	5.558	3.706	4.595
031	Temp	3G_TR_OU45	6.271	6.088	3.942	4.591	6.229	6.063	3.991	4.567
032	Temp	3H_TR_UIEQ	4.949	4.693	3.583	4.209	4.951	4.693	3.645	4.199
033	Temp	3I_TR_RIEQ	6.866	6.722	4.674	4.757	6.826	6.698	4.684	4.752
034	Temp	3J_LB_OU45	4.987	4.873	4.020	4.560	4.986	4.864	4.029	4.541
035	Temp	4A_LB_U	6.313	6.143	4.071	4.427	6.276	6.106	4.094	4.417
036	Temp	4B_LB_UI45	6.112	6.063	4.232	4.148	6.105	6.070	4.284	4.165
037	Temp	4C_RB_OEQ	6.564	6.538	4.257	3.821	6.534	6.507	4.288	3.874
038	Temp	4D_RB_IEQ	5.786	5.737	3.677	3.149	5.750	5.710	3.714	3.171
039	Temp	4E_RSD_U	5.224	5.150	3.615	4.726	5.186	5.121	3.649	4.713
040	Temp	4F_LSD_L	4.428	4.316	3.533	4.637	4.446	4.346	3.597	4.637
041	Temp	4G_TR_OL45	4.251	4.068	3.354	4.284	4.252	4.076	3.387	4.288
042	Temp	4H_TR_LIEQ	4.202	4.042	3.341	4.289	4.197	4.056	3.372	4.292
043	Temp	4I_TR_LIEQ	4.572	4.445	3.617	4.572	4.581	4.461	3.681	4.563
044	Temp	5A_LB_U	4.333	4.149	3.411	4.433	4.348	4.180	3.446	4.436
045	Temp	5B_LB_UI45	6.607	6.455	5.889	4.977	6.581	6.440	5.926	4.982
046	Temp	5C_RB_OEQ	6.637	6.472	5.991	4.980	6.611	6.456	6.024	4.983
047	Temp	5D_RB_IEQ	6.555	6.406	5.434	4.293	6.530	6.393	5.489	4.308
048	Temp	5E_RSD_U	5.575	5.378	4.900	4.743	5.545	5.359	4.945	4.741
049	Temp	5F_LSD_L	3.613	3.517	2.710	3.158	3.600	3.499	2.741	3.190
050	Temp	5G_TR_OU45	4.106	4.092	3.182	2.728	4.105	4.089	3.217	2.741
051	Temp	5H_TR_UIEQ	4.648	4.500	3.066	3.381	4.649	4.485	3.098	3.400
052	Temp	5I_TR_RIEQ	3.413	3.350	2.663	2.163	3.390	3.318	2.665	2.147
053	Temp	6A_LB_U	3.823	3.723	2.803	2.142	3.793	3.705	2.830	2.127
054	Temp	6B_LB_UI45	4.720	4.551	3.007	3.291	4.718	4.545	3.014	3.301
055	Temp	6C_RB_OEQ	5.722	5.616	4.174	4.502	5.694	5.581	4.191	4.497
056	Temp	6D_RB_IEQ	5.623	5.552	4.566	4.720	5.582	5.510	4.579	4.717
057	Temp	6E_RSD_U	5.062	5.019	4.120	4.367	5.058	5.012	4.141	4.365
058	Temp	6F_LSD_L	5.567	5.444	3.386	3.124	5.537	5.433	3.476	3.150
059	Temp	6G_TR_OL45	5.482	5.356	3.310	2.918	5.459	5.345	3.380	2.937
060	Temp	6H_TR_LIEQ	4.997	4.937	3.026	3.144	5.000	4.928	3.047	3.169
061	Temp	6I_TR_LIEQ	4.021	3.942	3.115	2.904	4.007	3.933	3.170	2.930
062	Temp	7A_LB_U	4.441	4.337	2.828	2.475	4.446	4.333	2.863	2.475
063	Temp	7B_LB_UI45	4.301	4.312	3.768	4.095	4.306	4.305	3.797	4.101
064	Temp	7C_RB_OEQ	4.213	4.096	3.233	3.472	4.214	4.114	3.267	3.498
065	Temp	7D_RB_IEQ	3.618	3.552	2.802	3.169	3.603	3.544	2.802	3.166
066	Temp	7E_RSD_U	4.116	4.082	3.146	2.588	4.125	4.082	3.186	2.582
067	Temp	7F_LSD_L	3.843	3.775	2.777	2.302	3.823	3.752	2.799	2.294
068	Temp	7G_TR_OU45	3.986	3.945	3.328	3.774	3.973	3.941	3.386	3.787
069	Temp	7H_TR_UIEQ	4.246	4.212	3.302	3.116	4.220	4.178	3.328	3.111
070	Temp	7I_TR_RIEQ	3.936	3.892	3.019	2.671	3.918	3.865	3.068	2.663
071	Temp	8A_LB_U	4.582	4.465	3.136	2.911	4.581	4.452	3.188	2.924
072	Temp	8B_LB_UI45	4.148	4.048	2.552	3.297	4.146	4.030	2.552	3.312
073	Temp	8C_RB_OEQ	4.412	4.394	3.611	3.624	4.383	4.365	3.648	3.636
074	Temp	8D_RB_IEQ	4.031	3.959	2.962	2.061	4.001	3.926	2.994	2.049
075	Temp	8E_RSD_U	5.489	5.351	3.684	3.672	5.439	5.299	3.728	3.707
076	Temp	8F_LSD_L	4.091	4.001	2.994	4.011	4.092	3.985	3.011	4.022
077	Temp	8G_TR_OL45	4.046	3.912	3.029	3.784	4.052	3.926	3.062	3.835
078	Temp	8H_TR_LIEQ	3.687	3.637	3.053	3.515	3.699	3.632	3.070	3.594
079	Temp	8I_TR_LIEQ	3.701	3.599	3.048	3.547	3.710	3.608	3.067	3.628
080	Temp	8J_RB_OL45	4.616	4.446	3.480	3.972	4.626	4.454	3.521	4.007
081	Temp	9A_LB_U	4.799	4.623	3.365	4.159	4.801	4.633	3.418	4.182
082	Temp	9B_LB_UI45	4.988	4.703	3.823	3.327	4.990	4.702	3.853	3.366
083	Temp	9C_RB_OEQ	4.535	4.602	3.956	5.596	4.542	4.607	3.931	5.632
084	Temp	9D_RB_IEQ	3.919	3.946	3.278	4.212	3.940	3.960	3.353	4.206
085	Temp	9E_RSD_U	3.808	3.801	3.334	4.948	3.849	3.845	3.388	4.983
086	Temp	9F_LSD_L	3.342	3.403	2.989	4.161	3.417	3.470	2.986	4.165
087	Temp	9G_TR_OU45	7.687	7.590	7.101	7.558	7.663	7.581	7.111	7.587
088	Temp	9H_TR_UIEQ	3.768	3.776	3.255	4.713	3.813	3.812	3.320	4.731
089	Temp	9I_TR_RIEQ	4.844	4.801	3.937	3.431	4.855	4.806	3.970	3.475
090	Temp	9J_LB_OU45	4.975	4.926	3.907	3.635	4.972	4.924	3.907	3.676
091	Temp	0A_LB_U	4.366	4.304	3.371	3.040	4.387	4.307	3.421	3.051
092	Temp	0B_LB_UI45	5.001	4.929	3.831	3.777	5.006	4.917	3.855	3.810
093	Temp	0C_RB_OEQ	4.906	4.737	3.418	4.410	4.875	4.707	3.474	4.416
094	Temp	0D_RB_IEQ	5.073	4.832	3.309	3.928	5.054	4.821	3.370	3.953
095	Temp	0E_RSD_U	5.229	5.139	4.364	4.922	5.215	5.111	4.366	4.933
096	Temp	0F_LSD_L	5.233	4.952	3.189	3.666	5.210	4.939	3.234	3.712
097	Temp	0G_TR_OL45	4.689	4.435	3.166	3.960	4.685	4.451	3.221	3.985
098	Temp	0H_TR_LIEQ	5.505	5.236	3.416	3.875	5.463	5.189	3.489	3.900
099	Temp	0I_TR_LIEQ	1.487	1.498	1.509	1.593	1.484	1.493	1.507	1.588
100	Temp	0I_TR_LIEQ	1.518	1.520	1.573	1.620	1.515	1.515	1.564	1.612
101	Temp	0I_TR_LIEQ	1.581	1.614	1.579	1.618	1.573	1.607	1.574	1.611

表 5.3: 4 ファイルについてのチャンネルごとの圧縮サイズ（1）

チャンネル	物理量名	センサ位置	正規分布の場合（ビット/サンプル）				Laplace分布の場合（ビット/サンプル）			
102	Temp	0I_TR_LIEQ	1.544	1.523	1.522	1.608	1.538	1.518	1.516	1.602
103	Temp	0I_TR_LIEQ	1.570	1.568	1.507	1.622	1.565	1.564	1.504	1.616
104	Strain	3USA3LIW	3.860	3.706	3.003	3.288	3.871	3.754	3.019	3.283
105	Strain	3USA3LIV	5.514	5.391	2.921	3.583	5.453	5.348	2.920	3.596
106	Strain	3USA9LOW	9.624	9.461	7.382	8.355	9.572	9.428	7.411	8.360
107	Strain	3USA9LOV	8.751	8.579	7.006	8.143	8.732	8.560	7.028	8.146
108	Strain	3USA9LIW	3.973	3.832	2.657	3.106	3.962	3.836	2.688	3.085
109	Strain	3USA9LIV	3.818	3.722	2.591	2.900	3.807	3.713	2.609	2.910
110	Strain	3USA3LOW	5.438	5.332	2.861	3.618	5.392	5.291	2.870	3.632
111	Strain	3USA3LOV	9.167	8.978	7.278	8.372	9.125	8.943	7.315	8.366
112	Strain	3USA15LOW	4.769	4.673	3.099	3.638	4.750	4.652	3.115	3.669
113	Strain	3USA15LOV	4.628	4.516	2.799	3.409	4.580	4.491	2.801	3.398
114	Strain	3USA15LIW	3.995	3.934	2.000	2.612	3.957	3.897	1.983	2.617
115	Strain	3USA15LIV	3.095	2.965	2.319	2.791	3.102	2.990	2.328	2.781
116	Strain	3Uweld3LW	4.051	3.967	2.608	3.435	4.020	3.961	2.604	3.437
117	Strain	3Uweld3LV	2.975	2.807	1.965	2.263	2.941	2.784	1.951	2.263
118	Strain	3Uweld9LW	3.820	3.722	1.955	2.497	3.800	3.699	1.940	2.486
119	Strain	3Uweld9LV	3.683	3.599	2.303	2.828	3.666	3.575	2.299	2.833
120	Strain	3Uweld15LW	4.628	4.504	2.473	3.162	4.599	4.482	2.474	3.180
121	Strain	3Uweld15LV	3.161	3.124	1.934	2.542	3.107	3.071	1.910	2.546
122	Strain	3USA9ROW	3.353	3.187	2.511	2.907	3.340	3.193	2.501	2.906
123	Strain	3USA9ROV	3.780	3.671	2.297	2.678	3.746	3.652	2.288	2.683
124	Strain	3USA9RIW	4.846	4.738	2.576	3.225	4.814	4.705	2.559	3.219
125	Strain	3USA9RIV	3.388	3.283	2.226	2.756	3.337	3.241	2.220	2.766
126	Strain	3USA3ROW	3.285	3.209	2.312	2.597	3.221	3.157	2.315	2.596
127	MicroW	3O-U-Naka	1.466	1.417	1.309	1.444	1.461	1.413	1.306	1.440
128	MicroW	3O-U-Naka	2.173	2.190	2.147	2.931	2.171	2.190	2.183	2.965
129	MicroW	3O-U-Naka	1.310	1.340	1.562	1.469	1.308	1.338	1.552	1.466
130	Temp	045U	11.608	11.611	6.989	1.519	11.626	11.626	6.987	1.510
131	Temp	105U	10.961	10.932	5.545	1.001	10.986	10.950	5.565	1.001
132	Temp	0I_TR_LIEQ	5.711	5.539	4.974	5.259	5.696	5.539	5.033	5.327
133	Temp	0J_RB_OL45	5.691	5.515	4.982	4.706	5.659	5.492	5.017	4.703
134	MicroW	3O-U-Naka	1.101	1.122	1.091	1.582	1.101	1.121	1.091	1.575
135	MicroW	3O-U-Naka	1.585	1.578	1.654	1.395	1.576	1.568	1.648	1.394
136	Temp	A_REF1	1.299	1.292	1.256	1.424	1.296	1.287	1.249	1.424
137	Strain	3USA3RIV	2.113	2.097	1.912	1.872	2.105	2.091	1.889	1.853
138	Strain	3USA15RIW	2.075	2.075	1.999	1.952	2.065	2.065	1.972	1.928
139	Strain	3USA15RIV	4.387	4.316	2.632	2.904	4.395	4.320	2.644	2.902
140	Strain	3Uweld3RW	2.329	2.312	2.340	2.308	2.365	2.358	2.353	2.327
141	Strain	3Uweld3RV	2.375	2.364	2.311	2.287	2.418	2.406	2.323	2.318
142	Strain	3Uweld9RW	2.432	2.405	2.029	2.033	2.449	2.407	1.997	2.014
143	Strain	3Uweld9RV	2.276	2.249	2.164	2.085	2.274	2.260	2.143	2.070
144	Strain	3USA15ROW	3.788	3.742	2.572	2.531	3.839	3.784	2.580	2.559
145	Strain	3USA15ROV	4.400	4.349	2.847	2.793	4.393	4.341	2.881	2.815
146	Strain	3Ucover3W	3.061	3.019	2.193	2.116	3.081	3.049	2.183	2.101
147	Strain	3Ucover3V	2.553	2.525	2.285	2.269	2.580	2.553	2.319	2.283
148	Strain	3Ucover9W	2.604	2.590	2.192	2.149	2.614	2.614	2.195	2.138
149	Strain	3Ucover9V	2.144	2.137	2.081	2.050	2.139	2.130	2.060	2.028
150	Strain	3Ucover15W	2.557	2.534	2.413	2.340	2.602	2.559	2.461	2.383
151	Strain	3Ucover15V	2.541	2.530	2.486	2.344	2.570	2.581	2.502	2.367
152	Strain	3Uweld15RW	2.379	2.346	2.344	2.273	2.418	2.382	2.375	2.287
153	Strain	3Uweld15RV	2.285	2.305	2.263	2.209	2.317	2.317	2.249	2.218
154	Strain	4Lcover27W	2.132	2.158	2.046	2.027	2.133	2.164	2.031	2.007
155	Strain	4Lcover27V	2.263	2.249	2.064	2.094	2.270	2.255	2.061	2.086
156	Strain	5Lcover27W	2.374	2.360	2.264	2.274	2.408	2.394	2.288	2.294
157	Strain	5Lcover27V	2.794	2.735	2.353	2.335	2.812	2.761	2.385	2.360
158	Strain	6Lcover27W	2.418	2.377	2.058	2.102	2.432	2.390	2.030	2.090
159	Strain	6Lcover27V	2.838	2.774	2.156	2.145	2.862	2.789	2.164	2.134
160	Strain	2IVUSCcoT	2.288	2.312	2.258	2.232	2.314	2.338	2.271	2.241
161	Strain	2IVUSCcoR	2.209	2.205	2.218	2.141	2.213	2.223	2.223	2.137
162	Strain	2ISUSCcoT	2.150	2.155	2.163	2.129	2.157	2.161	2.177	2.118
163	Strain	2ISUSCcoR	2.197	2.184	2.165	2.143	2.197	2.189	2.147	2.134
164	Strain	2OVUSCco1T	2.412	2.400	2.336	2.245	2.452	2.434	2.362	2.256
165	Strain	2OVUSCco1R	2.127	2.148	2.137	2.134	2.124	2.142	2.139	2.126
166	Strain	2OVUSCco2T	2.319	2.320	2.296	2.237	2.340	2.351	2.326	2.246
167	Strain	2OVUSCco2R	2.628	2.611	2.562	2.424	2.655	2.634	2.585	2.462
168	Curr	4.75R3HPCC	4.139	4.177	3.058	2.988	4.135	4.170	3.047	3.036
169	Field	4.75R3HEP1	2.641	2.636	2.605	3.180	2.656	2.667	2.603	3.256
170	Field	4.75R3HEP2	3.704	3.712	3.125	3.234	3.758	3.772	3.143	3.318
171	Field	4.75R3HEP3	3.537	3.562	3.035	3.392	3.614	3.631	3.091	3.479
172	Curr	7.75R3STR1	4.777	4.778	3.368	4.514	4.831	4.826	3.375	4.497
173	Curr	7.75R3STR2	5.556	5.551	4.011	4.564	5.572	5.568	4.094	4.550
174	Volt	9.75R3BOT1	2.384	2.398	2.235	5.470	2.417	2.420	2.270	5.482
175	Volt	9.75R3BOT2	2.483	2.512	2.336	6.303	2.523	2.542	2.345	6.343
176	Volt	9.75R3BOT3	2.147	2.155	2.335	6.779	2.140	2.160	2.292	6.774
177	Volt	9.75R3BOT4	2.249	2.273	2.228	5.416	2.285	2.300	2.246	5.439
178	Volt	9.75R3BOT5	2.312	2.303	2.322	6.263	2.340	2.328	2.309	6.315
179	Volt	9.75R3BOT6	2.060	2.065	2.339	6.751	2.059	2.052	2.241	6.747
180	Flux	8.5UVDM1	2.920	2.927	2.448	5.636	2.955	2.964	2.464	5.631
181	Flux	8.5UVDM2	3.822	3.834	3.067	7.390	3.878	3.869	3.109	7.407
182	Curr	8.5UVRG1	11.026	11.043	6.151	5.521	11.065	11.082	6.163	5.550
183	Curr	4.75R3SAR1	2.851	2.834	2.493	3.629	2.871	2.868	2.512	3.707
184	PhotoD	3O-CC01-LU	2.010	2.014	1.529	1.197	1.977	1.992	1.523	1.196
185	PhotoD	3O-CC01-LU	2.098	2.089	1.011	1.037	2.095	2.081	1.011	1.037
186	PhotoD	3O-CC01-LD	2.190	2.212	1.011	1.043	2.189	2.215	1.011	1.043
187	PhotoD	3O-CC01-S1	8.946	8.950	1.072	1.840	8.954	8.973	1.072	1.821
188	PhotoD	3O-CC01-S2	9.343	9.376	1.126	1.950	9.374	9.389	1.126	1.939
189	MicroW	3O-CC01-S3	2.264	2.273	1.011	1.097	2.279	2.282	1.011	1.097
190	PhotoD	3O-CC01-OB	6.648	6.664	1.011	1.065	6.652	6.669	1.011	1.065
191	Temp	A_REF2	1.209	1.283	1.219	1.465	1.208	1.278	1.218	1.461
192	Strain	9Lcover27W	2.575	2.687	2.351	2.264	2.599	2.698	2.400	2.274
193	Strain	9Lcover27V	4.565	4.755	2.862	2.625	4.573	4.746	2.909	2.666
194	Strain	7Lcover27W	2.670	2.817	2.097	2.132	2.696	2.826	2.107	2.128
195	Strain	7Lcover27V	7.968	7.974	5.455	4.809	8.002	8.000	5.500	4.835
196	Strain	8Lcover27W	7.339	7.330	4.674	3.714	7.373	7.366	4.678	3.781
197	Strain	8Lcover27V	2.716	2.844	2.310	2.264	2.730	2.855	2.372	2.284
198	Strain	0Lcover27W	3.023	3.096	2.462	2.488	3.060	3.117	2.503	2.526
199	Strain	0Lcover27V	3.024	3.136	2.107	2.042	3.068	3.162	2.109	2.031
200	Strain	1Lcover27W	2.508	2.636	1.968	1.972	2.510	2.631	1.956	1.955
201	Strain	1Lcover27V	4.337	4.537	2.524	2.259	4.337	4.514	2.557	2.268
202	Strain	2Lcover27W	2.174	2.305	1.858	1.852	2.171	2.283	1.826	1.833

表 5.4: 4 ファイルについてのチャンネルごとの圧縮サイズ（2）

チャンネル	物理量名	センサ位置	正規分布の場合（ビット/サンプル）				Laplace 分布の場合（ビット/サンプル）			
203	Strain	2Lcover27V	2.436	2.565	2.240	2.208	2.451	2.568	2.270	2.211
204	Strain	3Lcover27W	2.341	2.416	2.309	2.175	2.362	2.421	2.285	2.184
205	Strain	3Lcover27V	5.540	5.763	3.261	2.771	5.568	5.778	3.234	2.808
206	Strain	3Lcover21W	4.267	4.454	2.667	2.424	4.266	4.448	2.687	2.454
207	Strain	3Lcover21V	4.791	4.491	2.851	3.322	4.770	4.468	2.835	3.361
208	Strain	3Lcover33W	4.120	3.797	2.865	3.221	4.106	3.779	2.851	3.241
209	Strain	3Lcover33V	4.125	3.866	2.287	2.518	4.075	3.816	2.281	2.534
210	Strain	3Lweld21RW	4.375	4.114	2.707	3.022	4.367	4.099	2.709	3.037
211	Strain	3Lweld21RV	9.923	9.760	7.257	7.703	9.873	9.715	7.246	7.715
212	Strain	3Lweld21LW	3.051	2.955	1.788	1.850	2.986	2.893	1.778	1.833
213	Strain	3Lweld21LV	3.901	3.695	2.333	2.600	3.876	3.664	2.308	2.619
214	Strain	3Lweld33RW	3.215	3.057	1.779	1.905	3.171	3.005	1.762	1.893
215	Strain	3Lweld33RV	3.327	3.213	1.917	1.996	3.286	3.160	1.886	1.981
216	Strain	3Lweld33LW	3.872	3.821	2.190	2.692	3.823	3.778	2.188	2.711
217	Strain	3Lweld33LV	9.920	9.776	7.466	7.692	9.869	9.742	7.471	7.706
218	Strain	3LSA21ROW	3.196	3.102	1.693	1.895	3.115	3.030	1.679	1.877
219	Strain	3LSA21ROV	3.959	3.757	2.357	2.686	3.902	3.704	2.321	2.698
220	Strain	3LSA21RIW	9.727	9.603	7.312	7.623	9.676	9.561	7.323	7.642
221	Strain	3LSA21RIV	3.776	3.588	2.257	2.543	3.715	3.521	2.232	2.557
222	Strain	3LSA21LOW	9.831	9.653	7.259	7.641	9.769	9.605	7.254	7.658
223	Strain	3LSA21LOV	3.535	3.319	2.176	2.437	3.467	3.241	2.133	2.448
224	Strain	3LSA21LIW	3.550	3.311	2.177	2.537	3.508	3.271	2.155	2.542
225	Strain	3LSA21LIV	3.646	3.452	2.195	2.512	3.576	3.387	2.176	2.524
226	Strain	3LSA33ROW	2.409	2.354	1.781	1.848	2.380	2.307	1.761	1.830
227	Strain	3LSA33ROV	3.794	3.568	2.074	2.319	3.737	3.515	2.059	2.314
228	Strain	3LSA33RIW	2.689	2.703	1.758	1.970	2.621	2.634	1.741	1.957
229	Temp	DIV#1	2.562	2.620	2.556	3.365	2.596	2.637	2.582	3.455
230	Temp	DIV#2	2.788	2.853	2.539	3.470	2.820	2.877	2.559	3.548
231	Temp	DIV#3	2.790	2.978	2.814	3.837	2.814	3.013	2.840	3.866
232	Temp	DIV#4	4.068	4.249	3.944	4.908	4.069	4.242	3.977	4.958
233	Temp	DIV#5	4.260	4.460	4.097	5.129	4.246	4.462	4.113	5.178
234	Temp	DIV#6	2.782	2.899	2.701	3.718	2.818	2.935	2.739	3.778
235	Temp	DIV#7	3.493	3.668	3.275	4.240	3.589	3.724	3.310	4.232
236	Temp	DIV#8	4.100	4.207	3.793	4.502	4.106	4.193	3.830	4.490
237	Temp	DIV#9	3.457	3.564	3.210	4.094	3.543	3.644	3.271	4.086
238	Temp	DIV#10	3.638	3.782	3.482	4.388	3.706	3.832	3.498	4.369
239	Strain	3LSA33RIV	7.061	7.031	4.936	4.775	7.091	7.060	4.957	4.785
240	Strain	3LSA33LOW	2.283	2.320	2.159	2.184	2.293	2.337	2.152	2.181
241	Strain	3LSA33LOV	2.679	2.797	2.285	2.279	2.702	2.810	2.302	2.304
242	Strain	3LSA33LIW	2.278	2.312	2.227	2.219	2.309	2.320	2.243	2.239
243	Strain	3LSA33LIV	2.236	2.326	2.046	2.038	2.245	2.324	2.012	2.015
244	Strain	9OVLSCco2T	1.003	1.003	1.011	1.001	1.003	1.003	1.011	1.001
245	Strain	9OVLSCco2R	2.331	2.440	2.141	2.087	2.345	2.431	2.116	2.071
246	Temp	A_REF3	1.234	1.267	1.265	1.296	1.233	1.267	1.265	1.292
247	Temp	DIV#53	2.370	2.364	2.386	2.361	2.409	2.400	2.404	2.403
248	Temp	DIV#54	2.209	2.214	2.254	2.224	2.233	2.220	2.270	2.232
249	Temp	DIV#11	2.936	3.110	2.968	4.041	2.974	3.159	2.999	4.042
250	Temp	DIV#12	2.546	2.719	2.671	3.781	2.568	2.741	2.699	3.823
251	Temp	DIV#13	3.720	3.928	3.633	4.702	3.774	3.956	3.649	4.716
252	Temp	DIV#14	2.655	2.636	2.825	4.091	2.687	2.665	2.868	4.105
253	Temp	DIV#15	2.167	2.236	2.255	3.332	2.174	2.250	2.275	3.426
254	Temp	DIV#16	3.909	4.044	3.632	4.602	3.936	4.063	3.674	4.602
255	Temp	DIV#17	4.228	4.364	3.939	4.831	4.223	4.356	3.959	4.882
256	Temp	DIV#18	2.428	2.552	2.562	3.597	2.454	2.574	2.579	3.653
257	Temp	DIV#19	3.904	4.058	3.811	4.625	3.930	4.069	3.829	4.628
258	Temp	DIV#20	2.575	2.654	2.612	3.711	2.609	2.674	2.630	3.763
259	Temp	DIV#21	3.217	3.378	3.004	3.874	3.307	3.446	3.047	3.894
260	Temp	DIV#22	3.314	3.534	3.236	4.350	3.411	3.594	3.264	4.335
261	Temp	DIV#23	2.867	2.929	2.730	3.683	2.896	2.964	2.764	3.749
262	Temp	DIV#24	3.483	3.561	3.252	4.104	3.568	3.632	3.316	4.101
263	Temp	DIV#25	2.623	2.692	2.652	3.931	2.659	2.716	2.686	3.957
264	Temp	DIV#26	2.996	3.016	2.760	3.420	3.043	3.067	2.778	3.487
265	Temp	DIV#27	4.522	4.681	4.231	5.228	4.522	4.697	4.222	5.292
266	Temp	DIV#28	2.764	2.795	2.841	4.263	2.781	2.815	2.885	4.265
267	Temp	DIV#29	1.375	1.370	1.367	1.405	1.373	1.368	1.366	1.399
268	Temp	DIV#30	1.453	1.443	1.400	1.443	1.448	1.439	1.400	1.439
269	MicroW	1O-M-Naka	2.065	2.054	2.047	2.690	2.053	2.045	2.031	2.721
270	MicroW	10O-M-Naka	2.211	2.220	2.228	3.353	2.221	2.240	2.240	3.352
271	Curr	4.75R3SAR2	1.489	1.497	1.539	1.522	1.488	1.496	1.537	1.517
272	Curr	4.75R3CCR1	1.488	1.488	1.544	1.880	1.486	1.487	1.542	1.848
273	Flux	4.75R3BDM1	1.567	1.555	2.582	4.136	1.563	1.552	2.622	4.134
274	Flux	4.75R3BDM2	1.445	1.436	1.775	4.524	1.442	1.435	1.723	4.481
275	Field	VV-Mag01	1.750	1.772	1.760	2.051	1.739	1.757	1.745	2.037
276	Curr	HCPS-HIRG1	1.738	1.731	2.070	4.531	1.713	1.703	1.987	4.489
277	Curr	HCPS-HMRG1	6.353	6.387	6.277	5.755	6.392	6.422	6.330	5.755
278	Curr	HCPS-HORG1	5.852	5.848	5.824	5.536	5.868	5.880	5.824	5.555
279	Curr	PCPS-1VRG1	5.670	5.671	5.606	5.179	5.664	5.657	5.601	5.269
280	Curr	PCPS-1SRG1	7.889	7.899	7.986	8.260	7.911	7.917	8.031	8.292
281	Curr	PCPS-OVRG1	5.696	5.713	5.656	5.213	5.683	5.711	5.649	5.305
282	Pres	OVU	4.608	4.593	4.554	3.344	4.563	4.543	4.466	3.076
283	D_Pres	OVU	9.203	9.264	9.218	8.400	9.204	9.269	9.241	8.358
284	Temp	OVU	4.111	4.197	4.129	4.827	4.114	4.201	4.146	4.903
285	Temp	OVU	4.739	4.731	4.563	5.218	4.787	4.781	4.572	5.358
286	Temp	OVU	4.960	4.983	4.778	5.484	5.060	5.077	4.833	5.502
287	Temp	OVU	4.771	4.784	4.613	5.286	4.834	4.849	4.625	5.395
288	Pres	OVU	4.003	4.069	4.068	2.521	3.661	3.797	3.780	2.318
289	Temp	OVU	1.341	1.558	1.686	1.928	1.336	1.553	1.680	1.922
290	Temp	OVU	1.419	1.096	1.259	1.683	1.416	1.096	1.259	1.668
291	Temp	OVU	1.437	1.719	1.593	1.219	1.426	1.708	1.586	1.216
292	Pres	ISU	4.601	4.601	4.489	3.318	4.545	4.544	4.401	3.044
293	D_Pres	ISU	7.698	7.739	7.664	6.326	7.697	7.747	7.630	6.302
294	Temp	ISU	4.582	4.649	4.457	5.417	4.596	4.673	4.447	5.464
295	Temp	ISU	2.776	2.798	2.842	2.915	2.793	2.819	2.853	2.949
296	Temp	ISU	2.148	2.177	2.192	1.698	2.166	2.195	2.199	1.681
297	Temp	ISU	2.211	2.201	2.272	1.665	2.221	2.229	2.274	1.650
298	Pres	ISU	3.965	3.995	4.028	2.416	3.642	3.708	3.745	2.234
299	Temp	ISU	1.166	1.774	1.822	1.815	1.165	1.762	1.817	1.807
300	Temp	ISU	1.180	1.168	1.166	1.760	1.178	1.165	1.166	1.742
301	Temp	ISU	1.235	1.722	1.302	1.604	1.233	1.712	1.299	1.594
302	Pres	IVU	4.532	4.531	4.505	3.390	4.487	4.480	4.430	3.122
303	D_Pres	IVU	6.926	6.941	6.823	5.350	6.927	6.963	6.814	5.406

表 5.5: 4 ファイルについてのチャンネルごとの圧縮サイズ (3)

チャンネル	物理量名	センサ位置	正規分布の場合（ビット/サンプル）				Laplace 分布の場合（ビット/サンプル）			
304	Temp	IVU	3.879	3.987	3.821	4.780	3.922	4.022	3.883	4.824
305	Temp	IVU	4.770	4.772	4.621	5.286	4.834	4.828	4.636	5.397
306	Temp	IVU	4.957	4.971	4.787	5.573	5.047	5.074	4.846	5.557
307	Temp	IVU	4.536	4.556	4.372	5.015	4.550	4.575	4.366	5.177
308	Pres	IVU	3.915	3.983	4.009	2.451	3.592	3.683	3.737	2.262
309	Temp	IVU	1.564	1.847	1.078	1.929	1.557	1.832	1.078	1.925
310	Temp	IVU	1.236	1.751	1.949	1.391	1.232	1.745	1.934	1.388
311	Temp	IVU	1.101	1.008	1.061	1.917	1.102	1.008	1.061	1.905
312	Pres	OVL	3.952	3.953	3.939	2.451	3.589	3.647	3.617	2.267
313	D_Pres	OVL	9.744	9.744	9.738	9.003	9.734	9.725	9.738	8.954
314	Temp	OVL	4.512	4.549	4.377	5.370	4.517	4.558	4.365	5.429
315	Temp	OVL	4.779	4.769	4.627	5.212	4.845	4.827	4.642	5.343
316	Temp	OVL	4.835	4.845	4.680	5.266	4.914	4.916	4.708	5.382
317	Temp	OVL	4.452	4.420	4.266	4.629	4.436	4.407	4.257	4.620
318	Pres	OVL	4.380	4.382	4.304	2.627	4.262	4.217	4.093	2.408
319	Temp	OVL	2.067	2.004	1.897	2.362	2.054	1.989	1.889	2.378
320	Temp	OVL	1.169	1.455	1.426	1.468	1.168	1.450	1.421	1.461
321	Temp	OVL	1.427	1.009	2.031	1.891	1.423	1.009	1.995	1.878
322	Pres	ISL	3.927	3.946	3.905	2.411	3.607	3.666	3.641	2.228
323	D_Pres	ISL	7.067	7.032	6.992	5.978	7.072	7.027	6.965	5.965
324	Temp	ISL	3.449	3.445	3.359	4.226	3.545	3.536	3.442	4.222
325	Temp	ISL	3.727	3.637	3.557	2.911	3.790	3.713	3.635	2.932
326	Temp	ISL	2.785	2.789	2.753	1.981	2.819	2.832	2.800	1.948
327	Temp	ISL	3.532	3.427	3.434	3.399	3.636	3.555	3.555	3.489
328	Pres	ISL	4.340	4.285	4.263	2.727	4.250	4.183	4.131	2.494
329	Temp	ISL	1.507	1.858	1.021	1.617	1.495	1.849	1.021	1.606
330	Temp	ISL	1.247	1.093	1.039	1.465	1.242	1.092	1.039	1.458
331	Temp	ISL	1.263	2.026	1.035	1.702	1.260	2.017	1.035	1.691
332	Pres	IVL	4.008	4.037	4.008	2.477	3.696	3.755	3.737	2.282
333	D_Pres	IVL	8.101	8.059	8.112	6.966	8.038	7.988	8.013	6.917
334	Temp	IVL	3.673	3.692	3.534	4.686	3.745	3.762	3.613	4.695
335	Temp	IVL	5.019	5.025	4.840	5.606	5.126	5.132	4.906	5.582
336	Temp	IVL	5.057	5.054	4.836	5.641	5.160	5.155	4.921	5.613
337	Temp	IVL	4.664	4.630	4.503	5.178	4.692	4.653	4.501	5.327
338	Pres	IVL	4.392	4.363	4.371	2.822	4.316	4.272	4.253	2.582
339	Temp	IVL	1.430	1.492	2.609	3.905	1.426	1.486	2.500	3.915
340	Temp	IVL	1.397	1.481	2.323	3.658	1.393	1.475	2.142	3.697
341	Temp	IVL	1.602	1.604	1.527	2.442	1.595	1.595	1.487	2.408
342	Curr	PS_HI	1.007	1.005	1.382	2.178	1.007	1.005	1.377	2.181
343	Curr	PS_HM	1.003	1.003	1.139	2.836	1.003	1.003	1.139	2.865
344	Curr	PS_HO	1.069	1.022	1.753	1.525	1.068	1.022	1.740	1.528
345	Curr	PS_OV	1.399	1.816	1.158	2.727	1.394	1.799	1.156	2.712
346	Volt	H1_I_BL_J	1.629	1.628	2.429	4.749	1.620	1.620	2.385	4.721
347	Volt	H2_I_BL_J	1.629	1.642	2.639	4.660	1.618	1.632	2.462	4.611
348	Volt	H2_O_BL_J	1.592	1.633	2.445	4.681	1.585	1.621	2.383	4.641
349	Volt	H1_O_BL_J	1.574	1.651	2.304	4.537	1.568	1.642	2.210	4.483
350	Volt	H1_M_BL_J	1.556	1.388	2.392	4.557	1.551	1.385	2.305	4.480
351	Volt	H2_M_BL_J	1.741	1.465	2.207	4.470	1.728	1.458	2.158	4.423
352	Volt	H1_I_BL_K	1.628	1.575	2.385	4.658	1.617	1.566	2.350	4.620
353	Volt	H2_I_BL_K	1.586	1.586	2.458	4.652	1.580	1.576	2.287	4.603
354	Volt	H2_O_BL_K	1.637	1.611	2.367	4.672	1.626	1.604	2.294	4.627
355	Volt	H1_O_BL_K	1.629	1.655	2.227	4.530	1.620	1.648	2.129	4.472
356	Volt	H1_M_BL_K	1.567	1.598	2.413	4.517	1.559	1.586	2.323	4.451
357	Volt	H2_M_BL_K	1.639	1.674	2.256	4.446	1.629	1.661	2.182	4.395
358	Volt	OV_BL_J	1.675	1.433	2.247	4.251	1.665	1.428	2.185	4.203
359	Volt	IS_BL_J	1.563	1.626	2.170	4.326	1.555	1.619	2.106	4.278
360	Volt	IV_BL_J	1.698	1.542	2.188	4.372	1.690	1.536	2.125	4.337
361	Volt	OV_BL_K	1.610	1.512	2.101	4.243	1.599	1.507	2.024	4.190
362	Volt	IS_BL_K	1.751	1.526	2.154	4.331	1.739	1.522	2.091	4.286
363	Volt	IV_BL_K	1.577	1.396	2.308	4.374	1.569	1.393	2.240	4.340
364	Curr	PS_IS	1.959	1.970	1.101	3.328	1.953	1.968	1.100	3.397
365	Curr	PS_IV	1.211	1.218	1.768	2.240	1.210	1.218	1.759	2.242
366	Volt	PS_HI_1	1.576	1.644	1.778	2.042	1.557	1.619	1.750	2.041
367	Volt	PS_HM_1	1.125	1.144	1.377	1.877	1.125	1.143	1.374	1.856
368	Volt	PS_HO_1	1.658	1.719	1.948	2.054	1.639	1.699	1.943	2.054
369	Volt	PS_OV_1	1.896	1.968	2.508	2.296	1.872	1.937	2.517	2.324
370	Volt	PS_IS_1	3.236	3.698	3.621	3.358	3.239	3.655	3.654	3.409
371	Volt	PS_IV_1	2.145	2.209	2.197	2.233	2.194	2.258	2.180	2.257
372	Volt	PS_HI_2	1.680	1.680	2.149	2.278	1.662	1.661	2.201	2.278
373	Volt	PS_HM_2	1.954	1.927	1.878	2.133	1.945	1.913	1.867	2.146
374	Volt	PS_HO_2	1.722	1.719	1.954	1.969	1.708	1.702	1.941	1.959
375	Volt	PS_OV_2	5.125	5.406	4.914	4.974	5.166	5.418	5.015	5.013
376	Volt	PS_IS_2	2.490	2.957	2.890	2.541	2.474	2.984	2.936	2.612
377	Volt	PS_IV_2	1.821	2.300	2.087	2.171	1.794	2.284	2.116	2.185
378	BV	H-I	1.462	1.383	1.829	1.904	1.456	1.378	1.809	1.846
379	BV	H-M	1.423	1.481	1.522	1.925	1.419	1.476	1.507	1.857
380	BV	H-O	1.479	1.612	1.819	1.811	1.474	1.604	1.793	1.741
381	Volt	H1I	1.304	1.206	1.221	1.532	1.302	1.204	1.221	1.526
382	Volt	H1M	1.684	1.538	1.485	1.636	1.677	1.530	1.477	1.624
383	Volt	H1O	1.219	1.221	1.663	1.573	1.218	1.220	1.651	1.561
384	Volt	H2I	1.501	1.189	1.098	1.869	1.497	1.187	1.098	1.858
385	Volt	H2M	1.541	1.451	1.235	1.626	1.533	1.444	1.235	1.615
386	Volt	H2O	1.368	1.698	1.817	1.762	1.366	1.688	1.804	1.747
387	JV	H1_I_JV_J	1.467	1.141	1.205	1.709	1.459	1.141	1.204	1.682
388	JV	H1_I_JV_K	1.656	1.558	1.427	1.944	1.643	1.553	1.421	1.920
389	JV	H1_M_JV_J	1.271	1.436	1.453	1.803	1.268	1.432	1.446	1.780
390	JV	H1_M_JV_K	1.392	1.565	1.375	1.928	1.387	1.559	1.368	1.915
391	JV	H1_O_JV_J	1.453	1.483	1.403	1.838	1.446	1.477	1.399	1.819
392	JV	H1_O_JV_K	1.616	1.605	1.247	1.757	1.608	1.598	1.246	1.739
393	JV	H2_I_JV_J	1.401	1.413	1.303	1.685	1.397	1.407	1.302	1.663
394	JV	H2_I_JV_K	1.300	1.648	1.842	1.903	1.295	1.640	1.833	1.882
395	JV	H2_M_JV_J	1.519	1.567	1.298	1.919	1.515	1.562	1.296	1.896
396	JV	H2_M_JV_K	1.683	1.270	1.089	1.805	1.677	1.267	1.089	1.789
397	JV	H2_O_JV_J	1.221	1.188	1.217	1.867	1.220	1.188	1.215	1.795
398	JV	H2_O_JV_K	1.537	1.575	1.805	1.973	1.530	1.567	1.777	1.898
399	BField	Can_H1_1	1.044	1.042	1.070	1.804	1.044	1.042	1.070	1.786
400	BField	Can_H1_2	1.414	1.323	1.064	1.825	1.409	1.321	1.064	1.813
401	BField	Can_H1_3	1.507	1.626	1.298	2.023	1.502	1.615	1.297	2.024
402	BField	Can_H1_4	1.103	1.091	1.571	1.569	1.103	1.091	1.563	1.557
403	BField	Can_H1_5	1.019	1.016	1.141	1.040	1.019	1.016	1.141	1.040
404	Level	Ring_Buffe	1.051	1.031	1.222	1.982	1.051	1.031	1.222	1.978
405	Pres	Out_Header	1.817	1.668	1.236	2.012	1.802	1.652	1.235	2.012

表 5.6: 4 ファイルについてのチャンネルごとの圧縮サイズ（4）

ファイル名	サンプル点数	サイズ（バイト）
00010_00001_00000_A_00263_19981122.rsl	10794	8743652
00010_00001_00000_A_00264_19981122.rsl	10794	8743652
00010_00001_00000_A_00317_19981209.rsl	2712	2197232
00010_00001_00000_A_00318_19981209.rsl	22921	18566522

表 5.7: 性能試験に用いた計測ファイル。

BField	磁場（ヘリカルコイル用）	Level	ヘリウムレベル（高さ）
BV	バランス電圧（超伝導磁石）	MicroW	マイクロ波電力
Curr	電流	PhotoD	プラズマからの輻射
D_Press	差圧	Pres	ヘリウム圧力
Field	磁場（ポロイダルコイル用）	Strain	歪
Flux	磁束	Temp	温度
JV	接続部電圧	Volt	電圧

表 5.8: 性能試験に用いた計測ファイルの物理量名一覧

表 5.3, 5.4, 5.5, 5.6 の 11 の列はそれぞれチャンネル番号，物理量名（表 5.8 参照），センサ位置，正規分布を仮定した場合の 4 個のファイルの圧縮サイズ（ビット/チャンネル），Laplace 分布を仮定した場合の 4 個のファイルの圧縮サイズ（ビット/チャンネル）である。

図 5.1 および図 5.2 は，これら 4 個のファイルを 405×4 チャンネル分のデータと見て，チャンネルごとの圧縮サイズをヒストグラムで表したものである。元データは 16 ビット/サンプルであるので，ほとんどすべてのチャンネルが半分（8 ビット/サンプル）以下に縮んだことになる。正規分布，Laplace 分布とも，算術平均は約 3.4 ビット/チャンネル，中央値（メディアン）は約 3.1 ビット/チャンネルである。

5.3 速度の測定

表 5.9 は，上記の 4 個のファイルのうち最初のもの（「ファイル 263」）と最後のもの（「ファイル 318」）について，圧縮・伸張時間を与えたものである。NIFSq が本稿のアルゴリズムを実装した圧縮ツール，Zip と LHA は一般によく使われている圧縮ツールである。圧縮・伸張時間は標準的なパーソナルコンピュータ（CPU はクロック 400MHz の Pentium II）で計測した。オペレーティングシステムは時間計測の容易さから Linux（バージョン

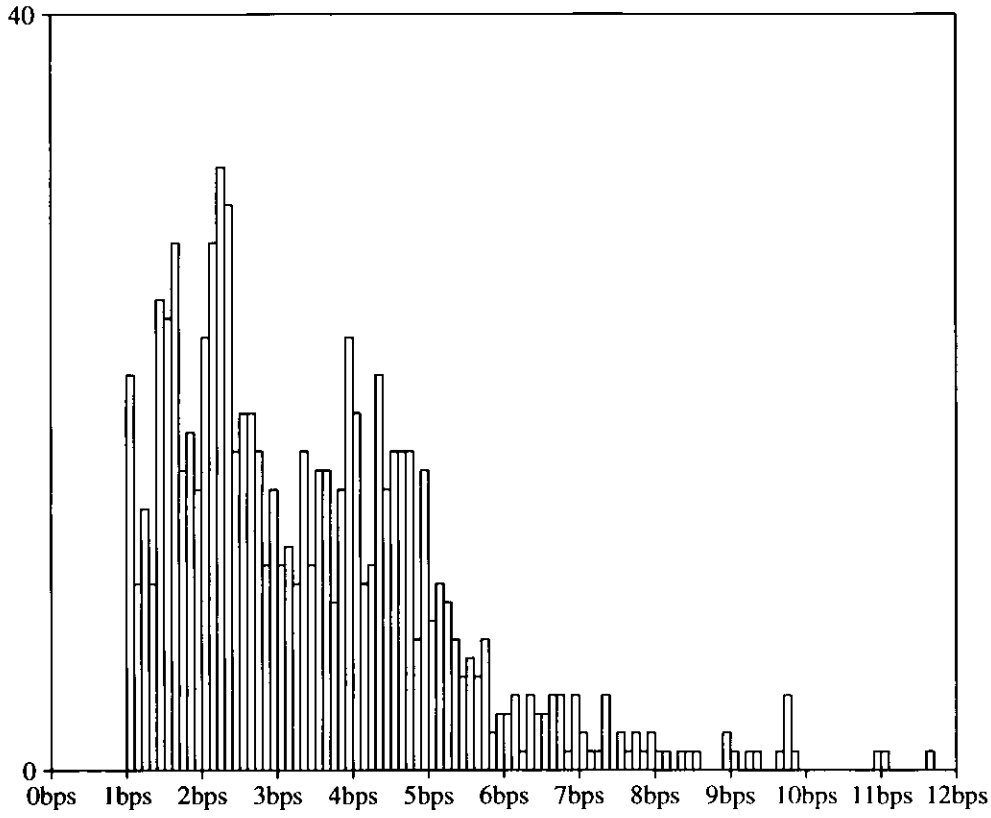


図 5.1: 正規分布を仮定したときの圧縮サイズのヒストグラム。横軸はサンプルあたりのビット数（bps)。縦軸はチャンネル数。

		サイズ（バイト）	圧縮時間（秒）	伸張時間（秒）
ファイル 263		8743652		
Zip		4794261	6.31	1.29
LHA		4769586	9.19	1.86
NIFSq	正規	2003136	2.25	2.04
	Laplace	2000022		
ファイル 318		18566522		
Zip		9822477	14.56	2.70
LHA		9807894	19.94	3.92
NIFSq	正規	3956084	4.94	4.38
	Laplace	3958382		

表 5.9: 圧縮サイズと時間の比較。

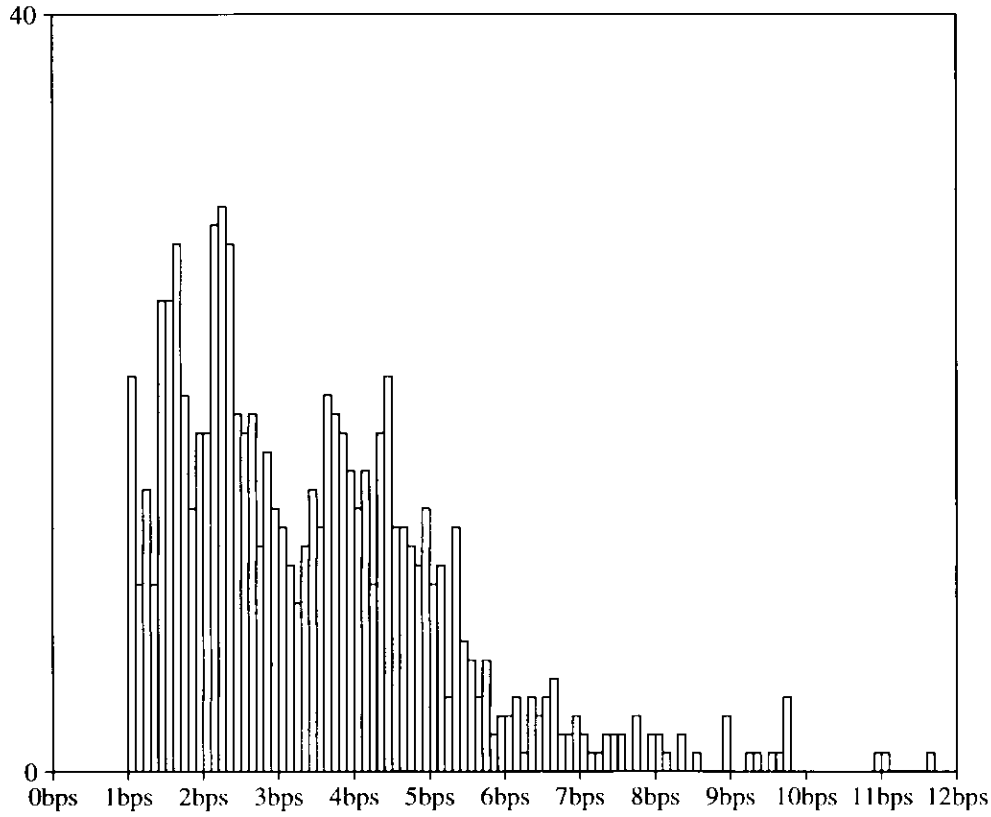


図 5.2: Laplace 分布を仮定したときの圧縮サイズのヒストグラム。横軸はサンプルあたりのビット数 (bps)。縦軸はチャンネル数。

2.0.35) を用いたが，Windows 系オペレーティングシステムでも同様な結果が得られる。

このように，同様な条件で計測したデータであっても，前者のデータは予測誤差分布を Laplace 分布と仮定する方が若干よく縮み，後者のデータは正規分布と仮定する方が若干よく縮んでいる。ただしこの程度の差は有意な差とはいえない。いずれにしても，典型的な圧縮率は 4 : 1 あるいはそれ以上といってよいであろう。

速度については，現在のバージョンは速度について十分に最適化しておらず，改良の余地が多分にあるが，それでも約 2M サンプル/秒 (4M バイト/秒) のスループットが得られた。特に圧縮の速度は一般的な圧縮ツールの数倍である。

なお，可逆性のテストとして，圧縮・伸張の後で元ファイルとの照合を行ったが，完全な一致が確認できた。

5.4 予測に利用する過去のデータの個数

本アルゴリズムには，過去何点のデータに基づいて符号を定めるかというパラメータがある。現在は16点に固定しているが，これを変化させると表5.10のように圧縮サイズがわずかに変化する（正規分布を仮定して符号化を行った)。この二つのファイルの例では32個が最適であるが，16個でも圧縮率はほとんど変わらない上に，16個なら32個の場合に比べてメモリの利用が半分ですむので，16個を選んだ。いずれにしてもこの点数は圧縮率にはわずかしか効かない。

過去の点数	ファイル263	ファイル318
4	2047936	4035482
8	2013792	3969786
16	2003136	3956084
32	2001712	3955032
64	2004540	3959208

表5.10: 符号を定めるための過去のデータ点数を変えたときの圧縮ファイルサイズの変化。

5.5 いくつかのデータの例

ここではいくつかの実際のデータの例を用いて，どのようなデータがどの程度縮むかを示す（第4章の図4.2，4.3，4.4も参照されたい)。

本節では `00010_00001_00000_A_00263_19981122.rsl`（「ファイル263」）を例にとり，いくつかの代表的なチャンネルをプロットし，圧縮率との関係を調べる。このファイルは1998年11月22日0時30分36秒を起点とし，405チャンネルについて4秒間隔で10794点（約12時間）にわたって取得した生データである。

まず，図5.3はチャンネル104（点3USA3LIWの歪（ストレイン））のグラフである。ゆるやかに変化する成分と激しくランダムに変化する成分が重畳されているように見える。しかも，ランダムに変化する成分の振幅は時間とともにゆるやかに変化している。

図5.4と5.5は，同じデータの最初の101点（点0–100）と中程の101点（点7000–7100）を抜き出したものである。比較のため，縦軸のスケールを同じにした。ランダムに変化する成分の振幅の違いが読み取れる。

このようなデータの統計的性質の変化に追随できるのが適応型のアルゴリズムの利点で

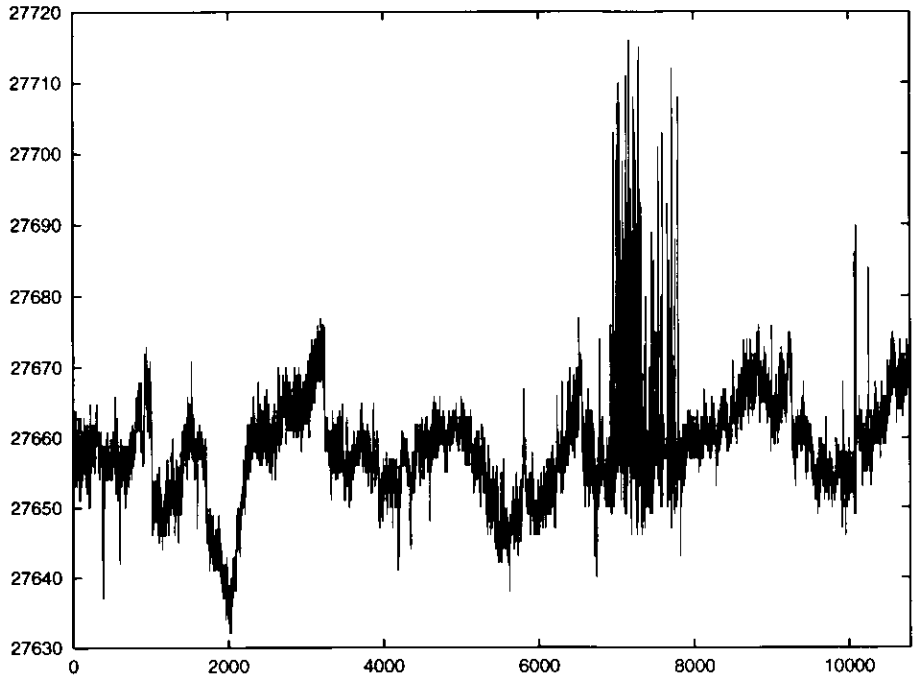


図 5.3: ファイル 263 のチャンネル 104 のグラフ。

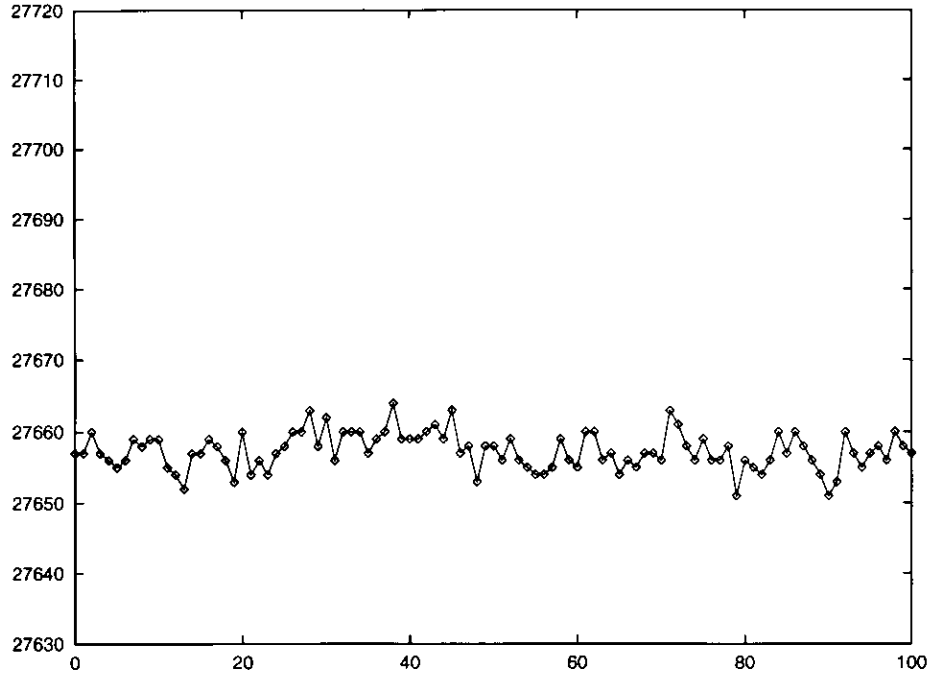


図 5.4: 図 5.3 の最初の 101 点のグラフ。

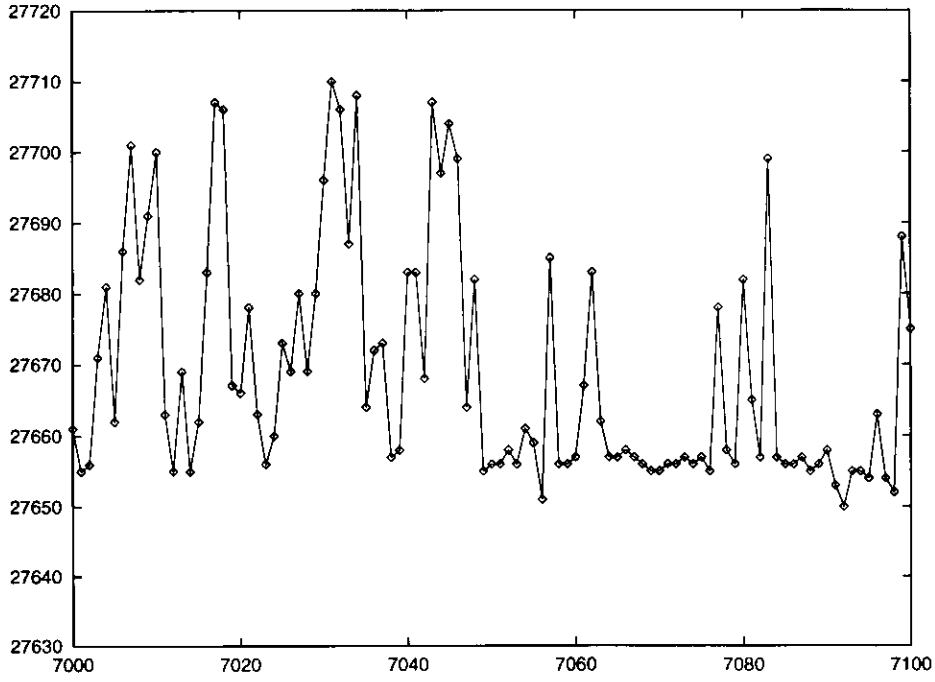


図 5.5: 図 5.3 の点 7000–7100 のグラフ。

ある。本アルゴリズムでは約 3.9 ビット/サンプルに圧縮できた。

より縮みやすい例として，チャンネル 191（点 A_REF2 の温度）の生データのグラフを図 5.6 に示す。データ全域にわたって末位の 2〜3 ビットしか変化していない。本アルゴリズムでは約 1.2 ビット/サンプルに縮み，ここで扱った例では最も縮みやすいチャンネルの一つである。このようなほとんど動きのないチャンネルが非常によく縮むのは明らかであろう。

逆に，縮みにくいチャンネルの例として，チャンネル 182（点 8.5UVRG1 での電流）の生データのグラフを図 5.7 に示した。図 5.8 はこの最初の 101 点を示したものである。非常に大きなノイズが重畳されており，約 11 ビット/サンプルにしか縮まない。

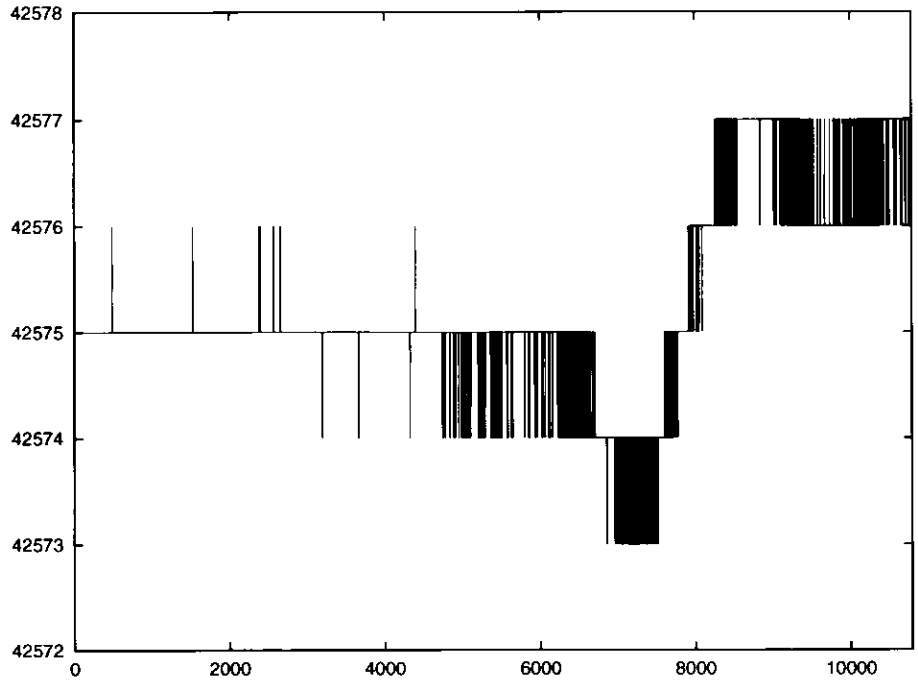


図 5.6: ファイル 263 のチャンネル 191 のグラフ。

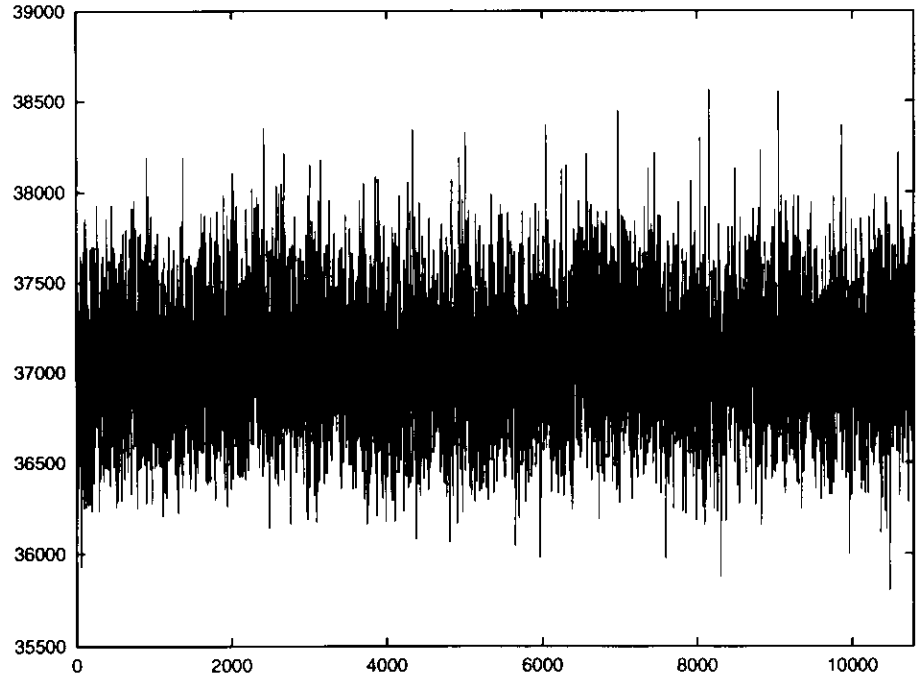


図 5.7: ファイル 263 のチャンネル 182 のグラフ。

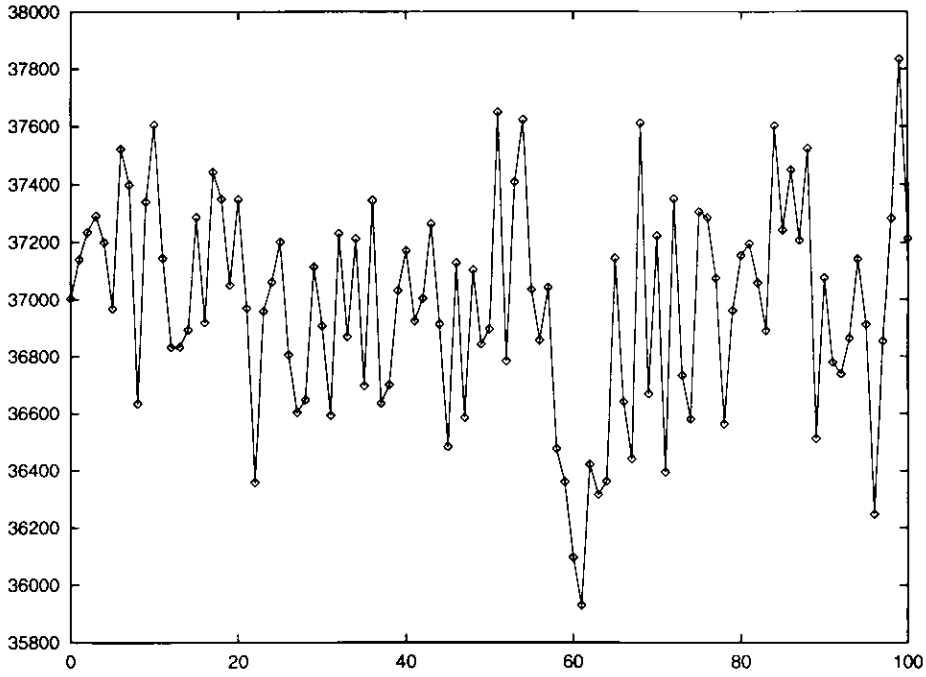


図 5.8: ファイル 263 のチャンネル 182 のグラフの最初の 101 点。

第 6 章

結論と今後の展望

6.1 結論

本研究の目的は，LHD 制御データ処理装置の出力に代表されるような多チャンネル時系列計測データを可逆圧縮するための既存の諸アルゴリズムの評価と，このようなデータを高速に可逆圧縮するための新アルゴリズムの開発であった。

その結果，図 5.1，5.2 および表 5.9 が示すように，今回試験した 16 ビット/サンプルで採取された多チャンネル時系列データの大多数のチャンネルを 2〜8 ビット/サンプルに圧縮し，しかも圧縮時間は従来の汎用圧縮ツールの数分の 1 程度に抑えることができた。このことから，当初の目的は十分達成されたと言ってよいであろう。

このアルゴリズムの特徴は次の 3 点に集約できる。

- 高速性 …… 標準的なパーソナルコンピュータで 2M サンプル/秒のスループットが得られる
- 可逆性 …… 整数演算だけを用い，情報量の損失を生じないアルゴリズムを採用している
- 実時間性 …… 過去のデータだけから符号を構成するので，すべてのデータの到着を待たずにその時点での最新データが復号できる

このアルゴリズムを実装した単体の圧縮ツール NIFSq[注1]，および他のプログラムから呼び出す形の圧縮ライブラリ NIFSqlib は，C 言語のソースコードの形で，`http://www.matsusaka-u.ac.jp/~okumura/NIFSq/` から入手できる。現在の LHD 制御データ処理

[注1] NIFS の（あるいは，粋（いき）な（nifty な））圧縮ツール squeezer のつもりで命名した。

装置に組み込まれているものは，この NIFSqlib の一つのバージョンに日本サン・マイクロシステムズ（株）の田村和一（まさかず）氏が手を加えたものである。NIFSq については，Solaris（Sun Mocrosystems 社の UNIX），Linux（フリーソフトの UNIX），および Windows 95/NT 上で動作確認されている。

6.2 今後の課題

今後の課題として，さらに多様な計測データについて実験を重ね，アルゴリズムの改善を続けることがまず必要である。たとえば，現在の NIFSq では予測残差分布（正規分布か Laplace 分布か）を圧縮時に指定するようになっている。これは，過去の少数のデータだけから高速かつ確実に分布形まで判断するのが難しいためであるが，この点を改良して，自動的にチャンネルごとに最適な残差分布を選択するのが本来は望ましい。また，残差の分散に相当するパラメータは，現在は過去 16 サンプルの絶対値の和から求めている。この 16 という数値は自由に変えられるようになっているが，これもチャンネルごとに自動的に最適な値を選ぶのが本来は望ましいし，近い過去ほど重みを大きくして荷重和をとるのが望ましいかもしれない。これらの点について，多様なデータ例を採取しつつ，検討を重ねることが一つの課題である。

また，ソフトウェアについては，ユーザインターフェースを改良した上で，Macintosh も含めた多様な計算機環境で利用可能にし，利用者層を広げることも課題の一つである。そのためには，現在 C で書かれているソフトウェアを C++ や Java に書き直すことも考慮している。

これ以外に，浮動小数点化された計測データの圧縮や，動画像データの可逆圧縮も視野にいれて研究を続けるつもりである。特に動画像については一般には MPEG 標準などの不可逆圧縮が用いられているが，物理実験を CCD カメラなどで撮影しネットワークで配送する目的には，可逆圧縮が望ましい。

謝辞

本研究の場を提供していただいた核融合科学研究所の本島修先生に感謝いたします。また，研究の便宜を図っていただき貴重なご意見を多数いただいた核融合科学研究所の山口作太郎氏，貴重なご意見をいただいた山口大学の刈谷丈治氏，職業能力開発大学校の寺町康昌氏，日本サン・マイクロシステムズ（株）の田村和一（まさかず）氏に感謝いたします。

付録 A

LHA のアルゴリズム

A.1 圧縮ファイルの構造

LHA の圧縮ファイルの形式は次のように非常に単純である。

ヘッダ 1
圧縮データ 1
ヘッダ 2
圧縮データ 2
⋮
ヘッダ n
圧縮データ n

ヘッダはレベル 0，レベル 1，レベル 2 の 3 通りのバージョンがある。レベル 0 ヘッダは，1988 年ごろ吉崎栄泰氏によって作られた LHarc[注1] のヘッダを流用したもので，現在ではほとんど使われていない。レベル 1 ヘッダは，この LHarc のヘッダとの互換性を重んじたもので，LHarc でも元ファイル名の一覧だけは見られるように工夫されている。レベル 2 ヘッダはまったく新しいもので，レベル 1 までのヘッダに存在したファイル名の長さの制約を除いたものである。現在ではレベル 2 ヘッダを使うべきである。表 A.1，表 A.2，表 A.3 にそれぞれレベル 0，レベル 1，レベル 2 ヘッダ，表 A.4 にレベル 1，2 共通の拡

[注1] LHarc は LHA の前身で，著者によって作られた LZARI（LZ77 と算術符号化を組み合わせたもの）の算術符号化部分を吉崎氏が動的 Huffman 符号化に置き換え，複数のファイルを一つにまとめる機能を追加したものである。

張ヘッダの構造を示す。なお，数値欄はすべて little endian[注2] で格納する。これらの表で m は元ファイル名の長さ（バイト数）である。

bytes	content
1	header size n
1	header checksum (algebraic sum modulo 256)
5	compression method ID (see Table A.6)
4	compressed size
4	original (uncompressed) size
2	last modified time (in MS-DOS format)
2	last modified date (in MS-DOS format)
1	MS-DOS file attribute (usually `0x20`)
1	header level (`0x00`)
1	pathname length m (≤ 233)
m	pathname (separator: \)
2	CRC of original file (mod $x^{16}+x^{15}+x^2+x$)
$n-m-24$	extension (used by earlier versions of UNIX LHA)

表 A.1: LHA のレベル 0 ヘッダ。

これらのヘッダで "compression method ID" と書いた欄は，圧縮方法とパラメータ（具体的には後述の `DICSIZ` と `PBIT`）を示すものである。これは 5 バイトの文字列で，今日 LHA および LHA 互換ソフトで使われている主なものは表 A.6 の通りである。

A.2 LZ77 符号化部

LZ77 アルゴリズム（特に LHA 前段の実装）では，過去に現れた文字列が再び現れた場合に，その文字列の長さと文字列へのポインタを出力する。たとえば

```
ABCDEFGBCDEHI
```

という内容のファイルが入力されたならば，2 度目に現れた "BCDE" は「長さ 4 バイト，6 バイト前」という「長さ・位置」のペアに符号化する。

[注2] Intel x86 形式のバイトオーダで，下位バイトが先，上位バイトが後になる。

bytes	content
1	size of basic header = $25 + m$ (= 0 if end of archive)
1	checksum (algebraic sum modulo 256) of basic header
	basic header
5	compression method ID (see Table A.6)
4	compressed size plus size of extended headers
4	original (uncompressed) size
2	last modified time (in MS-DOS format)
2	last modified date (in MS-DOS format)
1	reserved (0x20)
1	header level (0x01)
1	filename length m (≤ 230)
m	filename (without directory name)
2	CRC of original file (mod $x^{16} + x^{15} + x^2 + x$)
1	OS (0x00 = generic, 'M' = MS-DOS, 'm' = Mac, 'U' = UNIX)
2	size of 1st extended header n_1 (0 if none)
n_1	1st extended header (optional)
n_2	2nd extended header (optional)
⋮	⋮

表 A.2: LHA のレベル 1 ヘッダ。

LZ77 には三つのパラメータがある。

- DICSIZ ····· 何バイト前まで遡って一致文字列を探すか
- MAXMATCH ····· 何バイトの一致まで照合するか
- THRESHOLD ····· 何バイト以上の一致まで「長さ・位置」のペアに符号化するか

仮に DICSIZ が 5 バイトであれば，さきほどの例で 6 バイト前にある "BCDE" は照合できない。また，MAXMATCH が 3 バイトであれば "BCD" までの一致しか照合できない。また，THRESHOLD が仮に 5 バイトであれば，4 バイトの一致は無視されることになる。

LHA では MAXMATCH = 256，THRESHOLD = 3 としている。DICSIZ の値は可変で，ヘッダの "compression method ID" 欄に表 A.6 の形式で書き込まれている。

LHA 前段の LZ77 符号化部は，DICSIZ バイトの範囲内に THRESHOLD バイト以上の一致文字列が見つからなかった場合は，その位置の 1 文字分の文字コード c をそのまま出力する。一致文字列が見つかった場合は，そのうち長さが最大のもの（複数ある場合は最も

bytes	content
	basic header
2	total header size (basic and extended headers)
5	compression method ID (see Table A.6)
4	compressed size (excluding headers)
4	original (uncompressed) size
4	last modified time (seconds since 00:00 GMT, January 1, 1970)
1	reserved (0x20)
1	header level (0x02)
2	CRC of original file (mod $x^{16} + x^{15} + x^2 + x$)
1	OS (0x00 = generic, 'M' = MS-DOS, 'm' = Mac, 'U' = UNIX)
2	size of first extended header n_1
n_1	1st extended header
n_2	2nd extended header (optional)
⋮	⋮

表 A.3: LHA のレベル 2 ヘッダ。

bytes	content
1	extended header ID (see Table A.5)
$n_i - 3$	header content
2	size of next extended header n_{i+1} (0 if none)

表 A.4: LHA の拡張ヘッダの構造。

header ID	bytes ($n_i - 3$)	content
0x00	2	CRC16 of headers
0x01	–	filename (without directory name)
0x02	–	directory name (separators/terminator: 0xff)
0x3f	–	comment
0x40	2	MS-DOS file attribute
0x50	2	UNIX permission (ex. 0644)
0x51	4	UNIX GID and UID
0x54	4	last modified time (seconds since January 1, 1970)

表 A.5: LHA のおもな拡張ヘッダ ID。

compression method ID	DICSIZ	PBIT
-lh0-	（無圧縮）	–
-lh4-	4K	4
-lh5-	8K	4
-lh6-	up to 32K	5
-lh7-	up to 64K	5

表 A.6: LHA ヘッダの compression method ID 欄。DICSIZ はスライド辞書のサイズ，PBIT は一致位置の符号表のサイズを出力する欄のビット数。

現在位置に近いもの）を選び，その長さ（バイト数）から THRESHOLD を引いて 256 を加えた値を c として出力する。さらに，一致文字列が見つかった場合は，その現在位置との距離（バイト数）から 1 を引いた値 p も出力する。この c, p の可能な範囲は次の通りである。

$$0 \leq c \leq \texttt{MAXMATCH} - \texttt{THRESHOLD} + 256 = 509$$

$$0 \leq p < \texttt{DICSIZ}$$

たとえばさきほどの例

ABCDEFGBCDEHI

を LHA の前段で変換すると，次のようになる。

$$\begin{aligned}
c_1 &= \text{'A'} = \texttt{0x41}, & p_1 &= \text{null} \\
c_2 &= \text{'B'} = \texttt{0x42}, & p_2 &= \text{null} \\
c_3 &= \text{'C'} = \texttt{0x43}, & p_3 &= \text{null} \\
c_4 &= \text{'D'} = \texttt{0x44}, & p_4 &= \text{null} \\
c_5 &= \text{'E'} = \texttt{0x45}, & p_5 &= \text{null} \\
c_6 &= \text{'F'} = \texttt{0x46}, & p_6 &= \text{null} \\
c_7 &= \text{'G'} = \texttt{0x47}, & p_7 &= \text{null} \\
c_8 &= 4 + 256 = 260, & p_8 &= 6 - 1 = 5 \\
c_9 &= \text{'H'} = \texttt{0x48}, & p_9 &= \text{null} \\
c_{10} &= \text{'I'} = \texttt{0x49}, & p_{10} &= \text{null}
\end{aligned}$$

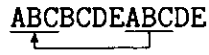
このように DICSIZ バイトの範囲内で最大一致文字列を探すには，概念的には DICSIZ バイトの環状バッファを使えばよい。この環状バッファのことをスライド辞書（sliding

dictionary)とも呼ぶ。実際には,照合のためにさらに MAXMATCH バイトが必要である。さらに LHA では,文字列照合を効率化するために,バッファのサイズを 2×DICSIZ+MAXMATCH バイトとし,DICSIZ バイト処理するごとにバッファの内容を DICSIZ バイトだけ平行移動している。

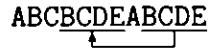
また,LHA では,圧縮を改善するために,次のような工夫(“lazy evaluation”)をしている。たとえば

ABCBCDEABCDE

という入力があり,すでに ABCBCDE まで処理したとしよう。次に ABCDE と最大一致する文字列を探し,長さ 3 バイトの ABC が見つかる。図で描けば



のようになる。しかし,ABCDE の頭の A をやりすごして次の BCDE との最大一致を探せば,



のように BCDE が見つかり,このほうが一致文字列が長いので,より圧縮率が良くなる。

以上が LZ77 符号化部のアルゴリズムである。この LZ77 符号化部の出力は,中間結果用の 16K バイト分のバッファに保存される。この中間バッファのサイズは,小さすぎると,次の Huffman 符号化部を呼び出す回数が増えるために圧縮速度が低下し,符号表を出力する回数も増すために圧縮率も低下する。大きすぎると,より広いファイル領域にわたって同じ Huffman 符号を用いることになるため,データの変化への適応性が悪くなる。

この中間バッファが満杯になると,その全内容を Huffman 符号化ルーチンで処理し,符号を構成して符号表を出力し,続いて中間バッファの内容を符号化して出力する。その際,c は 510 通りしかないので,そのまま度数分布を調べて最適な Huffman 符号を構成することができる。しかし,p は DICSIZ 通り(数千~数万通り)の値をとりうるので,何らかのグループ分けをしないと符号表の大きさが膨大になってしまう。そこで,p については

$$k = \begin{cases} 0 & (p = 0) \\ \lfloor \log_2 p \rfloor + 1 & (p = 1, 2, \dots) \end{cases}$$

で定義される k によってグループ分けをする。この k は

$$0 \leq k < \mathtt{NP} \stackrel{\text{def}}{=} \lceil \log_2 \mathtt{DICSIZ} \rceil + 1$$

を満たす NP 通りの整数である。Huffman 符号化部は，c と k についてはおのおの最適な可変長符号化をして出力するが，同じ k 値をもつ一群の p を区別するための番号は，固定長で出力する（表 A.7 参照)。

p	k	固定長部分のビット数
0	0	0
1	1	0
2, 3	2	1
4, 5, 6, 7	3	2
8, ..., 15	4	3
⋮	⋮	⋮
$2^{k-1}, \ldots, 2^k - 1$	k	$k-1$

表 A.7: p と k の関係。

A.3 Huffman 符号化部

LHA の Huffman 符号化部は，符号語長の上限を 16 ビットに制限した Huffman 符号を構成する。しかし，真に最適な長さ制限のある Huffman 符号を構成するのは時間がかかる上に，単純に構成した Huffman 符号の符号語長が 16 ビットを超えることはまれであるので，著者が考案した高速で近似的に最適な長さ制限のある Huffman 符号構成アルゴリズム（後述）を使っている。このアルゴリズムは Zip や gzip などでも踏襲されている。

実際の Huffman 符号構成ルーチン `make_tree(`n`, freq[], len[], code[])` は，c, k (p をグループ分けしたグループ番号)，および後述の t のどれに対しても使えるように工夫されている。したがって，以下で c と記した部分は，k や t でもかまわない。

このルーチンは，アルファベットのサイズ n と各シンボルの出現頻度

$$\texttt{freq[}c\texttt{]}, \quad c = 0, 1, \ldots, n-1$$

を与えると，各シンボルに対応する符号語のビット長

$$\texttt{len[}c\texttt{]}, \quad c = 0, 1, \ldots, n-1$$

と，実際の符号語

$$\texttt{code[}c\texttt{]}, \quad c = 0, 1, \ldots, n-1$$

を出力する。戻り値 r = `make_tree(...)` は，もし `freq[`c`]` $\neq 0$ を満たす c が一つしかなければ $r = c$ である（一つもなければ $r = 0$ である）。そうでなければ $n \leq r < 2n-1$ を満たす特に意味のない整数 r を返す。この整数値 r は現在の実装では Huffman 木を表すデータ構造の根ノードの番号であり，

$$\texttt{freq}[r] = \sum_{c=0}^{n-1} \texttt{freq}[c]$$

を満たす。

このルーチンのアルゴリズムは次の通りである。

まず，通常の（符号語の長さ制限のない）Huffman のアルゴリズムで符号語を構成する（実際は各シンボル $c = 0, \ldots, n-1$ の符号語長 `len[`c`]` だけ定めればよい）。

続いて，次のようなアルゴリズムで符号語長を 16 ビットに制限する。

```
for (i = 0; i <= 16; i++) len_cnt[i] = 0;
for (c = 0; c < n; c++)
    if (len[c] <= 16) len_cnt[len[c]]++;
cum = 0;
for (i = 16; i > 0; i--)
    cum += len_cnt[i] << (16 - i);
while (cum != (1 << 16)) {
    len_cnt[16]--;
    for (i = 15; i > 0; i--) {
        if (len_cnt[i] != 0) {
            len_cnt[i]--;  len_cnt[i+1] += 2;  break;
        }
    }
    cum--;
}
```

これで `len_cnt[`i`]` には長さ i ビットの符号語の個数が入る。

続いて，シンボルを頻度順に並べ，最も頻度の小さい `len_cnt[16]` 個の符号語長を 16 ビットに定め，次に頻度の小さい `len_cnt[15]` 個の符号語長を 15 ビットに定め，以下同様に続けて，最も頻度の大きい `len_cnt[0]` 個の符号語長を 0 ビットに定め，それに従って符号語長 `len[`c`]` を更新する。

このようにして各シンボル $c = 0, \ldots, n-1$ の符号語長 `len[`c`]` が定まったならば，次のアルゴリズムで各シンボル c に符号語 `code[`c`]` を対応させる。

```
start[0] = 0;
for (i = 0; i < 16; i++)
```

```
        start[i + 1] = (start[i] + len_cnt[i]) << 1;
    for (c = 0; c < n; c++) code[c] = start[len[c]]++;
```

このアルゴリズムで code[c] にはシンボル c の符号語が通常の2進法で入る。たとえば

$$\texttt{len}[c] = 4, \quad \texttt{code}[c] = 5 = (101)_2$$

であれば，シンボル c には4ビットの符号語 0101 が当てられる。この4ビットの符号語を実際に送出するルーチンは LHA のソースコードでは putbits(len[c], code[c]) である。

このアルゴリズムでは，符号表は各符号語の長さの表 len[c] だけから生成されることに注意されたい。したがって，圧縮データに付随させて送るべきものは len[c] の表だけである。各符号語長は 0 以上 16 以下の整数であるので，単純に符号化すると1文字あたり4〜5ビットを要し，510通りの文字について符号表を作れば 2K バイト以上になってしまう。

そこで，この符号表も，各文字を符号化するのと同じ make_tree() を使って Huffman 符号化することにする。ただし，実際にはそのブロックで使われなかった文字も多いので，使われなかった文字 (len[c] = 0) についてはランレングス符号化を併用することにする。

具体的には len[c] の表の出力を次のように行う。

まず len[c] の表の中で，0 はそのまま，0 以外はそれに 2 を加え，その値を可変長符号化して出力する。例外として，0 が3個〜18個続けば，1 を可変長符号化して出力し，続いて 0 の個数から 3 を引いた値を4ビット固定長で出力する。また，0 が 20 個以上続けば，2 を可変長符号化して出力し，続いて 0 の個数から 20 を引いた値を9ビット固定長で出力する。

このアルゴリズムで可変長符号化される値を t で表せば，t は次のように 0 から 18 までの NT = 19 通りの値をとり得る。

$t = 0$	1個の 0
$t = 1$	“後続の4ビットの値 + 3” 個の 0
$t = 2$	“後続の9ビットの値 + 20” 個の 0
$t = 3$	1個の 1
$t = 4$	1個の 2
⋮	⋮
$t = 18$	1個の 16

この可変長符号（ここでは長さ制限のある Huffman 符号）を構成するには，さきほどの make_table() を n = NT として呼び出せばよい。

以上のアルゴリズムを実装したルーチン count_t_freq() は次の通りである。ここで c_len[c] は c を可変長符号化したときの符号語長である。

```
for (i = 0; i < NT; i++) t_freq[i] = 0;
n = NC;
while (n > 0 && c_len[n - 1] == 0) n--;
i = 0;
while (i < n) {
    k = c_len[i++];
    if (k == 0) {
        count = 1;
        while (i < n && c_len[i] == 0) {  i++;  count++;  }
        if (count <= 2) t_freq[0] += count;
        else if (count <= 18) t_freq[1]++;
        else if (count == 19) {  t_freq[0]++;  t_freq[1]++;  }
        else t_freq[2]++;
    } else t_freq[k + 2]++;
}
```

この後で t の度数分布 t_freq[t] を数えて make_tree(NT, t_freq, pt_len, pt_code) を呼び出せば，pt_len[] と pt_code に符号語の長さと実際の符号語が入る。なお，t_len[] と t_code という名前にしないで pt_len[] と pt_code にしたのは，後で同じ配列を一致位置 p の符号化にも流用するためである。

しかしこのようにしても，最後には pt_len[] の表を出力しなければ復号できない。この表はもう Huffman 符号化せずに直接書き出す。しかし，少しでも表をコンパクトにするために，次のような工夫を凝らしている。

まず，出力したいものは pt_len[i], $i = 0, \ldots, 18$ であった。これらのうち最後の数個は 0 であることが多いので，それらを除いた個数をまず 5 ビット固定長で出力する。次に pt_len[i] の値を順に，表 A.8 のような符号表で出力する。

さらに，この場合の pt_len[i] の表は $i = 3$ から数個 0 が続くことが多いので，$i = 3$ から始まる 0 個〜3 個の 0 の連はその長さを 2 ビットで出力する。

この部分のルーチン write_pt_len(n, nbit, i_special) は具体的には次の通りである。この場合，引数は n = NT = 19, nbit = TBIT = $\lceil \log_2 \mathrm{NT} \rceil = 5$, i_special = 3 にセットしてから呼び出す。

```
while (n > 0 && pt_len[n - 1] == 0) n--;
putbits(nbit, n);
i = 0;
while (i < n) {
```

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	1110
8	11110
9	111110
	⋮

表 A.8: pt_len の可変長符号。

```
    k = pt_len[i++];
    if (k <= 6) putbits(3, k);
    else putbits(k - 3, (1 << (k - 3)) - 2);
    if (i == i_special) {
        while (i < 6 && pt_len[i] == 0) i++;
        putbits(2, (i - 3) & 3);
    }
}
```

以上述べた各ルーチンを呼び出して実際に符号表を出力するルーチン write_c_len() は次の通りである。

```
n = NC;
while (n > 0 && c_len[n - 1] == 0) n--;
putbits(CBIT, n);
i = 0;
while (i < n) {
    k = c_len[i++];
    if (k == 0) {
        count = 1;
        while (i < n && c_len[i] == 0) {  i++;  count++;  }
        if (count <= 2) {
            for (k = 0; k < count; k++)
                putbits(pt_len[0], pt_code[0]);
        } else if (count <= 18) {
            putbits(pt_len[1], pt_code[1]);
            putbits(4, count - 3);
```

```
        } else if (count == 19) {
            putbits(pt_len[0], pt_code[0]);
            putbits(pt_len[1], pt_code[1]);
            putbits(4, 15);
        } else {
            putbits(pt_len[2], pt_code[2]);
            putbits(CBIT, count - 20);
        }
    } else putbits(pt_len[k + 2], pt_code[k + 2]);
}
```

A.4 一致位置の符号化

一致位置の符号化は，すでに概略を述べた通り，一致位置 p をグループ化してその番号 k の度数分布を p_freq[k], $k = 0, \ldots,$ NP に求めておく。この表 p_len[k] の要素のうち最後の 0 の連なりを除いたものの個数をまず出力する。その欄に要するビット数を PBIT とすると，初期の頃の LHA では DICSIZ がたかだか $2^{13} = 8$K バイトであったので，NP = 14 であり，したがって PBIT = 4 で済むと考えた。しかしその後 DICSIZ を $2^{15} = 32$K バイト以上にすることが可能になり，PBIT = 5 を要するようになった。そこで，表 A.6 のようにヘッダの “compression method ID” 欄の内容によって PBIT の値を変えている。

実際のアルゴリズムは既出の make_tree() と write_pt_len() を使って次のように簡潔に書くことができる。

```
root = make_tree(NP, p_freq, pt_len, pt_code);
if (root >= NP) {
    write_pt_len(NP, PBIT, -1);
} else {
    putbits(PBIT, 0);  putbits(PBIT, root);
}
```

実際に各 p を出力する部分も次の通り非常に簡潔である。

```
c = 0;  q = p;  while (q) {  q >>= 1;  c++;  }
putbits(pt_len[c], pt_code[c]);
if (c > 1) putbits(c - 1, p & (0xFFFFU >> (17 - c)));
```

付録 B

LHD 制御データ処理装置の圧縮ファイルフォーマット

B.1 概要

LHD 制御データ処理装置のデータファイルには，A/D 変換器の出力をそのまま 16 ビット符号なし整数として納めた「生データ」，およびこれを較正曲線により物理量に変換し IEEE 4 バイト浮動小数点形式で納めた「物理量データ」，さらに物理量データをユーザの指定により編集した「編集データ」がある。また，これらはいずれも，データ取得のモードにより，常時取得される「低速データ」，随時取得される「高速データ」および「クエンチデータ」に分けられる。

拡張子	ヘッダの「データ種類」欄	データ種類
.rsl	RS␣␣	低速生データ
.rfa	RF␣␣	高速生データ
.rqu	RQ␣␣	クエンチ生データ
.psl	PS␣␣	低速物理量データ
.pfa	PF␣␣	高速物理量データ
.pqu	PQ␣␣	クエンチ物理量データ
.esl	ES␣␣	低速編集データ
.efa	EF␣␣	高速編集データ
.equ	EQ␣␣	クエンチ編集データ

表 B.1: 拡張子とヘッダの「データ種類」欄（␣は空白文字 0x20）。

これらのデータ種類に従ってデータファイルは表 B.1 の拡張子が付く。圧縮されている

場合はこれにさらに拡張子 .x が付く。したがって，たとえば低速生データを圧縮したものは .rsl.x という拡張子になる。しかし，ファイル名は保管場所の移動などに伴って変化することがあるので，ファイルのヘッダにそのファイルについての必要な情報が納められている。

ヘッダは 512 バイトで，先頭 4 バイトがファイル種類を表す（表 B.1）。また，ヘッダの最後のバイトで圧縮の有無を表す。各データ種類についてヘッダの詳しい構造を表 B.2, B.3, B.4, B.5 に示した。

バイト数	内容
4	データ種類 (RS␣␣)
4	実験 ID (真空冷却)
4	実験 ID (励磁)
4	実験 ID (プラズマ)
1	実験種別 (1: 真空冷却, 2: 励磁, 3: プラズマ)
1	予備
1	データ収集先名
1	予備
4	データファイル通番
4	先頭データ時刻 (1970 年元旦からの秒数)
4	サンプリング間隔 (秒)
4	タグ数
4	タグ毎データ数
471	予備
1	圧縮フラグ (1: 圧縮, 0: 非圧縮)

表 B.2: 低速生データファイルのヘッダ。

これらのヘッダで，4 バイトの数値欄はすべて big endian すなわち上位桁が先，下位桁が後に来る形式で納められている。たとえば $258 = 0 \times 256^3 + 0 \times 256^2 + 1 \times 256 + 2$ という値は，1 バイト目・2 バイト目が 0，3 バイト目が 1，4 バイト目が 2 になる。これはネットワークでデータをやりとりする際に広く用いられている形式で，LHD 制御データ処理装置のサーバの CPU（SPARC チップ）のデータ形式でもある。パーソナルコンピュータで使われている Intel 系チップの little endian 形式とは順序が逆であるので，注意しなければならない。

さらに物理量ファイルについては，この 512 バイトのヘッダの直後に，表 B.6 に示す 64 バイトの「タグ属性」欄が，タグ数（チャンネル数）だけ続く。

バイト数	内容
4	データ種類 (RF␣␣, RQ␣␣)
4	実験 ID (真空冷却) (クエンチでは空欄)
4	実験 ID (励磁) (クエンチでは空欄)
4	実験 ID (プラズマ) (クエンチでは空欄)
1	実験種別 (1: 真空冷却, 2: 励磁, 3: プラズマ) (クエンチでは空欄)
1	予備
1	データ収集先名
1	予備
4	データファイル通番
4	先頭データ時刻 (1970 年元旦からの秒数)
4	サンプリング間隔 (μ 秒)
4	タグ数
1	トリガ種類 (1: ハードウェア, 2: ソフトウェア) (クエンチでは空欄)
3	予備
4	プレトリガデータ数 (個)
4	ポストトリガデータ数 (個)
463	予備
1	圧縮フラグ (1: 圧縮, 0: 非圧縮)

表 B.3: 高速生データ・クエンチ生データファイルのヘッダ。

B.2 非圧縮生データ

圧縮されていない生データは，A/D 変換器の出力をそのまま 16 ビットの符号なし整数として納めたものである。バイト順序は big endian である。チャンネル数（タグ数）を m，サンプル点の数（チャンネルごとのデータ数）を n とすれば，次のようなプログラムでデータを読むことができる。ここで用いた ntohs() はネットワークバイト順序（big endian 形式）からホストバイト順序（そのマシン固有のバイト順序）に short 値（16 ビット値）を変換するためのライブラリ関数である。

```
int i, j;
unsigned short x;

for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        fread(x, 2, 1, infile);
        idata[i][j] = ntohs(x);
```

バイト数	内容
4	データ種類 (PS␣␣, ES␣␣)
4	実験 ID (真空冷却)
4	実験 ID (励磁)
4	実験 ID (プラズマ)
32	実験名
10	実験責任者 (login ID)
2	予備
1	実験種別 (1: 真空冷却, 2: 励磁, 3: プラズマ)
1	予備
1	データ収集先名
1	予備
4	データファイル通番 (編集データでは空欄)
10	先頭データ年月日 YYYY-MM-DD
2	予備
8	先頭データ時刻 HH:MM:SS
10	最終データ年月日 YYYY-MM-DD
2	予備
8	最終データ時刻 HH:MM:SS
4	サンプリング間隔 (秒)
4	タグ数
4	タグ毎データ数
391	予備
1	圧縮フラグ (1: 圧縮, 0: 非圧縮)

表 B.4: 低速物理量・低速編集データファイルのヘッダ。

```
    }
}
```

B.3 低速物理量データ

低速物理量データは，生データを較正曲線により物理量に変換し，広く用いられている IEEE 754 規格の 4 バイト浮動小数点数として納めたものである。バイト順序はやはり big endian である。チャンネル数（タグ数）を m，サンプル点の数（チャンネルごとのデータ数）を n とすれば，次のようなプログラムでデータを読むことができる。サンプル点ごと

バイト数	内容
4	データ種類 ($PF_{\sqcup\sqcup}$, $PQ_{\sqcup\sqcup}$, $EF_{\sqcup\sqcup}$, $EQ_{\sqcup\sqcup}$)
4	実験 ID (真空冷却) (クエンチでは空欄)
4	実験 ID (励磁) (クエンチでは空欄)
4	実験 ID (プラズマ) (クエンチでは空欄)
32	実験名 (クエンチでは空欄)
10	実験責任者 (login ID) (クエンチでは空欄)
2	予備
1	実験種別 (1: 真空冷却, 2: 励磁, 3: プラズマ) (クエンチでは空欄)
1	予備
1	データ収集先名
1	予備
4	データファイル通番 (編集ファイルでは元ファイルの通番)
10	先頭データ年月日 YYYY-MM-DD
2	予備
8	先頭データ時刻 HH:MM:SS
10	最終データ年月日 YYYY-MM-DD
2	予備
8	最終データ時刻 HH:MM:SS
4	サンプリング間隔 (μ 秒)
4	タグ数
4	タグ毎データ数
1	トリガ種類 (1: ハードウェア, 2: ソフトウェア) (クエンチでは空欄)
3	予備
4	プレトリガデータ数 (個) (編集データでは空欄)
4	ポストトリガデータ数 (個) (編集データでは空欄)
379	予備
1	圧縮フラグ (1: 圧縮, 0: 非圧縮)

表 B.5: 高速物理量・クエンチ物理量・高速編集・クエンチ編集データファイルのヘッダ。

のデータの最初に 4 バイト整数で経過時間（秒）が入る。

```
int i, j, t;
float x;

for (i = 0; i < n; i++) {
    fread(&t, 4, 1, infile);
    time[i] = ntohl(t);
```

バイト数	内容
1	種類 ('S': センサータグ, 'V': 高次元タグ) (生データ以外では空欄)
1	データ収集先
5	被測定機器
10	測定場所
6	測定項目
3	タグ番号
10	単位
8	センタータグ名
20	タグ名称 (先頭 20 バイト)

表 B.6: タグ属性。

```
    for (j = 0; j < m; j++) {
        fread(x, 4, 1, infile);
        *(int *)&x = ntohl(*(int *)&x);
        fdata[i][j] = x;
    }
}
```

B.4 高速物理量データ

高速物理量データは，データ行列の行と列の順序が低速物理量データの逆である。また，経過時間の単位は μ 秒である。チャンネル数（タグ数）を m，サンプル点の数（チャンネルごとのデータ数）を n とすれば，次のようなプログラムでデータを読むことができる。

```
int i, j, t;
float x;

for (i = 0; i < n; i++) {
    fread(&t, 4, 1, infile);
    time[i] = ntohl(t);
}
for (j = 0; j < m; j++) {
    for (i = 0; i < n; i++) {
        fread(x, 4, 1, infile);
        *(int *)&x = ntohl(*(int *)&x);
        fdata[i][j] = x;
    }
```

```
}
```

B.5 圧縮生データ

これは本稿で詳述した *NIFSq* アルゴリズムで圧縮している。512バイトのヘッダは圧縮していない。

B.6 圧縮物理量データ

本来ならば物理量データは生データから導かれるので，生データを圧縮して保存するのが最も効率的である。必要ならばヘッダに物理量への変換式を格納しておけばよい。しかし，従来のソフトウェアとの互換性維持のため，浮動小数点のままで圧縮したファイル形式も用意した。これは HDF と同様にデータを単純に zlib（Zip や gzip の圧縮アルゴリズムをライブラリ化したもの）で圧縮したものである。ただし 512 バイトのヘッダは圧縮しない。

圧縮物理量データを伸張する簡単なプログラムを挙げておく。コンパイル時に zlib をリンクする必要がある。

```
#include <stdio.h>
#include <stdlib.h>
#include <zlib.h>

#define INBUFSIZ   1024          /* 入力バッファサイズ（任意） */
#define OUTBUFSIZ  1024          /* 出力バッファサイズ（任意） */

z_stream z;              /* ライブラリとやりとりするための構造体 */

char inbuf[INBUFSIZ];            /* 入力バッファ */
char outbuf[OUTBUFSIZ];          /* 出力バッファ */
FILE *fin, *fout;                /* 入力・出力ファイル */

void do_decompress(void)         /* 伸張（復元） */
{
    int count, status;

    /* すべてのメモリ管理をライブラリに任せる */
    z.zalloc = Z_NULL;
    z.zfree = Z_NULL;
    z.opaque = Z_NULL;
```

```
/* 初期化 */
z.next_in = Z_NULL;
z.avail_in = 0;
if (inflateInit(&z) != Z_OK) {
    fprintf(stderr, "inflateInit: %s\n", (z.msg) ? z.msg : "???");
    exit(1);
}

z.next_out = outbuf;         /* 出力ポインタ */
z.avail_out = OUTBUFSIZ;     /* 出力バッファ残量 */
status = Z_OK;

while (status != Z_STREAM_END) {
    if (z.avail_in == 0) {  /* 入力残量がゼロになれば */
        z.next_in = inbuf;  /* 入力ポインタを元に戻す */
        /* データを読む */
        z.avail_in = fread(inbuf, 1, INBUFSIZ, fin);
    }
    status = inflate(&z, Z_NO_FLUSH); /* 展開 */
    if (status == Z_STREAM_END) break; /* 完了 */
    if (status != Z_OK) {   /* エラー */
        fprintf(stderr, "inflate: %s\n", (z.msg) ? z.msg : "???");
        exit(1);
    }
    if (z.avail_out == 0) { /* 出力バッファが尽きれば */
        /* まとめて書き出す */
        if (fwrite(outbuf, 1, OUTBUFSIZ, fout) != OUTBUFSIZ) {
            fprintf(stderr, "Write error\n");
            exit(1);
        }
        z.next_out = outbuf; /* 出力ポインタを元に戻す */
        z.avail_out = OUTBUFSIZ; /* 出力バッファ残量を元に戻す */
    }
}

/* 残りを吐き出す */
if ((count = OUTBUFSIZ - z.avail_out) != 0) {
    if (fwrite(outbuf, 1, count, fout) != count) {
        fprintf(stderr, "Write error\n");
        exit(1);
    }
}

/* 後始末 */
if (inflateEnd(&z) != Z_OK) {
```

```
        fprintf(stderr, "inflateEnd: %s\n", (z.msg) ? z.msg : "???");
        exit(1);
    }
}

int main(int argc, char *argv[])
{
    int i;

    if (argc != 3) return 1;
    if ((fin = fopen(argv[1], "r")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        exit(1);
    }
    if ((fout = fopen(argv[2], "w")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[2]);
        exit(1);
    }
    for (i = 0; i < 511; i++) fputc(fgetc(fin), fout);
    fputc(0, fout);
    do_decompress();
    fclose(fin);
    fclose(fout);
    return 0;
}
```

付録 C

NIFSq ソースコード

以下はスタンドアロン版 nifsq.c のソースコードである。ここに収録するにあたって，デバッグコードや冗長なコメントは省き，アルゴリズムがわかりやすい形に書き改めた。なお，この版ではデータファイルのヘッダについては考慮していない。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

typedef int            s32;     /* signed 32-bit integer */
typedef unsigned int   u32;     /* unsigned 32-bit integer */
typedef short          s16;     /* signed 16-bit integer */
typedef unsigned short u16;     /* unsigned 16-bit integer */
#define ABS16(x)     abs(x)     /* absolute value of 16-bit integer */

int to_nowhere = 0;             /* option -t turns this on */
s32 totalincount, totaloutcount;

#define METHOD_ILLEGAL      -1
#define METHOD_GAUSSIAN      8
#define METHOD_EXP           9
int method = METHOD_ILLEGAL;

/*************************************************************************
  Error and warning
*************************************************************************/

char *progname = "nifsq";
```

```
void error(int errno, char *format, ...)
{
    va_list argptr;

    if (errno != 0) perror(progname);
    fprintf(stderr, "%s: ", progname);
    va_start(argptr, format);
    vfprintf(stderr, format, argptr);
    va_end(argptr);
    fprintf(stderr, "\n");
    exit(1);
}

void warn(int errno, char *format, ...)
{
    va_list argptr;

    if (errno != 0) perror(progname);
    fprintf(stderr, "%s: ", progname);
    va_start(argptr, format);
    vfprintf(stderr, format, argptr);
    va_end(argptr);
    fprintf(stderr, "\n");
}

/***************************************************************************
  I/O
***************************************************************************/

#define INBUFSIZ  0x4000        /* input buffer size / 2 bytes */
#define OUTBUFSIZ 0x4000        /* output buffer size / 2 bytes */
u16 inbuf[INBUFSIZ];
u16 outbuf[OUTBUFSIZ];

int ifd, ofd;                   /* in/out file descriptors */

#define rightbits(n, x) ((x) & ((1 << (n)) - 1))

#ifndef O_BINARY                /* for MS-DOS compatibility */
#define O_BINARY  0             /* binary mode for open() */
#endif

static u16 magic[2]      = { 0xe0e1, 0x0a00 };
static u16 swap_magic[2] = { 0xe1e0, 0x000a };

int channels = 1;               /* number of channels.
                                   option -s changes this value */
int decompress_flag = 0;        /* option -d turns this on */
int swap_data = 0;              /* option -x turns this on */
int swap_read = 0;
int swap_write = 0;
```

```
int insize, inptr, inremain, outremain;
u32 bitbuf;

int ropen(char *name)
{
    if (strcmp(name, "-") == 0) {
        ifd = STDIN_FILENO;
    } else {
        errno = 0;
        ifd = open(name, O_RDONLY | O_BINARY, 0);
        if (ifd < 0) return 1;
    }
    insize = inptr = 0;
    inremain = 0;  bitbuf = 0;
    totalincount = 0;
    return 0;
}

int outcnt;

int wopen(char *name)
{
    if (to_nowhere) {
        ofd = -1;
    } else if (strcmp(name, "-") == 0) {
        ofd = STDOUT_FILENO;
    } else {
        errno = 0;
        ofd = open(name, O_WRONLY | O_CREAT | O_EXCL | O_BINARY, 0644);
        if (ofd < 0) return 1;
        /* memset(outbuf, 0, OUTBUFSIZ * sizeof(outbuf[0])); */
    }
    outcnt = 0;  outremain = 8 * sizeof(outbuf[0]);
    totaloutcount = 0;
    return 0;
}

void rclose(void)
{
    if (ifd != STDIN_FILENO) close(ifd);
    ifd = -1;
}

void swap_byteorder(int n, u16 a[])
{
    int i;

    for (i = 0; i < n; i++) a[i] = (a[i] << 8) | (a[i] >> 8);
}

void flush_outbuf(void)
{
```

```
    int i, t;
    char *buf;

    totaloutcount += outcnt;
    if (to_nowhere) {
        outcnt = 0;  return;
    }
    if (swap_write) swap_byteorder(outcnt, outbuf);
    t = outcnt * sizeof(outbuf[0]);
    buf = (char *)outbuf;
    errno = 0;
    while (t > 0) {
        i = write(ofd, buf, t);
        if (i == -1) error(errno, "write failed");
        t -= i;  buf += i;
    }
    if (! decompress_flag)
        memset(outbuf, 0, outcnt * sizeof(outbuf[0]));
    outcnt = 0;
}

void wclose(void)
{
    flush_outbuf();
    errno = 0;
    if (ofd != -1 && ofd != STDOUT_FILENO) {
        if (close(ofd) == -1) error(errno, "close failed");
    }
    ofd = -1;
}

void fill_inbuf(int suggested_size)
{
    int i, s;

    i = s = 0;
    errno = 0;
    suggested_size *= sizeof(inbuf[0]);
    while (suggested_size > 0) {
        i = read(ifd, (char *)inbuf + s, suggested_size);
        if (i <= 0) break;
        s += i;  suggested_size -= i;
    }
    if (i == -1) error(errno, "read error");
    insize = (s + (sizeof(inbuf[0]) - 1)) / sizeof(inbuf[0]);
    if (swap_read) swap_byteorder(insize, inbuf);
    inptr = 0;
    totalincount += insize;
}

/* Get n bits (n = 1..16) */
inline static unsigned getbits(int n)
```

```
{
    /* if sizeof(inbuf[0]) == 1, then 'if' must be 'while'. */
    if (n > inremain) {
        if (inptr >= insize) fill_inbuf(INBUFSIZ);
        bitbuf = (bitbuf << (8 * sizeof(inbuf[0]))) | inbuf[inptr++];
        inremain += 8 * sizeof(inbuf[0]);
    }
    return rightbits(n, bitbuf >> (inremain -= n));
}

/* Push back n bits (n = 1..16) */
inline static void ungetbits(int n)
{
    inremain += n;
}

/* Put n bits (n = 1..16) */
inline static void putbits(int n, unsigned x)
{
    /* if sizeof(outbuf[0]) == 1, then 'if' must be 'while'. */
    if (n >= outremain) {
        outbuf[outcnt] |= x >> (n - outremain);
        if (++outcnt >= OUTBUFSIZ) flush_outbuf();
        outremain += 8 * sizeof(outbuf[0]);
    }
    outbuf[outcnt] |= x << (outremain -= n);
}

/***************************************************************************
  tables & miscellany
***************************************************************************/

u32 *threshold;
char *codelentbl[32];

#define HISTLEN  16

u32 threshold_gaussian[34] = {
    (        0UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (    38586UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (    72466UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (    97792UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   129779UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   151924UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   231082UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   273613UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   413319UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   467414UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   568279UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   781447UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   944854UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  1273231UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
```

```
    (  1797242UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  2553170UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  3604112UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  5118826UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  7217784UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 10250072UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 14445092UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 20512534UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 28899692UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 41037441UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 57808882UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 82087246UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (115627259UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (164186854UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (231264010UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (328385723UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (462150893UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (611441397UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (842367377UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    0xffffffffUL  /* sentinel */
};

s32 threshold_exponential[34] = {
    (        0UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (    56920UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (    93548UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   153901UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   247768UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   308659UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   406927UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   667964UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (   832543UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  1236934UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  1626786UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  1851379UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  2491018UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  3253572UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  3728550UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  4982036UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  6491156UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  7457385UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (  9964072UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 12982495UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 14914769UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 19928144UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 27921174UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 39856288UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 55842347UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    ( 79712575UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (111684684UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (159422117UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (223225952UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
```

```
    (316294760UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (424777508UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (632825343UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    (850842661UL + (65536UL / HISTLEN - 1)) / (65536UL / HISTLEN),
    0xffffffffUL  /* sentinel */
};

char codelentbl_gaussian[32][16] = {
    { 1,2,3,4,7,7,7,7,8,8,8,8,8,8,8,8 },
    { 2,1,3,4,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 3,1,2,4,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 3,2,1,4,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 4,2,1,3,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 3,2,2,2,4,6,7,7,8,8,8,8,8,8,8,8 },
    { 4,2,2,2,3,6,7,7,8,8,8,8,8,8,8,8 },
    { 4,3,2,2,2,6,7,7,8,8,8,8,8,8,8,8 },
    { 4,4,2,2,2,4,6,6,8,8,8,8,8,8,8,8 },
    { 4,3,3,2,2,3,6,6,8,8,8,8,8,8,8,8 },
    { 6,4,3,2,2,2,7,7,8,8,8,8,8,8,8,8 },
    { 6,4,4,2,2,2,4,6,8,8,8,8,8,8,8,8 },
    { 6,4,3,3,2,2,3,6,8,8,8,8,8,8,8,8 },
    { 6,6,5,3,2,2,2,5,8,8,8,8,8,8,8,8 },
    { 6,6,4,3,3,2,2,3,8,8,8,8,8,8,8,8 },
    { 8,6,6,5,3,2,2,2,5,8,8,8,8,8,8,8 },
    { 8,6,6,4,3,3,2,2,3,8,8,8,8,8,8,8 },
    { 8,8,6,6,5,3,2,2,2,5,8,8,8,8,8,8 },
    { 8,8,6,6,4,3,3,2,2,3,8,8,8,8,8,8 },
    { 8,8,8,6,6,5,3,2,2,2,5,8,8,8,8,8 },
    { 8,8,8,6,6,4,3,3,2,2,3,8,8,8,8,8 },
    { 8,8,8,8,6,6,5,3,2,2,2,5,8,8,8,8 },
    { 8,8,8,8,6,6,4,3,3,2,2,3,8,8,8,8 },
    { 8,8,8,8,8,8,6,6,5,3,2,2,2,5,8,8 },
    { 8,8,8,8,8,8,6,6,4,3,3,2,2,3,8,8 },
    { 8,8,8,8,8,8,8,6,6,5,3,2,2,2,5,8 },
    { 8,8,8,8,8,8,8,6,6,4,3,3,2,2,3,8 },
    { 8,8,8,8,8,8,8,8,6,6,5,3,2,2,2,5 },
    { 8,8,8,8,8,8,8,8,6,6,4,3,3,2,2,3 },
    { 8,8,8,8,8,8,8,8,7,7,6,4,3,2,2,2 },
};

char codelentbl_exponential[32][16] = {
    { 1,2,3,4,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 2,1,3,4,6,7,8,8,8,8,8,8,8,8,8,8 },
    { 2,2,2,3,4,6,7,7,8,8,8,8,8,8,8,8 },
    { 3,2,2,2,4,6,7,7,8,8,8,8,8,8,8,8 },
    { 4,2,2,2,3,6,7,7,8,8,8,8,8,8,8,8 },
    { 3,3,2,2,3,4,6,6,8,8,8,8,8,8,8,8 },
    { 4,3,3,2,2,3,6,6,8,8,8,8,8,8,8,8 },
    { 4,4,3,2,2,3,4,5,8,8,8,8,8,8,8,8 },
    { 6,4,3,3,2,2,3,6,8,8,8,8,8,8,8,8 },
```

```
    { 5,4,4,3,2,2,3,4,8,8,8,8,8,8,8,8 },
    { 5,4,4,3,3,2,2,4,8,8,8,8,8,8,8,8 },
    { 6,5,5,3,3,2,2,3,6,7,8,8,8,8,8,8 },
    { 6,6,4,4,3,2,2,3,4,7,8,8,8,8,8,8 },
    { 6,6,4,4,3,3,2,2,4,7,8,8,8,8,8,8 },
    { 7,6,5,5,3,3,2,2,3,6,8,8,8,8,8,8 },
    { 7,6,6,4,4,3,2,2,3,4,8,8,8,8,8,8 },
    { 8,6,6,4,4,3,3,2,2,4,7,8,8,8,8,8 },
    { 8,7,6,5,5,3,3,2,2,3,6,8,8,8,8,8 },
    { 8,7,6,6,4,4,3,2,2,3,4,8,8,8,8,8 },
    { 8,8,6,6,4,4,3,3,2,2,4,7,8,8,8,8 },
    { 8,8,7,6,5,5,3,3,2,2,3,6,8,8,8,8 },
    { 8,8,7,6,6,4,4,3,2,2,3,4,8,8,8,8 },
    { 8,8,8,7,6,5,5,3,3,2,2,3,6,8,8,8 },
    { 8,8,8,7,6,6,4,4,3,2,2,3,4,8,8,8 },
    { 8,8,8,8,7,6,5,5,3,3,2,2,3,6,8,8 },
    { 8,8,8,8,7,6,6,4,4,3,2,2,3,4,8,8 },
    { 8,8,8,8,8,7,6,5,5,3,3,2,2,3,6,8 },
    { 8,8,8,8,8,7,6,6,4,4,3,2,2,3,4,8 },
    { 8,8,8,8,8,8,7,6,5,5,3,3,2,2,3,6 },
    { 8,8,8,8,8,8,7,6,6,4,4,3,2,2,3,4 },
    { 8,8,8,8,8,8,8,8,6,6,4,3,3,2,2,3 },
    { 8,8,8,8,8,8,8,8,7,7,6,4,3,2,2,2 },
};

u16 *data;                      /* this need not be malloc'ed */

#define MAX_CHANNELS  1024      /* maximum number of channels */

u32 sum1[MAX_CHANNELS], sum2[MAX_CHANNELS], sum3[MAX_CHANNELS];

u16 prevdata[MAX_CHANNELS];
u16 diff1[HISTLEN][MAX_CHANNELS],
    diff2[HISTLEN][MAX_CHANNELS],
    diff3[HISTLEN][MAX_CHANNELS];

s16 prevd1[MAX_CHANNELS];       /* previous d1 */
s16 prevd2[MAX_CHANNELS];       /* previous d2 */

char kprev[MAX_CHANNELS];       /* previous code number */

/*************************************************************************
  Make tables for Huffman encoding.  We use 32 codes (k = 0..31).
  For each k, we have 16 characters (b = 0..15) to compress.
  Characters are to be converted to codewords of lengths 1 to 8 bits.
  Lengths of codewords are given by codelentbl[k][b].

  Output: code[k][b]   ... right-adjusted codeword for character b.
          table[k][c] ... table for finding character (b) from codeword
                          c (left-adjusted 8-bit value, padded with
                          arbitrary bit strings) such that
                          if table[k][c] == b,
```

```
                        then code[k][b] == (c >> codelentbl[k][b]).
*************************************************************************/

static unsigned char code[32][16], table[32][256];

static void maketable(void)
{
    int c, c1, c2, b, k, len;

    for (k = 0; k < 32; k++) {
        c1 = 0;
        for (len = 1; len <= 8; len++) {
            for (b = 0; b < 16; b++) {
                if (codelentbl[k][b] == len) {
                    code[k][b] = c1 >> (8 - len);
                    c2 = c1 + (0x100 >> len);
                    for (c = c1; c < c2; c++) table[k][c] = b;
                    c1 = c2;
                }
            }
        }
        if (c1 != 0x100) error(0, "Error in codelentbl");
    }
}

void setmethod(int m)
{
    int i;

    if (m != METHOD_ILLEGAL && m == method) return;
    method = m;
    if (method == METHOD_GAUSSIAN) {
        threshold = threshold_gaussian;
        for (i = 0; i < 32; i++)
            codelentbl[i] = codelentbl_gaussian[i];
    } else if (method == METHOD_EXP) {
        threshold = threshold_exponential;
        for (i = 0; i < 32; i++)
            codelentbl[i] = codelentbl_exponential[i];
    } else {
        error(0, "Bad method: %d", m);
    }
    maketable();
}

/*************************************************************************
  Function bitlen(x) finds the bit length of the absolute value of
  signed 16-bit x.  For example, bitlen(0) = 0, bitlen(1) = bitlen(-1)
  = 1, bitlen(2) = bitlen(-2) = bitlen(3) = bitlen(-3) = 2, ...,
  bitlen(32767) = bitlen(-32767) = 15.  Note: bitlen(-32768) gives 15
  rather than 16.
*************************************************************************/
```

```
static int bitlentbl1[256] = {
    0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
    6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
    7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
    7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,
    8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8 };

static int bitlentbl2[129] = {
    8,9,10,10,11,11,11,11,12,12,12,12,12,12,12,12,
    13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,13,
    14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
    14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,14,
    15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
    15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
    15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,
    15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15,15 };

inline static int bitlen(s16 x)
{
    if (x < 0) x = -x;
    if ((x & 0xff00) == 0) return bitlentbl1[x];
    else                   return bitlentbl2[(u16)x >> 8];
}

/*************************************************************************
  Compression
*************************************************************************/

int read_data(void)
{
    if (inptr >= insize) fill_inbuf((INBUFSIZ / channels) * channels);
    if (insize != 0) {
        data = &inbuf[inptr];  inptr += channels;
        return 1;
    } else {
        return 0;               /* end of file */
    }
}

void compress(void)
{
    int i, b, c, k, p, d, d1, d2, d3;
    u32 z, pos;

    data = prevdata;;
    swap_read = swap_data;
    swap_write = 0;
    putbits(16, magic[0]);
```

```
putbits(16, magic[1]);
putbits(16, (u16)method);
putbits(16, (u16)channels);
for (i = 0; i < channels; i++) {
    for (p = 0; p < HISTLEN; p++)
        diff1[p][i] = diff2[p][i] = diff3[p][i] = 0;
    prevd1[i] = prevd2[i] = 0;
    sum1[i] = sum2[i] = sum3[i] = 0;
    prevdata[i] = 0;
    kprev[i] = 31;
}
for (pos = 0;  ; pos++) {
    if (read_data() == 0) {
        /* write end-of-file codeword */
        if (sum1[0] <= sum2[0]) {
            z = sum1[0];
        } else if (sum2[0] <= sum3[0]) {
            z = sum2[0];
        } else {
            z = sum3[0];
        }
        if (pos <= HISTLEN) {
            if (pos > 1) {
                z = (z * HISTLEN) / (pos - 1);
            } else {
                z = threshold[32];
            }
        }
        if (z < threshold[32]) {
            k = 31;
            while (z < threshold[k]) k--;
            c = codelentbl[k][15];
            putbits(c, code[k][15]);
            putbits(16, (u16)0x7fff);
        } else {
            putbits(16, (u16)0x8000);  putbits(1, 1);
        }
        break;
    }
    p = pos % HISTLEN;
    for (i = 0; i < channels; i++) {
        d1 = (s16)(data[i] - prevdata[i]);
        d2 = (s16)(d1 - prevd1[i]);  prevd1[i] = d1;
        d3 = (s16)(d2 - prevd2[i]);  prevd2[i] = d2;
        /* The next line could be
           if (sum[i] <= sum2[i] && sum1[i] <= sum3[i])
           but in practice compression tends to degrade. */
        if (sum1[i] <= sum2[i]) {
            d = d1;  z = sum1[i];
        } else if (sum2[i] <= sum3[i]) {
            d = d2;  z = sum2[i];
        } else {
```

```
    d = d3;  z = sum3[i];
}
if (pos <= HISTLEN) {
    if (pos > 1) {
        z = (z * HISTLEN) / (pos - 1);
    } else {
        z = threshold[32];
        if (pos == 0)
            d1 = d2 = d3 = prevd1[i] = prevd2[i] = 0;
    }
}
if (z < threshold[32]) {
    k = kprev[i];
    if (z < threshold[k]) {
        while (z < threshold[--k]);
    } else {
        while (z >= threshold[k + 1]) k++;
    }
    kprev[i] = (char)k;
    b = bitlen(d);
    c = codelentbl[k][b];
    putbits(c, code[k][b]);
    BITCOUNT(i, b + c);
    if (b != 0) {
        if (d >= 0) {
            putbits(b, (u16)rightbits(b - 1, d));
            if (d == 32767) {
                putbits(1, 0);
                BITCOUNT(i, 1);
            }
        } else if (d > -32767) {
            putbits(b, (u16)rightbits(b, -d));
        } else if (d == -32767) {
            putbits(16, (u16)0xfffe);
            BITCOUNT(i, 1);
        } else { /* if (d == -32768) */
            putbits(16, (u16)0xffff);
            BITCOUNT(i, 1);
        }
    }
} else {
    putbits(16, (u16)d);
    BITCOUNT(i, 16);
    if (d == -32768) {
        putbits(1, 0);  BITCOUNT(i, 1);
    }
    kprev[i] = 31;
}

sum1[i] -= diff1[p][i];
d1 = ABS16(d1);
sum1[i] += (unsigned)d1;
```

```
            diff1[p][i] = (u16)d1;
            sum2[i] -= diff2[p][i];
            d2 = ABS16(d2);
            sum2[i] += (unsigned)d2;
            diff2[p][i] = (u16)d2;
            sum3[i] -= diff3[p][i];
            d3 = ABS16(d3);
            sum3[i] += (unsigned)d3;
            diff3[p][i] = (u16)d3;
        }
        memcpy(prevdata, data, 2 * channels);
    }
    if (outremain != 16) putbits(outremain, 0);
}

/*************************************************************************
  Decompression
*************************************************************************/

void write_data(void)
{
    memcpy(&outbuf[outcnt], data, 2 * channels);
    if ((outcnt += channels) > OUTBUFSIZ - channels)
        flush_outbuf();
}

void decompress(void)
{
    int i, j, b, c, k, p, t, d, d1, d2, d3;
    u32 z, pos;

    swap_read = 0;
    swap_write = swap_data;
    i = getbits(16);  j = getbits(16);
    if (i == magic[0] && j == magic[1]) {
        /* good */
    } else if (i == swap_magic[0] && j == swap_magic[1]) {
        swap_read = 1;
        swap_byteorder(insize, inbuf);
    } else {
        error(0, "bad magic: %04x %04x", i, j);
    }
    i = getbits(16);  setmethod(i);
    channels = getbits(16);
    if (channels < 1)
        error(0, "bad number of channels: %d", channels);
    data = prevdata;;
    for (i = 0; i < channels; i++) {
        for (p = 0; p < HISTLEN; p++)
            diff1[p][i] = diff2[p][i] = diff3[p][i] = 0;
        prevd1[i] = prevd2[i] = 0;
        sum1[i] = sum2[i] = sum3[i] = 0;
```

```
    data[i] = 0;
    kprev[i] = 31;
}
for (pos = 0;  ; pos++) {
    p = pos % HISTLEN;
    for (i = 0; i < channels; i++) {
        if (sum1[i] <= sum2[i]) {
            z = sum1[i];
        } else if (sum2[i] <= sum3[i]) {
            z = sum2[i];
        } else {
            z = sum3[i];
        }
        if (pos <= HISTLEN) {
            if (pos > 1) {
                z = (z * HISTLEN) / (pos - 1);
            } else {
                z = threshold[32];
            }
        }
        if (z < threshold[32]) {
            k = kprev[i];
            if (z < threshold[k]) {
                while (z < threshold[--k]);
            } else {
                while (z >= threshold[k + 1]) k++;
            }
            kprev[i] = (char)k;
            t = getbits(8);
            b = table[k][t];
            c = codelentbl[k][b];
            ungetbits(8 - c);
            if (b != 0) {
                s32 mask = 1 << (b - 1);
                d = (s16)getbits(b);
                if (d & mask) {
                    d = -d;
                    if (d == -32767) {
                        d -= getbits(1);
                        BITCOUNT(i, 1);
                    }
                } else {
                    d |= mask;
                    if (d == 32767) {
                        if (getbits(1)) return;
                        BITCOUNT(i, 1);
                    }
                }
            } else {        /* b == 0 */
                d = 0;
            }
            BITCOUNT(i, b + c);
```

```
            } else {
                d = (s16)getbits(16);
                if (d == -32768) {
                    if (getbits(1)) return;
                    BITCOUNT(i, 1);
                }
                BITCOUNT(i, 16);
                kprev[i] = 31;
            }
            if (sum1[i] <= sum2[i]) {
                d1 = d;
                d2 = (s16)(d1 - prevd1[i]);
                d3 = (s16)(d2 - prevd2[i]);
            } else if (sum2[i] <= sum3[i]) {
                d2 = d;
                d1 = (s16)(d2 + prevd1[i]);
                d3 = (s16)(d2 - prevd2[i]);
            } else {
                d3 = d;
                d2 = (s16)(d3 + prevd2[i]);
                d1 = (s16)(d2 + prevd1[i]);
            }
            data[i] += (u16)d1;
            if (pos == 0) d1 = d2 = d3 = 0;
            prevd1[i] = d1;
            prevd2[i] = d2;

            sum1[i] -= diff1[p][i];
            d1 = ABS16(d1);
            sum1[i] += (unsigned)d1;
            diff1[p][i] = (u16)d1;
            sum2[i] -= diff2[p][i];
            d2 = ABS16(d2);
            sum2[i] += (unsigned)d2;
            diff2[p][i] = (u16)d2;
            sum3[i] -= diff3[p][i];
            d3 = ABS16(d3);
            sum3[i] += (unsigned)d3;
            diff3[p][i] = (u16)d3;
        }
        write_data();
    }
}

/*************************************************************************
  Usage and main
*************************************************************************/

static void usage(void)
{
    fprintf(stderr,
            "NIFSqueezer Version 2.01\n"
```

```
            "Usage: %s [options] filenames\n", progname);
    fprintf(stderr,
            "  Options: -d           : decompress\n"
            "           -e           : exponential (compression)\n"
            "           -x           : swap byte order of data file\n"
            "           -c           : output to stdout\n"
            "           -t           : test only, output to nowhere\n"
            "           -s channels  : number of channels (compression)\n"
            "  Compressesed files have extension '.x'.\n"
            "  Output files are in the current directory.\n"
            "  Existing files are not overwritten.\n");
    fprintf(stderr,
            "  Examples:\n"
            "    %s -s40 mydata  ... compress mydata (40 channels) to mydata.x\n"
            "    %s -d mydata.x  ... decompress mydata.x to mydata\n"
            "    %s -xs40 mydata ... if mydata has wrong endianness\n",
            progname, progname, progname);
    exit(0);
}

static int ratio(u32 a, u32 b) /* [(1000a + [b/2]) / b] */
{
    int i;

    for (i = 0; i < 3; i++)
        if (a <= 0xffffffffUL / 10UL) a *= 10;  else b /= 10;
    if ((u32)(a + (b >> 1)) < a) {  a >>= 1;  b >>= 1;  }
    if (b == 0) return 0;
    return (int)((a + (b >> 1)) / b);
}

int main(int argc, char *argv[])
{
    int i, j, c, m;
    char *p;
    char outfilename[1024];
    int to_stdout = 0;
    int files_processed = 0;

    m = METHOD_GAUSSIAN;
    progname = argv[0];
    for (i = 1; i < argc; i++) {
        if (argv[i][0] == '-') {
            for (j = 1; (c = argv[i][j]) != '\0'; j++) {
                if (c == 'd') {
                    decompress_flag = 1;
                } else if (c == 'e') {
                    m = METHOD_EXP;
                } else if (c == 't') {
                    to_nowhere = 1;
                } else if (c == 'c') {
                    to_stdout = 1;
```

```
        } else if (c == 's') {
            if (argv[i][j+1] != '\0') {
                channels = atoi(&argv[i][j+1]);
                break;
            } else if (++i < argc) {
                channels = atoi(argv[i]);
                break;
            } else {
                error(0, "-s without number of channels");
            }
            if (channels < 1 || channels > MAX_CHANNELS)
                error(0, "number of channels out of range: %d",
                      channels);
        } else if (c == 'x') {
            swap_data = 1;
        } else {
            usage();
        }
    }
} else {                /* if not an option */
    /* We want to output files in the current directory
       because data directory should be kept unwritable for
       safety.  MS-DOS users might need to change '/' to '\\'. */
    if (to_stdout) {
        p = "-";
    } else if ((p = strrchr(argv[i], '/')) != NULL) {
        p++;
    } else {
        p = argv[i];
    }
    if (strlen(p) >= sizeof(outfilename) - 2) {
        warn(0, "filename '%s' too long, skipping", p);
        continue;
    }
    strcpy(outfilename, p);
    if (decompress_flag) {
        if ((p = strrchr(outfilename, '.')) != NULL
            && strcmp(p, ".x") == 0) {
            *p = '\0';
        } else if (strcmp(outfilename, "-") != 0) {
            warn(0, "%s has no suffix .x", argv[i]);
            continue;
        }
        if (ropen(argv[i])) {
            warn(errno, "can't open %s", argv[i]);
            continue;
        }
        if (wopen(outfilename)) {
            warn(errno, "can't open %s", outfilename);
            rclose();
            continue;
        }
```

```
                fprintf(stderr, "%s -> %s\n",
                        argv[i], (to_nowhere) ? "" : outfilename);
                decompress();
            } else {
                if (! to_stdout) strcat(outfilename, ".x");
                if (ropen(argv[i])) {
                    warn(errno, "can't open %s", argv[i]);
                    continue;
                }
                if (wopen(outfilename)) {
                    warn(errno, "can't open %s", outfilename);
                    rclose();
                    continue;
                }
                fprintf(stderr, "%s -> %s\n",
                        argv[i], (to_nowhere) ? "" : outfilename);
                setmethod(m);
                compress();
            }
            rclose();
            wclose();
            c = ratio(totaloutcount, totalincount);
            fprintf(stderr, "out/in = %lu/%lu = %u.%03u\n",
                    2UL * totaloutcount, 2UL * totalincount,
                    c / 1000, c % 1000);
            files_processed++;
        }
    }
    if (files_processed == 0) usage();
    return 0;
}
```

参考文献

[1] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the Data Compression Conference, 1997, Snowbird, Utah*, pp. 201–210. IEEE Computer Society Press, 1997.

[2] Pervez M. Aziz, Henrik V. Sorensen, and Jan van der Spiegel. An overview of sigma-delta converters. *IEEE Signal Processing Magazine*, pp. 61–84, January 1996.

[3] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice-Hall, 1990.

[4] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center, 1994. SRC Research Report 124 (May 10, 1994).

[5] 船木陸議, 羅正華. Linux リアルタイム計測/制御開発ガイドブック. 秀和システム, 1999.

[6] S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, **IT-12** (3), pp. 399–401, 1966.

[7] Darrel Hankerson, Greg A. Harris, and Peter D. Johnson Jr. *Introduction to Information Theory and Data Compression.* CRC Press, 1998.

[8] P. G. Howard and J. S. Vitter. Error modeling for hierarchical lossless image compression. In *Proceedings of the Data Compression Conference, 1992, Snowbird, Utah*, pp. 269–278. IEEE Computer Society Press, 1992.

[9] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, **40**, pp. 1098–1101, 1952.

[10] J. Kariya, H. Okumura, M. Emoto, M. Shoji, Y. Teramachi, T. Ohska, Y. Shamoto, S. Yamaguchi, O. Motojima, and J. Yamamoto. A flexible and extensible monitoring system for large scale experiments. In T. Haruyama,

T. Mitsui, and K. Yamafuji, editors, *Proceedings of the Sixteenth International Cryogenic Engineering Conference/International Cryogenic Meterials Conference, Part 1*, pp. 673–676. Elsevier Science, 1997.

[11] 刈谷丈治, 奥村晴彦, 江本雅彦, 寺町康昌, 大須賀関雄, 山口作太郎, 本島修. 拡張性と柔軟性に富む LHD 超伝導コイル監視用システム. プラズマ・核融合学会誌, **74**, pp. 67–75, 1998.

[12] Jyrki Katajainen and Alistair Moffat. In-place calculation of minimum-redundancy codes. Preprint available from the authors, 1997.

[13] Lawrence L. Larmore and Daniel S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the Association for Computing Machinery*, **37** (3), pp. 464–473, 1990.

[14] Alistair Moffat, Andrew Turpin, and Jyrki Katajainen. Space-efficient construction of optimal prefix codes. In *Proceedings of the Data Compression Conference, 1995, Snowbird, Utah*, pp. 192–201. IEEE Computer Society Press, 1995.

[15] Mark Nelson and Jean loup Gailly. *The Data Compression Book.* M&T Books, New York, NY, second edition, 1996.

[16] H. Okumura, S. Yamaguchi, Y. Teramachi, J. Kariya, T. Ohska, Y. Tanahashi, T. Senba, N. Yanagi, Y. Yamazaki, J. Yamamoto, and O. Motojima. A scalable data acquisition system for superconducting coil experiment. In *Fusion Technology 1994: Proceedings of the 18th Symposium on Fusion Technology, Karlsruhe, Germany*, pp. 835–838. Elsevier, 1995.

[17] 奥村晴彦. ディスクを 2 倍使う方法: データ圧縮のすべて. *The BASIC*, pp. 2–10, 1989 年 3 月号.

[18] 奥村晴彦. C 言語による最新アルゴリズム事典. 技術評論社, 1991.

[19] 奥村晴彦. データ圧縮アルゴリズム概説. *C Magazine*, pp. 44–58, 1991 年 1 月号.

[20] 奥村晴彦. ファイル圧縮技術 (上). *bit*, pp. 4–13, 1994 年 12 月号.

[21] 奥村晴彦. ファイル圧縮技術 (下). *bit*, pp. 53–62, 1995 年 1 月号.

[22] 奥村晴彦. アナログ・ディジタル変換器出力の実時間圧縮. プラズマ・核融合学会誌, **73**, pp. 1135–1140, 1997.

[23] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing.* Prentice-Hall, second edition, 1999.

[24] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold (Chapman & Hall), 1993.

[25] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report Tech. Rep. JPL-79-22, Jet Propulsion Laboratory, Pasadena, CA, March 1979.

[26] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM J. Res. Devel.*, **20**, pp. 198–203, 1976.

[27] Tony Robinson. SHORTEN: Simple lossless and near-lossless waveform compression. Technical Report CUED/F-INFENG/TR.156, Cambridge University Engineering Department, Trumpington Street, Cambridge, CB2 1PZ, UK, December 1994.

[28] David Salomon. *Data Compression: The Complete Reference.* Springer, 1998.

[29] Michael Schindler. A fast renormalisation for arithmetic coding. In *Proceedings of the Data Compression Conference, 1998, Snowbird, Utah*, p. 572. IEEE Computer Society Press, 1998.

[30] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, **27**, pp. 379–423, 1948. Reprinted in [31].

[31] N. J. A. Sloane and Aaron D. Wyner, editors. *Claude Elwood Shannon: Collected Papers.* IEEE Press, 1993.

[32] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the Association for Computing Machinery*, **29** (4), pp. 928–951, 1982.

[33] K. Thompson and D. M. Ritchie. The UNIX time-sharing system. *Communications of the ACM*, **17** (7), pp. 365–375, 1974.

[34] John W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, 1977.

[35] Uresh Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, 1996.

[36] Sergio Verdú. Fifty years of shannon theory. *IEEE Transactions on Information Theory*, **44** (6), pp. 2057–2078, 1998.

[37] Marcelo J. Weinberger, Gadiel Seroussi, and Guillermo Sapiro. LOCO-I: A low complexity, context-based, lossless image compression algorithm. In *Proceedings of the Data Compression Conference, 1996, Snowbird, Utah*, pp. 140–149. IEEE Computer Society Press, 1996.

[38] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, **17** (6), pp. 8–19, 1984.

[39] Ross N. Williams. *Adaptive Data Compression.* Kluwer Academic Publishers,

1991.

[40] 山口作太郎, 庄司主, 三戸利行, 山崎耕造, 刈谷丈治, 奥村晴彦, 江本雅彦, 寺町康昌, 大須賀闘雄. 超伝導コイル実験監視システム. プラズマ・核融合学会誌, **73**, pp. 335–342, 1997.

[41] 安芳次. Java と分散オブジェクトによるデータ収集システム構築の基礎と応用. インターフェース, pp. 56–78, 1998 年 10 月号.

[42] 吉田茂, 岡田佳之. LZ 方式を使わない独自圧縮方式: パソコンから大型機に対応. 日経エレクトロニクス別冊 データ圧縮とディジタル変調 98 年版, pp. 235–242, 1998.

[43] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, **IT-23** (3), pp. 337–343, 1977.

[44] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, **IT-24** (5), pp. 530–536, 1978.

索引