

ロスレス音声コーデック

— 基本理論と実装 —

あいき 著

2019-09-22 版 発行

まえがき

ロスレス音声コーデックを作ってみませんか。ロスレス音声コーデックは、`wav` 等の音声ファイルを音質の劣化なく圧縮するソフトウェアです。平たく言うと、音声ファイルに特化した `zip` と考えてもらって問題ありません。ロスレス音声コーデックの特色は、なんと言っても音質の劣化が無いことです。皆さんが音楽を聞くときによく使う `mp3` はロッキーな音声コーデックであり、実は音質が劣化した状態で再生されています。何もこれは悪いことではなく、人間が聞こえない音声情報を削ることで、かなりの圧縮率を達成しています。一般にロスレス音声コーデックはロッキーなコーデックに比べて圧縮率は悪い傾向があります。しかし、音質が劣化しないことで、製作者側が聞かせたかった“本当の音”を再現することができます。なんだか宗教じみていますが、デジタルデータを再生するときに音質が損なわれないというお墨付きは重要です。

ロスレス音声コーデックと言えば、`FLAC`[28] が有名なコーデックとして挙げられます。しかし、実は `FLAC` は他のロスレス音声コーデック [16, 27, 30, 31, 32, 33, 35] に比べて展開速度が早い反面、圧縮率が悪いという欠点があります。本稿では ”`FLAC` を超える圧縮率を持つロスレス音声コーデック” の作成を目標として、その基本理論と実装を説明していきます。

目次

第1章 基本理論編	7
1.1 線形予測	8
1.1.1 予測とデータ圧縮	8
1.1.2 線形予測の理論	10
線形予測	10
Levinson-Durbin 再帰 (Levinson-Durbin recursion) の導出 . .	12
Levinson-Durbin 再帰アルゴリズム	15
PARCOR 格子型フィルター	16
PARCOR 係数による信号合成	18
PARCOR 係数の値域	19
1.1.3 線形予測の実装	20
実験	24
1.2 エントロピー符号	24
1.2.1 情報とその符号化	25
一意に復号可能な符号	26
McMillan (マクミラン) の不等式	27
1.2.2 情報量とエントロピー	29
情報量の定義	29
情報量の導出	30
エントロピー	32
平均符号長とエントロピー	34
様々なファイルのエントロピー計測	36
1.2.3 エントロピー符号の例	39
α, γ 符号とその確率分布	39
Golomb (ゴロム) 符号	40
Golomb (ゴロム) 符号のパラメータ設定	44

第2章 実装編	47
2.1 ALA コマンドラインツールの使用方法	47
2.1.1 ビルド方法	47
2.1.2 実行	48
エンコード	48
デコード	48
2.2 ALA ファイルフォーマット	48
2.2.1 ファイルヘッダフォーマット	48
2.2.2 ブロックフォーマット	49
2.3 ala_predictor.c	50
2.3.1 固定小数演算	50
固定小数による小数の表記	51
固定小数演算	52
2.3.2 PARCOR 予測フィルター	53
乗算時の右シフト対策の定数	55
前向き誤差計算	55
後ろ向き誤差計算	56
2.3.3 PARCOR 合成フィルター	56
2.3.4 MS 処理	57
MS 処理の実装解説	58
2.3.5 プリエンファシス/デエンファシス	60
プリエンファシスフィルター処理	60
デエンファシスフィルター処理	61
2.3.6 残差のエントロピー観察	62
元の信号と残差の分布	62
残差計算前後のエントロピーの変化	62
2.4 bit_stream.c	67
2.4.1 bit 単位の出力処理	67
1bit 出力処理	67
複数 bit 出力処理	68
2.4.2 bit 単位の取得処理	70
1bit 取得処理	70
複数 bit 取得処理	71
2.5 ala_coder.c	73
2.5.1 Rice 符号	73
Rice 符号化処理	73

	Rice 符号化されたデータの復号処理	74
2.5.2	残差の符号化と復号	75
	残差の符号化	75
	残差の復号	79
2.6	wav.c	80
	2.6.1 構造体定義	81
	2.6.2 公開 API	82
	wav ファイルの読み込み	82
	wav ファイルの書き込み	82
	2.6.3 使用例	82
2.7	main.c	83
	2.7.1 ALA のエンコード処理	83
	入力 wav データの取得	84
	エンコードするサンプル数の決定	84
	窓掛け	85
	PARCOR 係数の量子化	86
	PARCOR 係数のエンコード	87
	ブロック末尾のバイト境界揃え	87
	2.7.2 ALA のデコード処理	87
	シグネチャとフォーマットバージョンのチェック	88
	デコードするサンプル数の決定	89
	2.7.3 引数処理	89
2.8	FLAC との比較	90
	2.8.1 所感	91
2.9	性能向上の指針	91
	2.9.1 他のロスレス音声コーデックの概略	91
	FLAC(Free Lossless Audio Codec)	92
	Wavpack	92
	TTA(The True Audio)	92
	MPEG-4 ALS(The MPEG-4 Audio Lossless Coding)	92
	Monkey's Audio	93
	TAK(Tom's verlustfreier Audiokompressor)	93
	2.9.2 予測手法	93
	Burg 法	93
	1乗算型 PARCOR 格子型フィルター	96
	適応フィルター	96

2.9.3	他のエントロピー符号	97
	ランレンジス符号化	97
	ハフマン、適応的ハフマン	97
	ユニバーサル符号化	97
2.9.4	その他	98
	PARCOR 係数の符号化	98
参考文献		99

第1章

基本理論編

ロスレス音声コーデックによって.wav等の音声データファイルをエンコードするとき、大まかに言うと図1.1に示す処理が動いている。

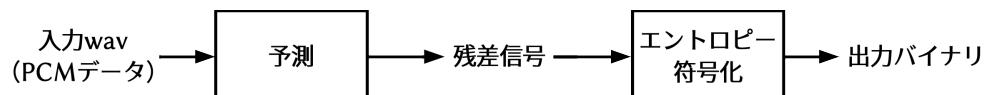


図1.1 ロスレス音声コーデックのエンコード処理概要

エンコードの処理の内容について、簡単に説明すると次のようになる。

ロスレス音声コーデックのエンコード手順

1. 入力の wav を取得する。
wav ファイルにおいて、音声信号データは PCM^aで符号化されている。
2. 入力の音声データパターンを解析して、音声波形の予測を行い、元波形と予測の差を取って残差 (residual) を計算する。
3. エントロピー符号 (entropy coding) によって残差信号をビット列に符号化 (エンコード) する。
4. 出力バイナリデータが得られる。

^a Pulse Code Modulation。信号の時間と振幅を量子化（離散化）して記録する方式を指す。

ここで、エントロピー符号とは、入力データの確率に基づいて、出力ビット列が短くなるように符号化を行う符号化手法である。次にデコードの処理手順について説明する。デコードは図1.2に示すように、エンコードの逆の手順を辿ることで元の wav に復元する。



図 1.2 ロスレス音声コーデックのデコード処理概要

ロスレス音声コーデックのデコード手順

1. 入力のバイナリデータを読み込む。
2. バイナリデータから残差を復号（デコード）する。
3. 残差とそれまでに復元した入力信号から予測を行って入力信号を復元する。
4. 元の wav ファイルが得られる。

ロスレス音声コーデックの場合は、入力した wav がビットパーフェクトに（1bit も差異もなく）復元される^{*1}。ビットパーフェクトの要請を満たすためには、エンコードとデコードの処理において、予測とエンタロピー符号の処理はそれぞれ元に戻るよう構成しなければならないことに注意する。予測した結果を元に戻すには、大雑把に言うと整数演算の性質を使うことで達成できる^{*2}。また、エンタロピー符号についても、構成した符号が一意に復号可能となる（曖昧さを残さずに元に戻すことができる）条件はよく知られている。

本章では、ロスレス音声コーデックの基礎を支える予測とエンタロピー符号の技術をそれぞれ紹介していく。

1.1 線形予測

1.1.1 予測とデータ圧縮

工学的な立場から予測というと、予測を行うまでに観測したデータから現在、あるいは未来の状態を推定することを言う。推定を行うための理論には必ず確率論が潜んでいる。ここでは確率論に立ち入らず、予測についての概要と、それが如何にしてデータ圧縮に役立つかを見していく。

例として、表 1.1 に示すような、時刻に対応した信号値の列が観測されたとしよう。

表 1.1において、時刻 9 に対応する信号値は未知 “?” であり予測する必要がある。この状況で、貴方なら時刻 9 に対応する信号値をどう予測するだろうか？信号値が 2 時刻で 1 ずつ上昇している所を観察すると、直感的には ? = 2 であると予測できるのではないだろ

^{*1} mp3, ogg 等のロッキーなコーデックは、デコード結果の波形は元の入力素材と一致しない。

^{*2} 浮動小数点演算はアーキテクチャごと（例：Intel 系プロセッサと ARM 系プロセッサ）に得られる結果が異なるため、ロスレス音声コーデックでは浮動小数点演算が使われることは基本的にはない。

表 1.1 信号値の例

時刻	0	1	2	3	4	5	6	7	8	9
信号値	-2	-2	-1	-1	0	0	1	1	2	?

うか。³

この様に予測できるのは、経験的に信号値の連続性を仮定しているからではないだろうか。実際、画像や音声ファイルの信号を観察すると、時間的、あるいは空間的に隣り合った信号値は大きく変化することはあまり見られない。例えば音声データにおいては、時刻が隣り合った信号値が大きく異なるとチクノイズとなって耳障りに聞こえてしまうので、自然な音声においてはなめらかな、即ち連続な信号値でなくてはならない。一般にマルチメディアのデータ圧縮は、まさにこの信号値の連続性に着目して圧縮を達成している。

予測が圧縮に活かされる例を見ていく。表 1.1 の信号値に対して $? = 2$ だったとして、直前の時刻の信号値がそのまま続くと予測してみる。⁴ 予測と同時に、予測に対しての誤差（残差）を計算しておく。残差は次の式 1.1 で定義される。

$$\text{残差} = \text{信号値} - \text{予測値} \quad (1.1)$$

予測値と残差を追記した結果を表 1.2 に示す。

表 1.2 予測の例

時刻	0	1	2	3	4	5	6	7	8	9
信号値	-2	-2	-1	-1	0	0	1	1	2	2
予測値	-	-2	-2	-1	-1	0	0	1	1	2
残差	-	0	1	0	1	0	1	0	1	0

表 1.2において、残差は 0, 1 だけで構成されていることが分かる。これはデータ圧縮にとって都合が良い。なぜなら、圧縮すべき対象の種類が少なく、かつ、その値が 0 付近に偏って出現しているからである。この性質によって残差は元の信号値よりも小さく圧縮することができる。

圧縮された残差を戻すときには、それまでに復元した信号値から圧縮時と全く同様の予測を行い、残差に予測値を加えれば良い（式 1.2）。

$$\text{信号値} = \text{残差} + \text{予測値} \quad (1.2)$$

³ 真面目な人であれば n 次の多項式を使って最小二乗法でフィッティングすると吹っ飛んだ値になるとか、ロジスティック回帰だ、SVR だ、ガウス過程回帰だと言い始めるかもしれない。

⁴ この予測方式を特に前値予測と呼ぶことがある。

マルチメディアのロスレス圧縮技術は、式 1.1, 1.2 に集約されると言って良い。この様に圧縮の仕組みは非常に単純だが、予測については大きな研究課題となる。予測を行う方が圧縮対象のデータを精度良く予測できていれば、残差はより小さくなるため、効率的な圧縮が実現できる。

1.1.2 線形予測の理論

この節では、音声圧縮の世界で伝統的に用いられている線形予測 (linear prediction) の仕組みを説明する。線形予測は大雑把にいって表 1.2 で示した予測形式の拡張として解釈できる。線形予測では、直前の時刻だけでなく、より過去の信号値を参照して予測を行うことを考える。ここで、過去の信号には係数（重み）をつけて予測を行う。この時、線型予測の理論に従うと一定の基準で最適な係数を求めることができる。最適な係数の求め方はよく知られている [19] ため、そのアルゴリズムも並べて解説する。

線形予測

時間について離散化した信号が $y(0), y(1), \dots, y(n)$ として得られたとする。ここで、 $y(n)$ を直前の $y(m)$ ($m = 0, \dots, n - 1$) によって予測する事を考える。線形予測では M 個の係数 $a(1), \dots, a(M)$ を用いた単純な線形結合

$$-a(1)y(n-1) - a(2)y(n-2) - \cdots - a(M)y(n-M) = -\sum_{m=1}^M a(m)y(n-m)$$

を使って予測を行う^{*5}。 M 個の係数を使用したとき、係数の次数が M であると言う。次数を明示的に表すために係数 a の下付き文字で次数を表し、 M 次の係数を $a_M(1), \dots, a_M(M)$ によって表す。線形結合による $y(n)$ の近似式は、次の式 1.3 で表される。

$$y(n) \approx -\sum_{m=1}^M a_M(m)y(n-m) \quad (1.3)$$

予測の誤差は、全ての n における二乗誤差の和 E によって測る:

$$\begin{aligned} E &= \sum_{n=-\infty}^{\infty} \left[y(n) - \left\{ -\sum_{m=1}^M a_M(m)y(n-m) \right\} \right]^2 \\ &= \sum_{n=-\infty}^{\infty} \left\{ y(n) + \sum_{i=1}^M a_M(i)y(n-i) \right\}^2 \end{aligned}$$

^{*5} ここで、係数に負号 $-$ が付いているのは、システムのフィードバック係数として捉えた時は負を付けるのが常識となっているからと考えられる。全ての係数の符号を反転させれば通常の和に戻るので、以下の導出にとって本質的な問題にならない。

ここで $a_M(0) = 1$ と定義すると、

$$E = \sum_{n=-\infty}^{\infty} \left\{ \sum_{m=0}^M a_M(m)y(n-m) \right\}^2 \quad (1.4)$$

とまとめられる。後は、この E を最小化するように係数 $a_M(1), \dots, a_M(M)$ を定めれば良い。

次に、誤差の最小化を考える。常套手段ではあるが、 E を $a_M(j)$ ($j = 1, \dots, M$) によって偏微分し、その結果を 0 とおいて解くことを考える。まず、 E の偏微分は、

$$\begin{aligned} \frac{\partial E}{\partial a_M(j)} &= \sum_{n=-\infty}^{\infty} \frac{\partial}{\partial a_M(j)} \left\{ \sum_{m=0}^k a_M(m)y(n-m) \right\}^2 \\ &= \sum_{n=-\infty}^{\infty} \frac{\partial}{\partial a_M(j)} \{a_M(0)y(n) + \dots + a_M(j)y(n-j) + \dots + a_M(M)y(n-k)\}^2 \\ &= \sum_{n=-\infty}^{\infty} 2y(n-j) \sum_{m=0}^M a_M(m)y(n-m) \\ &= 2 \sum_{m=0}^M a_M(m) \sum_{n=-\infty}^{\infty} y(n-j)y(n-m) \quad (\because \text{和の順序交換}) \\ &= 2 \sum_{m=0}^M a_M(m) \sum_{n'=-\infty}^{\infty} y(n')y(n'+j-m) \quad (n' = n - j \text{ とおいた}) \end{aligned}$$

ここで、**自己相関 (auto correlation)** $R(l)$ を次の式 1.5 で定義する：

$$R(l) = \sum_{n=-\infty}^{\infty} y(n)y(n+l) \quad (1.5)$$

自己相関 $R(l)$ を用いることで、偏微分の結果は式 1.6 の様に表せる。

$$\frac{\partial E}{\partial a_M(j)} = 2 \sum_{m=0}^M a_M(m) R(|j-m|) \quad (1.6)$$

次に、 $\frac{\partial E}{\partial a_M(j)} = 0$ ($j = 1, \dots, M$) とおいて解く事を考える。和の前に付いている係数 2 は両辺 2 で割ることで消すことが出来る。その上で $j = 1, \dots, M$ での式 (1.6) を並べてみると、

$$\begin{aligned} a_M(0)R(|0-1|) + a_M(1)R(|1-1|) + \dots + a_M(M)R(|k-1|) &= 0 \\ a_M(0)R(|0-2|) + a_M(1)R(|1-2|) + \dots + a_M(M)R(|k-2|) &= 0 \end{aligned}$$

⋮

$$a_M(0)R(|0-k|) + a_M(1)R(|1-k|) + \dots + a_M(M)R(|k-k|) = 0$$

より、行列形式で

$$\begin{bmatrix} R(1) & R(0) & R(1) & \dots & R(M-1) \\ R(2) & R(1) & R(0) & \dots & R(M-2) \\ \vdots & & & \ddots & \vdots \\ R(M) & R(M-1) & R(M-2) & \dots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_M(1) \\ a_M(2) \\ \vdots \\ a_M(M) \end{bmatrix} = \vec{0}$$

と表せられる。以下、

$$M = \begin{bmatrix} R(1) & R(0) & R(1) & \dots & R(M-1) \\ R(2) & R(1) & R(0) & \dots & R(M-2) \\ \vdots & & & \ddots & \vdots \\ R(M) & R(M-1) & R(M-2) & \dots & R(0) \end{bmatrix}, \quad \vec{a}_M = \begin{bmatrix} 1 \\ a_M(1) \\ a_M(2) \\ \vdots \\ a_M(M) \end{bmatrix} \quad (1.7)$$

として、 $M\vec{a}_M = \vec{0}$ を解くことを考える。

Levinson-Durbin 再帰 (Levinson-Durbin recursion) の導出

前節で求めた連立方程式 $M\vec{a}_M = \vec{0}$ をもう少し整理していく。数値解法的には、 M は正方行列にしておくのが望ましい。そこで、 M の一番上の行に $[R(0)R(1)\dots R(M)]$ を追加すると、

$$M\vec{a}_k = \begin{bmatrix} R(0) & R(1) & \dots & R(M) \\ R(1) & R(0) & \dots & R(M-1) \\ \vdots & & \ddots & \vdots \\ R(M) & R(M-1) & \dots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_M(1) \\ a_M(2) \\ \vdots \\ a_M(M) \end{bmatrix} - \begin{bmatrix} \sum_{m=0}^M a_M(m)R(m) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0}$$

と変形できる。よって、係数 $a_M(1), \dots, a_M(M)$ を求める問題は、次の連立方程式 1.8 を解くことに帰着できる。

$$\begin{bmatrix} R(0) & R(1) & \dots & R(M) \\ R(1) & R(0) & \dots & R(M-1) \\ \vdots & & \ddots & \vdots \\ R(M) & R(M-1) & \dots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_M(1) \\ a_M(2) \\ \vdots \\ a_M(M) \end{bmatrix} = \begin{bmatrix} \sum_{m=0}^M a_M(m)R(m) \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad (1.8)$$

連立方程式 1.8 を高速に解くアルゴリズムが、**Levinson-Durbin 再帰法 (Levinson-Durbin recursion)** である。以下、 $e_M = \sum_{i=0}^M a_M(m)R(m)$ とし、また自己分散行列 N_M を式 1.9 で定義する。

$$N_M = \begin{bmatrix} R(0) & R(1) & \dots & R(M) \\ R(1) & R(0) & \dots & R(M-1) \\ \vdots & & \ddots & \vdots \\ R(M) & R(M-1) & \dots & R(0) \end{bmatrix} \quad (1.9)$$

Levinson-Durbin 再帰法は、数学的帰納法によく似ており、次のステップにより係数を求めていく。⁶

1. 初期化ステップ: 1 次の係数 $a_1(1)$ を求める。
2. 再帰ステップ: k 次の係数 $a_k(1), \dots, a_k(k)$ から、 $k+1$ 次の係数 $a_{k+1}(1), \dots, a_{k+1}(k+1)$ を求める。

ここでは、1. および 2. の場合の解をそれぞれ見ていく。

■初期化ステップ $M = 1$ のとき、 \vec{a}_1, N_1 は次の様に定義される。

$$\vec{a}_1 = \begin{bmatrix} 1 \\ a_1(1) \end{bmatrix}, \quad N_1 \vec{a}_1 = \begin{bmatrix} e_1 \\ 0 \end{bmatrix}, \quad N_1 = \begin{bmatrix} R_0 & R_1 \\ R_1 & R_0 \end{bmatrix}$$

実際に $N_1 \vec{a}_1$ を計算してみると、

$$N_1 \vec{a}_1 = \begin{bmatrix} R_0 + R_1 a_1(1) \\ R_1 + R_0 a_1(1) \end{bmatrix} = \begin{bmatrix} e_1 \\ 0 \end{bmatrix}$$

従って、 $e_1 = R_0 + R_1 a_1(1)$ 、及び $R_1 + R_0 a_1(1) = 0$ から $a_1(1) = -\frac{R_1}{R_0}$ と求められる。⁷

■再帰ステップ 仮定として、

$$N_k \vec{a}_k = \begin{bmatrix} R(0) & R(1) & \dots & R(k) \\ R(1) & R(0) & \dots & R(k-1) \\ \vdots & & \ddots & \vdots \\ R(k) & R(k-1) & \dots & R(0) \end{bmatrix} \begin{bmatrix} 1 \\ a_k(1) \\ a_k(2) \\ \vdots \\ a_k(k) \end{bmatrix} = \begin{bmatrix} e_k \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

が成立していたとする。 $M = k+1$ の時、行列 N_{k+1} は

$$\begin{aligned} N_{k+1} &= \begin{bmatrix} R(0) & R(1) & \dots & R(k) & R(k+1) \\ R(1) & R(0) & \dots & R(k-1) & R(k) \\ \vdots & & \ddots & & \vdots \\ R(k) & R(k-1) & \dots & R(0) & R(1) \\ R(k+1) & R(k) & \dots & R(1) & R(0) \end{bmatrix} \\ &= \left[\begin{array}{c|c} & R(k+1) \\ N_k & R(k) \\ & \vdots \\ & R(1) \\ \hline R(k+1) & R(k) & \dots & R(1) & R(0) \end{array} \right] \end{aligned}$$

⁶ 参考資料 [19] で筆者は、「Levinson-Durbin 帰納法と言ったほうがいいんじゃないかな」と書いてあった。

⁷ $R_0 = \sum_{n=-\infty}^{\infty} y_n^2 > 0$ より、確率論的に至る所ゼロ除算の心配はない…が、デジタル信号においては y_n が全て 0 のときが往々にして起こる。この場合は、係数を全て 0 にする等の特殊処理が必要である。

となり、 N_k の行・列共に 1つ増えた行列となる。一方の \vec{a}_{k+1} は未知である。そこで、技巧的ではあるが \vec{a}_k に 0 を追加する事で拡張した次のベクトル $\vec{u}_{k+1}, \vec{v}_{k+1}$ を導入する。

$$\vec{u}_{k+1} = \begin{bmatrix} 1 \\ a_k(1) \\ a_k(2) \\ \vdots \\ a_k(k) \\ 0 \end{bmatrix}, \quad \vec{v}_{k+1} = \begin{bmatrix} 0 \\ a_k(k) \\ a_k(2) \\ \vdots \\ a_k(1) \\ 1 \end{bmatrix}$$

$\vec{u}_{k+1}, \vec{v}_{k+1}$ は互いに要素を反転したベクトルである。これら $\vec{u}_{k+1}, \vec{v}_{k+1}$ を用いて $N_{k+1}\vec{u}_{k+1}$ と $N_{k+1}\vec{v}_{k+1}$ を計算すると、まず $N_{k+1}\vec{u}_{k+1}$ は

$$\begin{aligned} N_{k+1}\vec{u}_{k+1} &= \left[\begin{array}{c|c} N_k & \begin{matrix} R(k+1) \\ R(k) \\ \vdots \\ R(1) \end{matrix} \\ \hline R(k+1) & R(k) & \dots & R(1) \end{array} \right] \begin{bmatrix} 1 \\ a_k(1) \\ a_k(2) \\ \vdots \\ a_k(k) \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} N_k\vec{a}_k \\ [R(k+1) \ R(k) \ \dots \ R(1)]\vec{a}_k \end{bmatrix} = \begin{bmatrix} e_k \\ 0 \\ \vdots \\ 0 \\ \sum_{j=0}^k a_k(j)R(k+1-j) \\ e_k \end{bmatrix} \end{aligned}$$

であり、もう一方の $N_{k+1}\vec{v}_{k+1}$ は、 N_{k+1} が**対称行列**なので、以下の様に、 $N_{k+1}\vec{u}_{k+1}$ の結果を反転したベクトルとなる。

$$N_{k+1}\vec{v}_{k+1} = \begin{bmatrix} \sum_{j=0}^k a_k(j)R(k+1-j) \\ 0 \\ \vdots \\ 0 \\ e_k \end{bmatrix}$$

そして、 \vec{a}_{k+1} は \vec{u}_{k+1} と \vec{v}_{k+1} の線形結合で表現できる。

$$\vec{a}_{k+1} = \vec{u}_{k+1} + \lambda_{k+1}\vec{v}_{k+1} \quad (1.10)$$

ここで λ_{k+1} は実数の重み係数で、**反射係数**と呼ばれる。この式 1.10 は、実際に

$N_{k+1}\vec{a}_{k+1} = N_{k+1}(\vec{u}_{k+1} + \lambda_{k+1}\vec{v}_{k+1})$ を計算することで導出できる。

$$N_{k+1}(\vec{u}_{k+1} + \lambda_{k+1}\vec{v}_{k+1}) = N_{k+1}\vec{u}_{k+1} + \lambda_{k+1}N_{k+1}\vec{v}_{k+1}$$

$$= \begin{bmatrix} e_k + \lambda_{k+1} \sum_{j=0}^k a_k(j)R(k+1-j) \\ 0 \\ \vdots \\ 0 \\ \sum_{j=0}^k a_k(j)R(k+1-j) + \lambda_{k+1}e_k \end{bmatrix}$$

ここで $\lambda_{k+1} = -\frac{\sum_{j=0}^k a_k(j)R(k+1-j)}{e_k}$ とすれば、

$$N_{k+1}\vec{a}_{k+1} = N_{k+1}(\vec{u}_{k+1} + \lambda_{k+1}\vec{v}_{k+1})$$

$$= \begin{bmatrix} e_k - \lambda_{k+1}^2 e_k \\ 0 \\ \vdots \\ 0 \\ \sum_{j=0}^k a_k(j)R(k+1-j) - \sum_{j=0}^k a_k(j)R(k+1-j) \end{bmatrix} = \begin{bmatrix} (1 - \lambda_{k+1}^2)e_k \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

となって e_{k+1} を求めることができる上、 $M = k + 1$ の時でも再帰が成立することが分かる。

Levinson-Durbin 再帰アルゴリズム

前節までの導出結果をまとめると、

1. $k = 1$ の時:

$$a_1(1) = -\frac{R(1)}{R(0)} \quad (1.11)$$

$$e_1 = R(0) + R(1)a_1(1) \quad (1.12)$$

2. k が求まった時、 $k + 1$ は:

$$\lambda_{k+1} = -\frac{\sum_{j=0}^k a_k(j)R(k+1-j)}{e_k} \quad (1.13)$$

$$e_{k+1} = (1 - \lambda_{k+1}^2)e_k \quad (1.14)$$

$$\vec{a}_{k+1} = \vec{u}_{k+1} + \lambda_{k+1}\vec{v}_{k+1} \quad (1.15)$$

ここで、

$$R(l) = \sum_{n=-\infty}^{\infty} y(n)y(n+l), \quad \vec{u}_{k+1} = \begin{bmatrix} 1 \\ a_k(1) \\ \vdots \\ a_k(k) \\ 0 \end{bmatrix}, \quad \vec{v}_{k+1} = \begin{bmatrix} 0 \\ a_k(k) \\ \vdots \\ a_k(1) \\ 1 \end{bmatrix}$$

なお、自己相関 $R(l)$ は過去から未来までの無限の信号和になっているので現実の計算機では計算出来ない。実装においては、自己相関の代わりに式 1.16 の標本自己相関 (sample autocorrelation) $\tilde{R}(l)$ を用いる。⁸

$$\tilde{R}(l) = \sum_{i=0}^n y(i)y(i-l) \quad (l = 0, \dots, k) \quad (1.16)$$

PARCOR 格子型フィルター

反射係数の符号を反転させた係数を **PARCOR 係数 (partial auto-correlation coefficient)** という。PARCOR 係数 γ_{k+1} は次の式 1.17 で計算できる。

$$\gamma_{k+1} = -\lambda_{k+1} = \frac{\sum_{j=0}^k a_k(j)R(k+1-j)}{e_k} \quad (1.17)$$

本節では、PARCOR 係数を用いた線型予測手法を見ていく。

線形予測の近似式 1.3 は近似であり誤差を含んでいる。式 1.3 は過去の信号を参照してより未来の信号を予測しているから、この誤差を特に**前向き誤差 (forward residual)** という。 M 次の係数を使用した際の、時刻 n における前向き誤差を $f_M(n)$ と表し、式 1.3 を近似を用いない形で書き換えると、前向き誤差は、

$$y(n) = f_M(n) - \sum_{m=1}^M a_M(m)y(n-m) \\ \iff f_M(n) = \sum_{m=0}^M a_M(m)y(n-m) \quad (a_M(0) = 1 \text{ とおいた}) \quad (1.18)$$

の形に表すことができる。前向き誤差があるということは、係数の適用方向を逆にしたときの誤差、即ち**後ろ向き誤差 (backward residual)** も存在する。 M 次係数を使用した際の、時刻 n における後ろ向き誤差 $b_M(n)$ は、次の式で定義される。

$$b_M(n) = \sum_{m=0}^M a_M(M-m)y(n-m) \quad (1.19)$$

⁸ 標本自己相関は自分自身との相関を計算するので $O(N^2)$ の計算量があるが、**ヴィーナー・ヒンチンの定理** (信号のパワースペクトラムは、その自己相間に等しい) を使って自己相関を計算すれば、実質 FFT と同等の計算量 $O(N \log N)$ で抑えることもできる。但し、巡回畳み込みや、パワースペクトラムの平均処理を考慮する必要がある。

これらの式 1.18, 1.19 を変形することで、線形予測の別の定式化を行うことを考える。Levinson-Durbin 法の係数更新式 1.13 に注目すると、 $\vec{a}_M = \vec{u}_M + \lambda_M \vec{v}_M = \vec{u}_M - \gamma_M \vec{v}_M$ は行毎に次の計算をしていることが観察できる。

$$a_M(m) = a_{M-1}(m) - \gamma_M a_{M-1}(M-m) \quad (m = 0, \dots, M) \quad (1.20)$$

この式 1.20 を前向き誤差の式 1.18 に代入すると、

$$\begin{aligned} f_M(n) &= \sum_{m=0}^M a_M(m)y(n-m) \\ &= \sum_{m=0}^M \{a_{M-1}(m) - \gamma_M a_{M-1}(M-m)\} y(n-m) \\ &= \sum_{m=0}^M a_{M-1}(m)y(n-m) - \gamma_M \sum_{m=0}^M a_{M-1}(M-m)y(n-m) \\ &= \sum_{m=0}^{M-1} a_{M-1}(m)y(n-m) - \gamma_M \sum_{m=1}^{M-1} a_{M-1}(M-m)y(n-m) \quad (\because a_{M-1}(M) = 0) \\ &= \sum_{m=0}^{M-1} a_{M-1}(m)y(n-m) - \gamma_M \sum_{m=0}^{M-1} a_{M-1}(M-1-m)y(n-1-m) \quad (\because m \rightarrow m+1) \\ &= f_{M-1}(n) - \gamma_M b_{M-1}(n-1) \end{aligned} \quad (1.21)$$

式 1.21 は、PARCOR 係数 γ_m ($m = 1, \dots, M$) から、前向き誤差を計算できることを示している。後ろ向き誤差についても、前向き誤差のときと同じく、後ろ向き誤差の式 1.19 に式 1.20 を代入すれば、

$$\begin{aligned} b_M(n) &= \sum_{m=0}^M a_M(M-m)y(n-m) \\ &= \sum_{m=0}^M \{a_{M-1}(M-m) - \gamma_M a_{M-1}(m)\} y(n-m) \\ &= \sum_{m=0}^M a_{M-1}(M-m)y(n-m) - \gamma_M \sum_{m=0}^M a_{M-1}(m)y(n-m) \\ &= \sum_{m=0}^M a_{M-1}(M-m)y(n-m) - \gamma_M \sum_{m=0}^{M-1} a_{M-1}(m)y(n-m) \quad (\because a_{M-1}(M) = 0) \\ &= \sum_{m=0}^{M-1} a_{M-1}(M-1-m)y(n-1-m) - \gamma_M \sum_{m=0}^{M-1} a_{M-1}(m)y(n-m) \quad (\because m \rightarrow m+1) \\ &= b_{M-1}(n-1) - \gamma_M f_{M-1}(n) \end{aligned} \quad (1.22)$$

となり、後ろ向き誤差も PARCOR 係数から計算できる。式 1.21, 1.22 は、図 1.3 に示すフィルターによって計算できる。このフィルターは、乗算器を交差させた特有の構造に

より、格子型フィルター (Lattice filter) と呼ばれる。

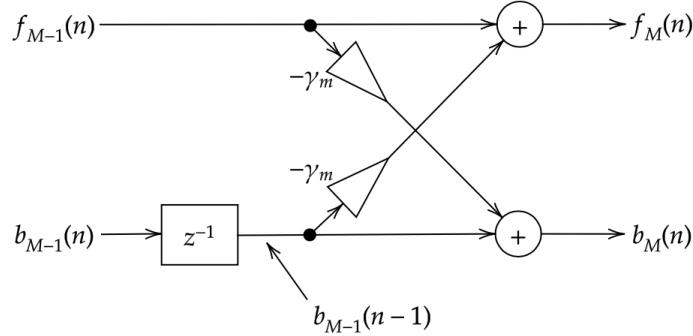


図 1.3 格子型フィルターによる誤差計算

更に図 1.4 の様に、 $f_0(n) = b_0(n) = y(n)$ とした上で格子型フィルターを連結することで、任意次数の前向き誤差と後ろ向き誤差を逐次計算できる。

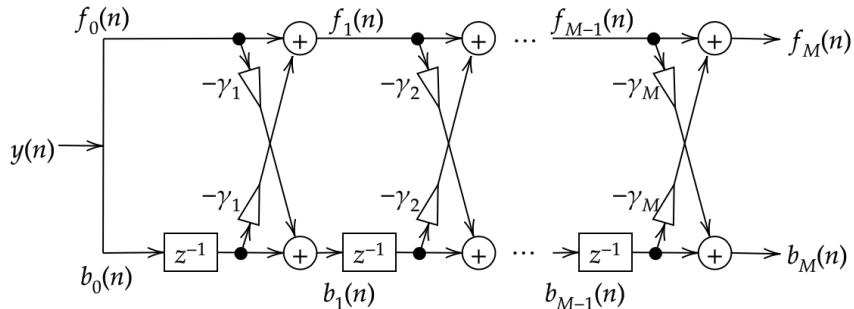


図 1.4 格子型フィルターの連結

PARCOR 係数による信号合成

前向き誤差の式 1.21 の移項操作により、式 1.23 のように、より高い次数の前向き誤差 $f_M(n)$ から 1 つ低い次数の前向き誤差 $f_{M-1}(n)$ を計算できる。

$$f_{M-1}(n) = f_M(n) + \gamma_M b_{M-1}(n-1) \quad (1.23)$$

式 1.23 の結果は重要である。これは計算済みの前向き誤差 $f_M(n)$ から、式 1.23, 1.22 を繰り返し計算することで元の信号 $y(n)$ を再合成できることを示している。格子型フィルターとして表現すると図 1.5 の様に、図 1.4 の格子型フィルターの前向き誤差の流れる（計算する）方向を逆転させたフィルターであることが分かる。

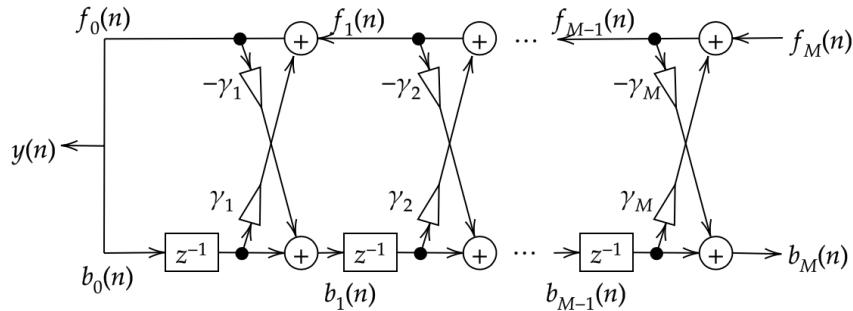


図 1.5 格子型フィルターによる信号合成

PARCOR 係数の値域

式 1.21, 1.22 を使って再度誤差最小化を考えると、PARCOR 係数の値域は $-1 \leq \gamma_m \leq 1$ ($m = 0, \dots, M$) であることが示せる。前向き誤差 $f_m(n)$ の全ての時刻における平均 E を

$$E[f_m(n)] = \lim_{N \rightarrow \infty} \frac{1}{2N} \sum_{n=-N}^N f_m(n)$$

で定義すると、前向き誤差の二乗平均 $E[f_m^2(n)]$ は次のように展開できる。

$$\begin{aligned} E[f_m^2(n)] &= E[(f_{m-1}(n) - \gamma_m b_{m-1}(n-1))^2] \\ &= E[f_{m-1}^2(n) - 2\gamma_m f_{m-1}(n)b_{m-1}(n-1) + \gamma_m^2 b_{m-1}^2(n-1)] \\ &= E[f_{m-1}^2(n)] - 2\gamma_m E[f_{m-1}(n)b_{m-1}(n-1)] + \gamma_m^2 E[b_{m-1}^2(n-1)] \end{aligned}$$

ここで、最後の式変形において平均 E の線形性を用いている。また、平均 E は全ての時刻 n についての平均だから、時刻をシフトして平均をとっても結果は変わらない。従って、 $E[b_{m-1}^2(n-1)] = E[b_{m-1}^2(n)]$ であること^{*9}により、次の結果を得る。

$$E[f_m^2(n)] = E[f_{m-1}^2(n)] - 2\gamma_m E[f_{m-1}(n)b_{m-1}(n-1)] + \gamma_m^2 E[b_{m-1}^2(n)] \quad (1.24)$$

$E[f_m^2(n)]$ を γ_m について偏微分すると、

$$\frac{\partial E[f_m^2(n)]}{\partial \gamma_m} = -2E[f_{m-1}(n)b_{m-1}(n-1)] + 2\gamma_m E[b_{m-1}^2(n)]$$

^{*9} $E[f_{m-1}(n)b_{m-1}(n-1)] \neq E[f_{m-1}(n)b_{m-1}(n)]$ には注意。E 内の時刻を全て同時にシフトしないと等号は成立しない。例えば、 $E[f_{m-1}(n)b_{m-1}(n-1)] = E[f_{m-1}(n+1)b_{m-1}(n)]$ は成立する。

偏微分の結果を 0 とおくことで、PARCOR 係数は次の式 1.25 でも求まることが分かる。

$$\gamma_m = \frac{E[f_{m-1}(n)b_{m-1}(n-1)]}{E[b_{m-1}^2(n)]} \quad (1.25)$$

更に、後ろ向き誤差 $b_m(n)$ についても、二乗平均 $E[b_m^2(n)]$ を γ_m について最小化することで、PARCOR 係数は次の式 1.26 でも求まることが分かる。

$$\gamma_m = \frac{E[f_{m-1}(n)b_{m-1}(n-1)]}{E[f_{m-1}^2(n)]} \quad (1.26)$$

再度式 1.24 に注目する。この式 1.24 を γ_m に関して平方完成すると、

$$\begin{aligned} E[f_m^2(n)] &= E[f_{m-1}^2(n)] - 2\gamma_m E[f_{m-1}(n)b_{m-1}(n-1)] + \gamma_m^2 E[b_{m-1}^2(n)] \\ &= E[b_{m-1}^2(n)] \left(\gamma_m^2 - 2\gamma_m \frac{E[f_{m-1}(n)b_{m-1}(n-1)]}{E[b_{m-1}^2(n)]} \right) + E[f_{m-1}^2(n)] \\ &= E[b_{m-1}^2(n)] \left(\gamma_m - \frac{E[f_{m-1}(n)b_{m-1}(n-1)]}{E[b_{m-1}^2(n)]} \right)^2 \\ &\quad - \frac{\{E[f_{m-1}(n)b_{m-1}(n-1)]\}^2}{E[b_{m-1}^2(n)]} + E[f_{m-1}^2(n)] \\ &= -\frac{\{E[f_{m-1}(n)b_{m-1}(n-1)]\}^2}{E[b_{m-1}^2(n)]} + E[f_{m-1}^2(n)] \quad (\because \text{式 1.25}) \end{aligned}$$

ここで、二乗平均は必ず非負 $E[f_m^2(n)] \geq 0$ だから、上式は必ず非負でなければならぬ。非負条件を変形すると、

$$\begin{aligned} &-\frac{\{E[f_{m-1}(n)b_{m-1}(n-1)]\}^2}{E[b_{m-1}^2(n)]} + E[f_{m-1}^2(n)] \geq 0 \\ \iff &\frac{\{E[f_{m-1}(n)b_{m-1}(n-1)]\}^2}{E[b_{m-1}^2(n)]} \leq E[f_{m-1}^2(n)] \\ \iff &\frac{\{E[f_{m-1}(n)b_{m-1}(n-1)]\}^2}{E[b_{m-1}^2(n)] E[f_{m-1}^2(n)]} \leq 1 \\ \iff &\gamma_m^2 \leq 1 \quad (\because \text{式 1.25, 1.26}) \end{aligned}$$

従って、PARCOR 係数 γ_m の値域が $-1 \leq \gamma_m \leq 1$ であることが示された。値域が制限されている性質は、符号化を行うにあたって都合が良い。一方、線形予測係数 $a_M(0), \dots, a_M(M)$ は値域に制限がなく、符号化の際に係数範囲についての追加情報を必要とする。

1.1.3 線形予測の実装

Levinson-Durbin 法の C 言語による実装をリスト 1.1 に示す。

リスト 1.1 Levinson-Durbin 再帰法の実装例

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <float.h>
6 #include <stdint.h>
7
8 /* (標本)自己相関の計算 */
9 static int32_t
10 calc_auto_correlation(double *auto_corr, const double *data, uint32_t
    num_sample, uint32_t max_order);
11
12 /* Levinson-Durbin再帰計算 */
13 static int32_t
14 levinson_durbin_recursion(double *lpc_coef, const double *auto_corr,
    uint32_t max_order);
15
16 int main(void)
17 {
18     int32_t num_sample = 300; /* サンプル数 */
19     int32_t max_delay = 10; /* LPC係数の数 */
20
21     int32_t smpl, i;
22     double *data = (double *)malloc(sizeof(double) * num_sample);
23     double *predict = (double *)malloc(sizeof(double) * num_sample);
24     double *auto_cor = (double *)malloc(sizeof(double) * (max_delay + 1));
25     double *coff = (double *)malloc(sizeof(double) * (max_delay + 1));
26     double error;
27
28     /* 波形の生成 */
29     for (smpl = 0; smpl < num_sample; smpl++) {
30         data[smpl] = sin(smpl * 0.01) + 0.5 * cos(4.0f * sin(smpl * 0.05));
31     }
32
33     /* 自己相関・Levinson-Durbin再帰計算 */
34     calc_auto_correlation(auto_cor, data, num_sample, max_delay + 1);
35     levinson_durbin_recursion(coff, auto_cor, max_delay);
36
37     /* 予測テスト */
38     for (smpl = 0; smpl < num_sample; smpl++) {
39         if (smpl < max_delay) {
40             /* 最初のmax_delayステップ分は元信号を単純コピー */
41             predict[smpl] = data[smpl];
42         } else {
43             /* 以降は予測 */
```

```

44     predict[smpl] = 0.0f;
45     for (i = 1; i <= max_delay; i++) {
46         predict[smpl] -= (coff[i] * data[smpl-i]);
47     }
48 }
49 }
50
51 /* 誤差計算・結果表示 */
52 error = 0.0f;
53 for (smpl = 0; smpl < num_sample; smpl++) {
54     error += pow(predict[smpl] - data[smpl], 2);
55     printf("No:%d Data:%f Predict:%f\n", smpl, data[smpl], predict[
56         smpl]);
57 }
58 printf("Error:%f\n", sqrt(error / num_sample));
59 free(data); free(predict);
60 free(auto_corr); free(coff);
61
62 return 0;
63 }
64
65 static int32_t levinson_durbin_recursion(double *lpc_coef, const double
66     *auto_corr, uint32_t max_order)
67 {
68     int32_t k, i;
69     double lambda;
70     double *u_vec, *v_vec, *a_vec, *e_vec;
71
72     if (lpc_coef == NULL || auto_corr == NULL) {
73         fprintf(stderr, "Data or result pointer point to NULL.\n");
74         return -1;
75     }
76
77     /*
78      * 0次自己相関(信号の二乗和)が0に近い場合、入力信号は無音と判定
79      * => 予測誤差、LPC係数は全て0として無音出力システムを予測。
80      */
81     if (fabs(auto_corr[0]) < FLT_EPSILON) {
82         for (i = 0; i < max_order + 1; i++) {
83             lpc_coef[i] = 0.0f;
84         }
85         return 0;
86     }
87
88     /* 初期化 */
89     a_vec = (double *)malloc(sizeof(double) * (max_order + 2)); /*
90         a_0, a_k+1を含めるとmax_order+2 */
91     e_vec = (double *)malloc(sizeof(double) * (max_order + 2)); /*
92         e_0, e_k+1を含めるとmax_order+2 */
93     u_vec = (double *)malloc(sizeof(double) * (max_order + 2));

```

```
91  v_vec = (double *)malloc(sizeof(double) * (max_order + 2));
92  for (i = 0; i < max_order + 2; i++) {
93      u_vec[i] = v_vec[i] = a_vec[i] = 0.0f;
94  }
95
96  /* 最初のステップの係数をセット */
97  a_vec[0] = 1.0f;
98  e_vec[0] = auto_corr[0];
99  a_vec[1] = - auto_corr[1] / auto_corr[0];
100 e_vec[1] = auto_corr[0] + auto_corr[1] * a_vec[1];
101 u_vec[0] = 1.0f; u_vec[1] = 0.0f;
102 v_vec[0] = 0.0f; v_vec[1] = 1.0f;
103
104 /* 再帰処理 */
105 for (k = 1; k < max_order; k++) {
106     lambda = 0.0f;
107     for (i = 0; i < k + 1; i++) {
108         lambda += a_vec[i] * auto_corr[k + 1 - i];
109     }
110     lambda /= (-e_vec[k]);
111     e_vec[k + 1] = (1 - lambda * lambda) * e_vec[k];
112
113     /* u_vec, v_vecの更新 */
114     for (i = 0; i < k; i++) {
115         u_vec[i + 1] = v_vec[k - i] = a_vec[i + 1];
116     }
117     u_vec[0] = 1.0f; u_vec[k + 1] = 0.0f;
118     v_vec[0] = 0.0f; v_vec[k + 1] = 1.0f;
119
120     /* resultの更新 */
121     for (i = 0; i < k+2; i++) {
122         a_vec[i] = u_vec[i] + lambda * v_vec[i];
123     }
124 }
125
126 /* 結果の取得 */
127 memcpy(lpc_coef, a_vec, sizeof(double) * (max_order + 1));
128
129 free(u_vec); free(v_vec);
130 free(a_vec); free(e_vec);
131
132 return 0;
133 }
134
135 static int32_t calc_auto_correlation(double *auto_corr, const double *
136 data, uint32_t num_sample, uint32_t max_order)
137 {
138     int32_t smpl, i;
139
140     if (max_order > num_sample) {
fprintf(stderr, "Max_order(%d) is larger than number of samples(%d)."
```

```

141     \n", max_order, num_sample);
142     return -1;
143 }
144 if (auto_corr == NULL || data == NULL) {
145     fprintf(stderr, "Data or result pointer point to NULL.\n");
146     return -2;
147 }
148 /* (標本)自己相関の計算 */
149 for (i = 0; i < max_order; i++) {
150     auto_corr[i] = 0.0f;
151     for (smpl = i; smpl < num_sample; smpl++) {
152         auto_corr[i] += data[smpl] * data[smpl - i];
153     }
154 }
155 }
156 return 0;
158 }
```

実験

実際にリスト 1.1 を実行し、元の信号と予測した信号をプロットしたグラフを図 1.6 に示す。

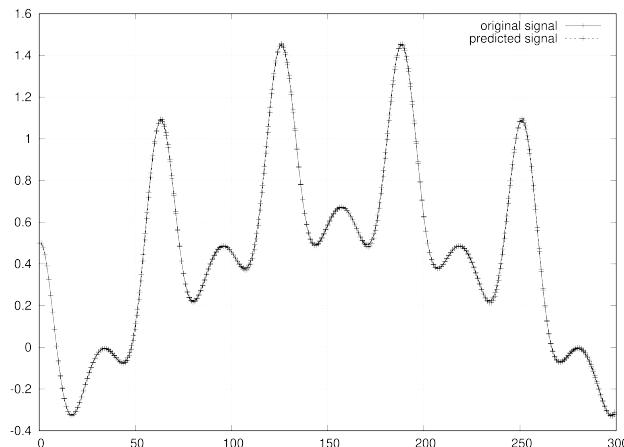


図 1.6 リスト 1.1 の実行結果

原信号が簡単すぎたのか、元の信号と予測した信号はほぼ重なっており、係数は少なめでも十分に予測できている事が分かる。

1.2 エントロピー符号

データ圧縮においては、圧縮後のサイズを小さくするために、いかに情報を短く符号化できるかが重要になる。短い符号化を達成するためには、原理的には情報をくまなく観察し、頻繁に登場する情報のパターンに対して短い符号を割り当ててやれば良い。

例えば、英語のアルファベットの中では'e'が最も出現確率が高いという統計 [18] がある。このパターンを利用して、'e'に短い符号を割り当てば英単語を効率的に圧縮することができる。この様に、情報の中に潜んでいるパターンの確率を基に符号化を行う手法をエントロピー符号化 (entropy coding) という。

本節では、[2, 15] を参考に、情報とその定量的基準の定義と、ロスレス音声コーデックでしばしば使われるエントロピー符号の例を見ていく。

1.2.1 情報とその符号化

我々の住む世界には様々な情報 (information) が溢れている。国語辞典 [14] を紐解くと、情報とは「事件や物事の内容や様子についての知らせ」と定義されている。例えば、人間同士の会話は事件や物事の知らせになっているので情報であるし、今読んでいるこの文章も何らかの技術を知らせるものであるから、情報の一つである。

情報理論では、情報は情報を発する源、つまり情報源 (information source) から生起（発生）するものと捉えている。また、生起した情報の時間的な系列は、過去に生起した情報に（確率的に）依存して生起すると考えられる。例えば、良くシャッフルされたトランプの山札からカード一枚ずつ引いていく時、引いたカードに応じて残りのカードを引く確率が変動し、これから引くカード、即ち情報の生起が定まってくる。

符号化 (coding, encoding) (エンコード) とは、情報を一定の規則に従って変換すること（言い換える）ことを指す。また、符号化の変換規則を符号 (code) という。例として、4つのアルファベット A, B, C, D を 0, 1 の列に割当てる符号を次のように定義する。

$$\begin{aligned} A &\rightarrow 00 \\ B &\rightarrow 01 \\ C &\rightarrow 10 \\ D &\rightarrow 11 \end{aligned}$$

この符号を用いてアルファベットの列 $AABD$ を符号化すると、

$$AABD \rightarrow 00\ 00\ 01\ 11$$

という列が得られる。符号化したままでは元の情報の内容は分からないので、符号の逆の

対応規則

$$\begin{aligned} 00 &\rightarrow A \\ 01 &\rightarrow B \\ 10 &\rightarrow C \\ 11 &\rightarrow D \end{aligned}$$

によって列 00 00 01 11 を戻すと、

$$00\ 00\ 01\ 11 \rightarrow AABD$$

元の情報に戻ることが分かる。この様に、符号化された情報を元の情報に戻す操作を**復号 (decoding) (デコード)** という。

一意に復号可能な符号

情報源を符号化するにあたって、符号が曖昧さを残さずに復号できる性質は実用上非常に重要である。何故なら、符号化した情報を受け取ったときに、それを完全に元の情報に復号できなければ、情報を正しく伝達できたとは言えないからである。本節では、曖昧さを残さずに復号できる符号の性質を見ていく。

議論に立ち入る前に、符号についての用語定義を行っておく。まず、符号語の数が r 個ある符号を r 元符号という。例えば、0, 1 から構成される符号は $r = 2$ で 2 元符号である^{*10}。また、符号の列が与えられた時、それを一意に（ただ一通りの方法で）復号できる符号を**一意に復号可能な符号 (uniquely decodable code)** という。一意に復号可能な符号とそうでない（一意に復号不可能な）2 元符号の例を表 1.3 に挙げる。

表 1.3 2 元符号の例

記号	符号 1	符号 2	符号 3
A	00	0	0
B	01	1	01
C	10	0	10
D	11	1	11

表 1.3 に挙げた符号 1, 2, 3 について、一意に復号可能かどうかを観察すると、

符号 1 一意に復号可能な符号である。全ての符号の列は、2 文字ずつ区切れば、 A, B, C, D のいずれかに重複なく対応付けることができる。

^{*10} 2 元符号のことを特にバイナリコード (binary code) と呼ぶこともある。

符号 2 一意に復号不可能な符号である。0だけを見た時、それが元の記号が A だったのか、 C だったのかを区別できない。同様に、1だけを見たときも B, D で区別できない。

符号 3 一意に復号不可能な符号である。符号の列 ”0110” を見た時、

$$\begin{aligned} 0110 &\rightarrow ADA \\ 0110 &\rightarrow BC \end{aligned}$$

という様に、2通りに復号できてしまう。

McMillan (マクミラン) の不等式

一意に復号可能な符号が持つ性質を特徴付ける定理として、次のマクミランの不等式 (McMillan inequality) がある。

定理 1 (マクミランの不等式). r 元符号が一意に復号可能ならば、記号の数 q に対応する符号の長さの組 l_1, l_2, \dots, l_q は、次の不等式を満たす。

$$\sum_{i=1}^q r^{-l_i} \leq 1 \quad (1.27)$$

Proof. 不等式の右辺は何乗しても 1 のままであることを使い、任意の自然数 n に対して $(\sum_{i=1}^q r^{-l_i})^n \leq 1$ を示す。左辺の $(\sum_{i=1}^q r^{-l_i})^n$ を展開すると、各項は $r^{-l_1}, r^{-l_2}, \dots, r^{-l_q}$ から n 回選択して乗じた項の和だから、 r^{-l_j} を選択したことを i_j と表すと、 n 回の乗算の選択を以下の式のように表せる。

$$r^{-l_{i_1}} \times r^{-l_{i_2}} \cdots \times r^{-l_{i_n}}$$

今、 $r^{-l_{i_1}}, r^{-l_{i_2}}, \dots, r^{-l_{i_n}}$ を選択したときの符号列の長さを $j = l_{i_1} + l_{i_2} + \cdots + l_{i_n}$ とおけば、

$$r^{-j} = r^{-l_{i_1}} \times r^{-l_{i_2}} \cdots \times r^{-l_{i_n}}$$

と表せる。こうした上で、 l_{\min}, l_{\max} をそれぞれ l_1, \dots, l_q の最小値と最大値とすると、 $l_{\min} \leq l_{i_1}, l_{i_2}, \dots, l_{i_n} \leq l_{\max}$ だから、 $nl_{\min} \leq j \leq nl_{\max}$ が成立する。これにより、

$$\left(\sum_{i=1}^q r^{-l_i} \right)^n = \sum_{j=nl_{\min}}^{nl_{\max}} w_j r^{-j} \quad (1.28)$$

と表せる。ここで、 w_j は長さ n の記号を符号化したときに、長さが j になるような符号の組み合わせ数である。構成した符号によっては、符号化したときの長さとして実現でき

ない長さ j が存在するが、そのような長さ j については $w_j = 0$ とおく^{*11}。更に w_j について注目する。長さ j の符号の列について、 r^j は長さ j の r 元符号が表現できる最大の記号数だから、不等式

$$w_j \leq r^j \quad (1.29)$$

が成立する。逆に $w_j > r^j$ と仮定すると、長さ j の符号列の組み合わせに重複が発生してしまうから、一意に復号可能な符号の仮定に反する。^{*12}この不等式 1.29 を式 1.28 に代入すると、

$$\begin{aligned} \left(\sum_{i=1}^q r^{-l_i} \right)^n &= \sum_{j=nl_{\min}}^{nl_{\max}} w_j r^{-j} \\ &\leq \sum_{j=nl_{\min}}^{nl_{\max}} r^j r^{-j} = \sum_{j=nl_{\min}}^{nl_{\max}} 1 = nl_{\max} - nl_{\min} + 1 \\ &\leq nl_{\max} \end{aligned}$$

が導かれる。 $(\sum_{i=1}^q r^{-l_i})^n > 1$ のとき、 n を大きくしていくと左辺 $(\sum_{i=1}^q r^{-l_i})^n$ は指数的に増加するのに対し、右辺 nl_{\max} は線形にしか増加していないので、不等式は成立しなくなる。従って、任意の自然数 n に対して不等式が成立するためには、

$$\left(\sum_{i=1}^q r^{-l_i} \right)^n \leq 1$$

でなければならない。この不等式を成り立たせる条件として、マクミランの不等式

$$\sum_{i=1}^q r^{-l_i} \leq 1$$

が得られる。 □

^{*11} w_j は多項係数から求められる。長さ l_i の符号を選んだ回数を $k_i \geq 0$ と書くと、

$$w_j = \frac{n!}{k_1! k_2! \dots k_q!}$$

となる。ここで、 $j = k_1 l_1 + k_2 l_2 + \dots + k_q l_q$ が成り立っている。 $j = k_1 l_1 + k_2 l_2 + \dots + k_q l_q$ を成り立てる $k_i (i = 1, \dots, q)$ の組み合わせが存在しない場合は、その長さ j を持つ符号列は出現しないから $w_j = 0$ となる。

^{*12} 例えば、2 元符号 0, 1 からなる長さ 1 の符号を考えてみる。この場合は $r = 2, j = 1$ で $r^j = 2^1 = 2$ となる。一方 w_1 は定義により長さ 1 の符号列の組み合わせを意味している。もし $w_1 = 3$ ならば、これは、長さ 1 の符号列が 3 通り存在することになる。しかし、長さ 1 の 2 元符号では、0, 1 の 2 通りしか符号の列が存在しない。3 通り以上符号の列を考えると、長さ 1 において 0, 1 のいずれかを使わざるを得ない。即ち、必ず重複が生じてしまい、その符号は一意に復号不可能になる。従って、2 元符号が一意に復号可能であるためには $w_1 \leq 2$ でなくてはならない。

定理 1 の命題の対偶「 $\sum_{i=1}^q r^{-l_i} > 1$ ならば、符号は一意に復号不可能」を用いることで、構成した符号が一意に復号不可能な符号なのか判定できる^{*13}。表 1.3 で挙げた符号について、一意に復号不可能な符号を正しく判定できているか判定してみると、次の結果が得られる。

符号 2 $r = 2, l_i = 1 (i = 1, 2, 3, 4)$ だから、

$$\sum_{i=1}^q r^{-l_i} = \sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-1} + 2^{-1} + 2^{-1} = 2 > 1$$

よって一意に復号不可能。

符号 3 $r = 2, l_1 = 1, l_2 = 2, l_3 = 2, l_4 = 2$ だから、

$$\sum_{i=1}^q r^{-l_i} = \sum_{i=1}^4 2^{-l_i} = 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} = \frac{5}{4} > 1$$

よって一意に復号不可能。

簡単な例において、マクミランの不等式が成立していることが確かめられる。

1.2.2 情報量とエントロピー

1.2.1 節で触れた情報を定量的に扱う尺度として**情報量**がある。情報量は、特定の情報の知らせを得ることで、不確実性がどの程度減少したかという尺度に基づいて測ることができる。不確実性を定性的に扱うには**確率**を使用する。本節では、情報量と、不確実性を定量的に扱う概念である**エントロピー**について見ていく。

情報量の定義

最初に、情報量の形式的な定義を与える。

定義 2 (情報量). 事象 A が生起したことを知ったときに得られる情報量を $I(A)$ を書くと、情報量は式 1.30 で定義される。

$$I(A) = -\log_2 P(A) \quad [bit] \tag{1.30}$$

ここで、 $P(A)$ は事象 A の生起する確率である。

なぜ情報量がこの式 1.30 の形で与えられるのかは後述する。まずは、情報量の計算例をいくつか挙げることで、情報量が直感的に妥当な尺度になっていることを観察する。

^{*13} 定理 1 の命題の逆、即ち「 $\sum_{i=1}^q r^{-l_i} \leq 1$ ならば、符号は一意に復号可能」は、**クラフトの不等式 (Kraft's inequality)** から導き出すことができる。この結果により、実は $\sum_{i=1}^q r^{-l_i} \leq 1$ が一意に復号可能な符号の必要十分条件であることが示せる。本稿では証明しない。

例 3 (コイン投げ). イカサマの無いコインを投げる事を考える。コイン投げによって得られる結果は、「コインの表が出る」か「コインの裏が出る」の 2通りしかありえない^{*14}。コインの表が出る確率を $P(\text{表})$ 、裏が出る確率を $P(\text{裏})$ と書くとすると、 $P(\text{表}) = P(\text{裏}) = 1/2$ である。従って、コインの表が出たことを知ったときに得られる情報量 $I(\text{表})$ は次のように計算できる：

$$I(\text{表}) = -\log_2 P(\text{表}) = -\log_2 \frac{1}{2} = \log_2 2 = 1$$

即ち、コインを投げてその結果を確認することで 1[bit] の情報が得られる。例えば、コインの表を 1、裏を 0 に対応付けることで、確かに 1bit でコインの表裏の情報をもれなく保存できるから、コイン投げによる情報量が 1bit となるのはもっともある。

例 4 (2 枚のコイン投げ). 2 枚のコインを同時に投げる事を考える。 $P(1 \text{ 枚目が表}) = P(2 \text{ 枚目が表}) = 1/2$ であり、各々のコインの表と裏が出る確率は、各々のコイン投げの結果に影響を及ぼさない、つまり独立だから、次の式が成り立つ。

$$P(1 \text{ 枚目が表かつ } 2 \text{ 枚目が表}) = P(1 \text{ 枚目が表})P(2 \text{ 枚目が表}) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$$

このように確率計算が分解できることを踏まえた上で、2 枚のコインを投げ、2 枚とも表だと知ったときに得られる情報量を計算すると、

$$\begin{aligned} I(1 \text{ 枚目が表かつ } 2 \text{ 枚目が表}) &= -\log_2 P(1 \text{ 枚目が表かつ } 2 \text{ 枚目が表}) \\ &= -\log_2 P(1 \text{ 枚目が表})P(2 \text{ 枚目が表}) \\ &= -\log_2 P(1 \text{ 枚目が表}) - \log_2 P(2 \text{ 枚目が表}) \\ &= I(1 \text{ 枚目が表}) + I(2 \text{ 枚目が表}) \quad (1.31) \\ &= 2 \end{aligned}$$

よって、2 枚のコインを同時に投げて、その結果を確認することで 2[bit] の情報が得られる。また、式 1.31 に注目すると、全体の情報量が、各々のコインの表裏を確認するときに得られる情報量の和に分解されていることが分かる。これは、情報量の持つ加法性を示している。

情報量の導出

情報量が式 1.30 の形で与えられるのは、例 4 で触れた情報量の加法性から自然に導かれる。ここでは [2] を参考に導出を行う。 x 通りの事象がすべて等確率で生起する情報源の情報量を $I(x)$ と表す。加法性により、 x, y 通りの事象が互いに独立ならば、

$$I(xy) = I(x) + I(y) \quad (1.32)$$

^{*14} コインが縁で直立することは無いものとする。

が成立する^{*15}。また、1通りの事象しか生起しない場合の情報量は0、即ち、

$$I(1) = 0 \quad (1.33)$$

が成立する。1通りの事象しか生起しない場合、情報の知らせを受けなくとも何が起きるか分かってしまうので、その場合の情報量が0になるのはもっともである。

これらの前提（式1.32, 1.33）の元で、定数 ε を用いて $I(x + \varepsilon x)$ を計算すると、

$$\begin{aligned} I(x + \varepsilon x) &= I((1 + \varepsilon)x) \\ &= I(1 + \varepsilon) + I(x) \\ \Leftrightarrow I(x + \varepsilon x) - I(x) &= I(1 + \varepsilon) \end{aligned} \quad (1.34)$$

式1.34の両辺を εx で割ると、

$$\frac{I(x + \varepsilon x) - I(x)}{\varepsilon x} = \frac{1}{x} \frac{I(1 + \varepsilon)}{\varepsilon} \quad (1.35)$$

なる関係式が得られる。ここで、 $\varepsilon \rightarrow 0$ の極限をとると、式1.35の左辺は $I(x)$ の導関数 $I'(x)$ そのものとなる。右辺の $I(1 + \varepsilon)/\varepsilon$ は、分子分母共に0に近づいていくが、その極限値を c とおく：

$$\lim_{\varepsilon \rightarrow 0} \frac{I(1 + \varepsilon)}{\varepsilon} = c$$

$I'(x)$ と c を用いて式1.35を書き直すと、

$$I'(x) = \frac{c}{x}$$

となる。更に、両辺の不定積分を計算すると、

$$\begin{aligned} \int I'(x) dx &= \int \frac{c}{x} dx \\ \Rightarrow I(x) &= c \log x + d \quad (d: 積分定数) \end{aligned}$$

となり、対数 \log が現れる。次に、 c, d が不定なので決定することを考える。まず、前提 $I(1) = 0$ （式1.33）より、

$$I(1) = c \log 1 + d = 0 \implies d = 0$$

また、コイン投げの例で観察したように、2つの情報が等確率で生起する時を情報量の基本単位とし、 $I(2) = 1[\text{bit}]$ と定めると、

$$I(2) = c \log 2 = 1 \implies c = \frac{1}{\log 2}$$

*15 例4で挙げたコイン投げの例では、2枚のコインを投げる事象は全4通り、1枚のコインを投げる事象は全2通りだから、 $I(4) = I(2) + I(2)$ となっている。

従って、 x 個からなる事象の情報量 $I(x)$ は次の式で表せる:

$$I(x) = \frac{\log x}{\log 2} = \log_2 x \quad (1.36)$$

次に、式 1.36 を用いることで、一般の情報量の定義（式 1.30）が得られることを見ていいく。式 1.36 により、 n 個の事象が等確率に生起する情報源から得られる情報量は $I(n) = \log_2 n$ である。一方、 n 個の事象の中から k 個をまとめて B とおくと、 B の部分が持つ情報量は $I(k) = \log_2 k$ となる。

B の中から何かの事象が生起したと分かったとしても、 n 個全体の中では何が生起したのかは確定しない。何が生起したのかを確定させるためには、 $I(k)$ に加え B の中から何が生起したのかを確定させる情報量 $I(B)$ が必要である。このことを式に表すと、

$$\begin{aligned} I(k) + I(B) &= I(n) \\ \log_2 k + I(B) &= \log_2 n \\ \implies I(B) &= \log_2 n - \log_2 k = -\log_2 \frac{k}{n} \end{aligned}$$

上式において k/n は B が生起する確率、即ち $P(B)$ に他ならない。従って、

$$I(B) = -\log_2 P(B)$$

となって、情報量の定義式 1.30 が情報量の加法性から自然に導かれることが確かめられた。

エントロピー

エントロピー (entropy) という言葉は熱力学から来ており、力学系の不確定度（無秩序）の度合いを表す量のことを指す。情報理論においてエントロピーは、情報量の期待値として定義される。

定義 5 (エントロピー). n 個の事象 A_1, A_2, \dots, A_n が存在する時、エントロピー $H(A)$ は式 1.37 で定義される。

$$H(A) = - \sum_{i=1}^n P(A_i) \log_2 P(A_i) \quad (1.37)$$

ここで $P(A_i)$ は A_i の生起する確率である。

後述するが、エントロピーは符号の長さに関する非常に重要な性質を持つ。また定義上、エントロピーは期待値であることを強調しておきたい。情報量は情報の知らせを受けてから確定する量である。一方、エントロピーは、情報の知らせを受ける前にどれほどの情報量が得られるかを推定する量となる。

例 6 (サイコロ投げ). イカサマがなく、全ての面が等確率で生起する六面サイコロを振る時、そのエントロピー H は次のように計算できる：

$$\begin{aligned} H &= - \sum_{i=1}^6 P(\text{サイコロの出目が } i) \log_2 P(\text{サイコロの出目が } i) \\ &= - \sum_{i=1}^6 \frac{1}{6} \log_2 \frac{1}{6} = -\log_2 \frac{1}{6} = \log_2 6 \end{aligned}$$

例 6 では全ての事象が等確率で生起する例を見た。この「全て等確率」という状況は確率に偏りが無く、最も予測のしづらい、言い換えると不確定度が最も高い状況であると言える。その性質を正確に表現した、エントロピーの重要な定理が存在する。

定理 7. n 個の事象を持つ情報源のエントロピーの最大値 H は、

$$H = \log_2 n \quad [\text{bit}] \quad (1.38)$$

で、それは、全ての事象が等確率 $\frac{1}{n}$ で生起するときのエントロピーである。

Proof. n 個の全ての事象が $1/n$ の確率で生起するならば、エントロピー H は定義により、

$$H = - \sum_{i=1}^n \frac{1}{n} \log_2 \frac{1}{n} = \log_2 n$$

と計算できる。この値が最大値であることを示すには、 $P_i \geq 0$, $\sum_{i=1}^n P_i = 1$ となるような任意の確率分布 P_1, \dots, P_n からなる情報源のエントロピーよりも $\log_2 n$ の方が大きいこと、即ち、

$$-\sum_{i=1}^n P_i \log_2 P_i \leq \log_2 n$$

が成立することを示せば良い。上式の不等式において、左辺から右辺を減じると、

$$\begin{aligned} -\sum_{i=1}^n P_i \log_2 P_i - \log_2 n &= \sum_{i=1}^n P_i \log_2 \frac{1}{P_i} + \log_2 \frac{1}{n} \\ &= \sum_{i=1}^n P_i \log_2 \frac{1}{P_i} + \sum_{i=1}^n P_i \log_2 \frac{1}{n} \\ &= \sum_{i=1}^n P_i \log_2 \frac{1}{nP_i} \\ &= \frac{1}{\log 2} \sum_{i=1}^n P_i \log \frac{1}{nP_i} \end{aligned}$$

ここで、対数関数に関する次の不等式を適用する。

$$\log x \leq x - 1 \quad (1.39)$$

この不等式の成立は、図 1.7 に示したグラフからも明らかである^{*16}。不等式を適用す

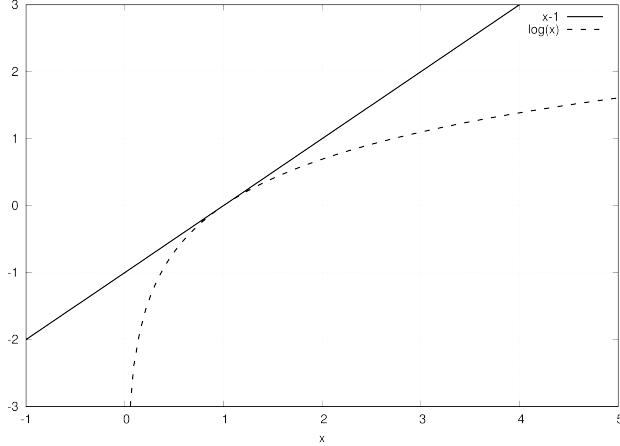


図 1.7 $x - 1$ と $\log x$ のグラフ

ると、

$$\begin{aligned} -\sum_{i=1}^n P_i \log_2 P_i - \log_2 n &= \frac{1}{\log 2} \sum_{i=1}^n P_i \log \frac{1}{nP_i} \\ &\leq \frac{1}{\log 2} \sum_{i=1}^n P_i \left(\frac{1}{nP_i} - 1 \right) = \frac{1}{\log 2} \sum_{i=1}^n \left(\frac{1}{n} - P_i \right) \\ &= \frac{1}{\log 2} \left(\sum_{i=1}^n \frac{1}{n} - \sum_{i=1}^n P_i \right) = 0 \end{aligned}$$

よって、

$$-\sum_{i=1}^n P_i \log_2 P_i - \log_2 n \leq 0 \iff -\sum_{i=1}^n P_i \log_2 P_i \leq \log_2 n$$

が成立するため、定理の成立が示された。 \square

定理 7 から、ある情報を記録するために必要な bit 数を計算することができる。例を挙げると、

例 8. 256 個の事象が全て等確率で生起すると考えると、エントロピー H は、

$$H = -\sum_{i=1}^{256} \frac{1}{256} \log_2 \frac{1}{256} = \log_2 256 = 8[\text{bit}]$$

^{*16} 念の為解析的に証明する。 $f(x) = x - 1 - \log x$ を微分すると、 $f'(x) = 1 - \frac{1}{x}$ となる。 $f'(x) = 0$ を x について解くと $x = 1$ が得られ、 $f(x)$ は $x = 1$ で極点を持つ。 $f(1) = 0$ であり、 $x < 1$ では $f(x) > 0$ かつ $f'(x) < 0$ 、 $x > 1$ では $f'(x) > 0$ であることから、 $f(x)$ は $x = 1$ において最小値 0 を取ることが分かる。即ち、 $f(x) = x - 1 - \log x \geq 0 \iff x - 1 \geq \log x$ が成立する。

と計算できる。256 個の事象が存在するとき、各々の事象の生起確率が未知であっても得られる情報量が 8[bit] を超えることはない。従って、8[bit] さえ用意すれば、256 個の事象が生起する情報源をもれなく記録することができる。

平均符号長とエントロピー

前節までの議論を統合し、符号とエントロピーとが密接に結びついていることを見ていく。我々が勝手に構成した符号について、その良し悪しを定量的に評価できなければ、どれほど良い圧縮を達成できるのか検討することができない。本節では、符号の良さを“平均符号長”によって評価した時、その最小値が実は“エントロピー”によって与えられることを見していく。

早速平均符号長を定義する。 N 通り存在する記号 s_1, s_2, \dots, s_N の各々の生起確率を $P(s_i)$ ($i = 1, \dots, N$)、また、各々の記号に対応する符号の長さを l_i ($i = 1, \dots, N$) と表す。この時、平均符号長 L は以下の式 1.40 に示すように、**符号の長さの期待値**として定義される。

$$L = \sum_{i=1}^N l_i P(s_i) \quad (1.40)$$

圧縮効率の良し悪しを探るためには、 L をなるべく小さくできれば良いのは明らかである。それでは、符号の長さ l_1, \dots, l_N を上手く選ぶことで、一意に復号可能な符号の平均符号長はどこまでも小さくできるのだろうか。実は、驚くべきことに、**平均符号長はエントロピーよりも小さくならない**。このことを定理として書くと、次のようになる。

定理 9 (平均符号長とエントロピー). 無記憶情報源 $S = \{s_1, s_2, \dots, s_N\}$ のエントロピーを $H(S)$ と表す。一意に復号可能な 2 元符号の平均符号長を L と表すと、以下の不等式が成立する。

$$L \geq H(S) \quad (1.41)$$

ここで、**無記憶情報源 (memoryless source)** とは、各記号 s_i ($i = 1, \dots, N$) が確率的に独立に生起する情報源のことである^{*17}。

Proof. 定理の証明に入る前に、補助定理として、2 つの確率分布 p_i, q_i ($i = 1, \dots, N$) に

*17 コイン投げやサイコロ投げは、その結果は以前に出た結果に依存せず、無記憶情報源となる。一方で、我々人間が書く文章は文脈、即ちそれまでに書いている背景に影響を受ける。人間の書く文章を情報源として考えると、それは必ずしも無記憶情報源にはならない。仮に無記憶情報源だとすると、それは常に一定の確率に従ってキーボードをタイプすることと同様（無限の猿定理を参照）であり、その結果できあがった文章は文法を無視した意味不明な文章になるのは想像に難くない。情報理論は、情報の意味や論理的側面は捨てる一方、純粹な確率・統計論的な性質を利用して議論を展開する [2]。

対して、

$$-\sum_{i=1}^N p_i \log_2 p_i \leq -\sum_{i=1}^N p_i \log_2 q_i \quad (1.42)$$

が成立することを示す。不等式の左辺から右辺を減じると、

$$\begin{aligned} (\text{左辺}) - (\text{右辺}) &= -\sum_{i=1}^N p_i \log_2 p_i + \sum_{i=1}^N p_i \log_2 q_i \\ &= \sum_{i=1}^N \log_2 \frac{q_i}{p_i} = \frac{1}{\log 2} \sum_{i=1}^N p_i \log \frac{q_i}{p_i} \\ &\leq \frac{1}{\log 2} \sum_{i=1}^N p_i \left(\frac{q_i}{p_i} - 1 \right) \quad (\because \text{式 1.39 を適用}) \\ &= \frac{1}{\log 2} \sum_{i=1}^N (q_i - p_i) = \frac{1}{\log 2} \left(\sum_{i=1}^N q_i - \sum_{i=1}^N p_i \right) \\ &= 0 \end{aligned}$$

よって、(左辺) \leq (右辺) が成り立つことから式 1.42 が成立していることが示された。

それでは、補助定理を使用して定理の証明に入る。一意に復号可能な符号を使用しているから、2元符号に対するマクミランの不等式

$$\sum_{i=1}^N 2^{-l_i} \leq 1$$

が成り立つ。ここで、不等式左辺を $Q = \sum_{i=1}^N 2^{-l_i}$ とおく。その上で、

$$Q_i = \frac{2^{-l_i}}{Q} \quad (i = 1, \dots, N)$$

なる量をおくと、 $Q_i \geq 0$ ($i = 1, \dots, N$)、 $\sum_{i=1}^N Q_i = \frac{1}{Q} \sum_{i=1}^N 2^{-l_i} = \frac{Q}{Q} = 1$ により、 Q_i は確率分布をなす。この時式 1.42 を適用すると、無記憶情報源のエントロピー $H(S)$ から以下の不等式を導出できる。

$$\begin{aligned} H(S) &= -\sum_{i=1}^N P(s_i) \log_2 P(s_i) \leq -\sum_{i=1}^N P(s_i) \log_2 Q_i \quad (\because \text{補助定理の式 1.42 を使用}) \\ &= -\sum_{i=1}^N P(s_i) \log_2 \frac{2^{-l_i}}{Q} = -\sum_{i=1}^N P(s_i) (\log_2 2^{-l_i} - \log_2 Q) \\ &= \sum_{i=1}^N l_i P(s_i) + \log_2 Q \sum_{i=1}^N P(s_i) \\ &= L + \log_2 Q \end{aligned} \quad (1.43)$$

再びマクミランの不等式を適用すると、 $Q = \sum_{i=1}^N 2^{-l_i} \leq 1$ が成立するから、 $\log_2 Q \leq 0$ 。この条件下で常に式 1.43 が成り立っているから、

$$H(S) \leq L$$

が導かれる。 \square

様々なファイルのエントロピー計測

定理 9 を利用すると、エントロピーを計算することで最小の平均符号長が分かるため、符号化によってどれくらい圧縮ができるのかを見積もることができる。コンピュータシステム上の任意のファイルを、無記憶情報源から生起した符号長 1byte = 8bit の 2 元符号とすれば、ファイルの各バイトの bit パターンの生起確率 $P(0x00), P(0x01), \dots, P(0xFF)$ を推定することで、ファイルの持つエントロピーを概算できる。

エントロピー計測のプログラムを掲載する。

リスト 1.2 エントロピーの計測プログラム

```

1 #include <stdio.h>
2 #include <stdint.h>
3 #include <math.h>
4
5 int main(int argc, char **argv)
6 {
7     FILE* fp;
8     int byte;
9     double entropy;
10    uint32_t count_sum; /* 頻度の総和 */
11    uint32_t byte_count[0x100] = { 0, }; /* 各バイトの出現数カウント */
12
13    /* 引数が少ない場合は使用方法を表示 */
14    if (argc < 2) {
15        fprintf(stderr, "Usage: %s filename\n", argv[0]);
16        return 1;
17    }
18
19    /* バイナリモードでファイルオープン */
20    fp = fopen(argv[1], "rb");
21    if (fp == NULL) {
22        fprintf(stderr, "Cannot open %s.\n", argv[1]);
23        return 1;
24    }
25
26    /* 各バイトの出現頻度をカウント */
27    count_sum = 0;
28    while ((byte = fgetc(fp)) != EOF) {
29        byte_count[byte & 0xFF]++;
30        count_sum++;

```

```

31     }
32
33     /* ファイルクローズ */
34     fclose(fp);
35
36     /* エントロピー計算 */
37     entropy = 0.0f;
38     for (byte = 0; byte < 0x100; byte++) {
39         /* 頻度0のパターンについては  $0 * \log_2(0) = 0$  だから加算しない */
40         if (byte_count[byte] > 0) {
41             /* 経験確率の計算 */
42             double prob = (double)byte_count[byte] / count_sum;
43             entropy -= prob * log2(prob);
44         }
45     }
46
47     /* 結果を表示 */
48     printf("entropy: %8.6f [byte]\n", entropy);
49
50     /* 正常終了 */
51     return 0;
52 }
```

実際に [20] にあるファイルに対してエントロピーを計測した結果を表 1.4 に示す。

表 1.4 [20] で挙げられているファイルのエントロピー計測値

ファイル名	ファイルの中身	エントロピー [byte]
ptt5	バイナリ (FAX で送信されているデータ)	1.21
kennedy.xls	バイナリ (EXCEL スプレッドシート)	3.57
alice29.txt	テキスト (英文小説)	4.57
fields.c	テキスト (C ソースコード)	5.01
sum	バイナリ (SPARC 実行可能形式)	5.33

本稿はロスレス音声コーデックに関するものだから、音声ファイル (.wav) のエントロピーを計測した。結果を表 1.5 に示す。

簡単に観察すると、一般に音声ファイルのエントロピーはテキストファイルやバイナリファイルに比べて高く、圧縮しづらいファイルであることが分かる。また、同じ音声ファイルでも、落語音声のような人間のみが喋っている音声よりも、ポップスの様に複数の楽器のとボーカルがミックスされた上音圧が引き上げられている音声の方がエントロピーが高いことが観察できる。

表 1.5 筆者手元の音声ファイル (.wav) のエントロピー

ファイル名	ジャンル	エントロピー [byte]
4-02 井戸の茶椀.wav	落語	6.37
1-01 火焰太鼓.wav	落語	6.67
02 My Song.wav	ピアノ	7.14
09 瑠璃子.wav	ピアノ	7.33
30-自分だけの輝き.wav	インストゥルメンタル	7.37
29-重ねる努力.wav	インストゥルメンタル	7.65
4-02 Let's アイカツ!(Short サイズ).wav	J-POP	7.76
5-02 カレンダーガール (TV-size).wav	J-POP	7.78

1.2.3 エントロピー符号の例

本節では、音声データ圧縮で頻繁に使われているエントロピー符号の例を挙げる。例として挙げる符号は、いずれも整数を対象にした2元符号である。また、これらの符号は整数の値が小さければより短い符号を割り当てる性質を持っている。この性質は、0に近い値が頻繁に登場する音声データの残差符号化に適している。

α, γ 符号とその確率分布

簡単なエントロピー符号の例として、ここでは α 符号 (unary 符号)、 γ 符号の2つを取り上げる。それらの符号の定義を表 1.6 に載せる^{*18}。

正整数 x を符号化するときの各符号のアルゴリズムは次のようになる。

α 符号 $x - 1$ 個分の 0 を出力した後に、1 を 1 回出力する

γ 符号 x を 2 進数で表したときの桁数から 1 を引いた数 ($= \lfloor \log_2 x \rfloor$) だけ 0 を出力した
後に、 x を 2 進数表記した符号を出力

繰り返しになるが、これらの符号は小さい整数値に対してより短い符号を割り当てる傾向がある。その傾向は、これらの符号が持つ確率分布によって見ることができる。[21] を参考に、 α 符号と γ 符号が持つ確率分布を導き出すことを考える。確率分布を考えるために情報量の式 1.30 が使える。情報量を符号の長さと考え、整数 x に対して $l(x)[bit]$ の

^{*18} これらの符号は 0 と 1 が反転しても全く同じ符号として定義されることに注意。例えば、 α の 0 と 1 を反転したとき、整数 1 は 0 に、整数 4 は 1110 に符号化される。

表 1.6 α 符号, γ 符号

整数値	α 符号	γ 符号
1	1	1
2	01	010
3	001	011
4	0001	00100
5	00001	00101
6	000001	00110
7	0000001	00111
8	00000001	0001000
\vdots	\vdots	\vdots
16	略	000010000
17	略	000010001
\vdots	\vdots	\vdots
32	略	00000100000

符号を割り当てたとすると、符号 x の生起確率 $P(x)$ は次のように求めることができる。

$$\begin{aligned} l(x) &= -\log_2 P(x) \\ \iff P(x) &= 2^{-l(x)} \end{aligned} \quad (1.44)$$

式 1.44 に従うと、整数 x に対する α 符号の長さ $l_\alpha(x)$ は $l_\alpha(x) = x$ と表せるから、 α 符号の確率分布 $P_\alpha(x)$ は次のように求まる。

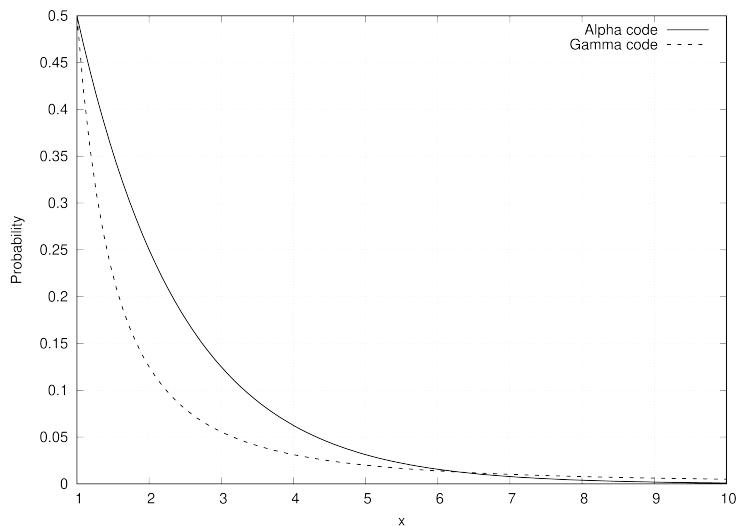
$$P_\alpha(x) = 2^{-l_\alpha(x)} = 2^{-x} \quad (1.45)$$

γ 符号に関する同様の考え方。「 x を 2 進数で表したときの桁数から 1 を引いた数」は長さ $\lfloor \log_2 x \rfloor$ 、残りの「 x を 2 進数表記した符号」は長さ $\lfloor \log_2 x \rfloor + 1$ を持つから、整数 x に対する γ 符号の長さ $l_\gamma(x)$ は $l_\gamma(x) = 2\lfloor \log_2 x \rfloor + 1$ となる。 $l_\gamma(x)$ を用いて、 γ 符号の確率分布 $P_\gamma(x)$ は次のように求められる。

$$\begin{aligned} P_\gamma(x) &= 2^{-l_\gamma(x)} = 2^{-2\lfloor \log_2 x \rfloor - 1} \\ &\approx 2^{-2\log_2 x - 1} = 2^{\log_2 x^{-2} - 1} = 2^{-1} \times 2^{\log_2 x^{-2}} \\ &= \frac{1}{2x^2} \end{aligned} \quad (1.46)$$

α 符号と γ 符号の確率分布を図 1.8 に示す。

これらの確率分布から読み取れる重要な性質は、符号化対象の整数値がこの確率分布に従っている場合、効率の良い圧縮が達成できる点である。図 1.8 に注目すると、 γ 符号は

図 1.8 α 符号と γ 符号の確率分布

α 符号よりも分布の裾野 (x が大きい時の値) が大きく、 α 符号よりも大きい整数 x が出現しやすい記号の符号化に適していることが分かる。この様に、応用の場面では、符号化対象のデータの偏りに合わせて適切な符号を選択することが重要になる。

Golomb (ゴロム) 符号

ロスレス音声コーデックで最もよく使われる符号が **Golomb (ゴロム) 符号** である。Golomb 符号は符号化パラメータ m を持ち、 m により出力する符号が異なる。特に、 m が 2 の幂数 $m = 2, 4, 8, 16, \dots$ のときの Golomb 符号を **Rice (ライス) 符号** と呼ぶ。各種パラメータを設定したときの Golomb 符号の出力符号を表 1.7 に示す。

表 1.7 を見ると、整数値が大きくなるにつれて符号長は長くなるが、パラメータ m によってその符号長の伸びを制御している事が分かる。具体的な符号化アルゴリズムは次のように表せる。

表 1.7 各種符号パラメータ m を使用したときの Golomb 符号

整数値	$m = 2$	$m = 3$	$m = 4$	$m = 5$
0	10	10	100	100
1	11	110	101	101
2	010	111	110	110
3	011	010	111	1110
4	0010	0110	0100	1111
5	0011	0111	0101	0100
6	00010	0010	0110	0101
7	00011	00110	0111	0110
8	000010	00111	00100	01110
9	000011	00010	00101	01111
10	0000010	000110	00110	00100
11	0000011	000111	00111	00101
12	00000010	000010	000100	00110
13	00000011	0000110	000101	001110
14	000000010	0000111	000110	001111
15	000000011	0000010	000111	001100
16	0000000010	00000110	0000100	001101

Golomb 符号化

1. 整数 x を m で割った時の商を $q = \lfloor \frac{x}{m} \rfloor$ 、剰余を r とする。
2. $q + 1$ を α 符号化する。
3. r を以下の手順で符号化する。
 - (a) $\log_2 m$ が整数 ($\iff m$ が 2 の幂数) の場合は、 $\log_2 m$ bit で r の 2 進数表記した符号を出力する。
 - (b) $\log_2 m$ が整数ではない場合は、 $b = \lceil \log_2 m \rceil$ として、更に以下の手順に従う。
 - i. $r < 2^b - m - 1$ であれば、 $b - 1$ bit で r の 2 進数表記した符号を出力する。
 - ii. $r \geq 2^b - m - 1$ であれば、 b bit で $b + 2^b - m$ の 2 進数表記した符号を出力する。

Rice 符号の場合は、 m が 2 の幂数であることを用いると、剰余 r の値による場合分けのコストがない上、 m による除算が右シフト演算に、剰余が $m - 1$ との AND 演算によっ

て実現できるため、高速な実装が実現できる。

Golomb 符号のアルゴリズムを概観すると、Golomb 符号はパラメータ m による商 q と剰余 r に分けて符号化している事が分かる。これを直感的な図にしてみると図 1.9 のようになる。

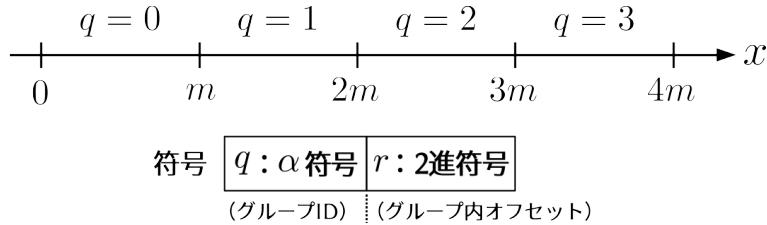


図 1.9 Golomb 符号の直感的な説明図

整数の数直線を長さ m の区間で区切り、商 q によってどの区間にに入っているのかを示し、 r によって区間内の位置を定めていると考えることができる。

Golomb 符号が従う確率分布を導出する。議論の単純化のため、パラメータ m は 2 の幂乗 (Rice 符号) に限定する。整数 x に対する Golomb 符号の長さ $l_G(x)$ は、商部分の α 符号は長さは $q + 1$ 、剰余部分は長さ $\log_2 m$ だから、

$$l_G(x) = q + 1 + \log_2 m = \left\lfloor \frac{x}{m} \right\rfloor + 1 + \log_2 m$$

よって、Golomb 符号が従う分布 $P_G(x)$ は、

$$\begin{aligned} P_G(x) &= 2^{-l_G(x)} = 2^{-\left(\lfloor \frac{x}{m} \rfloor + 1 + \log_2 m\right)} = \frac{1}{2m} 2^{-\lfloor \frac{x}{m} \rfloor} \\ &\approx \frac{1}{2m} 2^{-\frac{x}{m}} \end{aligned}$$

となる。 $P_G(x)$ が確率分布となるためには、全確率 $\int_0^\infty P_G(x)dx$ が 1 となる必要がある。実際に $\int_0^\infty P_G(x)dx$ は、

$$\begin{aligned} \int_0^\infty P_G(x)dx &= \frac{1}{2m} \int_0^\infty 2^{-\frac{x}{m}} dx = \frac{1}{2m} \left[-m \frac{2^{-\frac{x}{m}}}{\log 2} \right]_0^\infty \\ &= -\frac{1}{2} \left(0 - \frac{1}{\log 2} \right) = \frac{1}{2 \log 2} \end{aligned}$$

この結果により、 $P_G(x)$ を $\frac{1}{2 \log 2}$ で割ることで $P_G(x)$ が確率分布となる。その結果を改めて $P_G(x)$ と表すと、

$$P_G(x) = \left(\frac{1}{2 \log 2} \right)^{-1} \frac{1}{2m} 2^{-\frac{x}{m}} = \frac{\log 2}{m} 2^{-\frac{x}{m}}$$

ここで、 $\lambda = \frac{\log 2}{m}$ とおいて整理すると、

$$\begin{aligned} P_G(x) &= \lambda 2^{-\frac{\lambda x}{\log 2}} \\ \implies \log P_G(x) &= \log \lambda - \frac{\lambda x}{\log 2} \log 2 = \log \lambda - \lambda x \\ \implies P_G(x) &= \exp(\log \lambda - \lambda x) \\ &= \lambda \exp(-\lambda x) \end{aligned} \quad (1.47)$$

式 1.47 はパラメータ λ を持つ**指数分布**に他ならない。符号化対象の整数 x は常に離散値を取っているが、確率変数が離散値を取る場合の指数分布は**幾何分布**をなす。幾何分布は $0 < \rho < 1$ を満たすパラメータ ρ を用いて、式 1.48 で表せる。

$$P_G(x) = \rho^x (1 - \rho) \quad (1.48)$$

幾何分布はある事象が成功する（あるいは、失敗する）までに必要とする試行回数の確率分布である。パラメータ ρ は試行一回あたりの失敗（あるいは、成功）確率を示している。様々なパラメータ ρ の値に対応する幾何分布の確率分布関数を図 1.10 に示す。

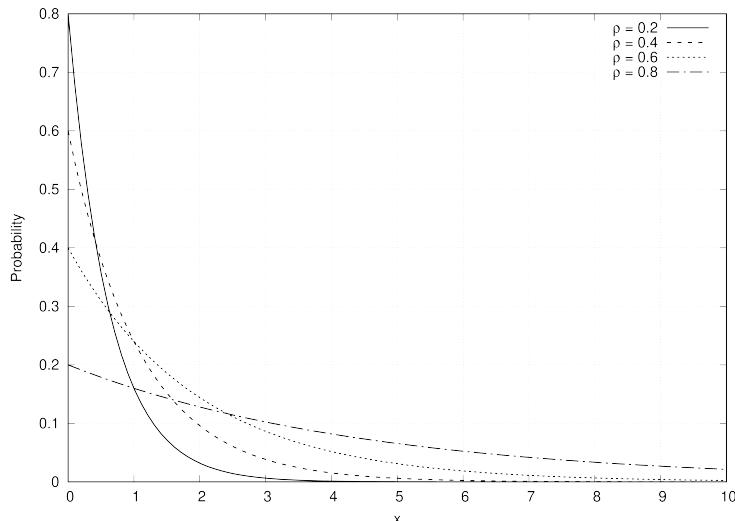


図 1.10 幾何分布 (Golomb 符号が従う分布 $P_G(x)$) の確率分布

以上の議論により、Golomb 符号は符号化対象のデータが幾何分布に従っているときに効率的に圧縮できることが分かる。Golomb 符号がマルチメディアのデータ圧縮に使用される主な理由として、予測による残差の分布は幾何分布に近い形を取るという経験的事実と、ハフマン符号等で必要になるテーブルといった、符号化のための追加記憶領域を必要としない点が挙げられる。

Golomb (ゴロム) 符号のパラメータ設定

前節で導入した Golomb 符号にはパラメータ m が伴っている。 m によって効率的に符号化できる整数の分布が変わってしまうため、パラメータの調整を行うことが実用上重要な。ここでは、[22] に記載のある、JPEG2000 で採用されている手法に従って最適なパラメータの導出を行う。

手法の鍵となるアイデアは、観測した整数 x の平均 $E[x]$ に合わせて m を上手く選ぶことで、 $P_G(x)$ の分布を α 符号の持つ分布に近づける事である。

まず、幾何分布の定義式 1.48 に従って平均 $E[x]$ を求めると、

$$\begin{aligned} E[x] &= \sum_{x=0}^{\infty} x P_G(x) = \sum_{x=0}^{\infty} x(1-\rho)\rho^x \\ &= (1-\rho) \sum_{x=0}^{\infty} x \rho^x = (1-\rho) \frac{\rho}{(1-\rho)^2} \\ &= \frac{\rho}{1-\rho} \end{aligned} \quad (1.49)$$

となり^{*19}、パラメータ ρ から平均 $E[x]$ が求まることが分かる。

次に $P_G(x)$ の分布を α 符号に近づける事を考えるが、式 1.48 のままではパラメータ m が陽に現れないため、議論が進まない。そこで、幾何分布の整数を m 個毎にまとめ、商 q についての確率分布 $P_{G_m}(q)$ を以下の式で定義する。

$$P_{G_m}(q) = \sum_{r=0}^{m-1} P_G(qm+r) \quad (1.50)$$

式 1.50 に式 1.48 を代入すると、

$$\begin{aligned} P_{G_m}(q) &= \sum_{r=0}^{m-1} P_G(qm+r) = \sum_{r=0}^{m-1} \rho^{qm+r}(1-\rho) \\ &= (1-\rho)\rho^{qm} \sum_{r=0}^{m-1} \rho^r \\ &= (1-\rho)\rho^{qm} \frac{1-\rho^m}{1-\rho} \quad (\because \text{等比級数の和の公式}) \\ &= \rho^{qm}(1-\rho^m) \end{aligned} \quad (1.51)$$

が得られる。この式 1.51 において、 $\rho^m = \frac{1}{2}$ であれば、 $P_{G_m}(q) = (\frac{1}{2})^q \times \frac{1}{2} = 2^{-q-1}$ となって確率分布が α 符号に一致し、理想的な符号化が行える。現実の残差は 0 に偏っ

^{*19} ここで、 $\sum_{x=0}^{\infty} x \rho^x = \frac{\rho}{(1-\rho)^2}$ を用いている。証明は、等比級数の和 $\sum_{x=0}^{\infty} \rho^x = \frac{1}{1-\rho}$ の両辺を ρ で微分すると $\sum_{x=0}^{\infty} x \rho^{x-1} = \frac{1}{(1-\rho)^2}$ となり、この両辺に ρ を乗じることで欲しい式が得られる。

て出現することは稀という仮定^{*20}から、図 1.10 から観察できるように $\rho \approx 1$ とすると、 ρ^m は次のように近似できる。

$$\begin{aligned}\rho^m &= \{1 - (1 - \rho)\}^m \approx 1 - m(1 - \rho) \\ &\approx 1 - m \frac{1 - \rho}{\rho} \\ &= 1 - \frac{m}{E[x]} \quad (\because \text{式 1.49 を使用})\end{aligned}\tag{1.52}$$

ρ^m の近似式 1.52 に $\rho^m = \frac{1}{2}$ の条件を入れることで、 m についての式を得る。

$$\begin{aligned}\rho^m &\approx 1 - \frac{m}{E[x]} = \frac{1}{2} \\ \implies m &= \frac{E[x]}{2}\end{aligned}\tag{1.53}$$

式 1.53 が最適なパラメータ m を求める式の一つである。JPEG2000においては、高速化のため Rice 符号のパラメータ k を $m = 2^k$ として、式 1.53 の両辺の \log_2 を取り、以下の式 1.54 で最適なパラメータ k を設定する。

$$k = \max \left\{ 0, \left\lceil \log_2 \left(\frac{E[x]}{2} \right) \right\rceil \right\}\tag{1.54}$$

$E[x]$ は推定値で置き換えて良いため、観測した整数の移動平均等から適応的にパラメータ m, k を求めることができる。

^{*20} この仮定から、ほぼ無音の音声に対して圧縮率が悪くなる。

第2章

実装編

基本理論をベースに、筆者はロスレス音声コーデック ALA を扱うコマンドラインツール（以下、ALA コマンドラインツールと呼ぶ）を C 言語フルスクラッチで作成した。本ツールは、wav ファイルのエンコードと、エンコード済みバイナリファイルを wav ファイルにデコードする機能を持つ。

ソースファイルは [github](https://github.com/MrAiki/ALA) にアップしている。以下の URL を参照のこと。

- <https://github.com/MrAiki/ALA>

ALA は以下のソースファイルから構成され、それぞれ以下の役割を持つ。

```
main.c メインエントリ (main 関数)
ala_predictor.c, ala_predictor.h 予測/合成モジュール
ala_coder.c, ala_coder.h 符号化/復号モジュール
ala_utility.c, ala_utility.h 計算時ユーティリティ
bit_stream.c, bit_stream.h bit 単位の入出力モジュール
wav.c, wav.h wav ファイル入出力モジュール
Makefile メイクファイル
```

本章では、まず ALA コマンドラインツールの使用方法を述べ、次に ALA のファイルフォーマットを解説する。続いて、各ソースファイルについて実装説明を行う。

2.1 ALA コマンドラインツールの使用方法

2.1.1 ビルド方法

ALA コマンドラインツールをビルドするには、コマンドライン上で `make` を実行する^{*1}。

^{*1} gcc が必要である。Windows でビルドする場合は、Windows Subsystem for Linux の使用を推奨。

```
$ make
```

^{*2}gcc により各ソースがコンパイルされる。ビルドに成功すれば、成果物として実行可能形式ファイル ala が出力される。

2.1.2 実行

エンコード

手元の wav ファイルをエンコードする際には、ala をビルドした上で以下のコマンドを実行する。

```
$ ./ala -e 入力ファイル.wav 出力ファイル
```

エンコード処理が実行され、成功した場合は圧縮済みのバイナリファイルが出力される。

デコード

エンコード済みのバイナリファイルをデコードする際は、以下のコマンドを実行する。

```
$ ./ala -d 入力ファイル 出力ファイル.wav
```

デコード処理により wav ファイルが復元される。

2.2 ALA ファイルフォーマット

ALA コマンドラインツールでエンコードしたファイル（以下、ALA ファイルと呼ぶ）の構成概要図を図 2.1 に示す。

ALA ファイルの先頭には、ALA のフォーマット情報やエンコードした wav のフォーマット情報を含むヘッダが記録されている。ヘッダに続き、一定サンプル数単位で波形データを符号化したブロックがエンコードした波形の時系列順に並んでいる。

本節では ALA のヘッダとブロックのフォーマットを紹介する。なお、ALA はバイナリデータを全てビッグエンディアンで記録している。

2.2.1 ファイルヘッダフォーマット

表 2.1 に、ALA ファイルのヘッダフォーマットを示す。

ALA では実装の簡略化のため、末尾を除くブロックのサンプル数と、PARCOR 係数の次数を同一にしている。一般に圧縮率を高めるには、波形の特性に合わせてブロックサイズや PARCOR 係数（線形予測係数）の次数を可変にする [16, 32] 方策が取られる。

^{*2} \$はコマンドプロンプトであり、実際には入力する必要はない。

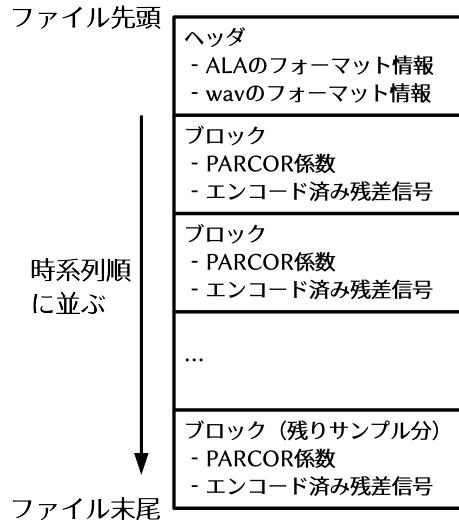


図 2.1 ALA ファイルの構成概要

表 2.1 ALA のヘッダフォーマット

bit 幅	内容	補足
32	'A', 'L', 'A', '\0'	ALA ファイルであることを示すシグネチャ
16	フォーマットバージョン	バイナリフォーマットのバージョン
8	チャンネル数	
32	サンプル数	1 チャンネルあたりのサンプル数を示す
32	サンプリングレート	
8	サンプルあたり bit 数	
16	ブロックあたりサンプル数	ファイル末尾ではこの数値より小さくなるので注意
8	PARCOR 係数次数	全ブロックで同一の次数を用いる

2.2.2 ブロックフォーマット

表 2.2 に、ALA ファイルのブロックフォーマットを示す。

ブロック先頭の同期コードは、デコード時に現在読み出している位置が正しいかどうかを判定する目的で使用する。しかし、この同期コードはおまじない程度にしか効果がない。PARCOR 係数や、エンコード済みの残差の bit 列の中に 0xFFFF が表れた場合、その場所をブロック先頭と誤判定する可能性がある。より厳格な対策として、ブロックデー

表2.2 ALAのブロックフォーマット

bit 幅	内容	補足
16	0xFFFF	ブロック先頭を示す同期コード
x	PARCOR 係数	$x = 16 \times \text{PARCOR 係数次数} \times \text{波形チャンネル数}$ 1 チャンネル目係数列、2 チャンネル目係数列、... の順序で記録
不定	エンコード済みの残差	1 チャンネル目残差、2 チャンネル目残差、... の順序で記録

タのチェックサム値や巡回冗長検査 (CRC16、CRC32 等) の計算結果をブロックに記録しておき、デコード時に検査する対策がとられる [28, 32, 16, 33]。ALA では実装の簡略化のため、シグネチャと同期コード以外にデータの正しさをチェックする機能を省いている。

2.3 ala_predictor.c

`ala_predictor.c` には、予測と合成処理を行う関数群が集められている。

- PARCOR 係数計算
- PARCOR 予測/合成フィルター
- プリエンファシス/デエンファシス
- チャンネル間相関除去 (MS 処理)

本節では、これらの計算処理について説明する。計算処理は全て固定小数演算により実現されているため、コードの説明の前置きとして固定小数演算についての補足説明を入れる。また、プリエンファシス/デエンファシスと、チャンネル間相関除去については基本理論で触れていない為、本節で簡易に補足を入れる。なお、PARCOR 係数計算については、リスト 1.1 (1.1.3 節) で既に取り上げているため説明を省略する。

2.3.1 固定小数演算

固定小数 (fixed-point float) とは、小数点が固定された bit 位置にある小数の表現法である。一方、C 言語で `float`, `double` 等の型を持つ浮動小数点数は、指数部の値により小数点が移動する。固定小数点は整数によって表すことができ、また、固定小数同士の演算も整数演算によって実行できる。ここでは、[24, 25] を参考にしながら固定小数演算

についての解説を行う。

固定小数を用いる理由は主に 2 点ある。1 つは、浮動小数演算を用いると CPU や FPU の差異により計算結果が異なる場合がある。例えば、微小な数値を扱う際に、丸めの方向が異なったり、桁落ちが発生した時の動作が異なる [23]。この特性があると、エンコードした時の演算結果とデコードした時の演算結果に差異が発生し、ビットパーカーフェクトに復元できなくなる可能性がある。一方、整数演算は、同一の整数型で同一の演算を行えば結果を復元することが可能である。固定小数を用いるもう 1 つの理由は、整数演算を備えているプラットフォームは多く存在しているため、移植がしやすいという点が挙げられる。

固定小数による小数の表記

簡単な例として、下位 4bit を小数部、上位 4bit を整数部とした 8bit 固定小数を考える。この固定小数における小数の表示例を表 2.3 に示す。なお、整数部の上位 1bit は符号 bit として、2 の補数表現により負数を表現する。

表 2.3 下位 4bit を小数部、上位 4bit を整数部とした 8bit 固定小数の数値例

10 進小数	8bit 固定小数 (2 進数)	10 進小数	8bit 固定小数 (2 進数)
7.9375	0111 1111	-0.0625	1111 1111
7.875	0111 1110	-0.125	1111 1110
7.8125	0111 1101	-0.1875	1111 1101
7.75	0111 1100	-0.25	1111 1100
⋮	⋮	⋮	⋮
0.1875	0000 0011	-7.8125	1000 0011
0.125	0000 0010	-7.875	1000 0010
0.0625	0000 0001	-7.9375	1000 0001
0.0	0000 0000	-8.0	1000 0000

表 2.3 の観察から、固定小数は符号付き整数と全く同様の順序関係を持っていることが分かる。また、最大値 7.9375 は最小値 -8.0 よりも絶対値が小さいことや、正数よりも負数の方が多い性質を持つ。

一般に小数部に n bit 使用した際に、整数で表現された固定小数 f は以下の式 2.1 で 10 進小数 d に変換できる。

$$d = f \times 2^{-n} \quad (2.1)$$

逆に、10 進小数 d は以下の式 2.2 で固定小数 f に変換できる。

$$f = \text{round}(d \times 2^n) \quad (2.2)$$

ここで round は整数への丸めを行う関数である^{*3}。式 2.1, 2.2 で現れる 2 の幂乗数による乗算は、C 言語では nbit のシフト演算により変換を行うことができる。

固定小数演算

固定小数どうしの加算と減算は、整数の加算と減算と全く同様に計算できる。nbit の小数部を持つ固定小数 f_1, f_2 に対応する 10 進小数を d_1, d_2 で表すと、加算 $d_1 + d_2$ と減算 $d_1 - d_2$ は式 2.1 により、

$$\begin{aligned}d_1 + d_2 &= f_1 \times 2^{-n} + f_2 \times 2^{-n} = (f_1 + f_2) \times 2^{-n} \\d_1 - d_2 &= f_1 \times 2^{-n} - f_2 \times 2^{-n} = (f_1 - f_2) \times 2^{-n}\end{aligned}$$

となるため、整数加算と減算をそのまま行えば良い。

固定小数どうしの乗算は、整数乗算を行う際に小数部 bit 分だけ右シフトを行う必要がある。何故なら、式 2.1 を用いて乗算 $d_1 \times d_2$ を計算してみると、

$$\begin{aligned}d_1 \times d_2 &= f_1 \times 2^{-n} \times f_2 \times 2^{-n} \\&= (f_1 \times f_2 \times 2^{-n}) \times 2^{-n}\end{aligned}$$

となる。固定小数どうしの乗算を行うことで、 $f_1 \times f_2 \times 2^{-n}$ の結果は得られるが、最後の 2^{-n} の乗算が不足している事が分かる。この為、整数乗算結果に対して 2^{-n} を乗ずる（C 言語では nbit の算術右シフトを行う）ことが必要である^{*4}。

乗算の計算例を挙げる。表 2.3 を参考に、 7.75×-0.25 を考えてみる。 7.75 に対応する整数値は $01111100 = 124$ 、 -0.25 に対応する整数値は $11111100 = -4$ である。整数乗算の結果 $124 \times -4 = -496$ が得られ、これに $2^{-4} = 1/16$ を乗ずることで -31 が得られる。 -31 に 2^{-4} を乗じて 10 進小数に戻すと -1.9375 が得られ、これは 7.75×-0.25 の結果に一致していることが確認できる。

固定小数の乗算処理を実装する際に注意すべき点^{*5}を 2 点挙げる。整数のオーバーフローと、右シフト時の切り捨てである。

まず、固定小数は整数部の bit 幅が狭くなっているため、乗算によってオーバーフローが発生しやすい。例えば、24bit の量子化 bit 数を持つ wav ファイルを扱う際に小数部の bit 数を 24 とすると、32bit 整数型 (int32_t 等) を使用した際にオーバーフローが発生するので注意が必要である。対応としては、64bit 整数型 (int64_t 等) を使うか、オーバーフローが発生しないように乗算前に数値を右シフトしておくことが考えられる。

もう 1 点の右シフト時の切り捨てについても注意すべきである。乗算後の右シフトを何も考えずに行うと、乗算結果が負に偏って表れやすくなる。何故なら、C 言語の算術右

^{*3} 丸めは一意に定まらない。最も近い整数に丸めることもあれば、無条件に切り捨て/切り上げを行う操作を差して丸めと言う場合もある。

^{*4} 固定小数に対して 2^{-n} を乗ずるタイミングは、乗算前でも良い。

^{*5} ⇔ 筆者が躊躇した点

シフト演算は、正整数 1 に対する 1bit 右シフト $1 \gg 1$ の結果は 0 になる一方で、負整数 -1 に対する 1bit 右シフト $-1 \gg 1$ は -1 のままとなるからである。この現象に対しては、乗算結果に 0.5 を加えてから右シフトを行う（正の無限大方向に切り上げる）ことで偏りをある程度緩和できる。

次に固定小数どうしの除算について見ていく。除算では、乗算とは逆に小数部 bit 数だけ左シフトを行う必要がある。理由としては、再度 n bit の小数部を持つ固定小数 f_1, f_2 とそれに対応する 10 進小数 d_1, d_2 を用いた際に、 d_1/d_2 は、

$$\begin{aligned} d_1/d_2 &= (f_1 \times 2^{-n})/(f_2 \times 2^{-n}) \\ &= (f_1/f_2 \times 2^{-n})/2^{-n} = (f_1/f_2 \times 2^{-n}) \times 2^n \end{aligned}$$

となり、 2^n の乗算が追加で必要になることが分かる。ALAにおいては除算は使用していないため、立ち入った説明は行わない。

2.3.2 PARCOR 予測フィルター

PARCOR 係数による予測は、端的に言えば格子型フィルター（図 1.4、式 1.21、式 1.22）をそのままコードに落とし込んだものである。式 1.21、式 1.22 を再掲する。

$$\begin{aligned} f_M(n) &= f_{M-1}(n) - \gamma_M b_{M-1}(n-1) \quad (\text{式 1.21 を再掲}) \\ b_M(n) &= b_{M-1}(n-1) - \gamma_M f_{M-1}(n) \quad (\text{式 1.22 を再掲}) \end{aligned}$$

また、PARCOR 係数による格子型予測フィルターの図（図 1.4）をここに再掲する。

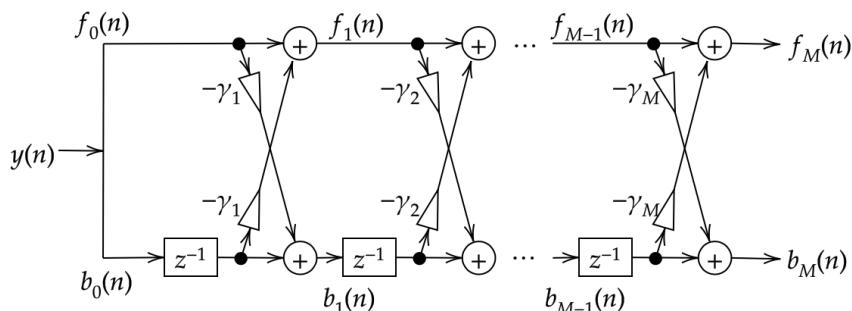


図 2.2 格子型フィルターの連結（図 1.4 を再掲）

PARCOR 係数を用いた予測フィルターの実装部分をリスト 2.1 に示す。

リスト 2.1 PARCOR 予測フィルター実装 (ala_predictor.c)

```
283 /* PARCOR係数により予測/誤差出力(32bit整数入出力) */
284 ALAPredictorApiResult ALALPCSSynthesizer_PredictByParcorCoefInt32(
285     struct ALALPCSSynthesizer* lpc,
```

```
286     const int32_t* data, uint32_t num_samples,
287     const int32_t* parcor_coef, uint32_t order, int32_t* residual)
288 {
289     uint32_t samp, ord;
290     int32_t* forward_residual;
291     int32_t* backward_residual;
292     int32_t mul_temp;
293     /* 丸め誤差軽減のための加算定数 = 0.5 */
294     const int32_t half = (1UL << 14);
295
296     /* 引数チェック */
297     if (lpc == NULL || data == NULL
298         || parcor_coef == NULL || residual == NULL) {
299         return ALAPREDICTOR_APIRESULT_INVALID_ARGUMENT;
300     }
301
302     /* 次数チェック */
303     if (order > lpc->max_order) {
304         return ALAPREDICTOR_APIRESULT_EXCEED_MAX_ORDER;
305     }
306
307     /* オート変数にポインタをコピー */
308     forward_residual = lpc->forward_residual;
309     backward_residual = lpc->backward_residual;
310
311     /* 誤差計算 */
312     for (samp = 0; samp < num_samples; samp++) {
313         /* 格子型フィルタにデータ入力 */
314         forward_residual[0] = data[samp];
315         /* 前向き誤差計算 */
316         for (ord = 1; ord <= order; ord++) {
317             mul_temp
318                 = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
319                     backward_residual[ord - 1] + half, 15);
320             forward_residual[ord] = forward_residual[ord - 1] - mul_temp;
321         }
322         /* 後ろ向き誤差計算 */
323         for (ord = order; ord >= 1; ord--) {
324             mul_temp
325                 = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
326                     forward_residual[ord - 1] + half, 15);
327             backward_residual[ord] = backward_residual[ord - 1] - mul_temp;
328         }
329         /* 後ろ向き誤差計算部にデータ入力 */
330         backward_residual[0] = data[samp];
331         /* 残差信号 */
332         residual[samp] = forward_residual[order];
333     }
334 }
```

固定小数演算を行うにあたり、小数部の bit 幅は 15 としている（16bit とすると、32bit 整数の乗算時にオーバーフローが発生する可能性があるため^{*6}）。このソースに対して幾つか補足を入れる。

乗算時の右シフト対策の定数

リスト 2.2 乗算時の定数 (ala_predictor.c)

```
293 /* 丸め誤差軽減のための加算定数 = 0.5 */
294 const int32_t half = (1UL << 14);
```

乗算時の右シフトによる切り捨てに対するための定数 `half` を定義している。小数部の bit 幅が 15 のため、0.5 は 1 を 14bit 左シフトして得ることができる。

前向き誤差計算

リスト 2.3 前向き誤差計算 (ala_predictor.c)

```
313 /* 格子型フィルタにデータ入力 */
314 forward_residual[0] = data[samp];
315 /* 前向き誤差計算 */
316 for (ord = 1; ord <= order; ord++) {
317     mul_temp
318     = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
319         backward_residual[ord - 1] + half, 15);
320     forward_residual[ord] = forward_residual[ord - 1] - mul_temp;
321 }
```

式 1.21 に従って低次の前向き誤差から順次前向き計算を行っている。途中、固定小数乗算を行い変数 `mul_temp` に代入している。注目すべきは `ALAUTILITY_SHIFT_RIGHT_ARITHMETIC` マクロである。これは算術右シフト演算を確実に行うためのマクロであり、`ala_utility.h` に定義がある。

リスト 2.4 算術右シフトマクロ (ala_utility.h)

```
11 /* 算術右シフト */
12 #if (((int32_t)-1) >> 1) == ((int32_t)-1)
13 /* 算術右シフトが有効な環境では、そのまま右シフト */
14 #define ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(sint32, rshift) ((sint32) >> (
15     rshift))
16 /* 算術右シフトが無効な環境では、自分で定義する ハッカーのたのしみのより引用 */
17 /* 注意)有効範囲:0 <= rshift <= 32 */
18 #define ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(sint32, rshift) \
19     (((uint64_t)(sint32) + 0x80000000UL) >> (rshift)) - (0x80000000UL >> (
20     rshift)))
```

^{*6} 例えば、16bit 整数の最大値 65535 どうしの乗算は $65535 \times 65535 = 4294836225$ 。これは、符号付き 32bit 整数の最大値を超えておりオーバーフローしている。

 20 #endif

一般に整数型に対する右シフト演算 (`>>`) は、算術右シフト演算になるとは限らない [17] ため、符号なし整数を用いた算術右シフト演算を定義している。

後ろ向き誤差計算

リスト 2.5 後ろ向き誤差計算 (ala_predictor.c)

```

321  /* 後ろ向き誤差計算 */
322  for (ord = order; ord >= 1; ord--) {
323      mul_temp
324          = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
325              forward_residual[ord - 1] + half, 15);
325      backward_residual[ord] = backward_residual[ord - 1] - mul_temp;
326  }
327  /* 後ろ向き誤差計算部にデータ入力 */
328  backward_residual[0] = data[samp];

```

式 1.22 に従って後ろ向き誤差を更新している。前向き誤差とは異なり、変数 `backward_residual[ord]` は次のループで計算で使用するために、コウジイ…から低次後ろ向き誤差に向かって更新を行わなければならない。

2.3.3 PARCOR 合成フィルター

PARCOR 係数による残差からの合成処理についても格子型フィルター（図 1.5、式 1.23、式 1.22）の実装を素直に行うだけである。式 1.5、式 1.22 を再掲する。

$$f_{M-1}(n) = f_M(n) + \gamma_M b_{M-1}(n-1) \quad (\text{式 1.23 を再掲})$$

$$b_M(n) = b_{M-1}(n-1) - \gamma_M f_{M-1}(n) \quad (\text{式 1.22 を再掲})$$

また、PARCOR 係数による格子型合成フィルターの図（図 1.5）をここに再掲する。

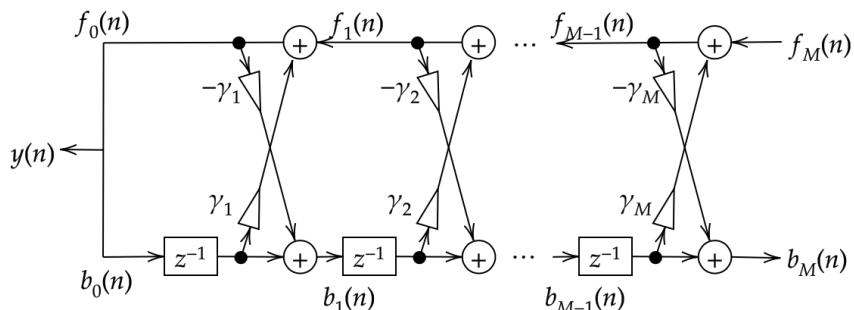


図 2.3 格子型フィルターの連結（図 1.5 を再掲）

合成処理部分の実装をリスト 2.6 に示す。

リスト 2.6 PARCOR 合成フィルター実装 (ala_predictor.c)

```

362  /* 格子型フィルタによる音声合成 */
363  for (samp = 0; samp < num_samples; samp++) {
364      /* 誤差入力 */
365      forward_residual = residual[samp];
366      for (ord = order; ord >= 1; ord--) {
367          /* 前向き誤差計算 */
368          forward_residual
369              += (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
370                  backward_residual[ord - 1] + half, 15);
370      /* 後ろ向き誤差計算 */
371      mul_temp
372          = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(parcor_coef[ord] *
373              forward_residual + half, 15);
373      backward_residual[ord] = backward_residual[ord - 1] - mul_temp;
374  }
375  /* 合成信号 */
376  output[samp] = forward_residual;
377  /* 後ろ向き誤差計算部にデータ入力 */
378  backward_residual[0] = forward_residual;
379 }
```

残差計算の時とは異なり、前向き誤差と後ろ向き誤差は両方とも高次から低次の残差に向かって計算するため、前向き誤差と後ろ向き誤差をそれぞれ個別の `for` 文によって更新する必要がない。また、式 1.23 に注目すると、前向き誤差は次のサンプルの計算で使われることは無いので配列に記録する必要がなく、単一の変数 `forward_residual` への積和演算を行えば良い。これにより合成処理は予測時よりもわずかに早く計算を行える。

2.3.4 MS 処理

CD 等に収録されたステレオ音源は、収録環境（マイクの配置、マスタリング等）に依存するが L (左) チャンネルと R (右) チャンネルは互いに相関を持っていることが多い。チャンネル間の相関（他チャンネルの影響）を除去できれば、よりチャンネルの独立性を高めた信号処理を行いやすくなる。チャンネル間の相関を除去する手法の 1 つに **MS 処理**がある。MS 処理は多数のコーデックで有効性が認められている [26, 28]。本節では MS 処理について概要と実装を説明する。

MS 処理は、ある時刻における L チャンネル信号 L と R チャンネル信号 R を次の式 2.3, 2.4 によって M (Mid, ミッド) チャンネル信号 M と S (Side, サイド) チャンネル信号 S に変換する。

$$M = L + R \quad (2.3)$$

$$S = L - R \quad (2.4)$$

M, S はそれぞれ中心、側面成分を示している。一方、相関除去の立場から考えてみると、 M は L チャンネルと R チャンネルの符号が同一（同相）の成分が強められ、符号が逆（逆相）が弱められた成分、 S は M と逆の性質を持った成分と考えられる。

M, S は以下の式 2.5, 2.6 によって L, R に戻すことができる。

$$L = (M + S)/2 \quad (2.5)$$

$$R = (M - S)/2 \quad (2.6)$$

MS 处理の実装解説

MS 处理の実装をリスト 2.7 に示す。

リスト 2.7 LR 信号を MS 信号に変換 (ala_predictor.c)

```

480 /* LR -> MS(int32_t) */
481 ALAPredictorApiResult ALAChannelDecorrelator_LRtoMSInt32(int32_t **data,
482     uint32_t num_channels, uint32_t num_samples)
483 {
484     uint32_t smpl;
485     int32_t mid, side;
486
487     /* 引数チェック */
488     if ((data != NULL)
489         || (data[0] != NULL) || (data[1] != NULL)
490         || (num_channels < 2)) {
491         return ALAPREDICTOR_APIRESULT_INVALID_ARGUMENT;
492     }
493
494     /* サンプル単位でLR -> MS処理 */
495     for (smpl = 0; smpl < num_samples; smpl++) {
496         /* 注意：除算は右シフト必須(/2ではだめ。0方向に丸められる) */
497         mid = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(data[0][smpl] + data
498             [1][smpl], 1);
499         side = data[0][smpl] - data[1][smpl];
500         data[0][smpl] = mid;
501         data[1][smpl] = side;
502     }
503
504     return ALAPREDICTOR_APIRESULT_OK;
505 }
```

単純な MS 处理の式 2.3, 2.4 とは異なり、以下のコードにより変換を行っている。

リスト 2.8 MS 変換処理部分 (ala_predictor.c)

```

496     /* 注意：除算は右シフト必須(/2ではだめ。0方向に丸められる) */
497     mid = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(data[0][smpl] + data
498         [1][smpl], 1);
499     side = data[0][smpl] - data[1][smpl];
```

M チャンネル成分が 2 で割られている。これで問題なく処理ができる理由は、式 2.3, 2.4 による変換の下位 1bit は一致しているという整数演算の性質に基づく。整数の下位 1bit は奇数か偶数かを示す bit とも捉える事ができる。この整数の奇偶に着目すると、奇数と偶数の加算と減算の結果の奇偶は一致することが分かる。念の為、くどいかもしれないが、奇数と偶数の全ての加減算のパターンを確認してみると、

$$\begin{array}{ll} \text{奇数} + \text{奇数} = \text{偶数} & \text{奇数} - \text{奇数} = \text{偶数} \\ \text{奇数} + \text{偶数} = \text{奇数} & \text{奇数} - \text{偶数} = \text{奇数} \\ \text{偶数} + \text{奇数} = \text{奇数} & \text{偶数} - \text{奇数} = \text{奇数} \\ \text{偶数} + \text{偶数} = \text{偶数} & \text{偶数} - \text{偶数} = \text{偶数} \end{array}$$

よって、 M か S の片方の奇偶が分かれればもう片方の奇偶を判別できる。この事実を用いれば、 M か S 片方の奇偶、つまり下位 1bit は冗長な情報となるので削除できる。本実装では M チャンネルの下位 1bit を右シフトによって削除している⁷。

次に、MS 信号を LR 信号に戻す処理の実装をリスト 2.9 に示す。

リスト 2.9 MS 信号を LR 信号に変換 (ala_predictor.c)

```

506 /* MS -> LR(int32_t) */
507 ALAPredictorApiResult ALAChannelDecorrelator_MStoLRI32(int32_t **data,
508     uint32_t num_channels, uint32_t num_samples)
509 {
510     uint32_t smpl;
511     int32_t mid, side;
512
513     /* 引数チェック */
514     if ((data != NULL)
515         || (data[0] != NULL) || (data[1] != NULL)
516         || (num_channels < 2)) {
517         return ALAPREDICTOR_APIRESULT_INVALID_ARGUMENT;
518     }
519
520     /* サンプル単位でMS -> LR処理 */
521     for (smpl = 0; smpl < num_samples; smpl++) {
522         side = data[1][smpl];
523         mid = (data[0][smpl] << 1) | (side & 1);
524         data[0][smpl] = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(mid + side
525             , 1);
526         data[1][smpl] = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(mid - side
527             , 1);
528     }

```

*7 コメントに記したが、右シフトでないと成立しないことに要注意。例えば、 $L=-741$, $R=322$, $M=L+R=-419$, $S=L-R=-1063$ のとき、 $(L+R)>>1$ は-210 と評価される。 -210 を 1bit 左シフトすると-420 と評価されるが、下位 1bit を S から補えば-419 となり、元に戻る。 $(L+R)/2$ は-209 と評価され、 M は元に戻らない。筆者は単純な除算を行って検証してしまい、休日 2 日が失われた。

```
528     return ALAPREDICTOR_APIRESULT_OK;
529 }
```

変換処理部分の実装をリスト 2.10 に示す。

リスト 2.10 LR 復元処理部分 (ala_predictor.c)

```
522     side = data[1][smp1];
523     mid = (data[0][smp1] << 1) | (side & 1);
524     data[0][smp1] = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(mid + side
525           , 1);
      data[1][smp1] = (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(mid - side
           , 1);
```

まず S チャンネル成分の下位 1bit の情報を使って MS 成分を復元してから、後は式 2.5, 2.6 によって LR チャンネル成分を復元している。

2.3.5 プリエンファシス/デエンファシス

線形予測の前処理として、プリエンファシスという処理が行われる事が多い [1, 11]。プリエンファシスは、音声の高周波数成分を強調する処理であり、しばしばタップ数 1 の FIR フィルターで実現される。具体的に処理を式に起こしてみると、時刻 n における出力 $y(n)$ は、現在の入力 $x(n)$ と直前時刻の入力 $x(n-1)$ を用いて式 2.7 で定義される。

$$y(n) = x(n) - \mu x(n-1) \quad (2.7)$$

ここで μ は FIR フィルターの係数であり、プリエンファシス処理においては $0.9 \leq \mu \leq 1$ の範囲に設定される。もし $\mu = 1$ ならば式 2.7 は入力データの差分計算に該当するため、直流成分を完全に打ち消す効果を持つ。

プリエンファシス処理を施したデータを元に戻す処理にデエンファシスがある。これはプリエンファシスとは逆に低周波数成分を強調する処理であり、次の式 2.12 で定義される。

$$y(n) = x(n) + \mu x(n-1) \quad (2.8)$$

何故プリエンファシス処理が上手くいくのか、筆者が調べた限り、その定性的な説明はなされていない。一般に、プリエンファシス処理によって、収録時に失われがちな高周波数成分が相対的に持ち上がり、スペクトラムが平坦になるのが良い分析結果をもたらすという説明が多くなされる。[11] では実験的に、プリエンファシスを使用した場合は、線形予測で得られる係数が安定（発散しにくい状態）になりやすいとの結果を示している。

プリエンファシスフィルター処理

プリエンファシスフィルター処理の実装をリスト 2.11 に示す。

リスト 2.11 プリエンファシスフィルター処理 (ala_predictor.c)

```

384 /* プリエンファシス(int32, in-place) */
385 ALAPredictorApiResult ALAEmpHASISFilter_PreEmphasisInt32(
386     int32_t* data, uint32_t num_samples, int32_t coef_shift)
387 {
388     uint32_t smpl;
389     int32_t prev_int32, tmp_int32;
390     const int32_t coef_numer = (int32_t)((1 << coef_shift) - 1);
391
392     /* 引数チェック */
393     if (data == NULL) {
394         return ALAPREDICTOR_APIRESULT_INVALID_ARGUMENT;
395     }
396
397     /* フィルタ適用 */
398     prev_int32 = 0;
399     for (smpl = 0; smpl < num_samples; smpl++) {
400         tmp_int32 = data[smpl];
401         data[smpl] -= (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(prev_int32 *
402                         coef_numer, coef_shift);
403         prev_int32 = tmp_int32;
404     }
405     return ALAPREDICTOR_APIRESULT_OK;
406 }
```

入力データ `data` をその場で (in-place で) 書き換える関数である。フィルター計算は固定小数乗算を使用しており、式 2.7 の μ に該当する係数との乗算を、 $(1 << \text{coef_shift}) - 1$ との乗算処理と、`coef_shift` の右シフト処理に分けて実現している。ALA バージョン 1.0.0 では、`coef_shift` は 5 に設定されているため、フィルター係数 μ の値は $\mu = (2^5 - 1)/2^5 = 31/32 = 0.96875$ に相当する。

デエンファシスフィルター処理

デエンファシスフィルター処理の実装をリスト 2.12 に示す。

リスト 2.12 デエンファシスフィルター処理 (ala_predictor.c)

```

408 /* デエンファシス(int32, in-place) */
409 ALAPredictorApiResult ALAEmpHASISFilter_DeEmphasisInt32(
410     int32_t* data, uint32_t num_samples, int32_t coef_shift)
411 {
412     uint32_t smpl;
413     const int32_t coef_numer = (int32_t)((1 << coef_shift) - 1);
414
415     /* 引数チェック */
416     if (data == NULL) {
417         return ALAPREDICTOR_APIRESULT_INVALID_ARGUMENT;
418     }
419 }
```

```

420  /* フィルタ適用 */
421  for (smpl = 1; smpl < num_samples; smpl++) {
422      data[smpl] += (int32_t)ALAUTILITY_SHIFT_RIGHT_ARITHMETIC(data[smpl -
423          1] * coef_numer, coef_shift);
424  }
425  return ALAPREDICTOR_APIRESULT_OK;
426 }
```

デエンファシスフィルター処理は簡単である。式 2.8 で示したように、前のサンプル値にフィルター係数を乗じ、現在のサンプル値に加算するだけで良い。

2.3.6 残差のエントロピー観察

本節では MS 処理、プリエンファシス、PARCOR 予測の 3 つの予測手法によって、どれほど音声データのエントロピーが変化するかを観察する。

元の信号と残差の分布

信号の分布は、入力 wav の元の信号（残差計算前）とその残差の計算結果に対して PCM 値の出現頻度を計測することで観察した。頻度の絶対値は音源により異なるため、出現頻度の総和を取って出現頻度を割ることで経験確率を算出した。

表 1.5 で挙げた 8 個の wav ファイルに対して信号の分布を計測した結果を図 2.4-2.11 に示す。グラフにおいて、横軸（PCM の値）の範囲は -2048 から 2048 に制限し、縦軸（経験確率）は差を見やすくするために対数軸を用いている。

全体的に、残差計算後の分布の方が 0 における頻度ピーク値が大きく、また、分布の 0 から離れたところの頻度（裾野）が鋭く減少している事が読み取れる。これは、図 1.48 で見た幾何分布の概形に近く、よりエントロピー符号化に適した分布に変換できていることを意味する。

ファイル個別に気になる点を述べる。1-01 火焰太鼓.wav の元の信号の分布は奇特である。0 にピークはあるものの、次の緩やかなピークが -100 周辺に存在する。これは、収録時に発生している背景ノイズと、直流成分（オフセット）の影響を受けているものと考えられる。振幅値の振れ幅が大きい 4-02 Let's アイカツ!(Short サイズ).wav と 5-02 カレンダーガール (TV-size).wav では、他の分布と異なり残差の減衰が緩やか（裾野が広い）だが、減衰が早いためグラフの横軸を [-4000, 4000] まで広げた時に残差の頻度の方が小さくなっていることを確認した。

残差計算前後でのエントロピーの変化

残差計算前後でのエントロピーの計算結果を表 2.4 に示す。表 1.5 の計測を行った際はデータを 1 バイト (8bit) に区切って計測していたが、ここでは PCM の bit 幅 (16bit)

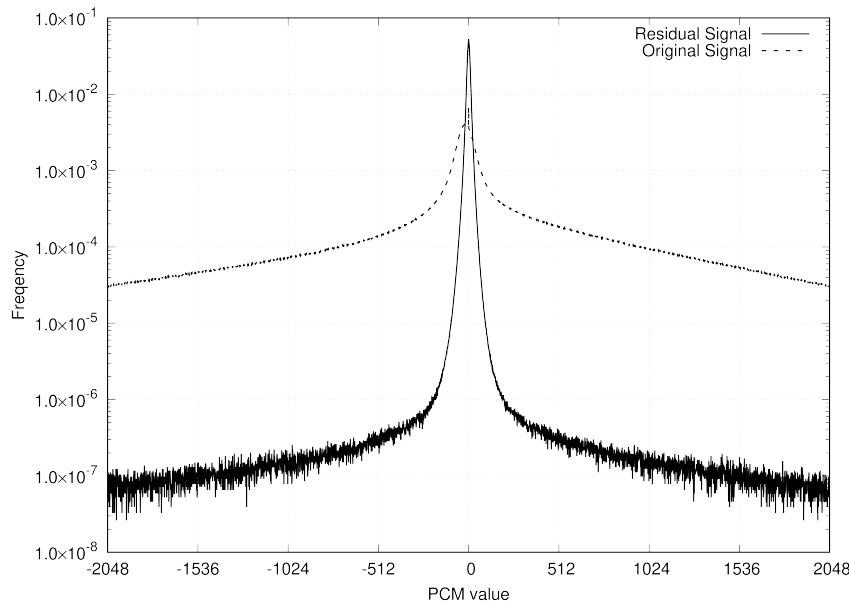


図 2.4 4-02 井戸の茶椀.wav に対する信号分布計測結果

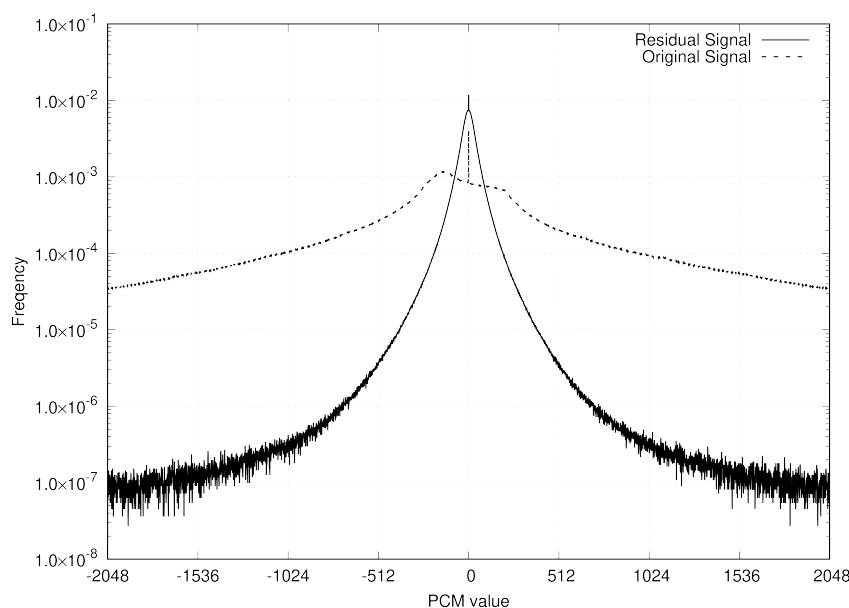


図 2.5 1-01 火炎太鼓.wav に対する信号分布計測結果

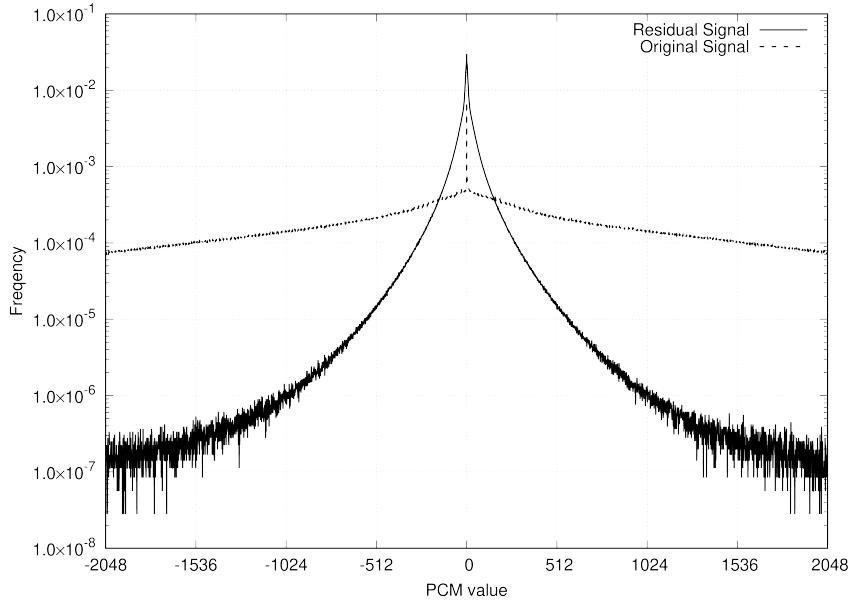


図 2.6 02 My Song.wav に対する信号分布計測結果

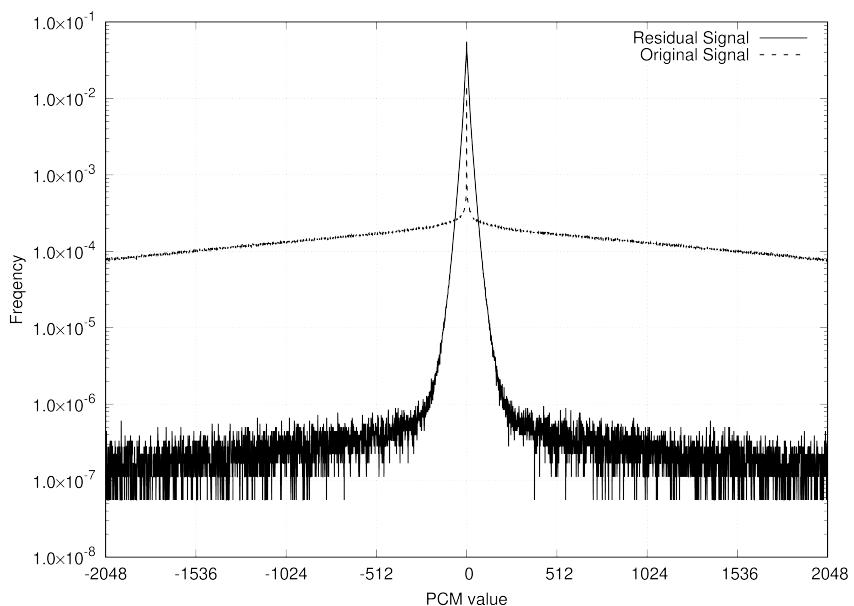


図 2.7 09 瑠璃子.wav に対する信号分布計測結果

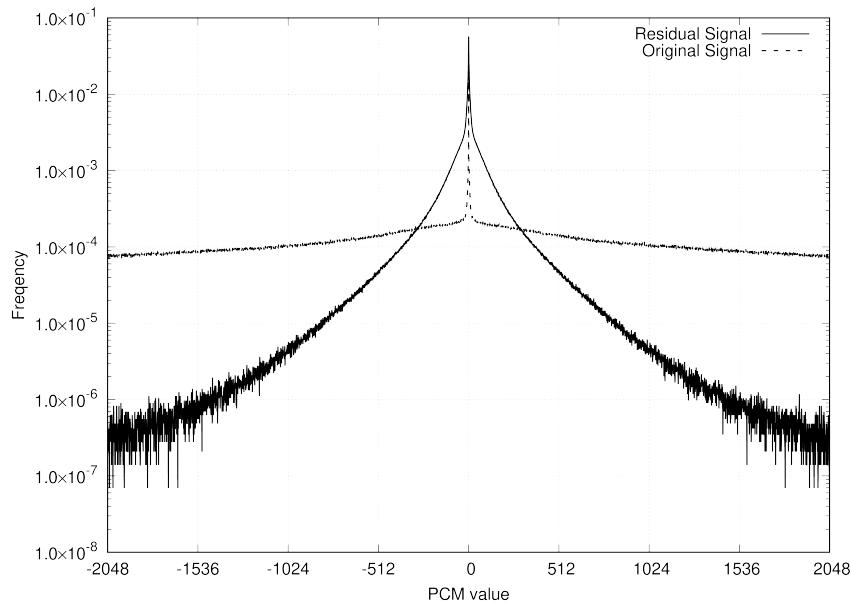


図 2.8 30-自分の輝き.wav に対する信号分布計測結果

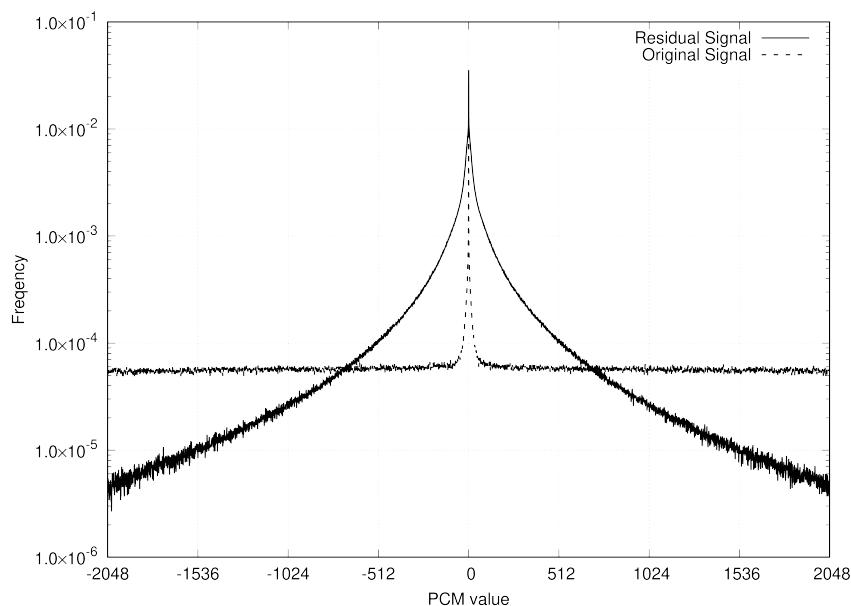


図 2.9 29-重ねる努力.wav に対する信号分布計測結果

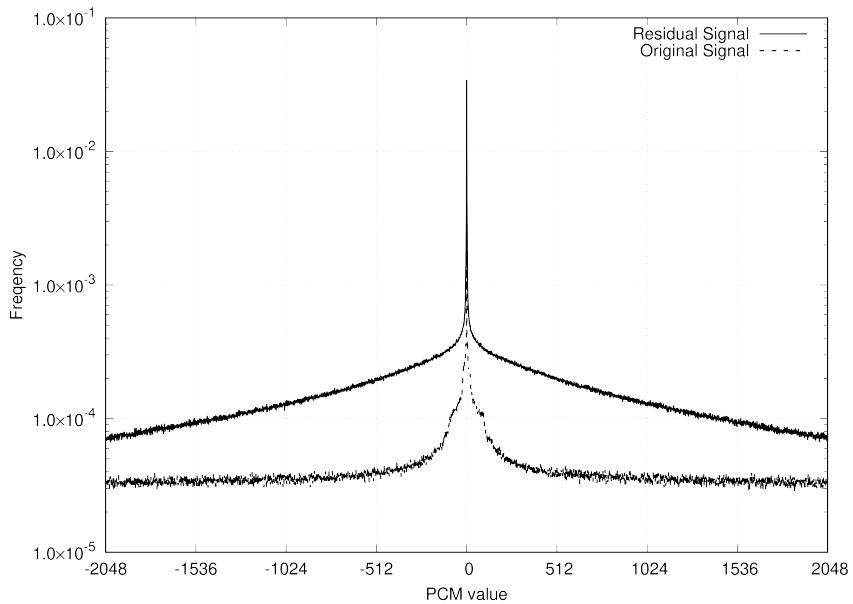


図 2.10 4-02 Let's アイカツ! (Short サイズ).wav に対する信号分布計測結果

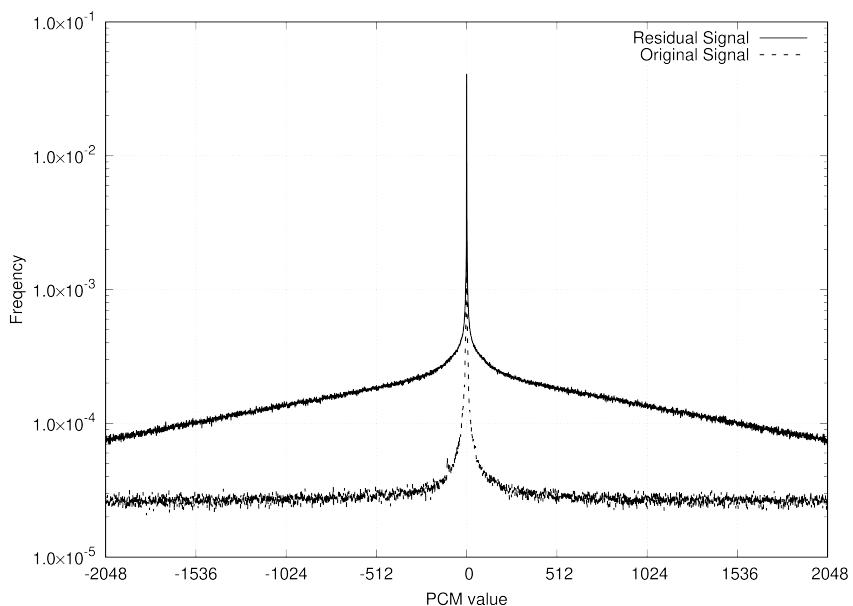


図 2.11 5-02 カレンダーガール (TV-size).wav に対する信号分布計測結果

に区切って計測を行ったため、残差計算前のエントロピー値が表 1.5 と異なることに注意いただきたい。

表 2.4 残差計算前後のエントロピー計算結果

ファイル名	元の信号 [byte]	残差 [byte]
4-02 井戸の茶椀.wav	11.04	5.44
1-01 火焰太鼓.wav	11.91	7.88
02 My Song.wav	13.04	7.86
09 瑠璃子.wav	13.66	6.01
30-自分だけの輝き.wav	13.73	8.27
29-重ねる努力.wav	14.59	9.42
4-02 Let's アイカツ!(Short サイズ).wav	15.10	12.82
5-02 カレンダーガール(TV-size).wav	15.16	12.88

全てのファイルにおいて、残差計算によってエントロピーが減少することを確認した。ただしエントロピーの減少度合いは音源依存で、4-02 井戸の茶椀.wav に見られるようにエントロピーが半分以下になるファイルがある一方、5-02 カレンダーガール(TV-size).wav のように、エントロピーが 1 割程度しか減少しないファイルが存在した。

2.4 bit_stream.c

bit_stream.c には、bit 単位でファイル入出力を行う関数群が集められている。このモジュールは bit 単位の入出力機能に限らず、CPU の差異により発生するバイトオーダー(エンディアン)を揃える目的も兼ねている。実装は [7] の bitio.c を参考に、僅かな機能拡張を行っている。

2.4.1 bit 単位の出力処理

1bit 出力処理

1bit だけ出力を行う処理をリスト 2.13 に示す。

リスト 2.13 1bit 出力処理 (bit_stream.c)

```

177 /* 1bit出力 */
178 BitStreamApiResult BitStream_PutBit(struct BitStream* stream, uint8_t
                                         bit)
179 {
180     /* 引数チェック */
181     if (stream == NULL) {
182         return BITSTREAM_APIRESULT_INVALID_ARGUMENT;

```

```

183     }
184
185     /* 読み込みモードでは実行不可能 */
186     if (stream->flags & BITSTREAM_FLAGS_FILEOPENMODE_READ) {
187         return BITSTREAM_APIRESULT_INVALID_MODE;
188     }
189
190     /* バイト出力するまでのカウントを減らす */
191     stream->bit_count--;
192
193     /* ビット出力バッファに値を格納 */
194     if (bit != 0) {
195         stream->bit_buffer |= (uint8_t)(1 << stream->bit_count);
196     }
197
198     /* バッファ出力・更新 */
199     if (stream->bit_count == 0) {
200         if (fputc(stream->bit_buffer, stream->fp) == EOF) {
201             return BITSTREAM_APIRESULT_IOERROR;
202         }
203         stream->bit_buffer = 0;
204         stream->bit_count = 8;
205     }
206
207     return BITSTREAM_APIRESULT_OK;
208 }
```

1bitだけ出力する場合、その値は即座にファイルに書き出されず、一旦1バイト分のバッファ `stream->bit_buffer` に蓄えられる。bitの書き出しカウント数は `stream->bit_count` に記録されており、このカウントが0になったときに始めてバイト単位の書き出し (`fputc` 関数) が実行される。

複数 bit 出力処理

複数 bit 出力を行う処理をリスト 2.14 に示す。

リスト 2.14 複数 bit 出力処理 (`bit_stream.c`)

```

210  /*
211  * valの右側(下位)n_bits 出力(最大64bit出力可能)
212  * BitStream_PutBits(stream, 3, 6);は次と同じ:
213  * BitStream_PutBit(stream, 1); BitStream_PutBit(stream, 1);
214  *           BitStream_PutBit(stream, 0);
215  */
216 BitStreamApiResult BitStream_PutBits(struct BitStream* stream, uint32_t
217 n_bits, uint64_t val)
218 {
219     /* 引数チェック */
220     if (stream == NULL) {
221         return BITSTREAM_APIRESULT_INVALID_ARGUMENT;
```

```

220     }
221
222     /* 読み込みモードでは実行不可能 */
223     if (stream->flags & BITSTREAM_FLAGS_FILEOPENMODE_READ) {
224         return BITSTREAM_APIRESULT_INVALID_MODE;
225     }
226
227     /* 出力可能な最大ビット数を越えている */
228     if (n_bits > sizeof(uint64_t) * 8) {
229         return BITSTREAM_APIRESULT_INVALID_ARGUMENT;
230     }
231
232     /* 0ビット出力では何もしない */
233     if (n_bits == 0) {
234         return BITSTREAM_APIRESULT_OK;
235     }
236
237     /* valの上位ビットから順次出力
238      * 初回ループでは端数(出力に必要なビット数)分を埋め出力
239      * 2回目以降は8bit単位で出力 */
240     while (n_bits >= stream->bit_count) {
241         n_bits = n_bits - stream->bit_count;
242         stream->bit_buffer |= (uint8_t)BITSTREAM_GETLOWERBITS(stream->
243             bit_count, val >> n_bits);
244         if (fputc(stream->bit_buffer, stream->fp) == EOF) {
245             return BITSTREAM_APIRESULT_IOERROR;
246         }
247         stream->bit_buffer = 0;
248         stream->bit_count = 8;
249     }
250
251     /* 端数ビットの処理:
252      * 残った分をバッファの上位ビットにセット */
253     assert(n_bits <= 8);
254     stream->bit_count -= n_bits;
255     stream->bit_buffer |= (uint8_t)(BITSTREAM_GETLOWERBITS(n_bits, val) <<
256         stream->bit_count);
257 }
```

コメントにもあるようにアプリケーション側で出力対象の値を 1bit 単位に分けて出力をすればリスト 2.14 に相当する処理を実現できるが、その処理の簡略化のために本関数を用意している。

1 バイト (8bit) を超える数値出力の際には、本モジュールはビッグエンディアンで出力する。即ち、出力対象の引数値 *val* の最上位 bit から順に出力をを行う。この処理はリスト 2.15 の *while* ループに表れている。

```

237  /* valの上位ビットから順次出力
238  * 初回ループでは端数(出力に必要なビット数)分を埋め出力
239  * 2回目以降は8bit単位で出力 */
240  while (n_bits >= stream->bit_count) {
241      n_bits = n_bits - stream->bit_count;
242      stream->bit_buffer |= (uint8_t)BITSTREAM_GETLOWERBITS(stream->
243          bit_count, val >> n_bits);
244      if (fputc(stream->bit_buffer, stream->fp) == EOF) {
245          return BITSTREAM_APIRESULT_IOERROR;
246      }
247      stream->bit_buffer = 0;
248      stream->bit_count = 8;
249  }

```

この while ループは、ループの度に 1 バイトの出力をを行う。出力バッファ stream->bit_buffer に BITSTREAM_GETLOWERBITS マクロによって取り出した出力対象の 1 バイトデータを設定し、fputc 関数を実行している。このループにおいて引数変数 n_bits は出力対象の値 val の bit 位置を出力しているかを示すカウンタとなる。また、BITSTREAM_GETLOWERBITS マクロは第二引数の値から、第一引数で指定した数だけ下位 bit から取り出す処理を行う。

while ループを抜けた残りの bit は、リスト 2.16 に示す処理でバッファ stream->bit_buffer に記録され、次回の書き出し処理への備えを行う。

リスト 2.16 残った bit の記録処理 (bit_stream.c)

```

250  /* 端数ビットの処理:
251  * 残った分をバッファの上位ビットにセット */
252  assert(n_bits <= 8);
253  stream->bit_count -= n_bits;
254  stream->bit_buffer |= (uint8_t)(BITSTREAM_GETLOWERBITS(n_bits, val) <<
255      stream->bit_count);

```

2.4.2 bit 単位の取得処理

1bit 取得処理

1bit だけ取得を行う処理をリスト 2.17 に示す。

リスト 2.17 1bit の取得処理 (bit_stream.c)

```

259  /* 1bit取得 */
260  BitStreamApiResult BitStream_GetBit(struct BitStream* stream, uint8_t*
261      bit)
262  {
263      int32_t ch;
264
265      /* 引数チェック */
266      if (stream == NULL || bit == NULL) {

```

```

266     return BITSTREAM_APIRESULT_INVALID_ARGUMENT;
267 }
268
269 /* 読み込みモードでない場合は即時リターン */
270 if (!(stream->flags & BITSTREAM_FLAGS_FILEOPENMODE_READ)) {
271     return BITSTREAM_APIRESULT_INVALID_MODE;
272 }
273
274 /* 入力ビットカウントを1減らし、バッファの対象ビットを出力 */
275 if (stream->bit_count > 0) {
276     stream->bit_count--;
277     (*bit) = (stream->bit_buffer >> stream->bit_count) & 1;
278     return BITSTREAM_APIRESULT_OK;
279 }
280
281 /* 1バイト読み込みとエラー処理 */
282 if ((ch = fgetc(stream->fp)) == EOF) {
283     if (feof(stream->fp)) {
284         /* ファイル終端に達した */
285         return BITSTREAM_APIRESULT_EOS;
286     } else {
287         /* それ以外のエラー */
288         return BITSTREAM_APIRESULT_IOERROR;
289     }
290 }
291
292 /* カウンタとバッファの更新 */
293 stream->bit_count = 7;
294 stream->bit_buffer = (uint8_t)ch;
295
296 /* 取得したバッファの最上位ビットを出力 */
297 (*bit) = (stream->bit_buffer >> 7) & 1;
298
299 return BITSTREAM_APIRESULT_OK;
300 }

```

常に読み込み予定の bit を含むデータ 1 バイト分をバッファ stream->bit_buffer に取得しておき、バッファに残っている bit 数のカウンタ stream->bit_count が残っている限りはそこから 1bit 取り出す。カウンタ stream->bit_count が 0 になったら次のデータを fgetc 関数によって 1 バイト取得してバッファ stream->bit_buffer とカウンタ stream->bit_count を補充し、1bit を取り出す。

複数 bit 取得処理

最後に複数 bit 取得を行う処理をリスト 2.18 に示す。

リスト 2.18 複数 bit 取得処理 (bit_stream.c)

```

302 /* n_bits 取得(最大64bit)し、その値を右詰めして出力 */
303 BitStreamApiResult BitStream_GetBits(struct BitStream* stream, uint32_t

```

```
n_bits, uint64_t *val)
304 {
305     int32_t ch;
306     uint64_t tmp = 0;
307
308     /* 引数チェック */
309     if (stream == NULL || val == NULL) {
310         return BITSTREAM_APIRESULT_INVALID_ARGUMENT;
311     }
312
313     /* 読み込みモードでない場合は即時リターン */
314     if (!(stream->flags & BITSTREAM_FLAGS_FILEOPENMODE_READ)) {
315         return BITSTREAM_APIRESULT_INVALID_MODE;
316     }
317
318     /* 入力可能な最大ビット数を越えている */
319     if (n_bits > sizeof(uint64_t) * 8) {
320         return BITSTREAM_APIRESULT_INVALID_ARGUMENT;
321     }
322
323     /* 最上位ビットからデータを埋めていく
324      * 初回ループではtmpの上位ビットにセット
325      * 2回目以降は8bit単位で入力しtmpにセット */
326     while (n_bits > stream->bit_count) {
327         n_bits -= stream->bit_count;
328         tmp |= BITSTREAM_GETLOWERBITS(stream->bit_count, stream->bit_buffer)
329             << n_bits;
330         /* 1バイト読み込みとエラー処理 */
331         if ((ch = fgetc(stream->fp)) == EOF) {
332             if (feof(stream->fp)) {
333                 /* 途中でファイル終端に達していたら、ループを抜ける */
334                 goto END_OF_STREAM;
335             } else {
336                 /* それ以外のエラー */
337                 return BITSTREAM_APIRESULT_IOERROR;
338             }
339         stream->bit_buffer = (uint8_t)ch;
340         stream->bit_count = 8;
341     }
342
343 END_OF_STREAM:
344     /* 端数ビットの処理
345      * 残ったビット分をtmpの最下位ビットにセット */
346     stream->bit_count -= n_bits;
347     tmp |= (uint64_t)BITSTREAM_GETLOWERBITS(n_bits, (uint32_t)(stream->
348         bit_buffer >> stream->bit_count));
349
350     /* 正常終了 */
351     *val = tmp;
352     return BITSTREAM_APIRESULT_OK;
```

 352 }

取得データは一時変数 `tmp` に対して上位 `bit` から順に書き込まれ、最後に引数 `val` に反映される。処理の構造としては `BitStream_PutBits` とほぼ同様であり、取得 bit 数 `n_bits` が少なくなるまで `while` ループを回し、ループの度に 1 バイト取得を行う。

2.5 ala_coder.c

このモジュールは、予測によって得られた残差信号を符号化してファイルに書き出す処理と、ファイルから残差を復号する処理を分担している。なお、本モジュールのファイル入出力は bit 単位で行う必要があるため、`bit_stream.c` に依存している。

2.5.1 Rice 符号

ALA では Golomb 符号化のパラメータを 2 の幂数に限定している為、Rice 符号のみを考えればよい。`ala_coder.c` では、Rice 符号化と復号処理を 1.2.3 節で示した定義通りに実装している。

Rice 符号化処理

Rice 符号化を行う処理をリスト 2.19 に示す。

リスト 2.19 Rice 符号化処理 (`ala_coder.c`)

```

35 static void ALACoder_PutRiceCode(
36     struct BitStream* strm, uint32_t rice_parameter, uint32_t val)
37 {
38     uint32_t i, quot, rest;
39
40     assert(strm != NULL);
41
42     /* 商と剰余の計算 */
43     quot = val >> ALAUtility_Log2Ceil(rice_parameter);
44     rest = val & (rice_parameter - 1);
45
46     /* 商部分の出力 */
47     for (i = 0; i < quot; i++) {
48         BitStream_PutBit(strm, 0);
49     }
50     BitStream_PutBit(strm, 1);
51
52     /* 剰余部分の出力 */
53     BitStream_PutBits(strm, ALAUtility_Log2Ceil(rice_parameter), rest);
54 }
```

符号化対象の数値 `val` と Rice 符号のパラメータ `rice_parameter` を用いて商 `quot` と

剩餘 `rest` を計算し、商は α 符号（1.2.3節）、剩餘は2進数で出力を行っている。

剩餘 `rest` の計算においては `rice_parameter` は2の冪乗数であることを用いて、AND演算(&)によって剩餘計算を行っている⁸。

`ALAUtility_Log2Ceil` 関数は引数の整数 x に対して $\lceil \log_2(x) \rceil$ を高速に計算する関数であり、`ala_utility.c` に定義がある（リスト2.20）。

リスト 2.20 `ALAUtility_Log2Ceil` 関数 (`ala_utility.c`)

```

56 /* NLZ(最上位ビットから1に当たるまでのビット数)を計算する黒魔術 */
57 /* ハッカーのたのしみ参照 */
58 static uint32_t nlz10(uint32_t x)
59 {
60     x = x | (x >> 1);
61     x = x | (x >> 2);
62     x = x | (x >> 4);
63     x = x | (x >> 8);
64     x = x & ~(x >> 16);
65     x = (x << 9) - x;
66     x = (x << 11) - x;
67     x = (x << 14) - x;
68     return nlz10_table[x >> 26];
69 }
70
71 /* ceil(log2(val)) を計算する */
72 uint32_t ALAUtility_Log2Ceil(uint32_t val)
73 {
74     assert(val != 0);
75     return 32U - nlz10(val - 1);
76 }
```

実装は[37]を引用したものであるが、筆者の力量不足により、このコードが上手く動く理由を説明できない。[17, 36]に解説とソースコードがあるため、参考にしていただきたい。

Rice符号化されたデータの復号処理

Rice符号化されたデータの復号を行う処理をリスト2.21に示す。

リスト 2.21 Rice復号処理 (`ala_coder.c`)

```

57 static uint32_t ALACoder_GetRiceCode(
58     struct BitStream* strm, uint32_t rice_parameter)
59 {
60     uint32_t quot, rest;
61     uint8_t bit;
62
63     assert(strm != NULL);
```

⁸ 整数 val と 2 の冪数 p に対して、 $val \& (p-1)$ は $val \% p$ に等しい。直感的には、 $p-1$ との AND を行なうことは下位 $\log_2(p)$ bit のマスクを取る演算と考えられ、それがまさに剩餘演算に相当している。 $p=2, 4$ あたりで試しに計算を行ってみると良い。

```

64  /* 商部分を取得 */
65  quot = 0;
66  BitStream_GetBit(strm, &bit);
67  while (bit == 0) {
68      quot++;
69      BitStream_GetBit(strm, &bit);
70  }
71 }
72
73 /* 剰余部分を取得 */
74 if (rice_parameter == 1) {
75     /* 1の剰余は0 */
76     rest = 0;
77 } else {
78     /* ライス符号の剰余部分取得 */
79     uint64_t bitsbuf;
80     BitStream_GetBits(strm, ALAUtility_Log2Ceil(rice_parameter), &bitsbuf
81                     );
82     rest = (uint32_t)bitsbuf;
83 }
84 return (rice_parameter * quot + rest);
85 }
```

符号化処理（リスト 2.19）の逆の操作を行っているだけである。まず商 `quot` を α 符号の定義（1 が出現するまでの bit 数）に従って取得し、次に Rice 符号のパラメータを元に剰余 `rest` を取得する。

2.5.2 残差の符号化と復号

残差の符号化と復号処理は、Rice 符号のパラメータを入力データに適応しながら変更し、残差を出力、あるいは取得する処理に要約される。Rice 符号のパラメータの更新処理は式 1.54 で見たように平均値の更新処理に帰着される。平均値は有限のサンプル数では厳密に計算できないので、替わりに推定平均値の計算を行う。符号化と復号処理で全く同じ様に推定平均値を更新することで、パラメータを適応的に変更しながらでも、可逆な符号化が実現できる。

残差の符号化

残差の符号化処理をリスト 2.22 に示す。

リスト 2.22 残差の符号化処理 (ala_coder.c)

```

110 /* 符号付き整数配列の符号化 */
111 ALACoderApiResult ALACoder_PutdataArray(
112     struct ALACoder* coder, struct BitStream* strm,
113     const int32_t** data, uint32_t num_channels, uint32_t num_samples)
```

```

114 {
115     uint32_t smpl, ch, uint;
116
117     /* 引数チェック */
118     if ((strm == NULL) || (data == NULL) || (coder == NULL)) {
119         return ALACODER_APIRESULT_INVALID_ARGUMENT;
120     }
121
122     /* 各チャンネルの平均値をセット/記録 */
123     for (ch = 0; ch < num_channels; ch++) {
124         uint64_t mean_uint = 0;
125         for (smpl = 0; smpl < num_samples; smpl++) {
126             mean_uint += ALAUTILITY_SINT32_TO_UINT32(data[ch][smpl]);
127         }
128         mean_uint /= num_samples;
129         /* 平均の最大は符号無し16bit整数の最大値に制限 */
130         mean_uint = ALAUTILITY_MIN(mean_uint, UINT16_MAX);
131         BitStream_PutBits(strm, 16, mean_uint);
132         coder->estimated_mean[ch] = ALACODER_UINT32_TO_FIXED_FLOAT(mean_uint)
133             ;
134     }
135
136     /* 各チャンネル毎に符号化 */
137     for (ch = 0; ch < num_channels; ch++) {
138         for (smpl = 0; smpl < num_samples; smpl++) {
139             /* 符号なし整数に変換 */
140             uint = ALAUTILITY_SINT32_TO_UINT32(data[ch][smpl]);
141             /* ライス符号化 */
142             ALACoder_PutRiceCode(strm, ALACODER_CALCULATE_RICE_PARAMETER(coder
143                 ->estimated_mean[ch]), uint);
144             /* 推定平均値を更新 */
145             ALACODER_UPDATE_ESTIMATED_MEAN(coder->estimated_mean[ch], uint);
146         }
147     }
148 }
```

符号化処理の前に、推定平均値の初期値を決める必要がある。推定平均値の初期値を設定する処理をリスト2.23に示す。

リスト2.23 推定平均値の初期化 (ala_coder.c)

```

122     /* 各チャンネルの平均値をセット/記録 */
123     for (ch = 0; ch < num_channels; ch++) {
124         uint64_t mean_uint = 0;
125         for (smpl = 0; smpl < num_samples; smpl++) {
126             mean_uint += ALAUTILITY_SINT32_TO_UINT32(data[ch][smpl]);
127         }
128         mean_uint /= num_samples;
129         /* 平均の最大は符号無し16bit整数の最大値に制限 */
130         mean_uint = ALAUTILITY_MIN(mean_uint, UINT16_MAX);
```

```

131     BitStream_PutBits(strm, 16, mean_uint);
132     coder->estimated_mean[ch] = ALACODER_UINT32_TO_FIXED_FLOAT(mean_uint)
133     ;
134 }
```

初期値は適当に 1 等で決め打ちしても処理としては問題ないが、推定平均値の収束を早くするために、入力データの平均値を推定値の初期値に設定している。また、初期値は残差の直前に記録しておく。

処理マクロについて補足する。ALAUTILITY_SINT32_TO_UINT32 マクロは符号付き整数を符号なし整数に変換する処理を行う。Rice 符号は正整数を対象にしているため、符号化にあたってこの変換は必須である。マクロは ala_utility.h において次のように定義される。

リスト 2.24 ALAUTILITY_SINT32_TO_UINT32 マクロ (ala_utility.h)

```

27 /* 符号付き 32bit 数値を符号なし 32bit 数値に一意変換 */
28 #define ALAUTILITY_SINT32_TO_UINT32(sint) (((int32_t)(sint) < 0) ? ((
29     uint32_t)((-((sint) << 1)) - 1)) : ((uint32_t)((sint) << 1)))
```

このマクロによって、正整数は正偶数に、負整数は正奇数に変換される。この変換は絶対値の小さい方から大きい順に変換後の数値が並ぶため、Rice 符号のような符号化対象の数値の絶対値の大きさに追従して出力符号長が長くなる符号にとって都合が良い。符号付き整数を符号なし整数に変換する手法は他にも幾つか存在し、[9] によると、整数を符号 bit とその絶対値に分けて符号化する方法や、符号を負数に対応させる方法が挙げられている。

ALACODER_UINT32_TO_FIXED_FLOAT マクロは整数値を固定小数化するマクロである。マクロの定義をリスト 2.25 に示す。

リスト 2.25 整数値の固定小数化 (ala_coder.c)

```

13 /* 符号なし整数を固定小数に変換 */
14 #define ALACODER_UINT32_TO_FIXED_FLOAT(u32) ((u32) << (
15     ALACODER_NUM_FRACTION_PART_BITS))
```

左シフト演算によって、小数部を 0 にした状態の固定小数点数を作成している。固定小数化する理由としては、平均推定値を更新する際、率直に整数を使うと切り捨てによる精度落ちが無視できないためである。

符号化と推定平均値の更新処理をリスト 2.26 に示す。

リスト 2.26 符号化処理コア部分 (ala_coder.c)

```

135 /* 各チャンネル毎に符号化 */
136 for (ch = 0; ch < num_channels; ch++) {
137     for (smpl = 0; smpl < num_samples; smpl++) {
138         /* 符号なし整数に変換 */
139         uint = ALAUTILITY_SINT32_TO_UINT32(data[ch][smpl]);
```

```

140     /* ライス符号化 */
141     ALACODER_PutRiceCode(strm, ALACODER_CALCULATE_RICE_PARAMETER(coder
142         ->estimated_mean[ch]), uint);
143     /* 推定平均値を更新 */
144     ALACODER_UPDATE_ESTIMATED_MEAN(coder->estimated_mean[ch], uint);
145 }

```

1サンプル毎にサンプルを符号なし整数に変換し、Rice 符号としての出力と、推定平均値の更新処理が行われる。ここで、推定平均値を Rice 符号パラメータに変換する処理は ALACODER_CALCULATE_RICE_PARAMETER マクロが、推定平均値の更新は ALACODER_UPDATE_ESTIMATED_MEAN がその役割を果たす。以下、それらのマクロの説明を行う。

ALACODER_CALCULATE_RICE_PARAMETER マクロは、式 1.54 に基づいて Rice 符号のパラメータを計算するマクロであり、リスト 2.27 に示す形で定義される。

リスト 2.27 ALACODER_CALCULATE_RICE_PARAMETER マクロ (ala_coder.c)

```

21 /* Rice符号のパラメータ計算 2 ** ceil(log2(E(x)/2)) = E(x)/2の2の累乗切り上
22 #define ALACODER_CALCULATE_RICE_PARAMETER(mean) \
23     ALAUtility_RoundUp2Powered(ALAUTILITY_MAX(
        ALACODER_FIXED_FLOAT_TO_UINT32((mean) >> 1), 1UL))

```

式 1.54 をここに再掲する。

$$k = \max \left\{ 0, \left\lceil \log_2 \left(\frac{E[x]}{2} \right) \right\rceil \right\} \quad (\text{式 1.54 を再掲})$$

両辺の 2 の累乗を取った 2^k が計算すべき Rice 符号パラメータとなる。 k は単純には式 1.54 を使えばよいが、2 の対数の切り上げを行う関数 ALAUtility_Log2Ceil は負荷が高い^{*9}。そこで、 2^k を以下の式 2.9 によって計算している。

$$\begin{aligned} 2^k &= \max \left\{ 2^0, 2^{\lceil \log_2(\frac{E[x]}{2}) \rceil} \right\} = \max \left\{ 1, \text{clp} \left(\frac{E[x]}{2} \right) \right\} \\ &= \text{clp} \left(\max \left\{ 1, \frac{E[x]}{2} \right\} \right) \end{aligned} \quad (2.9)$$

ここで、 $\text{clp}(x) = 2^{\lceil \log_2(x) \rceil}$ は x を 2 の累乗数に切り上げる関数である [17]。 $\lceil \log_2(x) \rceil$ は x を 2 進数表示するのに十分な bit 数であり、 $2^{\lceil \log_2(x) \rceil}$ により x よりも大きい最小の 2 の累乗数が得られる。 $\text{clp}(x)$ は $\lceil \log_2(x) \rceil$ の計算を介さずに直接計算ができるため、若干の速度向上が期待できる。

^{*9} パラメータ更新処理は符号化対象の全てのサンプルで 1 回ずつ実行されるため、負荷へのインパクトが大きい。

ALACODER_UPDATE_ESTIMATED_MEAN マクロは、推定平均値を更新するマクロであり、リスト 2.28 に示す形で定義される。

リスト 2.28 ALACODER_UPDATE_ESTIMATED_MEAN マクロ (ala_coder.c)

```
17 /* 推定平均値の更新マクロ(指數移動平均により推定平均値を更新) */
18 #define ALACODER_UPDATE_ESTIMATED_MEAN(mean, uint) { \
19     (mean) = (ALACoderFixedFloat)(119 * (mean) + 9 * \
20         ALACODER_UINT32_TO_FIXED_FLOAT(uint) + (1UL << 6)) >> 7; \
20 }
```

リスト 2.28 は、現在の推定平均値 $E[x]$ と新しい入力値 x に対して、更新後の推定平均値 $E'[x]$ を以下の式 2.10 によって計算している。

$$E'[x] = \frac{119E[x] + 9x}{128} = \frac{119}{128}E[x] + \frac{9}{128}x \quad (2.10)$$

右シフトによる切り捨て誤差の影響を緩和するため、小数の 0.5 に相当する ($1UL << 6$) を加算している。

式 2.10 は指數移動平均による平均値の推定式に他ならない。音声信号は画像に比べ信号の bit 幅 (ダイナミックレンジ) が広いため、信号の変化が急激になりやすい。筆者は、単純な移動平均を用いていると、信号変化の傾向についていけなくなる可能性を踏まえ、応答性の高い平均を求める意図で更新式 2.10 を使用している。

残差の復号

残差の復号処理をリスト 2.29 に示す。

リスト 2.29 残差の復号処理 (ala_coder.c)

```
150 /* 符号付き整数配列の復号 */
151 ALACoderApiResult ALACoder_GetdataArray(
152     struct ALACoder* coder, struct BitStream* strm,
153     int32_t** data, uint32_t num_channels, uint32_t num_samples)
154 {
155     uint32_t ch, smpl, uint;
156
157     /* 引数チェック */
158     if ((strm == NULL) || (data == NULL) || (coder == NULL)) {
159         return ALACODER_APIRESULT_INVALID_ARGUMENT;
160     }
161
162     /* 平均値初期値の取得 */
163     for (ch = 0; ch < num_channels; ch++) {
164         uint64_t bitsbuf;
165         BitStream_GetBits(strm, 16, &bitsbuf);
166         coder->estimated_mean[ch] = ALACODER_UINT32_TO_FIXED_FLOAT(bitsbuf);
167     }
168 }
```

```

169  /* 各チャンネル毎に復号 */
170  for (ch = 0; ch < num_channels; ch++) {
171      for (smpl = 0; smpl < num_samples; smpl++) {
172          /* ライス符号を復号 */
173          uint = ALACoder_GetRiceCode(strm, ALACODER_CALCULATE_RICE_PARAMETER
174              (coder->estimated_mean[ch]));
175          /* 推定平均値を更新 */
176          ALACODER_UPDATE_ESTIMATED_MEAN(coder->estimated_mean[ch], uint);
177          /* 符号付き整数に変換 */
178          data[ch][smpl] = ALAUTILITY_UINT32_TO_SINT32(uint);
179      }
180  }
181  return ALACODER_APIRESULT_OK;
182 }
```

先頭に記録された平均値の初期値を取得し、残差を復号しながら推定平均値を符号化の時と同様に更新する。大まかな処理内容について難しい点はないため省略するが、一点 ALAUTILITY_UINT32_TO_SINT32 マクロは補足が必要である。このマクロは ALAUTILITY_SINT32_TO_UINT32 マクロで変換された符号なし整数を符号付き整数に戻すマクロであり、ala_utility.h のリスト 2.30 で定義される。

リスト 2.30 ALAUTILITY_UINT32_TO_SINT32 マクロ (ala_utility.h)

```

29 /* 符号なし32bit数値を符号付き32bit数値に一意変換 */
30 #define ALAUTILITY_UINT32_TO_SINT32(uint) ((int32_t)((uint) >> 1) ^ -(  

31     int32_t)((uint) & 1))
```

$-(\text{int32_t})((\text{uint}) \& 1)$ は uint が奇数のときに-1(0xFFFFFFFF)、偶数のときに0と評価される。この値と $((\text{uint}) >> 1)$ の \wedge (XOR) を取ると、 uint が奇数のときは $(\text{uint} >> 1)$ を bit 反転した数値が得られ、これは負数を2の補数表現に従って表現した場合 $-((\text{uint} >> 1) + 1)$ と等しくなる。一方、 uint が偶数のときは $(\text{uint} >> 1)$ がそのまま得られる。この結果に従うと $\text{uint} = 0, 1, 2, 3, 4, \dots$ をマクロに適用すると順次 $0, -1, 1, -2, 2, \dots$ という結果が得られ、ALAUTILITY_SINT32_TO_UINT32 マクロの逆の変換が得られていることが分かる。

2.6 wav.c

本モジュールは wav ファイルの入出力を行う機能を提供する。bit 単位の入出力処理を行っているが、bit_stream.c を使用しておらず独自の入出力処理を実装している。これは、他の用途で転用できるようにするためである。

本稿では wav ファイルと本モジュールの詳細な解説は省略し、wav.h で宣言されている構造体と API (関数) の機能についての説明に留める。本モジュールは簡易実装に過ぎず、全ての wav ファイルの入出力には対応していないことに注意されたい。本モジュー

ルは音声データを示す `data` チャンク以外は全て無視して処理を行う。

2.6.1 構造体定義

`wav` ファイルを扱うための構造体 `WAVFile` はリスト 2.31 で定義される。

リスト 2.31 `wav` ファイルハンドル (`wav.h`)

```
32 /* WAVファイルハンドル */
33 struct WAVFile {
34     struct WAVFormat format; /* フォーマット */
35     WAVPcmData** data; /* 実データ */
36 };
```

`wav` ファイルに関する操作(関数)はこの構造体を介して行う。`WAVFile` は、`wav` ファイルのフォーマット情報が入った `WAVFormat` と、波形データ本体である `WAVPcmData` から構成される。`WAVFormat` の定義はリスト 2.32 に示す通り。

リスト 2.32 `wav` ファイルフォーマット (`wav.h`)

```
23 /* WAVファイルフォーマット */
24 struct WAVFormat {
25     WAVDataFormat data_format; /* データフォーマット */
26     uint32_t num_channels; /* チャンネル数 */
27     uint32_t sampling_rate; /* サンプリングレート */
28     uint32_t bits_per_sample; /* 量子化ビット数 */
29     uint32_t num_samples; /* サンプル数 */
30 };
```

波形の情報を示す情報が詰まっている。波形データフォーマット `WAVDataFormat` は現状 PCM フォーマットのみしか対応していない。波形データの型の実体は `int32_t` であることを踏まえると、本モジュールは整数型(最大 32bit)の PCM のみサポートしていることに注意。

波形データへの読み書きは、リスト 2.33 の `WAVFile_PCM` を使用できる。(直接配列にアクセスしても良い)

リスト 2.33 波形データアクセサ (`wav.h`)

```
38 /* アクセサ */
39 #define WAVFile_PCM(wavfile, samp, ch) (wavfile->data[(ch)][(samp)])
```

サンプル位置 `samp` とチャンネル `ch` を指定してデータへのアクセスを行う。ここでも一点注意。`wav` モジュールを扱う側で bit 幅を意識させないように、波形データは一律で **32bit** 整数で取得される。波形が 32bit より小さい量子化 bit 数であっても、**32bit** 幅に拡張される。例えば、量子化 bit 数が 16bit の波形データにおいて、0x7FFF は 32bit 整数に(左 16bit シフトして)拡張され 0x7FFF0000 で取得される。一方、`wav` ファイルへ

の書き込みの際、0x7FFF0000 と書き込んだデータは 16bit 整数に（右 16bit シフトして）変換され、0x7FFF に丸められて書き込まれる。

2.6.2 公開 API

wav ファイルの読み込み

wav ファイルの読み込みを行う関数をリスト 2.34 に示す。

リスト 2.34 wav ファイル読み込み (wav.h)

```
45 /* ファイルからWAVファイルハンドルを作成 */
46 struct WAVFile* WAV_CreateFromFile(const char* filename);
```

引数としてファイル名を指定し、読み込みに成功した場合は **WAVFile** 構造体へのポインタが返り値として得られる（読み込みに失敗した場合は NULL が返る）。

なお、**WAVFile** 構造体は波形フォーマット情報 **WAVFormat** だけから新しく作ることもできる。wav ファイルの新規作成時に使用する。

リスト 2.35 WAVFile を波形フォーマット指定で新規作成 (wav.h)

```
48 /* フォーマットを指定して新規にWAVファイルハンドルを作成 */
49 struct WAVFile* WAV_Create(const struct WAVFormat* format);
```

wav ファイルの書き込み

WAVFile をファイル名指定で書き込む関数をリスト 2.36 に示す。書き込みの成否は返り値で得られる。

リスト 2.36 wav ファイル書き込み (wav.h)

```
54 /* ファイル書き出し */
55 WAVApiResult WAV_WriteToFile(
56     const char* filename, const struct WAVFile* wavfile);
```

2.6.3 使用例

本モジュールの簡単な使用例として、wav ファイルの量子化 bit 幅を変換して書き出すプログラム **wavbitchange.c** をリスト 2.37 に示す。

リスト 2.37 wav ファイルの bit 幅変換 (wavbitchange.c)

```
1 #include "wav.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char** argv)
6 {
```

```
7   struct WAVFile* wavfile;
8
9   /* 引数の数が誤っている場合は使用方法を表示 */
10  if (argc != 4) {
11      fprintf(stderr, "Usage: program bits infile outfile\n");
12      return 1;
13  }
14
15  /* wavファイルの読み込み */
16  if ((wavfile = WAV_CreateFromFile(argv[2])) == NULL) {
17      fprintf(stderr, "Failed to get wav data.\n");
18      return 1;
19  }
20
21  /* 変換後のbitの取得 */
22  wavfile->format.bits_per_sample = atoi(argv[1]);
23  if ((wavfile->format.bits_per_sample != 8)
24      && (wavfile->format.bits_per_sample != 16)
25      && (wavfile->format.bits_per_sample != 24)
26      && (wavfile->format.bits_per_sample != 32)) {
27      fprintf(stderr, "Wavfile bps must be 8/16/24/32.\n");
28      return 1;
29  }
30
31  /* bitを変更して保存 */
32  WAV_WriteToFile(argv[3], wavfile);
33
34  /* ハンドル破棄 */
35  WAV_Destroy(wavfile);
36
37  return 0;
38 }
```

2.7 main.c

main 関数が含まれる main.c は、引数処理を行ってエンコード処理とデコード処理を呼び分ける。エンコード処理およびデコード処理は、表 2.1, 2.2 のフォーマットに従ってバイナリファイルの読み書きを実行しているに過ぎない。

2.7.1 ALA のエンコード処理

ALA のエンコード処理手順は、次のように要約される。

ALA のエンコード処理手順 (main.c の do_encode 関数)

1. 入力の wav ファイルデータを取得する。
2. ヘッダ情報を書き出す。
3. ブロックのエンコード処理を行う。以下の処理を wav ファイルデータの末尾に達するまで繰り返す。
 - (a) wav のチャンネル数がステレオ (2) 以上であれば、MS 処理を行う。
 - (b) double 形式のデータにプリエンファシスと窓掛けを行ってから、PARCOR 係数を計算する。
 - (c) 整数データにプリエンファシスを適用する。
 - (d) 整数データに PARCOR 格子型フィルターによる予測を行い、残差を算出する。
 - (e) 同期コードと PARCOR 係数をエンコードする。
 - (f) 残差をエンコードする。

大まかな処理内容は上記内容で十分と思われるため、細かい補足について触れていく。

入力 wav データの取得

入力 wav データは整数型で取得すると同時に、解析用に高精度な double 型に変換している（リスト 2.38）。

リスト 2.38 wav データ取得 (main.c)

```

100  /* 入力データ取得 */
101  for (ch = 0; ch < num_channels; ch++) {
102    for (smpl = 0; smpl < num_samples; smpl++) {
103      input[ch][smpl] = WAVFile_PCM(in_wav, smpl, ch) * pow(2, -31);
104      input_int32[ch][smpl] = WAVFile_PCM(in_wav, smpl, ch);
105      /* 情報が失われない程度に右シフト */
106      input_int32[ch][smpl] >>= (32 - in_wav->format.bits_per_sample);
107    }
108  }
```

既に 2.6.1 節で注意したが、WAVFile 構造体に含まれる波形データは 32bit 整数型で記録されている。このまま PARCOR 係数の解析を行うと（特に自己相関関数の）数値が大きくなり過ぎてしまい、精度落ちの可能性がある^{*10}。そのため、解析用のデータは、整数型データに 2^{-31} を乗じた double 型のデータとして記録する。

エンコードするサンプル数の決定

エンコードするサンプル数は、リスト 2.39 に示す式で決定している。

^{*10} IEEE 754 の規格に従う浮動小数点数は、0 近傍に多くの数値が含まれるため数値表現精度が高いが、0 から離れるに従って精度が落ちる [38]。

リスト 2.39 エンコードするサンプル数の決定 (main.c)

```
136  /* エンコードするサンプル数の決定 */
137  num_encode_samples = ALAUTILITY_MIN(ALA_NUM_SAMPLES_PER_BLOCK,
138      num_samples - enc_offset_sample);
```

ブロックに含まれるサンプル数は、ファイル末尾を除き、リスト 2.40 で定義される `ALA_NUM_SAMPLES_PER_BLOCK` マクロで固定である。

リスト 2.40 `ALA_NUM_SAMPLES_PER_BLOCK` マクロ (main.c)

```
25 /* ブロックあたりサンプル数 */
26 #define ALA_NUM_SAMPLES_PER_BLOCK 4096
```

式 `num_samples - enc_offset_sample` は残りサンプル数を示すため、`ALA_NUM_SAMPLES_PER_BLOCK` との最小値を取ることで、通常は固定サンプル数、末尾は残りサンプル数でエンコードを行える。

窓掛け

PARCOR 係数を計算する前に、分析精度を高めるために窓関数 (window function) を適用する。デジタル信号処理では、窓関数を適用することを窓掛けと呼ぶことがある。

窓関数は短時間の信号に適用する関数である。形式的には、窓関数 $w(n)$ は信号の分析区間長 N に対し $n = 0, \dots, N - 1$ で定義される関数であり、 $w(n)$ の値に制限を設けていないため、様々な定義が存在する。窓関数の適用処理は、窓関数 $w(n)$ と分析対象の信号 $x(n)$ の要素積を取る。即ち、窓関数を適用した信号を $y(n)$ と書くと、

$$y(n) = w(n)x(n) \quad n = 0, \dots, N - 1 \quad (2.11)$$

と表すことができる。

何故窓関数を使用する必要があるのか概略を述べる。デジタル信号の分析においては、分析対象の信号区間に周期性を仮定することが多い。現実の信号の信号区間を切り出してその区間に周期性を仮定すると、区間の左端と右端が不連続だった場合に周波数分析結果にスペクトル漏れ等の歪みが発生する [10]。スペクトル漏れとは、本来存在すべき周波数の成分が他の周波数成分に漏れてしまう現象である。このような現象に対策するために窓関数が導入される。多くの窓関数は区間の端点で 0 (あるいは 0 に近い数値) を取るため、式 2.11 から分かるように区間の端点が 0 に近くなり不連続性とスペクトル漏れを緩和することができる。

当然、使用する窓関数により分析結果は変化する。応用においては、分析対象の音声と用途に合わせて窓関数を選択するのは無論のこと、音声コーデックによっては独自の窓関数を設計することすらある [39]。窓関数は奥が深く、本稿で窓関数の詳細を説明するのは荷が重い。窓関数についての詳細な説明は各種信号処理の本を参照いただきたい [5, 6, 10, 11]。

実装の説明に移る。ALA は性能の良さと処理の単純さから、式 2.12 のサイン窓を使用している。

$$w(n) = \sin\left(\frac{\pi n}{N-1}\right) \quad (2.12)$$

ブロック単位で窓関数を適用する処理をリスト 2.41 に示す。

リスト 2.41 窓関数の適用処理 (main.c)

```

151  /* 窓の作成 */
152  ALAUtility_MakeSinWindow(window, num_encode_samples);
153  /* 窓掛け */
154  for (ch = 0; ch < num_channels; ch++) {
155      ALAUtility_ApplyWindow(window, input_ptr[ch], num_encode_samples);
156 }
```

`ALAUtility_MakeSinWindow` 関数は式 2.12 の定義式に従って窓関数係数 $w(n)$ を生成し、`ALAUtility_ApplyWindow` 関数は式 2.11 に従って、1 チャンネル分のデータに対して窓関数を適用している。

PARCOR 係数の量子化

`double` 精度で求めた PARCOR 係数を、リスト 2.42 に示す処理で小数部が 15bit の固定小数点数へ変換（量子化）する。

リスト 2.42 PARCOR 係数の量子化 (main.c)

```

171  /* PARCOR係数量子化 */
172  for (ch = 0; ch < num_channels; ch++) {
173      /* PARCOR係数の0次成分は0.0のはずなので処理をスキップ */
174      parcor_coef_int32[ch][0] = 0;
175      for (ord = 0; ord < ALA_PARCOR_ORDER; ord++) {
176          /* 整数へ丸める */
177          parcor_coef_int32[ch][ord]
178              = (int32_t)ALAUtility_Round(parcor_coef[ch][ord] * pow(2.0f,
179                  15));
180          /* roundによる丸めによりビット幅をはみ出してしまうことがあるので範囲制限
181          */
182          parcor_coef_int32[ch][ord]
183              = ALAUTILITY_INNER_VALUE(parcor_coef_int32[ch][ord], INT16_MIN,
184                  INT16_MAX);
185      }
186  }
```

1.1.2 節で見たように、PARCOR 係数の値域は理論的に最小 -1 、最大 1 に制限されている。この数値を小数部が 15bit の固定小数に変換するには、式 2.2 で見たように、 2^{15} を乗じて丸めれば良い。ただし、C89 準拠の環境では `round` 関数が存在しないため、ALA では独自実装（`ALAUtility_Round` 関数）を使用している。

PARCOR 係数のエンコード

量子化した PARCOR 係数は、リスト 2.43 に示す処理でエンコードする。

リスト 2.43 PARCOR 係数のエンコード (main.c)

```

207  /* 各チャネルのPARCOR係数 */
208  for (ch = 0; ch < num_channels; ch++) {
209      /* 0次係数は0だから飛ばす */
210      for (ord = 1; ord < ALA_PARCOR_ORDER + 1; ord++) {
211          BitStream_PutBits(out_strm, 16, ALAUTILITY_SINT32_TO_UINT32(
212              parcor_coef_int32[ch][ord]));
213      }

```

PARCOR 係数をチャネルごとに 16bit の列で書き出す。小数部が 15bit の固定小数点数を記録するためには、符号 bit を付け加えると 16bit 必要になる。

ブロック末尾のバイト境界揃え

1 ブロックの残差のエンコードが終わった後、BitStream_Flush 関数を実行して、ブロックのバイナリデータ終端をバイト単位に揃えている（リスト 2.44）。

リスト 2.44 ブロック末尾のバイト境界揃え (main.c)

```

218  /* バイト境界に揃える */
219  BitStream_Flush(out_strm);

```

この処理は必須ではないが、ブロックがバイト境界に揃っていれば標準ライブラリ関数を用いたバイト単位の読み取りがしやすくなり、また、同期コードがバイナリエディタで観認しやすくなる。

2.7.2 ALA のデコード処理

ALA のデコード処理手順は、次のように要約される。

ALA のデコード処理手順 (main.c の do_decode 関数) —

1. 入力のバイナリデータを開く。
2. ヘッダ情報を取得する。
3. ブロックのデコード処理を行う。以下の処理をヘッダから取得したサンプル数に達するまで繰り返す。
 - (a) 同期コードをチェックし、PARCOR 係数を取得する。
 - (b) 残差をデコードする。
 - (c) PARCOR 格子型フィルターによる合成を行う。
 - (d) デエンファシスを適用する。
 - (e) チャンネル数がステレオ (2) 以上であれば、MS を LR に戻す。
4. 復元した波形データから wav ファイルを出力する。

ヘッダを取得した後、各ブロックのデコード処理は、エンコードの処理手順を逆転したものであると言える。エンコード処理の説明（2.7.1 節）と同じく、以下では補足説明を入れる。

シグネチャとフォーマットバージョンのチェック

ヘッダ情報の取得の際に、同時にリスト 2.45 に示す処理でシグネチャとフォーマットバージョンのチェックを行う。

リスト 2.45 シグネチャとフォーマットバージョンのチェック (main.c)

```

283  /* シグネチャ */
284  BitStream_GetBits(in_strm, 32, &bitsbuf);
285  /* シグネチャの確認 */
286  if ( (((bitsbuf >> 24) & 0xFF) != 'A')
287      || (((bitsbuf >> 16) & 0xFF) != 'L')
288      || (((bitsbuf >> 8) & 0xFF) != 'A')
289      || (((bitsbuf >> 0) & 0xFF) != '\0')) {
290      fprintf(stderr, "Invalid_signature.\n");
291      return 1;
292  }
293  /* フォーマットバージョン */
294  BitStream_GetBits(in_strm, 16, &bitsbuf);
295  /* シグネチャの確認 */
296  if (bitsbuf != ALA_FORMAT_VERSION) {
297      fprintf(stderr, "Unsupported_format_version:%d\n", (uint16_t)bitsbuf
298      );
299  }

```

BitStream_GetBits 関数で 32bit のシグネチャを一気に読み取り、1 バイトずつシグネチャ文字との一致を確認する。得られるバイナリデータはビッグエンディアンのため、上位 bit から順に調べる。また、取得したフォーマットバージョン番号が

ALA_FORMAT_VERSION マクロ定数と不一致だった場合は、無条件でデコード失敗とする。ALA バージョン 1.0.0 の段階では、ALA_FORMAT_VERSION マクロはリスト 2.46 で定義している。

リスト 2.46 ALA_FORMAT_VERSION マクロ (main.c)

```
22 /* フォーマットバージョン */
23 #define ALA_FORMAT_VERSION 1
```

デコードするサンプル数の決定

デコードするサンプル数は、リスト 2.47 に示す式で決定している。

リスト 2.47 デコードするサンプル数の決定 (main.c)

```
361 /* デコードサンプル数の確定 */
362 num_decode_samples = ALAUTILITY_MIN(num_block_samples, num_samples -
dec_offset_sample);
```

ヘッダから取得したブロックサンプル数 num_block_samples を基に、エンコードと全く同じ処理でデコードサンプル数を決定する。

2.7.3 引数処理

main 関数はコマンドラインの引数を処理し、エンコード (do_encode 関数) 処理とデコード (do_decode 関数) 処理を呼び分ける。main 関数の処理をリスト 2.48 に示す。

リスト 2.48 main 関数 (main.c)

```
441 /* メインエントリ */
442 int main(int argc, char** argv)
443 {
444     const char* option;
445     const char* input_file;
446     const char* output_file;
447
448     /* 引数が足らない */
449     if (argc < 4) {
450         print_usage(argv);
451         return 1;
452     }
453
454     /* 引数文字列の取得 */
455     option = argv[1];
456     input_file = argv[2];
457     output_file = argv[3];
458
459     /* エンコード/デコード呼び分け */
460     if (strcmp(option, "-e") == 0) {
```

```

461     if (do_encode(input_file, output_file) != 0) {
462         fprintf(stderr, "Failed to encode.\n");
463         return 1;
464     }
465 } else if (strcmp(option, "-d") == 0) {
466     if (do_decode(input_file, output_file) != 0) {
467         fprintf(stderr, "Failed to decode.\n");
468         return 1;
469     }
470 } else {
471     print_usage(argv);
472     return 1;
473 }
474
475     return 0;
476 }
```

コマンドラインの第一引数 (`argv[1]`) にオプション文字列が入っているため、それが"`-e`"と一致するならば `do_encode` 関数を実行し、"`-d`"と一致するならば `do_decode` 関数を実行する。

2.8 FLACとの比較

表1.5で挙げた wav ファイルに対して、ロスレス音声コーデック FLAC と ALA の圧縮率比較を行った。圧縮後のサイズ比較結果を表 2.5 に、圧縮率の比較結果を表 2.6 に示す。圧縮率は (圧縮後のサイズ/圧縮前のサイズ) × 100 により計算している。

使用した FLAC のバージョンは 1.3.2 である。また、FLAC には圧縮率を決めるオプションが存在するが、比較においては全てデフォルト（指定なし）を使用している。

表 2.5 FLAC と ALA の圧縮サイズ比較

ファイル名	元サイズ	ALA[byte]	FLAC[byte]
4-02 井戸の茶椀.wav	306378284	103688028	68009913
1-01 火焰太鼓.wav	297902124	142929310	128413795
02 My Song.wav	71360044	32529466	31704702
09 瑠璃子.wav	35781164	13089913	13511689
30-自分だけの輝き.wav	28811392	13444905	13486930
29-重ねる努力.wav	21694240	11691313	11684283
4-02 Let's アイカツ!(Short サイズ).wav	22819884	17378477	17197100
5-02 カレンダーガール(TV-size).wav	18321964	14020229	13884712

表 2.6 FLAC と ALA の圧縮率比較

ファイル名	ALA[%]	FLAC[%]
4-02 井戸の茶椀.wav	33.8	22.2
1-01 火焰太鼓.wav	48.0	43.1
02 My Song.wav	45.6	44.4
09 瑠璃子.wav	36.6	37.8
30-自分だけの輝き.wav	46.7	46.8
29-重ねる努力.wav	53.9	53.9
4-02 Let's アイカツ!(Short サイズ).wav	76.1	75.4
5-02 カレンダーガール(TV-size).wav	76.5	75.8

2.8.1 所感

FLAC の圧縮率を超えたとは言えないのが正直な所感である。全体的に FLAC の方が優れた圧縮率を示した。特に、4-02 井戸の茶椀.wav は圧縮率換算で 10% 以上の差が発生している。この差が生じた原因について調査した所、4-02 井戸の茶椀.wav は MS 処理の結果 Side 成分が完全に無音になっており^{*11}、その無音の符号化において差が出ていることが判明した。ALA では無音でも Rice 符号で符号化を行うが、FLAC では無音を含む定数信号に対してランレングス符号化を行う [29] ため、無音部分の圧縮率が高くなり、結果有意な差が発生したものと考えられる。

09 瑠璃子.wav や 30-自分だけの輝き.wav においては僅かではあるが ALA の方が圧縮率が良い。両音源ともにピアノがメインの音源であるから、周波数構造が明確に表れている音源に対しては FLAC とそこそくに対抗できる実力があると思われる。

2.9 性能向上の指針

2.8 節で見たように、ALA が FLAC を超える圧縮率を持つためには更なる工夫が必要である。本節では、圧縮率を高めるための知見や筆者の失敗談について触れる。

2.9.1 他のロスレス音声コーデックの概略

ロスレス音声コーデックの今後の性能改善のためには、まず現状を知ることが重要と思われる。本節では、FLAC を始めとした代表的なロスレス音声コーデックについて、その

^{*11} つまり、モノラル音源をアップミックスしてステレオ化した音源。

概要とアルゴリズムを簡単に説明する。ロスレス音声コーデックの比較を行ったサイトとしては、[43, 44, 45, 46, 48] が挙げられる。特に [46] はロスレス音声コーデック自体の存在意義を説いているため、新しくコーデックを作る際は一読すべきである。

FLAC(Free Lossless Audio Codec)

FLAC[28] は 2019 年時点で最もよく知られたロスレス音声コーデックである。実装はオープンかつライセンスは BSD 系の為、商用利用しやすい^{*12}。圧縮率は他のコーデックと比べて劣るが、アルゴリズムが簡潔であり、非常にデコード速度が早い。アルゴリズムは基本的には線型予測と Rice 符号の組み合わせからなる。Rice 符号のパラメータを特定のサンプル単位で調節することで、圧縮率の向上を図っている。

Wavpack

Wavpack[27] は平均して FLAC よりも圧縮率が高く、またロッシー符号化を兼ねたハイブリッドモード等、多機能であることで知られる。実装はオープンであり、使用技術についても Web に公開されている [26]。内部的な理論としては、予測は適応フィルター (Sign-Sign LMS ベース)、符号化は Golomb 符号化を改良した符号を使用している。

TTA(The True Audio)

TTA[27] も平均して FLAC よりも圧縮率が高い。予測は適応フィルターベースの手法を用いることで線形予測ベースの手法に必要な係数計算の手間を省き、高速なエンコード速度を誇る。符号化は Rice 符号ベースの手法を用いている。実装がシンプルで他のコーデックに比べて読みやすいため、実用レベルの実装を知る意味ではまず TTA を参照すべきである。

MPEG-4 ALS(The MPEG-4 Audio Lossless Coding)

MPEG4-ALS[16] は MPEG-4 標準に含まれるロスレス音声コーデックである。圧縮率と処理速度において飛び抜けた点は無い。基本となる技術は、予測においては線形予測、符号化においては Rice 符号の他、圧縮率の高い算術符号ベースの手法も使用できる。要素技術が非常に多く、それらを適切に選択することで高い圧縮率を達成できるが、エンコーダーを扱う側が習熟する必要があると思われる。実装はオープンではある [34] もの、開発の一端を担った NTT が特許を多数取得しているため、実装にあたっては注意が必要である。

^{*12} ただし、コアライブラリ以外は GPL ライセンスが適用されているため注意。

Monkey's Audio

Monkey's Audio[35] はロスレス音声コーデックの中でも圧縮率が高い^{*13}ことが知られ、しばしば他コーデックの圧縮率と比較される。Monkey's Audio は音声データをブロックに分けずに符号化するため、ストリーミングエンコード/デコードが事実上できない。高压縮率の秘訣はブロックに分けないことで達成していると思われる。実装はオープンであるものの、筆者としては難解であり読み解くことができなかった。

TAK(Tom's verlustfreier Audiokompressor)

TAK[32] は Monkey's Audio に並ぶ圧縮率と FLAC 並のデコード速度を持つ非常に高性能なコーデックである。事実上、2019 年の段階では最強のロスレス音声コーデックであると言える。しかし、TAK は実装がクローズの上、商用利用を禁止している。更に、TAK が公式に配布している SDK には Windows 向け exe とライブラリが含まれるだけ^{*14}であり、マルチプラットフォーム対応ができていない。ffmpeg がリバースエンジニアリングしたソースが公開されている [42] が、筆者はこの実装をまだ理解しきっていない。

2.9.2 予測手法

予測についての取り組みを本節で述べる。

Burg 法

ALA では自己相関を計算することによる Levinson-Durbin 法ベースの線形予測を使用していたが、他にも線形予測手法の別の定式化が存在する。特に Burg 法は自己相関関数ベースの手法よりもより精度の高い解析ができる [41] 事が知られている。Burg 法は前向き誤差と後ろ向き誤差の二乗和の最小化により定式化される。

筆者は [40] を参考に、リスト 2.49 に示すように C 言語版の簡易実装を行った。

リスト 2.49 Burg 法の C 言語実装

```

1 #include <math.h>
2 #include <stdint.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <float.h>
7 #include <assert.h>
8

```

^{*13} 圧縮率の高いロスレス音声コーデックとしては他に La[30] と OptimFROG[31] が挙げられるが、エンコード/デコード速度が異常に遅いため、ここでは解説しない。

^{*14} wine を使用すれば Linux 環境で動かすことは可能。

```
9 static void burg_algorithm(const double* data, uint32_t num_samples,
10   double* coef, uint32_t coef_order)
11 {
12   uint32_t i, k;
13   double *a_vec;
14   double *f_vec;
15   double *b_vec;
16   double Dk, mu;
17   double tmp1, tmp2;
18
19   /* ベクトル領域割り当て */
20   a_vec = malloc(sizeof(double) * (coef_order + 1));
21   parcor_coef = malloc(sizeof(double) * (coef_order + 1));
22   f_vec = malloc(sizeof(double) * num_samples);
23   b_vec = malloc(sizeof(double) * num_samples);
24
25   /* 各ベクトル初期化 */
26   for (k = 0; k < coef_order + 1; k++) {
27     a_vec[k] = 0.0f;
28   }
29   a_vec[0] = 1.0f;
30   memcpy(f_vec, data, sizeof(double) * num_samples);
31   memcpy(b_vec, data, sizeof(double) * num_samples);
32
33   /* Dkの初期化 */
34   Dk = 0.0f;
35   for (i = 0; i < num_samples; i++) {
36     Dk += 2.0f * f_vec[i] * f_vec[i];
37   }
38   Dk -= f_vec[0] * f_vec[0] + b_vec[num_samples - 1] * b_vec[num_samples
39   - 1];
40
41   /* Burg 再帰アルゴリズム */
42   for (k = 0; k < coef_order; k++) {
43     /* 反射(PARCOR)係数の計算 */
44     mu = 0.0f;
45     for (i = 0; i < num_samples - k - 1; i++) {
46       mu += f_vec[i + k + 1] * b_vec[i];
47     }
48     mu *= -2.0f / Dk;
49     parcor_coef[k] = -mu;
50     assert(fabs(parcor_coef[k]) < 1.0f);
51
52     /* a_vecの更新 */
53     for (i = 0; i <= (k + 1) / 2; i++) {
54       tmp1 = a_vec[i]; tmp2 = a_vec[k + 1 - i];
55       a_vec[i] = tmp1 + mu * tmp2;
56       a_vec[k + 1 - i] = mu * tmp1 + tmp2;
57     }
58 }
```

```
58     /* f_vec, b_vecの更新 */
59     for (i = 0; i < num_samples - k - 1; i++) {
60         tmp1 = f_vec[i + k + 1]; tmp2 = b_vec[i];
61         f_vec[i + k + 1] = tmp1 + mu * tmp2;
62         b_vec[i] = mu * tmp1 + tmp2;
63     }
64
65     /* Dkの更新 */
66     Dk = (1.0f - mu * mu) * Dk - f_vec[k + 1] * f_vec[k + 1] - b_vec[
67         num_samples - k - 2] * b_vec[num_samples - k - 2];
68 }
69
70     /* 係数コピー */
71     memcpy(coef, a_vec, sizeof(double) * (coef_order + 1));
72
73     free(b_vec);
74     free(f_vec);
75     free(a_vec);
76     free(parcor_coef);
77 }
78
79 int main(void)
80 {
81     const uint32_t num_samples = 32 * 8192;
82     const uint32_t coef_order = 256;
83     uint32_t i, j;
84     double* data;
85     double* predict;
86     double* coef;
87     double err;
88
89     data = malloc(sizeof(double) * num_samples);
90     predict = malloc(sizeof(double) * num_samples);
91     coef = malloc(sizeof(double) * (coef_order + 1));
92
93     /* データ作成 */
94     for (i = 0; i < num_samples; i++) {
95         data[i] = sin(i * 0.01f) + cos(4.0f * sin(i * 0.05f));
96     }
97
98     /* Burgアルゴリズム */
99     burg_algorithm(data, num_samples, coef, coef_order);
100
101    /* 予測 */
102    err = 0.0f;
103    for (i = coef_order; i < num_samples; i++) {
104        predict[i] = 0.0f;
105        for (j = 1; j <= coef_order; j++) {
106            predict[i] -= coef[j] * data[i - j];
107        }
108        err += pow(data[i] - predict[i], 2);
```

```

108     }
109     printf("error:%e\n", sqrt(err / (num_samples - coef_order)));
110
111     free(data);
112     free(predict);
113     free(coef);
114
115     return 0;
116 }
```

数個の三角関数の和で表現されるような単純な信号に対しては、自己相関ベースの手法より誤差が小さいことを確認している。しかし、本実装を ALA に組み込んだときに、筆者の手元で調べた限りにおいては有意な性能改善を示さなかった。

1乗算型 PARCOR 格子型フィルター

式 1.21, 1.22 に注目すると、PARCOR 格子型フィルターは 1 次数につき 2 回の乗算が必要である。これは通常の線形予測式 1.18 では 1 次数につき 1 回の乗算で済んでいたのに対し、計算負荷が高くなっている。

各書籍 [6, 11] を見ると、格子型フィルターの乗算回数を 1 次数あたり 1 回にする 1 乗算型の PARCOR 格子型フィルターについての説明があった。[6, 11] によると、入力データに除算を行ってから 1 乗算型フィルターに入力するというフィルター構成が描かれていた。筆者はその実装に取り組んだが、合成までの処理を正しく行える実装を完成させることができなかった。

適応フィルター

適応フィルター (adaptive filter) とは、入力信号と予測誤差からフィルター係数を適応的に変化させるフィルターである。信号を無音（もしくは、白色雑音化）にする様に係数を更新すると、フィルター出力の振幅が小さくなり、圧縮に有利になる。適応フィルターとしては、LMS(least-mean square) フィルターが理論的にも実用的にも成熟している [12]。

適応フィルターはサンプル毎に係数更新を行う必要があるため、計算負荷が高い。そこで、Wavpack[27] や TTA[33] では入力信号の符号のみを用いた Sign LMS や、予測誤差についても符号をとる Sign-Sign LMS を改良して使用している。LMS フィルターの定式化の比較については [47] が簡潔である。

線型予測に適応フィルターを組み合わせると、さらなる圧縮率向上が望める。負荷増大と実装の複雑化が見込まれるため、ALA では採用を見送ったが、圧縮率向上の最も有望な方針と言える。

2.9.3 他のエントロピー符号

本稿では取り扱われることのなかったエントロピー符号について本節で幾つか採り上げる。

ランレンジス符号化

2.8 節で見たように、無音部分のエンコードでは 0 が連続して出現するため、ランレンジス符号化が有効に働く。ランレンジス符号化とは、同一の記号が並んだ長さを符号化する手法である。

もし実装するとなれば、音声データの音量を計り、音量がある閾値以下の場合にランレンジス符号化を適用するという処理が考えられる。当然、ランレンジス符号化を使用したか否かの識別情報をブロックに含める必要がある。ALA は実装が複雑になることを嫌い、ランレンジス符号化の採用を見送った。

ハフマン、適応的ハフマン

本稿では説明を省略したが、ハフマン符号化 (Huffman coding) は最も代表的なエントロピー符号化手法である。ハフマン符号は、符号の生起確率に基づいて符号を最適に（平均符号長がエントロピーと一致するように）構成することが可能である。しかし、ハフマン符号には符号の生起確率を記録したテーブルが必要になり、特に音声においては 16bit 幅の生起確率情報を全て記録しなければならない。

適応的ハフマン符号はテーブルを符号化/復号しながら構成するため、テーブルの保存は不要になる。筆者は [3, 4] を参考に適応的ハフマンを音声圧縮に試したが、圧縮率は Rice 符号と同程度であり、計算負荷が非常に大きいことが判明した。これは、テーブルの記憶領域がメモリに取られ、サンプルごとに実行されるテーブルの更新処理が非常にボトルネックになっているためである。

上記の理由があるため、筆者はハフマン符号を使用していない。しかし、TAKにおいてはハフマン符号化と Rice 符号化を組み合わせたような手法を実装している [32] とある。`ffmpeg` がリバースエンジニアリングにより実装したコード [42] にもハフマン符号に対応していると思しき固定のテーブル変数 `xcodes` があるが、その実装解説に至っていない。

ユニバーサル符号化

ユニバーサル符号化とは、符号化済みのデータを辞書と呼ばれる記録領域に保存し、辞書内のパターンと符号化対象の一一致情報を符号化する手法である。ユニバーサル符号化には大きく分けて LZ77 系と LZ78 系の手法がある。現在多くの圧縮ソフトウェア (`zip`, `gzip` 等多数) がユニバーサル符号化を元に実装を行っている。

音声圧縮に対してユニバーサル符号化が有効かどうか、筆者は [4] を参考に LZ77 系の LZSS と LZ78 系の LZW を実装して試してみたが、手元で調べた限りにおいては Rice 符号と比べ有意な改善結果を示していない。性能が上がらなかった原因として、音声データにおいては辞書内のデータと完全一致するパターンを発見できないことが多かった事が挙げられる。画像やテキストでは 8bit 幅のデータを対象にしているため、データ内に同一のパターンが発生しやすい。一方、音声データの多くは 16bit 幅のデータを持ち、例え周期性があったとしても、人工的なデータで無い限りは完全一致のパターンは出現しづらい。

2.9.4 その他

PARCOR 係数の符号化

PARCOR 係数の符号化についても幾つか工夫が存在する。これらの技術は NTT を中心に特許取得されているので、商品化に当たっては十分に注意する必要がある。

[8] では、得られた PARCOR 係数を元に圧縮率を概算し、その数値を元に最適な PARCOR 係数の次数を決定している。圧縮率が概算できる事実は有用である。白色雑音のようなエントロピーの高いデータは予測によってもエントロピーが減らないため、もし圧縮率を概算できて予測によって圧縮率が改善しないことが分かるのならば、元データをそのまま出力する手法が考えられる。

また [8, 13] で触れられているように、PARCOR 係数は $[-1, 1]$ の範囲で一様に得られるのではなく、次数ごとに偏りがあることが知られている。この性質に着目し、PARCOR 係数を \tan^{-1} 等の関数で非線形変換することで、より低 bit の係数記録領域で音質を損なわずに合成できることが示されている。

筆者は PARCOR 係数の非線形変換を試したが、16bit 幅を使う限りにおいては主だった改善が見られなかった。筆者の所感としては、PARCOR 係数の非線形変換は、低 bit で記録する前提において有効に働くものと考えられる。

参考文献

- [1] 青木直史, 『ディジタル・サウンド処理入門』, CQ 出版社, 2006 年
- [2] 甘利俊一, 『情報理論』, 筑摩書房 (ちくま学芸文庫), 2011 年
- [3] 植松友彦, 『文書データ圧縮アルゴリズム入門』, CQ 出版社, 1994 年
- [4] M. ネルソン／J.-L. ゲイリー 著, 萩原剛志・山口英 訳, 『データ圧縮ハンドブック』, ピアソン・エデュケーション, 1994 年
- [5] L.R.Rabiner, R.W.Schafer, 鈴木久喜 訳, 『音声のディジタル信号処理(上)』, コロナ社, 1983 年
- [6] L.R.Rabiner, R.W.Schafer, 鈴木久喜 訳, 『音声のディジタル信号処理(下)』, コロナ社, 1983 年
- [7] 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 年
- [8] Kamamoto, Yutaka, et al. "Low-complexity PARCOR coefficient quantization and prediction order estimation designed for entropy coding of prediction residuals." *Acoustical Science and Technology* 34.2 (2013): 105-112.
- [9] 昌達慶仁, 『圧縮処理プログラミング』, ソフトバンククリエイティブ, 2010 年
- [10] 城戸健一, 『ディジタルフーリエ解析(Ⅰ) —基礎編—』, コロナ社, 2007 年
- [11] J.D.Markel, A.H.Gray,Jr., 鈴木久喜 訳, 『音声の線形予測』, コロナ社, 1980 年
- [12] 藤井 健作, 棟安 実治, 『再考・適応アルゴリズム』, 電子情報通信学会 基礎・境界ソサイエティ Fundamentals Review, 2014, 8 卷, 4 号, p. 292-313
- [13] 中田和男, 『音声の高能率符号化』, 森北出版株式会社, 1986 年
- [14] 松井栄一編, 『日本語新辞典』, 小学館, 2005 年.
- [15] 宮川洋, 『情報理論』, コロナ社, 1979 年
- [16] Liebchen, Tilman, et al. "The MPEG-4 Audio Lossless Coding (ALS) standard-technology and applications." Proc. 119th AES Conv. 2005. APA
- [17] Warren,Jr.,Henry S., 滝沢 徹, 鈴木 貢, 赤池 英夫, 葛 毅, 藤波 順久 翻訳, 『ハッカーのたのしみ一本物のプログラマはいかにして問題を解くか』, エスアイビーアクセス, 2014 年
- [18] Qwerty、Dvorak 両配列における各キーの使用率, <http://www7.plala.or.jp/>

- dvorakjp/hinshutu.htm
- [19] Linear Prediction and Levinson-Durbin Algorithm <http://www.emptyloop.com/technotes/A%20tutorial%20on%20linear%20prediction%20and%20Levinson-Durbin.pdf>
 - [20] The Canterbury Corpus, <http://corpus.canterbury.ac.nz/descriptions/>
 - [21] γ 符号、 δ 符号、ゴロム符号による圧縮効果, <https://naoya-2.hatenadiary.org/entry/20090804/1249380645>
 - [22] Zhu Li, Lec 03 Entropy and Coding II Hoffman and Golomb Coding <http://l.web.umkc.edu/lizhu/teaching/2016sp.video-communication/notes/lec03.pdf>
 - [23] 浮動小数点を利用する際に知っておきたいこと, <https://blogs.microsoft.com/jpvsblog/2014/10/28/93/>
 - [24] 固定小数, <http://www.sage-p.com/compone/toda/fixdec.htm>
 - [25] 第 10 回「固定小数点の演算」(201302) , <http://www.hdlab.co.jp/web/a060onepoint/201302.php>
 - [26] 7.11 Wavpack, <http://www.wavpack.com/WavPack.pdf>
 - [27] WavPack Audio Compression <http://www.wavpack.com>
 - [28] FLAC - Free Lossless Audio Codec, <https://xiph.org/flac/index.html>
 - [29] FLAC - format, <https://xiph.org/flac/format.html>
 - [30] Lossless Audio Homepage, <http://www.lossless-audio.com>
 - [31] OptimFROG, <http://losslessaudio.org>
 - [32] TAK, <http://thbeck.de/Tak/Tak.html>
 - [33] TTA, http://tausoft.org/wiki/True_Audio_Codec_Overview
 - [34] MPEG-4 Audio Lossless Coding (ALS), https://www.nue.tu-berlin.de/menue/research/research_topic/compression_and_transmission/mpeg_4_audio_lossless_coding_als/parameter/en/#c230252
 - [35] Monkey's Audio - a fast and powerful lossless audio compressor <https://www.monkeysaudio.com/index.html>
 - [36] ビットを数える・探すアルゴリズム, <http://www.nminoru.jp/~nminoru/programming/bitcount.html>
 - [37] nlz.c.txt, <https://www.hackersdelight.org/hdcodetxt/nlz.c.txt>
 - [38] 数値解析(帝京大学)2016年度前期浮動小数点数補足 http://y-m.jp/wp-content/uploads/2016/04/na_2nd_floatingpoint.pdf
 - [39] Vorbis I specification, https://xiph.org/vorbis/doc/Vorbis_I_spec.html
 - [40] Burg' s Method, Algorithm and Recursion, <http://www.emptyloop.com/technotes/A%20tutorial%20on%20Burg's%20method,%20algorithm%20and%20recursion.html>

20recursion.pdf

- [41] Power spectral density estimate using Burg method, <https://jp.mathworks.com/help/dsp/ref/burgmethod.html>
- [42] takdec.c, https://ffmpeg.org/doxygen/2.0/libavcodec_2takdec_8c_source.html
- [43] Lossless audio codec comparison, <http://www.audiograaf.nl/downloads.html>
- [44] Lossless comparison - Hydrogenaudio, http://wiki.hydrogenaud.io/index.php?title=Lossless_comparison
- [45] Lossless audio compression, <http://www.firstpr.com.au/audiocomp/lossless/>
- [46] Why Lossless Audio Codecs generally suck, <https://codecs.multimedia.cx/2010/11/why-lossless-audio-codecs-generally-suck/>
- [47] Lecture 5: Variants of the LMS algorithm, <https://www.cs.tut.fi/~tabus/course/ASP/SGN2206LectureNew5.pdf>
- [48] houyhnhnm のエキセントリックらぶらぶ音声データ講座 第4回～WAVE・音楽CD→可逆圧縮～, <http://www7a.biglobe.ne.jp/~fortywinks/music4.htm>

あとがき

『かつて、出会う街中がステージだった一』

2015年5月31日の初夏の暑い日。僕らはダイバーシティ東京にいた。就活で疲れていた僕たちは、何かしたい、いや、何となく、普段行くはずのないお台場に足を運んでいた。目的のイベントは幼女向けというのだから、これは目も当てられない現実逃避だった。しかし、このコンテンツの熱さは、当時誰もが認めていたし、僕も夢中だった。

ダイバーシティの階段を登るスパンコールの輝きに目を向ければ、そこには本物のアイドルがいた。そしてその日、僕たちは初めて人の形をした偶像というものを体全体に叩き込まれた。歌唱や振り付けはこうやるのだと、圧倒的なパフォーマンスに体が震えた。

それからは夢中になって楽しむ日々が続いた。NHKホール、大田区産業プラザ、中野サンプラザ、堂島リバーフォーラム、ラクーア、そして東京ドームシティ。僕らは出会う街中がステージの中にいたのだ。

2016年3月31日、最終放送日。本当はここでやめるべきだったのだ。本当にここでやめるべきだった。

劇場版で答えが出ていたのに、未練だけで、ただ怠惰に継続する日々を始めてしまった。ここからは苦悩の毎日である。

2017年3月26日のパシフィコ横浜、歌唱が途切れる前から僕は涙で何も見えていなかった。失われた歌は、もう帰ってこない。そして絶対に帰らないのが絶対のルールだと。僕の頭は理解しているはずなのに涙が止まらない。

その頃には、もはや初代の概念が崩れ去った放送と筐体を、僕は我慢して、ただ我慢して継続していた。

2018年2月4日の夕方、僕らは福岡市内で大号泣した。でも、ようやくこれで全てが終わらせられる。未練なく、これで決別できると僕は信じた。

2018年2月28日、武道館が終わった。これで終わったはずだった。終わってほしかった。いや、本当は終わっていたのだ。5thを過ぎてさえ未練を引きずった僕は、いい加減に責められるべきだし、順当に罰は下った。

2019年8月17日、こともあろうにパシフィコ横浜。不安がありながらも、確かな成長を目視できたり、十分に強い曲があった。

しかし、出てきた答えは打ち切りだった。その上、コンテンツに対する嘘と、過去方向への破壊を宣言された。思い出は“未来”の中に探しに行くのではなかったのか？かつて、出会う街中がステージだった。

ロスレス音声コーデック

— 基本理論と実装 —

2019年9月22日 ver 1.0

著者 あいき

印刷所 ねこのしっぽ 様