Replacing the expectations with autocorrelation coefficients yields the system of n linear equations

$$\begin{bmatrix} R(0) & R(1) & R(2) & \dots & R(n-1) \\ R(1) & R(0) & R(1) & \dots & R(n-2) \\ R(2) & R(1) & R(0) & \dots & R(n-3) \\ \vdots & \vdots & \vdots & & \vdots \\ R(n-1) & R(n-2) & R(n-3) & \dots & R(0) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} R(1) \\ R(2) \\ R(3) \\ \vdots \\ R(n) \end{bmatrix},$$

with the n coefficients c_j as the unknowns. This can be written compactly as $\mathbf{R}C = P$ and can easily be solved by Gaussian elimination or by inverting matrix $\mathbf{R}$. The point is that matrix inversion requires in general $O(n^3)$ operations, but ours is not a general case, because our matrix $\mathbf{R}$ is special. It is easy to see that each diagonal of $\mathbf{R}$ consists of identical elements. Such a matrix is called a Toeplitz matrix, after its originator, Otto Toeplitz, and it can be inverted by a number of specialized, efficient algorithms. In addition, our $\mathbf{R}$ is also symmetric.

Otto Toeplitz (1881–1940) came from a Jewish family that produced several teachers of mathematics. Both his father, Emil Toeplitz, and his grandfather, Julius Toeplitz, taught mathematics in a Gymnasium and they also both published mathematics papers. Otto was brought up in Breslau and attended a Gymnasium in that city. His family background made it natural that he also should study mathematics.

During his long, productive carrer, Toeplitz studied algebraic geometry, integral equations, and spectral theory. He worked also on summation processes, infinite-dimensional spaces and functions on them. In the 1930s he developed a general theory of infinite dimensional spaces and criticized Banach's work as being too abstract.

Toeplitz operators and Toeplitz matrices bear his name.

FLAC employs a recursive, efficient method, known as the Levinson-Durbin algorithm, to solve this system of n equations. This algorithm was first proposed by Norman Levinson in 1947 [Levinson 47], improved by J. Durbin in 1960 [Durbin 60], and further improved by several researchers. In its present form it requires only $3n^2$ multiplications. A description of this algorithm is outside the scope of this book, but can be found, together with a derivation, in [Parsons 87].

The author would like to thank Josh Coalson, for his help with this section.

7.11 WavPack

(This section written by David Bryant, WavPack's developer.)

WavPack [WavPack 06] is a completely open, multiplatform audio compression algorithm and software that supports three compression modes, lossless, high-quality lossy, and a unique hybrid compression mode. It handles integer audio samples up to 32-bits wide and also 32-bit IEEE floating-point data [IEEE754 85]. The input stream is partitioned by WavPack into blocks that can be either mono or stereo and are generally 0.5 seconds long (but the length is actually flexible). Blocks may be combined

in sequence by the encoder to handle multichannel audio streams. All audio sampling rates are supported by WavPack in all its modes.

WavPack defaults to the lossless mode. In this mode, the audio samples are simply compressed at their full resolution and no information is discarded. WavPack generally provides a reasonable compromise between compression ratio and encoding/decoding speed. However, for specific applications, an alternate compromise may be more desirable. For this reason, WavPack incorporates an optional "fast" mode that is fast and entails only a small penalty in compression performance and a "high" mode that aims for maximum compression (at a somewhat slower pace).

The lossy mode employs no subband coding or psychoacoustic noise masking. Instead, it is based on variable quantization in the time domain combined with mild noise shaping. This mode can operate from bitrates as low as 2.22 bits per sample up to fully lossless and offers considerably more flexibility and performance than the similar, but much simpler, ADPCM (Section 7.6). This makes the lossy mode ideal for high-quality audio encoding where the data storage or bandwidth requirements of the lossless mode might be prohibitive.

Finally, the hybrid mode combines the lossless and lossy modes into a single operation. Instead of a single output file being generated from a source, two files are generated. The first, with the extension `wv`, is a smaller lossy file that can be played on its own. The second is a "correction" file (`wvc`) that, when combined with the first, provides lossless restoration of the original audio. The two files together have essentially the same size as the output produced by the pure lossless mode, so the hybrid mode generates very little additional overhead.

Efficient hardware decoding was among the design goals of WavPack, and this weighed heavily (along with patent considerations) in the design decisions. As a result, even the most demanding WavPack mode will easily decode CD quality audio in real time on an ARM7 processor running at 75 MHz (and the default modes use considerably less than that).

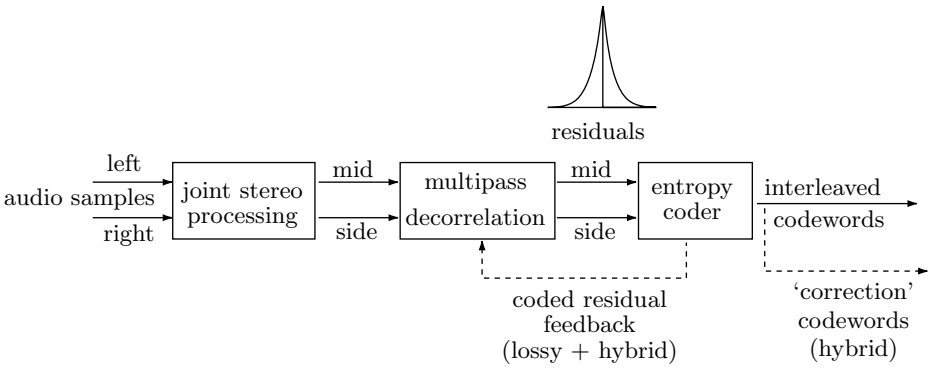

Figure 7.31: The WavPack Encoder.

Figure 7.31 is a block diagram of the WavPack encoder. Its main parts are the joint-stereo processing (removing inter-channel correlations), multipass decorrelation

(removing intra-channel correlations between neighboring audio samples), and the entropy coder.

Decorrelation

The decorrelation passes are virtually identical in all three modes of WavPack. The first step in the decorrelation process is to convert the left and right stereo channels to the standard difference and average (also referred to as side and mid) channels by $s(k) = l(k) - r(k)$ and $s'(k) = \text{int}\big((l(k) + r(k))/2\big)$. Notice that the integer division by 2 discards the least-significant bit of the sum $l(k) + r(k)$, but this bit can be reconstructed from the difference, because a sum $A + B$ and a difference $A - B$ have the same least-significant bit.

The second decorrelation step is prediction. WavPack performs multiple passes of very simple, single-tap linear predictive filters that employ the sign-sign LMS method for adaptation. (The sign-sign LMS method [adaptiv9 06] is a variation on the standard least-mean-squares method.) Having only one adjustable weighing factor per filter eliminates problems of instability and non-convergence that can occur when multiple taps are updated from a single result. The overall computational complexity of the process is simply controlled by the total number of filtering passes made. The default case performs five passes, as this provides a good compromise of compression vs. speed. In the "fast" mode, only two passes are made, while the "high" mode incorporates the maximum allowed 16 passes.

There are actually 13 variations of the filter depending on what sample value (or linear function of two sample values) is used as the input value $u(k)$ to the prediction filter. These variations are identified by a parameter named "term." In the case of a single channel, filtering depends on the value of this parameter as follows

$$u(k) = \begin{cases} 1 \le \text{term} \le 8, & s(k - \text{term}), \\ \text{term} = 17, & 2s(k-1) - s(k-2), \\ \text{term} = 18, & \big(3s(k-1) - s(k-2)\big)/2.0, \end{cases}$$

where $s(k)$ is the sample to be predicted and the recent sample history is $s(k-1)$, $s(k-2)$,..., $s(k-8)$.

For the dual channel case (which may be left and right or mid and side) there are three more filter variations available that use cross-channel information

$$\begin{aligned} &\texttt{if}(\text{term} = -1),\ u(k) = s'(k),\ u'(k) = s(k-1), \\ &\texttt{if}(\text{term} = -2),\ u(k) = s'(k-1),\ u'(k) = s(k), \\ &\texttt{if}(\text{term} = -3),\ u(k) = s'(k-1),\ u'(k) = s(k-1). \end{aligned}$$

Once $u(k)$ has been determined, an error (or residual) value $e(k)$ is computed as the difference $e(k) = s(k) - w(k)u(k)$, where $w(k)$ is the current filter weight value. The weight generally varies between +1 for highly correlated samples and 0 for completely uncorrelated samples. It can also become negative if there is a negative correlation between the sample values. Negative correlation would exist if $(-1 \times u(k))$ was better than $u(k)$ as a predictor of $s(k)$. This often happens with samples with large amounts of high frequencies.

Finally, the weight is updated for the next sample based on the signs of the filter input and the residual $w(k+1) = w(k) + d \cdot \text{sgn}(u(k)) \cdot \text{sgn}(e(k))$, where parameter d is the step-size of the adaptation, a small positive integer. (The `sgn` function used by WavPack returns +1, 0, or −1.) The rule for adapting $w(k)$ implies that if either $u(k)$ or $e(k)$ is zero, the weight is not updated. In cases where the input to the filter comes from the same channel as the sample being predicted (i.e., where "term" is positive), the magnitude of the weight is self limiting and is therefore allowed to exceed ±1. However, in the cross-channel cases (where "term" is negative), the weight could grow indefinitely, and so is clipped to ±1 after each adaptation.

Here's an example of an adaptation step. Assuming that the input $u(k)$ to the predictor is 105, sample $s(k)$ to be predicted is 100, the current filter weight $w(k)$ is 0.8, and the current stepsize d is 0.01, we first compute the residual $e(k) = s(k) - w(k) \cdot u(k) = 100 - 105 \cdot 0.8 = 16$ and then update the weight

$$\begin{aligned} w(k+1) &= w(k) + s \cdot \text{sgn}(e(k)) \cdot \text{sgn}(u(k)) \\ &= 0.8 + 0.01 \cdot \text{sgn}(100) \cdot \text{sgn}(16) \\ &= 0.8 + 0.01 \cdot 1 \cdot 1 = 0.81. \end{aligned}$$

We can see that if the original weight had been more positive (greater than 0.8), the prediction would have been closer to $s(k)$ and the residual would have been smaller. Therefore, we increase the weight in this iteration.

The magnitude of the stepsize is a compromise between fast adaptation and noisy operation around the optimum value. In other words, if the stepsize is large, the filter will adjust quickly to sudden changes in the signal, but the weight will jump around a lot when it is near the correct value. Too small a stepsize will cause the filter to adjust too slowly to rapidly changing audio.

In the lossless mode, the results of the decorrelation (the residuals $e(k)$) are simply passed to the entropy coder for exact translation. In the lossy and hybrid modes, in contrast, the decorrelated values may be coded inexactly based on a defined maximum allowable error. In this situation, the actual value coded is returned from the entropy coder and this value is used to update the filter weight and generate the reconstructed sample value. This is required because during subsequent decoding the exact sample values will not be available, and we must operate the encoder with only the information that we will have at that time.

Implementation

All sample data is presented to the decorrelator as 32-bit signed integers, even if the source is smaller. This is done to allow common routines, but is also required from an arithmetic standpoint because, for example, 16-bit data might clip at some steps if restricted to 16-bit storage. The routines are designed in such a way that they will not overflow on sample data with up to 25 bits of significance. This allows for standard 24-bit audio and the 25-bit significand of 32-bit IEEE data. Any significance beyond this must be handled by side channel data.

The filter weights are represented in 16-bit signed integers with 10 bits of fraction (for example $1024 = 0000010\underbrace{0\ldots00}_{10}{}_2$ represents 1.0), and the adjusting step-size may

range from 1 to 7 units of that resolution. For input values that fit in 16-bit signed integers, it is possible to perform the rounded weight multiply with a single 16-bit × 16-bit = 32-bit operation by `prediction = (sample * weight + 512) >> 10;`

Since "sample" and "weight" both fit in 16-bit integers, the signed product will fit in 32 bits. We add the 512 for rounding and then shift the result to the right 10 places to compensate for the 10 bits of fraction in the weight representation. This operation is very efficiently implemented in modern processors.

For those cases where the sample does not fit in 16 bits, we need more than 32 bits in the intermediate result, and several alternative methods are provided in the source code. If a multiply operation is provided with 40 bits of product resolution (as in the ColdFire EMAC or several Texas Instruments DSP chips), then the above operation will work (although some scaling of the inputs may be required). If only 32-bit results are provided, it is possible to perform separate operations for the lower and upper 16 bits of the input sample and then combine the results with shifts and an addition. This has been efficiently implemented with the MMX extensions to the `x86` instruction set. Finally, some processors provide hardware-assisted double-precision floating-point arithmetic which can easily perform the operation with the required precision.

As mentioned previously, multiple passes of the decorrelation filter are used. For the standard operating modes (fast, default, and high) stepsize is always 2 (which represents about 0.002) and the following terms are employed, in the order shown:

$$\begin{aligned}\text{fast mode terms} &= \{17, 17\},\\ \text{default mode terms} &= \{18, 18, 2, 3, -2\},\\ \text{high mode terms} &= \{18, 18, 2, 3, -2, 18, 2, 4, 7, 5, 3, 6, 8, -1, 18, 2\}.\end{aligned}$$

For single channel operation, the negative term passes (i.e. passes with cross-channel correlation) are skipped. Note, however, that the decoder is not fixed to any of these configurations because the number of decorrelation passes, as well as other information such as what terms and stepsize to use in each pass, is sent to the decoder as side information at the beginning of each block.

For lossless processing, it is possible to perform each pass of the decorrelation in a separate loop acting on an array of samples. This allows the process to be implemented in small, highly optimized routines. Unfortunately, this technique is not possible in the lossy mode, because the results of the inexact entropy coding must be accounted for at each sample. However, on the decoding side, this technique may be used for both lossless and lossy modes (because they are essentially identical from the decorrelator's viewpoint).

Generally, the operations are not parallelizable because the output from one pass is the input to the next pass. However, in the two-channel mode the two channels may be performed in parallel for the positive term values (and again this has been implemented in MMX instructions).

Entropy Coder

The decorrelation step generates residuals $e(k)$ that are signed numbers (generally small). Recall that decorrelation is a multistep process, where the differences $e(k)$ computed by a step become the input of the next step. Because of this feature of WavPack,

we refer to the final outputs $e(k)$ of the decorrelation process as residuals. The residuals are compressed by the entropy encoder. The sequence of residuals is best characterized as a two-sided geometric distribution (TSGD) centered at zero (see Figure 2.9 for the standard, one-sided geometric distribution). The Golomb codes (Section 2.5) provide a simple method for optimally encoding similar one-sided geometric distributions. This code depends on a single positive integer parameter m. To Golomb encode any nonnegative integer n, we first divide n into a magnitude $\texttt{mag} = \text{int}(n/m)$ and a remainder $\texttt{rem} = n \bmod m$, then we code the magnitude as a unary prefix (i.e. `mag` 1's followed by a single 0) and follow that with an adjusted-binary code representation of the remainder. If m is an integer power of 2 ($m = 2^k$), then the remainder is simply encoded as the binary value of `rem` using k bits. If m is not an integer power of 2, then the remainder is coded in either k or $k+1$ bits where $2^k < m < 2^{k+1}$. Because the values at the low side of the range are more probable than values at the high side, the shorter codewords are reserved for the smaller values.

Table 7.32 lists the adjusted binary codes for m values 6 through 11.

remainder to code	when $m=6$	when $m=7$	when $m=8$	when $m=9$	when $m=10$	when $m=11$
0	00	00	000	000	000	000
1	01	010	001	001	001	001
2	100	011	010	010	010	010
3	101	100	011	011	011	011
4	110	101	100	100	100	100
5	111	110	101	101	101	1010
6		111	110	110	1100	1011
7			111	1110	1101	1100
8				1111	1110	1101
9					1111	1110
10						1111

Table 7.32: Adjusted Binary Codes for Six m Values.

The Rice codes are a common simplification of this method, where m is a power of 2. This eliminates the need for the adjusted binary codes and eliminates the division required by the general Golomb codes (because division by a power of 2 is implemented as a shift). Rice codes are commonly used by lossless audio compression algorithms to encode prediction errors. However, Rice codes are less efficient when the optimum m is midway between two consecutive powers of 2. They are also less suitable for lossy encoding because of discontinuities resulting from large jumps in the value of m. For these reasons Rice codes were not chosen.

Before a residual $e(k)$ is encoded, it is converted to a nonnegative value by means of `if` $e(k) < 0$, $e(k) = -\big(e(k)+1\big)$. The original sign bit of $e(k)$ is appended to the end of the coded value. Note that the sign bit is always written on the compressed stream, even when the value to be encoded is zero, because -1 is also coded as zero. This is less efficient for very small encoded values where the probability of the value zero is significantly higher than neighboring values. However, we know from long experience

that an audio file may have long runs of consecutive zero audio samples (silence), but relatively few isolated zero audio samples. Thus, our chosen method is slightly more efficient when zero is not significantly more common than its neighbors. We show later that WavPack encodes runs of zero residuals with an Elias code.

The simplest method for determining m is to divide the residuals into blocks and determine the value of m that encodes the block in the smallest number of bits. Compression algorithms that employ this method have to send the value of m to the decoder as side information. It is included in the compressed stream before the encoded samples. In keeping with the overall philosophy of WavPack, we instead implement a method that dynamically adjusts m at each audio sample.

The discussion on page 67 (and especially Equation (2.2)) shows that the best value for m is the median of the recent sample values. We therefore attempt to adapt m directly from the current residual $e(k)$ using the simple expression

$$\begin{aligned}\texttt{if}\,\big(e(k)\geq m(k)\big)\,\texttt{then}\,m(k+1)&=m(k)+\text{int}\left[\frac{m(k)+127}{128}\right],\\ \texttt{else}\,m(k+1)&=m(k)-\text{int}\left[\frac{m(k)+126}{128}\right].\end{aligned}\tag{7.10}$$

The idea is that the current median $m(k)$ will be incremented by a small amount when a residual $e(k)$ occurs above it and will be decremented by an almost identical amount when the residual occurs below it. The different offsets (126 and 127) for the two cases were selected so that the median value can move up from 1, but not go below 1. This works very well for large values, but because the median is simply an integer, it tends to be "jumpy" at small values because it must move at least one unit in each step. For this reason the actual implementation adds four bits of fraction to the median that are simply ignored when the value is used in a comparison. This way, it can move smoothly at smaller values.

It is interesting to note that the initial value of the median is not crucial. Regardless of its initial value, m will move towards the median of the values it is presented with. In WavPack, m is initialized at the start of each block to the value that it left off on the previous block, but even if it started at 1 it would still work (but would take a while to catch up with the median).

There is a problem with using an adaptive median value in this manner. Sometimes, a residual will come along that is so big compared to the median that a Golomb code with a huge number of 1's will be generated. For example, if m was equal to 1 and a residual of 24,000 occurred, then the Golomb code for the sample would result in $\texttt{mag}=\texttt{int}(24{,}000/1)=24{,}000$ and $\texttt{rem}=24{,}000 \bmod 1=0$ and would therefore start with 24,000 1's, the equivalent of 3,000 bytes of all 1's. To prevent such a situation and avoid ridiculously long Golomb codes, WavPack generates only Golomb codes with 15 or fewer consecutive 1's. If a Golomb code for a residual $e(k)$ requires k consecutive 1's, where k is greater than or equal 16, then WavPack encodes $e(k)$ by generating 16 1's, followed by the Elias gamma code of k (Code C_1 of Table 2.6). When the residuals are distributed geometrically, this situation is exceedingly rare, so it does not affect efficiency, but it does prevent extraordinary and rarely-occurring cases from drastically reducing compression.

Recall that each encoded value is followed by the sign bit of the original residual $e(k)$. Thus, an audio sample of zero is inefficiently encoded as the two bits 00. Because of this, WavPack employs a special method to encode runs of consecutive zero residuals. A run of k zeros is encoded in the same Elias gamma code preceded by a single 1 to distinguish it from a regular code (this is done only when the medians are very small to avoid wasting a bit for every sample.)

One source of less than optimum efficiency with the method described so far is that sometimes the data is not distributed in an exactly geometrical manner. The decay for higher values may be faster or slower than exponential, either because the decorrelator is not perfect or the audio data has uncommon distribution characteristics. In these cases, it may be that the dynamically estimated median is not a good value for m, or it may be that no value of m can produce optimal codes. For these cases, a further refinement has been added that I have named Recursive Golomb Coding.

In this method, we calculate the first median as in Equation (7.10) and denote it by m. However, for values above the median we subtract m from the value and calculate a new median, called m', that represents the 3/4 population point of samples. Finally, we calculate a third median (m'') for the sample values that lie above the second median (m'), and we use this for all remaining partitions (Figure 7.33). The first three coding zones still represent 1/2, 1/4, and 1/8 of the residuals, but they don't necessarily span the same number of possible values as they would with a single m. However, this is not an issue for the decoder because it knows all the medians and therefore all the span sizes.

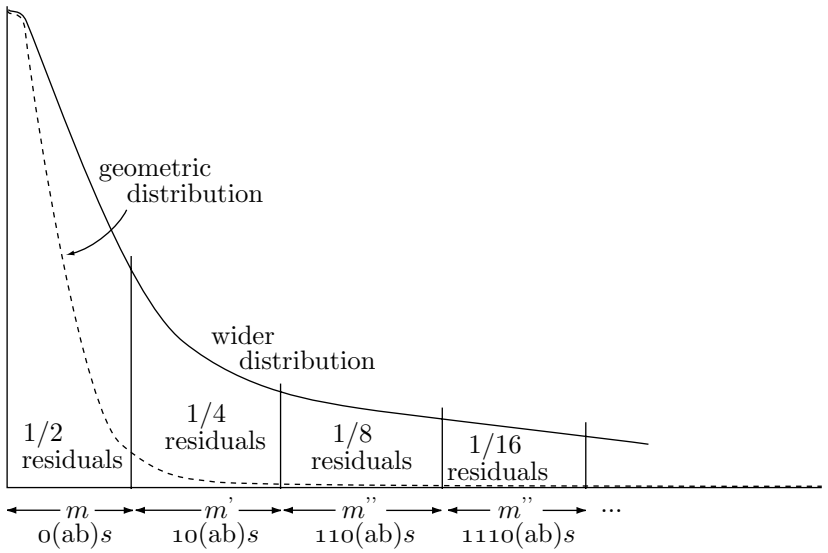

Figure 7.33: Median Zones for Recursive Golomb Coding.

In practice, recursive Golomb coding works like this. If the residual being coded is in the first zone, fine. If not, then subtract m from the residual and pretend we're starting all over. In other words, first we divide the residuals into two equal groups

(under m and over m). Then, we subtract m from the "over m" group (so they start at zero again) and divide those into two groups. Then, we do this once more. This way, the first several zones are guaranteed to have the proper distribution because we calculate the second and third medians directly instead of assuming that they're equal in width to the first one. Table 7.34 lists the first few zones where (ab) represents the adjusted-binary encoding of the remainder (residual modulo m) and S represents the sign bit.

range	prob.	coding
$0 \le \text{residual} < m$	1/2	$0(ab)S$
$m \le \text{residual} < m + m'$	1/4	$10(ab)S$
$m + m' \le \text{residual} < m + m' + m''$	1/8	$110(ab)S$
$m + m' + m'' \le \text{residual} < m + m' + 2m''$	1/16	$1110(ab)S$
...		

Table 7.34: Probabilities for Recursive Golomb Coding.

Lossy and hybrid coding

Recursive Golomb coding allows arbitrary TSGD integers to be simply and efficiently coded, even when the distribution is not perfectly exponential and even when it contains runs of zeros. For the implementation of lossy encoding, it is possible to simply transmit the unary magnitude and the sign bit, leaving out the remainder data altogether. On the decode side, the value at the center of the specified magnitude range is chosen as the output value. This results in an average of three bits per residual as illustrated by Table 7.35. The table is based on the infinite sum

$$1 + \sum_{n=1}^{\infty} \frac{n}{2^n},$$

where the 1 represents the sign bit and each term in the sum is a probability multiplied by the number of bits for the probability. The infinite sum adds up to two bits, and the total after adding the sign bit is three bits.

prob.	bits sent	# of bits	(# of bits) × prob.
1/2	$0S$	2	2/2
1/4	$10S$	3	3/4
1/8	$110S$	4	4/8
1/16	$1110S$	5	5/16
1/32	$11110S$	6	6/32
1/64	$111110S$	7	7/64
...	...	...	...
total 1			3

Table 7.35: Three Bits Per Residual.

This works fine. However, a lower minimum bitrate is desired and so a new scheme is employed that splits the samples 5:2 instead of down the middle at 1:1. In other words, a special "median" is found that has five residuals lower for every two that are higher. This is accomplished easily by modifying the procedure above to

$$\texttt{if}\ \big(s(k) > m(k)\big)\ \texttt{then}\ m(k+1) = m(k) + 5 \cdot \text{int}\left[\frac{m(k)+127}{128}\right],$$
$$\texttt{else}\ m(k+1) = m(k) - 2 \cdot \text{int}\left[\frac{m(k)+125}{128}\right].$$

The idea is that the median is bumped up five units every time the value is higher, and bumped down two units when it is lower. If there are two ups for every five downs, the median stays stationary. Because the median moves down a minimum of two units, we consider 1 or 2 to be the minimum (it will not go below this).

Now the first region contains 5/7 of the residuals, the second has 10/49, the third 20/343, and so on. Using these enlarged median zones, the minimum bitrate drops from 3 to 2.4 bits per residual, as shown by the following infinite sum

$$1 + \sum_{n=1}^{\infty} \frac{5n \times 2^{n-1}}{7^n}.$$

There is an efficiency loss with these unary codes because the number of 1's and 0's in the datastream is no longer equal. There is a single 0 for each residual, but since there are only 1.4 total bits per residual (not counting the sign bit), there must be only an average of 0.4 1's per residual. In other words, the probability for 0's is 5/7 and the probability for 1's is 2/7. Using the $\log_2$ method to determine the ideal number of bits required to represent those symbols, we obtain the numbers listed in Table 7.36a and derive an optimum multibit encoding scheme (Table 7.36b).

symbol	freq	$\log_2(1/\text{freq})$
0	5/7	0.485427
1	2/7	1.807355

(a)

input sequence	freq	ideal # bits	output sequence
00	25/49	0.970854	0
01	10/49	2.292782	10
1	14/49	1.807355	11

(b)

Table 7.36: Optimum Translation of Unary Magnitudes.

Table 7.37 shows that the average number of unary magnitude bits transmitted is reduced by the translation to $(73/84)\times 1.4 = 1.21666 \approx 1.22$ bits per sample. We use the 1.4 figure because only the unary magnitude data is subject to this translation; adding back the untranslated sign bit gets us to about 2.22 bits per sample. This represents the encoder running "flat out" lossy (i.e., no remainder data sent) however, the method

input sequence	freq	input bits	net* input	output sequence	output bits	net* output
00	25/49	2	50/49	0	1	25/49
01	10/49	2	20/49	10	2	20/49
1	14/49	1	14/49	11	2	28/49
totals	1		84/49			73/49

*The "net" values are the frequency multiplied by the number of bits.

Table 7.37: Net Savings from Unary Translation.

is so efficient that it is used for all modes. In the lossless case, the complete adjusted-binary representation of the remainder is inserted between the magnitude data and the terminating sign bit.

In practice, we don't normally want the encoder to be running at its absolute minimum lossy bitrate. Instead, we would generally like to be somewhere in-between full lossy mode and lossless mode so that we send some (but not all) of the remainder value. Specifically, we want to have a maximum allowed error that can be adjusted during encoding to give us a specific signal-to-noise ratio or constant bitrate. Sending just enough data to reach a specified error limit for each sample (and no more) is optimal for this kind of quantization because the perceived noise is equivalent to the RMS level of the error values.

To accomplish this, we have the encoder calculate the range of possible sample values that satisfy the magnitude information that was just sent. If this range is smaller than the desired error limit, then we are finished and no additional data is sent for that sample. Otherwise, successive bits are sent to the datastream, each one narrowing the possible range by half (i.e., 0 = lower half and 1 = upper half) until the size of the range becomes less than the error limit. This binary process is done in lockstep by the decoder, to remain synchronized with the bitstream.

At this point it should be clear how the hybrid lossless mode splits the bitstream into a lossy part and a correction part. Once the specified error limit has been met, the numeric position of the sample value in the remaining range is encoded using the adjusted binary code described above, but this data goes into the "correction" stream rather than the main bitstream. Again, on decoding, the process is exactly reversed so the decoder can stay synchronized with the two bitstreams.

With respect to the entropy coder, the overhead associated with having two files instead of one is negligible. The only cost is the difference resulting from coding the lossy portion of the remainder with the binary splitting method instead of the adjusted binary method. The rest of the cost comes on the decorrelator side where it must deal with a more noisy signal.

More information on the theory behind and the implementation of WavPack can be found in [WavPack 06].