A Multiply-Free Enumeration of Combinations With Replacement and Sign

Timothy B. Terriberry[†], *Member, IEEE* and Jean-Marc Valin[†‡], *Member, IEEE*

[†]Xiph.Org Foundation

[‡]CSIRO ICT Centre, Sydney, Australia

Abstract

Vector quantization of speech audio data increasingly uses algebraic codebooks, such as the popular unit-pulse codebook used in the Algebraic Code-Excited Linear Prediction (ACELP) algorithm [1]. While the simple encoding ACELP uses for the codebook indices is suitable for its relatively small codebooks, the existing encodings for larger codebooks either waste bits or require expensive operations, such as extended-precision multiplications and division. We propose CWRS, a simple enumerative algorithm for encoding these codebook indices that does not require multiplications or divisions. Although the basic implementation uses a lookup table, we give an alternate version that eliminates this table for a moderate speed penalty.

Index Terms

Speech coding, speherical quantization, variable dimension vector quantization

This work was undertaken for the Xiph.Org Foundation and sponsored in part by Australia's Commonwealth Scientific and Industrial Research Organization (CSIRO).

A Multiply-Free Enumeration of Combinations With Replacement and Sign

I. Introduction

Alebraic codebooks for vector quantization offer two important advantages over stochastic codebooks. First, they do not to need store the entire codebook in memory, allowing them to scale to much higher dimensions than a stochastic codebook. Second, the inherent structure in the codebook can greatly accelerate the search for an optimal codeword. Speech coding algorithms such as ACELP [1] have long employed them for quantizing unit vectors. Under some mild assumptions, the data is uniformly distributed [2], making it well-suited for representation by an algebraic codebook. Even in more recent codecs, like CELT[1], where these assumptions may not necessarily hold, the complexity reduction in time and space makes them a practical tool where a stochastic codebook would be completely infeasible.

CELT partitions the audio signal multiple bands of varying sizes, and codes one unit vector per band, with as many as 100 dimensions per vector. Even though CELT is CBR, the number of bits assigned to a band varies with each frame, allowing allocations as high as 64 bits for a single vector. Every combination of vector size and rate allocation requires a separate codebook, meaning that CELT uses a very large number of codebooks. The large storage requirments of stochastic codebooks would impose much greater limits not only on the codebook size, but also on the number of codebooks, restricting the sizes of each band and how bits are allocated to them. The impairment from this loss of flexibility would be much larger than any potential gain from a more efficient codebook.

The *unit-pulse* codebook consists of n-dimensional code vectors $\mathbf{y}$ formed by adding a fixed number, m, of signed unit pulses:

$$\mathbf{y} = \sum_{k=0}^{m-1} s_k \varepsilon_{x_k} \ . \tag{1}$$

Here s_k is the sign of the kth pulse, x_k is the position in the code vector, and ε_{x_k} is the elementary basis vector for dimension x_k. The signs are constrained so that if there is more than one pulse in a given position, they all have the same sign. That is, if $x_i = x_k$, then $s_i = s_k$. The resulting code vector $\mathbf{y}$ is not generally a unit vector, but one can easily obtain a unit vector by normalizing by $1/\left\|\mathbf{y}\right\|$.

[1] http://www.celt-codec.org/

This codebook does not provide an optimal tessellation of the n-dimensional unit sphere when $m > 1$, but such tessellations are not known for arbitrary dimension. Good tessellations can be found by solving large energy minimization problems, but at high rates the resulting stochastic codebooks do not fit in memory. They can be constructed from a regular tessellation of a circular section of the $n-1$ dimensional plane [3], but enumerating the points in such a section faces similar problems in higher dimensions. Some good solutions are known for specific dimensions, such as $n = 24$ [4], [5]. Subdivision of the convex regular polytopes (higher-dimensional analogs of the Platonic solids) provides another alternative, but makes quantization more computationally expensive and severely restricts the possible sizes of the codebooks [6]. This limits the granularity of rate allocation between bands and makes extremely low rates more difficult to achieve.

We propose new, simple, $O(n+m)$ algorithms for encoding and decoding the vectors in the unit-pulse codebook, outlined in Section II. In their basic form, these algorithms require $O(nm)$ storage for a lookup table, though multiple codebooks with different values of n and m can share a single table. Section III describes an alternate, $O(nm)$ version of the algorithm which does not require a lookup table. Unlike previous algorithms, discussed in Section IV, our algorithms waste no bits and use no multiplications or divisions. This makes them suitable for very large codebooks, which require extended-precision arithmetic.

II. Encoding

Under the uniform distribution assumption, every possible code vector must be encoded with the same number of bits. If there are $V(n, m)$ unique code vectors of dimension n with m pulses, then the easiest approach is to assign every code vector a unique index and store a single integer between 0 and $V(n, m) - 1$. Two procedures are then required: one to translate a given code vector into an index, and one to translate an index into a code vector.

A. Counting the Number of Pulse Vectors

We start by counting the number of possible code vectors. This is like counting the number of combinations of m items taken from a set of size n with replacement, except that every unique element also has a sign bit associated with it. This last condition eliminates the chances of finding a simple closed-form expression, however we can write down a simple recurrence relation:

$$\begin{aligned} V(n, m) = {} & V(n-1, m) \\ & + V(n, m-1) + V(n-1, m-1) \ . \end{aligned} \tag{2}$$

The $V(n-1,m)$ term corresponds to the number of code vectors that do not have any pulses in the first dimension; this is just the number of ways of distributing the pulses into the remaining dimensions. The $V(n,m-1)$ term corresponds to the number of code vectors that do have at least one pulse in the first dimension; this is the number of ways of distributing the remaining pulses into any of the dimensions.

The last term, however, double-counts the number of code vectors that have exactly one pulse in the first dimension. $V(n-1,m-1)$ is the number of ways of distributing the remaining pulses into the remaining dimensions. If one or more of the $m-1$ remaining pulses also lie in the first dimension, then the first pulse must keep the same sign. However, if none of the remaining pulses lie in the first dimension, then there are two choices: one for the positive sign, and one for the negative sign. One of these choices is already accounted for in the $V(n,m-1)$ term. The $V(n-1,m-1)$ term accounts for the second one.

Two base cases limit the recursion. There is exactly one way to place zero pulses in any number of dimensions, so $V(n,0)=1$. Similarly, there is no way to place a non-zero number of pulses in a zero-dimensional vector, so $V(0,m)=0$ for all $m>0$. The closed-form solution to this recurrence for $m>0$ is given by

$$V(n,m) = 2n \cdot {}_2F_1(1-m, 1-n; 2; 2) \ , \tag{3}$$

where ${}_2F_1(a,b;c;z)$ is the hypergeometric function

$${}_2F_1(a,b;c;z) = \sum_{k=0}^{\infty} \frac{a^{\overline{k}} b^{\overline{k}}}{c^{\overline{k}}} \cdot \frac{z^k}{k!} \ , \tag{4}$$

and $a^{\overline{k}}$ is the rising factorial: $a(a+1)(a+2)\ldots(a+k-1)$. This yields the number of combinations with replacement and sign (CWRS). This function is not easy to evaluate directly.

B. Pulse Vector to Index

To translate a code vector into an index, we partition the the total number of vectors according to the possibilities described above. Let i be the index we are computing, j be the index of the current dimension we are considering, k be the number of pulses already considered, and s_k and x_k be the sign and position of the kth pulse. We require that pulses be sorted in ascending order by position, so that $x_k \leq x_{k+1}$. Now let

$$U(n,m) = V(n,m-1) + V(n-1,m-1) \tag{5}$$

be the number of vectors that include at least one pulse in the first dimension. It is easy to see that

$$V(n,m) = \sum_{j=0}^{n-1} U(n-j,m) \ , \tag{6}$$

since this sums the number of vectors with no pulses in the first j dimensions, but at least one pulse in the next dimension. Thus $U(n-j,m)$ provides a convenient means of partitioning $V(n,m)$. Also, since U is the sum of two values of V, it obeys the same recurrence relation:

$$\begin{aligned} U(n,m) &= U(n-1,m) \\ &\quad + U(n,m-1) + U(n-1,m-1) \ , \end{aligned} \tag{7}$$

with the base cases $U(n,1) = U(1,m) = 2$.

We build the index by considering each pulse in turn and locating the partition in which it lies. Let p be the size of the current partition. If the current pulse comes after this partition, we add p to i and move on to the next partition. Otherwise, we further subdivide the current partition using the next pulse.

Pseudocode is given in Algorithm 1. At the start of the first iteration, $i = j = k = 0$. We have not placed any previous pulses, so we always require a sign. In this case we let $p = U(n,m)$. On subsequent iterations, we will start with $j = x_{k-1}$, and thus we have already determined the sign of any further pulses in that position. In that case, $p = \frac{1}{2}U(n-j,m-k)$. Now, if $j < x_k$, then there are no more pulses in dimension j, so we must advance to the next partition. Therefore we add p to i, set $j = j+1$, and update $p = U(n-j,m-k)$. This process is repeated until $j = x_k$. If $x_{k-1} \neq x_k$ (or if this is the first pulse), then we need to encode a new sign for this position. We split the partition in half, with the top half associated with the positive pulses and the bottom half associated with the negative. That is, if $s_k = -1$, then we add $\frac{1}{2}p$ to i. This allows the decoder to determine the sign associated with each position as soon as it encounters the first pulse in that position. Now we set $k = k+1$ and return to the beginning to encode the next pulse. When all the pulses have been processed, i will be a unique index between 0 and $V(n,m) - 1$. Fig. 1 shows how the entire code space is partitioned for a 3 dimensional vector with 3 pulses.

C. Index to Pulse Vector

To recover the code vector corresponding to a given index, we simply reverse the process. Let i be an index computed by the above procedure, and again start with $j = k = 0$. For the first pulse, we let $p = U(n-j,m-k)$. When $k > 0$, we start with $p = \frac{1}{2}U(n-j,m-k)$, and if $i < p$, then we have another pulse in the same position as the previous pulse. In this case we set $x_k = x_{k-1}$, $s_k = s_{k-1}$,

Algorithm 1 Convert a pulse vector (x_k, s_k) to an index i

Require: $x_k \leq x_{k+1}$

$i \Leftarrow 0$
$j \Leftarrow 0$
for $k = 0$ to $m - 1$ **do**
  $p \Leftarrow U(n - j, m - k)$
  if $k > 0$ **then**
    $p \Leftarrow \frac{p}{2}$
  end if
  while $j < x_k$ **do**
    $i \Leftarrow i + p$
    $j \Leftarrow j + 1$
    $p \Leftarrow U(n - j, m - k)$
  end while
  if $(k = 0$ or $x_k > x_{k-1})$ and $s_k = -1$ **then**
    $i \Leftarrow i + \frac{p}{2}$
  end if
end for
return i

$k = k + 1$, and move on to the next pulse. Now while $i \geq p$, there must be no more pulses in dimension j, so we subtract p from i, set $j = j + 1$, and update $p = U(n - j, m - k)$. Finally, when $i < p$, we know that the code vector must contain at least one pulse in dimension j, so we set $x_k = j$. If $i < \frac{1}{2}p$, then we set $s_k = 1$. Otherwise $s_k = -1$, and we subtract $\frac{1}{2}p$ from i. Finally we set $k = k + 1$ and return to the beginning to decode the next pulse. Eventually, we will have decoded m pulses and i will be 0. See Algorithm 2 for details.

Higher dimensions and larger numbers of pulses are possible so long as the total number of code vectors can fit in an integer register. Larger codebooks require extended-precision arithmetic. However, since the basic algorithms require only addition, subtraction, and comparison, such arithmetic is still relatively inexpensive.

Algorithm 2 Convert an index i to a pulse vector (x_k, s_k)

$j \Leftarrow 0$
for $k = 0$ to $m - 1$ **do**
  $p \Leftarrow U(n - j, m - k)$
  if $k > 0$ and ($\frac{p}{2} \leq i$ or $s_{k-1} = -1$) **then**
    $i \Leftarrow i + \frac{p}{2}$
  end if
  while $p \leq i$ **do**
    $i \Leftarrow i - p$
    $j \Leftarrow j + 1$
    $p \Leftarrow U(n - j, m - k)$
  end while
  $x_k \Leftarrow j$
  if $\frac{p}{2} \leq i$ **then**
    $s_k \Leftarrow -1$
    $i \Leftarrow i - \frac{p}{2}$
  else
    $s_k \Leftarrow 1$
  end if
end for
return (x_k, s_k)

III. Evaluating $U(n, m)$

If $O(nm)$ space is available, one can pre-compute a table of the required values of $U(n, m)$ directly from the recurrence, yielding an $O(n+m)$ algorithm. Using $U(n, m)$ requires only half as much storage as $V(n, m)$, since $U(n, m)$ is symmetric: $U(n, m) = U(m, n)$. It also saves m lookups when decoding or encoding a vector, but requires an extra n additions to compute the size of the codebook, which is needed to pack or unpack the resulting index.

However, small embedded devices may make even this amount of storage impractical. One can compute

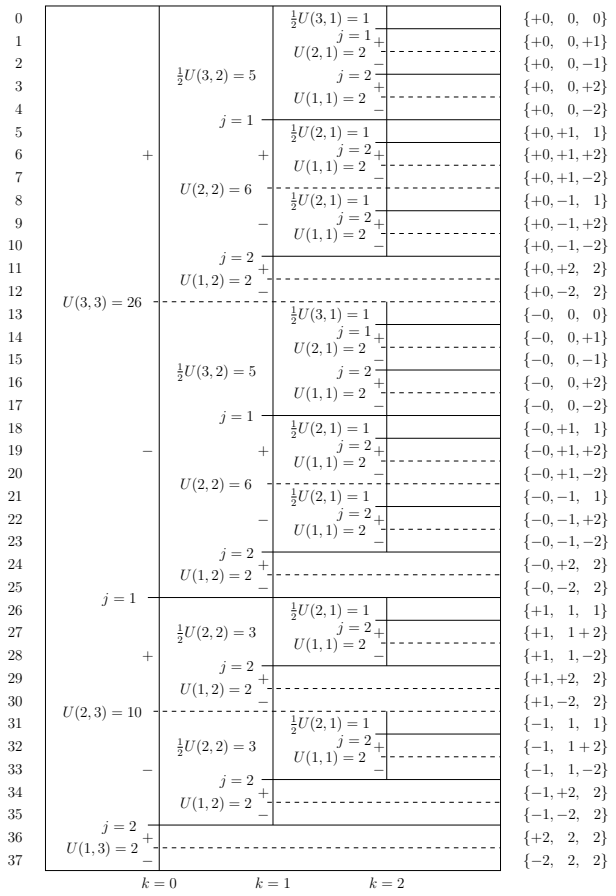

Fig. 1. Partitioning of the code space for a 3 dimensional vector with 3 pulses. Each column shows the decision levels for the kth pulse. Solid horizontal lines indicate where the position of the next pulse, j, is incremented. Dashed horizontal lines separate the positive and negative halves of each region. The left column indicates the index associated with each pulse vector. The right indicates the pulse vectors using the list of positions x_k, with the signs s_k attached to the first occurrence of each unique position.

$V(n,m)$ in $O(m)$ time using $O(1)$ storage by grouping the possibilities by the number of unique positions:

$$V(n,m) = \sum_{d=1}^{m} 2^d C(n,d) C(m-1,d-1),\ m > 0\ , \tag{8}$$

where $C(n,m)$ is the binomial coefficient. See the discussion of FPC in Section IV for details. However, (8) requires $3m$ multiplications and $2m$ divisions to evaluate and yields an $O(nm+m^2)$ algorithm.

A much better alternative uses n words of temporary storage to hold a single column of U, denoted u_m. Column u_{m+1} can be computed from u_m directly via (7). Similarly, u_{m-1} can be computed from

column u_m using the recurrence

$$\begin{aligned} U(n, m-1) = U(n, m) \\ - U(n-1, m) - U(n-1, m-1) \ . \end{aligned} \tag{9}$$

Encoding or decoding pulse k only uses values from column u_{m-k}. Thus we compute u_m from u_1 before the algorithm begins in $O(nm)$ time and replace it with u_{m-k-1} in $O(n)$ time at the end of each iteration. Further operations can be saved by noting that j never decreases, so only the first $n-j$ entries of the column need to be updated. The total algorithm is $O(nm)$, but requires no multiplications or divisions. Hence, the complexity scales linearly with the number of machine words required to represent an index, i. For practical values of n and m, it runs at about 2.5 time slower than a table lookup implementation, a penalty that is dwarfed by the complexity of the rest of the codec.

IV. Related Work

The interleaved single-pulse permutation (ISPP) coding method used in the AMR-WB codec encodes the signs of the pulses by varying the order the pulse positions x_k are transmitted [7]. A permutation of the pulse indices can easily save $m-1$ sign bits, but this becomes inefficient as the total number of pulses grows large, since the number of possible permutations, $m!$, grows much faster than 2^m. Therefore the exact encoding uses a complicated set of rules for each possible value of m to mitigate this inefficiency. This makes it hard to extend to an arbitrary number of pulses. Even using these rules, with 6 pulses it still wastes about 1.5 bits.

The factorial pulse coding (FPC) method [8] eliminates these problems by using an exact enumeration algorithm, as we do. It enumerates the same possibilities as CWRS, but in a completely different order, based on (8). FPC partitions the code space according to the number of unique, non-zero pulse positions, d. That is, the locations of the d unique pulses are identified with an index F into the $C(n, d)$ possibilities (combinations without replacement). The magnitudes of the pulses are identified with an index D into the $C(m-1, d-1)$ possibilities (combinations with replacement). Finally, 2^d bits are used to indicate the signs.

Translating a pulse vector to an index and back with FPC requires three separate stages and a total of $O(n+m)$ values of $C(n, m)$. There is no reason these cannot be computed with $O(nm)$ additions and subtractions using the same technique we used in Section III. However, to identify d from the index when decoding, one must skip past each of the terms of the sum in (8) until the correct one is located. This requires at least $O(\min(m, n))$ multiplications, and separating D and F out of the combined index

requires one additional division. The multiplications are required even in a table lookup implementation[2]. In contrast, CWRS computes a unified index in a single stage, which allows for smaller code size, and does not require any multiplications or divisions, which can be very expensive for large codebooks where extended-precision arithmetic is needed.

V. Conclusion

We have developed a new enumeration algorithm for a popular algebraic codebook used in various speech codecs. Every code vector is represented with the same number of bits, and that number is the smallest possible for the size of the codebook. The translation from vector to codeword and back is accomplished with two simple algorithms that run in $O(n+m)$ time with a small lookup table. All possible codebooks can share the same lookup table, allowing great flexibility in varying n and m without requiring excessive storage. When table storage is not available, we have given $O(nm)$ alternatives that still require no multiplications or divisions. Since the cost of the arithmetic scales linearly with the maximum codeword size, CWRS scales well to very large codebooks.

References

[1] C. Laflamme, J.-P. Adoul, H.-Y. Su, and S. Morissette, "On reducing computational complexity of codebook search in CELP coder through the use of algebraic codes," in *Proc. of the* 15^{th} *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'90)*, vol. 1, Apr. 1990, pp. 177–180.

[2] J.-P. Adoul and C. Lamblin, "A comparison of some algebraic structures for CELP coding of speech," in *Proc. of the* 12^{th} *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'87)*, vol. 12, Apr. 1987, pp. 1953–1956.

[3] J. Hamkins and K. Zeger, "Asymptotically dense spherical codes–part I: Wrapped spherical codes," *IEEE Transactions on Information Theory*, vol. 43, no. 6, pp. 1786–1798, Nov. 1997.

[4] N. J. A. Sloane, "Tables of sphere packings and spherical codes," *IEEE Transactions on Information Theory*, vol. 27, no. 3, pp. 327–338, May 1981.

[5] J.-P. Adoul and M. Barth, "Nearest neighbor algorithm for spherical codes from the Leech lattice," *IEEE Transactions on Information Theory*, vol. 34, no. 5-I, pp. 1188–1202, Sept. 1988.

[6] A. G. Burr, "Spherical codes for m-ary code shift keying," in *Proc. of the* 2^{nd} *National Conference on Telecommunications*, Apr. 1989, pp. 67–72.

[7] "AMR wideband speech codec: Transcoding functions," 3GPP Technical Specification TS 26.190, 2000.

[8] J. P. Ashley, E. M. Cruz-Zeno, U. Mittal, and W. Peng, "Wideband coding of speech using a scalable pulse codebook," in *Proc. of the 2000 IEEE Workshop on Speech Coding*, Sept. 2000, pp. 148–150.

[2]The multiplications could be eliminated by using an $O(\min(m,n))$ table for each codebook—that is, for each unique (n,m) pair. When the number of pulses is not fixed, the number of such pairs is large. This can require significantly more storage than $U(n,m)$ or $C(n,m)$, and is not suitable for the small embedded devices speech codecs must run on.