On Generating Large Binary Linear Error-Correcting Codes

R. E. Sabin
Computer Science Department
Loyola College
Baltimore, MD 21210 USA
E-Mail: RES@loyola.edu

Abstract. To determine the error-correcting capability of a large error-correcting code it may be necessary to generate the code, an intractable task. Using a stack-based algorithm and utilizing structural properties of a code can reduce time required. Timing results are reported for generating large codes using these methods on massively parallel platforms.

1. Introduction

Linear codes were the first error-correcting codes studied [5]. Among them are the well-known BCH codes, Reed-Solomon codes, and Goppa codes. A linear code of length n and dimension k is simply a k-dimensional subspace of F^n, the space of all n-tuples with elements from finite field F = GF(q).

In the search for good codes, the coding theorist needs to know the probability that a transmitted codeword will be correctly decoded. Important in such a determination is a knowledge of the weights of codewords. In some circumstances, the number of codewords with each weight that occurs, the weight distribution of the code, can be determined without the generation of the code: the MacWilliams identities [7] and Pless power moments [9] provide such means. When such methods are not applicable, the entire (n,k) linear code may be generated by taking all linear combinations of a k x n generating matrix whose rows are basis vectors of the code. Requiring at least q^k operations, linear code generation belongs to the set of computational problems referred to as NP-complete [1].

Here we discuss code generation techniques with a view to making the task more feasible for large (k > 30) binary codes. Empirical results are presented. A discussion of techniques appropriate for non-binary codes is presented in an appendix.

2. Linear Code Generation

Let C be a linear code in $GF(2)^n$ with generating array G whose k rows are a basis for C; C has length n and dimension k. A naive approach to code generation would involve counting in $GF(2)^n$, using the binary digits of each integer as the coefficients of the rows of G in the linear combination that is a codeword. Such a method is inefficient, requiring an average of k operations to produce a single codeword. An optimal algorithm to produce a code is one that has the minimum change property: every codeword, except the first, can be produced by adding a single basis vector to the previous codeword generated. Gray codes [4] provide such an algorithm. We

JCMCC **19** (1995), pp. 245-258

define the binary reflected Gray code of length k recursively with : interpreted as "followed by":

R (1) = 0 : 1

R (k) = 0 R(k-1) : 1 R'(k-1)

where 0 R(k-1) represents a 0 prefixed to each codeword of R(k-1) and R'(k-1) is the reflection of R(k-1), i.e., the codewords of R(k-1) listed in reverse order

In this paper, the term "binary Gray code" will refer exclusively to the binary reflected Gray code; also, to avoid confusion with the linear codewords being generated, we will refer to a Gray codeword, b, as a Gray "bit-sequence." Code C = { b· G : b ∈ R(k) }; each Gray bit-sequence "selects" rows of G to be combined, i.e., if b[i] is non-zero, G[i] is a summand of the codeword, c, being produced. Algorithm 1 generates codewords in this way:

Algorithm 1 - Binary linear code generation using binary Gray code

```
for j = 1 to k do                          /* initialize the bit-sequence */
    b[j] ← 0
c ← 0                                  /* c is initially the all zero codeword */
for j = 1 to 2^k - 1 do
    i ← 1
    while j mod 2^i = 0 do
        i ← i + 1
    c ← c + G[i]                                  /* alter the codeword */
    b[i] ← (b[i] + 1) mod 2                       /* alter the bit-sequence */
```

To generate the next linear codeword in the sequence, however, the position, i, of the bit of the Gray bit-sequence that should be complemented is needed, not the Gray bit-sequence itself. This transition sequence may be generated by maintaining a simple stack of integers [2]. The inner loop of algorithm 1 and all references to b are thus eliminated. In Algorithm 2, the stack is stored in array Stack, indexed [0...k] with the stack top at Stack[0]; t stores successive values in the transition sequence:

Algorithm 2 - Binary linear code generation using Gray code transition sequence

```
for j = 0 to k do
    Stack[j] ← j + 1                           /* initialize the stack */
c ← 0    t ← 0
while t ≤ k do
    t ← Stack[0]      /* the next value in the transition sequence */
    c ← c + G[t]
    Stack[0] ← 1          /* these 3 statements "pop" the stack */
    Stack[t-1] ← Stack[t]
    Stack[t] ← t + 1
```

Each loop iteration produces a single codeword. If each row of G, a sequence of bits, is considered to be a single integer or a sequence of integers, a codeword requires a minimum of three arithmetic operations. In this case, for codes with $k \geq 10$, Algorithm 2 reduces the number of operations of Algorithm 1 by a factor of approximately 2.7. To store the weight distribution of the code in array W, where W[i] is the number of codewords of Hamming weight i, append the following two lines to the body of the while loop of Algorithm 2:

$i \leftarrow$ weight of c
$W[i] \leftarrow W[i] + 1$

Weight determination of a single codeword is speedily accomplished if each row of G is considered a sequence of integers and a bit counting instruction is available [11]. If such a function is not available, table look-up is most efficient where, for a suitably large h, a table indexed $[0...2^h - 1]$ stores the number of bits in the binary representation of the index and n/h entries are summed to produce the weight of the codeword.

3. Optimizations

3.1 Utilizing computer architecture

Linear code C can be partitioned into cosets $x + C'$, where C' and H' are subspaces of C, $x \in H'$, $C = C' \oplus H'$ and $C' \cap H' = 0$ (the all zero vector). Thus, the generation of C can be divided into subtasks of equal size, each the generation of a distinct coset $x + C'$. Since each coset may be generated by the same instruction sequence, a single processor with array processing capabilities or a multi-processor system processing a single instruction stream are appropriate for linear code generation.

3.1.1 Single processor with vectorization

For large n, a vector processor can greatly speed code generation without radical alteration of the algorithm. Assume that the code to be generated is binary and that the vector length of the single processor is $v = 2^h$ and memory banks are allocated in column-major order (as in FORTRAN). If C has dimension k and generating array G, let subcodes H' and C' have dimensions h and $k - h$ respectively. Form array H indexed $[1...v]$ containing all codewords of H', each H[i] for $1 \leq i \leq v$ a distinct linear combination of the last h rows of G. Code C' has as its basis the first $k - h$ rows of G. As Algorithm 4 executes, a single element in the register originally containing H will contain all codewords in a single coset $x + C'$.

Weight determination is accomplished by use of v weight arrays, W_i, $1 \leq i \leq v$. In the initial for loop, as well as the inner for loop, the weight of H[i] is recorded in W_i. After the entire code is generated, the W_i are coalesced into a single weight table.

Alternatively, v copies of the stack may be maintained and the while loop may be nested in a vectorized for loop, although this approach has proven to be less efficient.

Algorithm 3 - Binary code generation utilizing vector length $v = 2^h$

```
for i = 0 to k do                                   /* initialize the stack */
        Stack[i] ← i + 1
for i = 1 to v do          /* take all linear combinations of last h rows of G */
        determine H[i]
t ← 1
while t ≤ k- h do
        for i = 1 to v do                                        /* vectorize */
                H[i] ← H[i] + G[t]
        Stack[0] ← 1
        Stack[t-1] ← Stack[t]
        Stack[t] ← t + 1
        t ← Stack[0]
```

3.1.2 The use of multiprocessors

If SPMD (Same Program Multiple Data) operations are supported by the system, an algorithm similar to Algorithm 3 allows each of p available processors to generate 1/p of the code. If $p = 2^h$, each processor is seeded with a distinct linear combination of h rows of G and G is truncated by the removal of those rows. Then each processor executes the same instruction, adding the same entry from the truncated generating array which has been broadcast to all processors. Theoretically, execution time is reduced by a factor on the order of p.

Algorithm 4 -Binary code generation utilizing $p = 2^h$ processors

```
for j = 1 to p do
        determine Seed_j      /* a linear combination of last h rows of G */
        send Seed_j to processor j
        send G' to processor j          /* G' is G minus the last h rows */
At each processor j:
        for i = 0 to k do                                  /* initialize the stack */
                Stack[i] ← i + 1
        i ← 1
        while i ≤ k-h do
                Seed_j ← Seed_j + G'[i]
                Stack[0] ← 1
                Stack[i-1] ← Stack[i]
                Stack[i] ← i + 1
                i ← Stack[0]
```

As with vectorization, weights of successive $Seed_j$ are determined and stored in p weight arrays that finally are coalesced into a single weight table.

3.2 Based on the structure of G

3.2.1 If G is systematic

If generating array G is in systematic form, i.e., G = [I | G″], the identity submatrix, I, can be ignored and shortened codewords of length n-k generated. The weight of each must be incremented by the number of rows that appeared in the linear combination used to produce the shortened codeword. This requires the generation of the Gray bit-sequence representing the linear combination and the determination of its weight. If Algorithm 2 or 3 is used, at least 3 additional operations are required: maintain the Gray bit-sequence, determine its weight, add that weight to the weight of the shortened codeword. If, when G and G″ are represented as arrays of integers, G″ has fewer columns than G, this method can improve performance.

3.2.2 If G has circulant structure

If the each row, except the first, of the generating array, G, is a cyclic shift by a constant number of places of the preceding row, G has circulant structure, and every linear combination of rows of G is a cyclic shift of some linear combination that includes the first row. Since such shifts have the same Hamming weight, we generate those codewords that result from linear combinations that include the first row. Entries on the weight table are incremented by the number of shifted versions of the generated codewords that exist.

All cyclic codes have a generating array with circulant structure: let the generator polynomial g(x), a factor of $x^n - 1$, be the first row of G. Each successive row is x times the preceding row. Quasi-cyclic codes also have generators of this type [8].

For binary codes, a Gray bit-sequence b_i, $1 \le i < 2^k$, represents a linear combination of rows of G summed to produce one codeword (as in 2.1). Let b_i = 1100...0 indicate the sum of rows 1 and 2 of the generating array. The number of trailing zeros z_i in b_i indicates the additional number of codewords that are "shifted" versions of this codeword. The sequence z_i is decreasing: $z_1 = k-1$ and z is decremented by 1 only when a new highest number in the transition sequence t_i is attained. For k = 4:

t:	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
b:	1000	1100	0100	0110	1110	1010	0010	0011	1011	1111	0111	0101	1101	1001	0001
z:	3	2	2	1	1	1	1	0	0	0	0	0	0	0	0

For Algorithm 5, the rows of generating array G are renumbered [0...k-1] and h, the highest value thus far produced in the transition sequence, is maintained. Then the number of "shifted" versions of the current codeword in the code plus one (for the codeword itself) is represented as s (s = z + 1) and is used to increment the weight table.

Algorithm 5 - Binary linear code generation utilizing shifting

```
for j = 0 to k do                                   /* initialize the stack */
    Stack[j] ← j + 1
c ← G[0]
h ← 0  /* initialize the highest value in the transition sequence */
s ← k                                   /* the number of cyclic shifts of c */
a ← weight of c
W[a] ← W[a] + s                                     /* record the weight */
t ← 0
while t < k do
    t ← Stack[0]
    If t > h                            /* a new highest transition value */
            h ← t
            s ← s-1
    c ← c + G[t]
    a ← weight of c
    W[a] ← W[a] + s                     /* increment the weight table */
    Stack[0] ← 1                                    /* pop the stack */
    Stack[t-1] ← Stack[t]
    Stack[t] ← t + 1
```

3.2.3 Reduction of the overhead of stack maintenance

Algorithms 2, 3, and 5 may be altered slightly to generate multiple codewords per loop iteration, thus reducing the number of iterations of the code-producing loop and the overall number of operations required for stack maintenance. To generate q^i codewords per iteration, select i of the original basis vectors; let G contain the remaining vectors and be indexed [1...k-i]. Each codeword generated in the loop is incremented by all linear combinations of the i selected basis vectors. For a binary code, the number of operations performed in the while loop is reduced to $(2 + 2^i)(2^{k-i} - 1)$. The optimal value of i depends upon the capabilities of the available compiler and hardware.

3.3. Avoiding generation of the entire code

If the entire weight table is not needed, probabilistic approaches permit the determination of minimum weight with a high degree of certainty [6], but here we limit discussion to methods that involve the actual generation of a portion of the code.

3.3.1 If G has circulant structure

If G has the structure described in 3.2.2, representatives of each weight class can be determined by use of Algorithm 5 with all references to s and h removed and Wt[a] + 1 replacing Wt[a] + s. We will refer to this variation as **Algorithm 5A**.

3.3.2 Determination of minimum weight

If the (n,k) binary code to be generated is cyclic, it has been shown [10] that to show that a code has minimum distance d, we need only examine those codewords whose first k bits have weight w^* satisfying

$$w^* \le \lceil d/p - 1 \rceil \quad \text{where } p = \lfloor n/k \rfloor.$$

If generating array G is placed in systematic form, these codewords may be generated by taking linear combinations that correspond to Gray bit-sequences with w^* or fewer 1's. An algorithm has been proposed to generate the desired subsequence of Gray bit-sequences [12]. This algorithm does not vectorize, but does lend itself to parallel processing. For sufficiently large w^*/k ratios, this method may be effectively used to accept or eliminate codes that are candidates for having the best minimum distance for any code of like length and dimension.

4. Empirical results

4.1 The array processing platform

The Cray Y-MP C90-16/512 supercomputer, at the Pittsburgh Supercomputing Center, has 16 vector processors. Each processor has 8 vector registers, each consisting of 64 8-byte scalar registers grouped together, and 4 functional units dedicated to vector processing. With a clock cycle of 4 nanoseconds, a processor can do, by use of chaining, a vectorized add and multiply in one clock cycle. C90 programs listed below ran on a single processor.

4.2 The MPMD platform

The Cray T3D-256 at the Pittsburgh Supercomputing Center is a scalable parallel supercomputer with 256 DEC Alpha processing elements with a peak aggregate speed of 38.4 Gflops and a total memory of 4 GBytes. The T3D's topology is that of a three-dimensional torus and the memory, 16MB per processor, is logically shared but physically distributed. The PSC T3D is attached to the C90 through which jobs are submitted. At the time of this research the T3D at was in pre-production status.

4.3 The generating programs

Binary code generators to evaluate the algorithms above were constructed in FORTRAN77 for execution on the C90 and the T3D. Programs reported below had the following characteristics:

1. All data structures, including the generating array contained integers, treated as sequences of 64 bits. (Integers are by default 64 bits on both the C90 and T3D.) The metrics for the code as well as the generating array were pre-built and fetched from a data file.
2. Selection of rows of the generating array was made by use of Algorithm 2, a stack that produces the transition sequence.
3. Addition of rows was accomplished by use of exclusive or (IEOR), a predefined, bit manipulation function available on all Cray systems.

4. Weight of codewords was determined by use of Popcnt, a predefined Cray function that returns the number of ones in the internal representation of its single argument. (Table look-up for weights was abandoned as it was found to increase execution time by roughly a factor of two.)
5. All programs executed on the C90 were compiled on the C90 with requests for aggressive optimization and for bringing inline of small routines. Vectorization, when used, was implicit, i.e., no compiler directives were used. The codeword producing loop was judged to have successfully vectorized only after examination of the listing (.l) file showing loop markings that resulted from compilation with the command cft77 -e m filename.
6. Programs executed on the T3D utilized Parallel Virtual Machine (PVM) routines for communication among processors. The generating array was input to a single processor (PE0), which readied the data for use, then broadcast the data to 63 additional PEs. The number of processing units was chosen to be equal to the vector stride of the C90 to allow easier comparison of test results from the two platforms. Each PE calculated a coset of equal size and determined a weight table for that portion of the code, sent the result to PE0 which added all weight tables to produce and print the final result.

Ten programs were constructed and tested:

Program A: Implementation of Algorithm 2.
The while loop, dependent upon values in a stack, does not vectorize. Intended for execution on the C90.

Program B: Implementation of Algorithm 3 with vector length 64.
Intended for execution on the C90.

Program C: Implementation of Algorithm 3 with vector length 64 and minimization of stack operations (as in 3.2.3).
Intended for execution on the C90. Program B was altered to generate 8 codewords per iteration of the inner for loop. It was empirically determined that the code was optimal at 8 codewords per iteration.

Program D: Implementation of Algorithm 4 with 64 processors.
Intended for execution on the T3D. Processing element 0 (PE0) functioned as the master and distributed the generating array and an initial codeword to PE1 through PE63. Then each processor, PE0-PE63, generated 1/64th of the code, sent back a weight distribution array which PE0 coalesced into one weight table and reported.

Program E: Implementation of Algorithm 5 with vector length 64.
Intended for execution on the C90. Algorithm 5 was altered (in a manor similar to the alteration of Algorithm 2 to produce Algorithm 4) to include a vectorizable for loop within the while loop that references a predetermined array, H, consisting of all codewords of a dimension 6 subcode.

Program F: Implementation of Algorithm 5 with vector length 64 and minimization of stack operations.

Intended for execution on the C90. Program E was altered to generate 8 codewords per iteration of the inner for loop.

Program G: Implementation of Algorithm 5 with 64 processors.
Intended for execution on the T3D with a methodology similar to Program D.

Program H: Implementation of Algorithm 5A with vector length 64.
Intended for execution on the C90. Algorithm 5A was altered in a manner similar to that used for Algorithm 5 (above) to allow vectorization.

Program I: Implementation of Algorithm 5A with vector length 64 and minimization of stack operations.
Intended for execution on the C90. Program H was altered to generate 8 codewords per iteration of the inner for loop.

Program J: Implementation of Algorithm 5A with 64 processors.
Intended for execution on the T3D with a methodology similar to Program D.

Note that Programs H, I, and J do not generate the entire weight table, but guarantee that a codeword of every possible weight is produced.

4.4 The test set

Five linear codes (n,k) were used for testing: (55,10), (55,20), (63,31), (105,24), and (111,36). For lengths 55 and 63, he bit-length of an integer on the C90 and the T3D, the generating array was of one dimension. For lengths 105 and 111, two-dimensional generating arrays were used.

5. Timing and Results

All runs were submitted in batch mode with the code and data sets fetched from mass storage. No significant operations involved floating point numbers, so the megaflop rate was not an appropriate metric for comparing results. Rather the run-time for each code is compared. In Table 1, execution time for a program run on the C90 represents the user CPU seconds reported by the job accounting utility (ja) for the execution of the program with the given data set. For jobs submitted to the T3D, ja does not provide appropriate statistics since its report of the execution time in the massively parallel environment includes time used for system requests initiated by the T3D such as disk I/O. For the T3D, a Cray-supplied function, secondr, was called at the beginning and end of each program. The difference in clock times so reported are displayed in Table 1. Program A was not used to generate the dimension 36 code, as the estimated execution time was in excess of 7 CPU hours. Figure 1 displays the CPU times for generation of codewords of the (111,36) code. Programs that utilize only the vectorization of the C90 are displayed as C90 1, while C90 2 indicates those programs that utilize vectorization and attempted to minimize stack operations. Results from programs run on the T3D are also displayed.

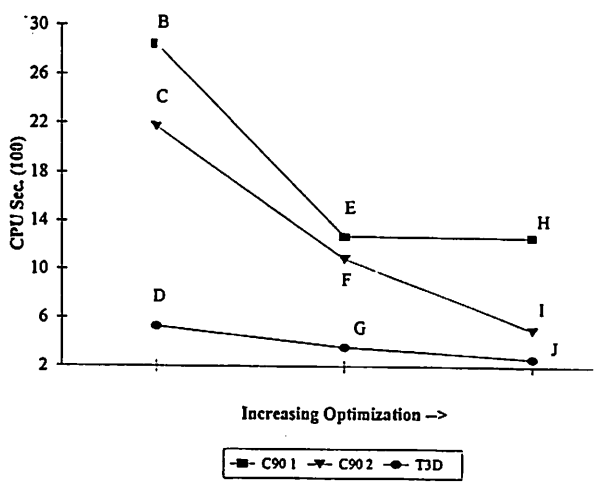
In each case, execution time includes the time required for input of the generating array, creation of any auxiliary arrays, generation of code words, determination of

weight tables, coalescing the weight tables to a single table, and the output of the result.

Table 1 - Execution time in CPU seconds for codes of varying length and dimension (n,k)

Prog.	Plat.	(55,10)	(55,20)	(63,31)	(105,24)	(111,36)
A	C90	.0014	.3643	747.8	6.845	——
B	C90	.0018	.0354	71.63	.7054	2846
C	C90	.0018	.0245	47.24	.5236	2172
D	T3D	.0663	.0863	13.10	.1833	528.7
E	C90	.0022	.0198	35.40	.3202	1274
F	C90	.0022	.0137	23.44	.1186	1088
G	T3D	.0805	.0887	7.1635	.1424	359.0
H	C90	.0022	.0118	25.12	.2708	1268
I	C90	.0022	.0051	5.120	.1276	504.2
J	T3D	.1805	.1035	6.961	.1276	266.1

Fig. 1 - Generation of (111,36) Code



6. Discussion

The author had previously generated several of these codes on single processors. On a Decstation 3100, the generating time for the dimension 31 code exceeded 10 CPU hours. On a VAX 6800, the dimension 20 code required approximately 5 CPU hours. As expected, the C90 performed much more efficiently. Vectorization on the C90 (Program B) reduced these times by 90%. Minimizing the stack operations, by generating eight codewords/loop iteration, in Program C, further dramatically reduced the time. Many linear codes have a generating array with circulant structure allowing the improved performance of Programs E and F. If only weight class representatives are needed, Program I shows the best performance. It is worthy of note that the use of a two-dimensional generating array does not always double the generation time for a code of like dimension but a generating array of one dimension, as would be expected. For example, the (111,36) code contains 32 times the number of codewords as the (63,31) code, yet Program F generates the larger code in 1088 seconds instead of the expected 1500 seconds.

For all codes of dimension 24 or greater, the massive parallelization of the T3D dramatically reduced the time needed to generate an entire code. Apparently, the overhead of PVM calls adversely effected run-times for smaller codes. Since such calls are a one-time expense, generation of larger codes, i.e., of codes with dimension greater than 36, should show a time savings of at least 80%. We note that in order to allow direct comparison of vectorization on the C90 with parallelization on the T3D, only 64 of the available 512 processors of the T3D were utilized. Were all processors used, a cyclic (128,40) code would be expected to be generated by Program G in approximately 0.2 CPU hours. A representative of each weight class of this code could be generated by Program J, using 512 processors, in less than 9 CPU minutes.

7. Conclusion

The use of massive parallelization promises to allow the examination of large codes that have hitherto been impossible to examine [14]. The author intends to further test massively parallel methodologies for code generation in the T3D environment, including the minimization of stack operations, and to use successful methods to further investigate a class of quasi-cyclic linear codes [13].

Author's Address: Loyola College, Comp. Sc. Dept., 4501 N. Charles St., Baltimore, MD 21210. E-mail: RES@loyola.edu

References

[1] E. R. Berlekamp, R. J. McEliece, and H. C. vanTilborg, "On the inherent intractability of certain coding problems," IEEE Trans. on Inf. Th. vol. IT-24 pp. 384-386, May, 1978.

[2] J. R. Bitner, G. Ehrlich, and E. M. Reingold, "Efficient generation of binary

reflected Gray code and its applicantions," Comm. of ACM vol. 19 pp. 517-521, Sept., 1976.
[3] M. Cohen, "Affine m-ary Gray codes," Inf. and Control vol 6 pp. 70-78, 1963.
[4] I. Flores, "Reflected number systems," IRE Trans. on Electron. Comput. vol. EC-5 pp. 79-82, June, 1956.
[5] R. W. Hamming, "Error detecting and error correcting codes," Bell Syst.Tech. J. vol. 29 pp. 147-160, 1950.
[6] J. S. Leon, "A probabilistic algorithm for computing minimum weights of large error-correcting codes," IEEE Trans. on Inf. Th. vol. 34 pp. 1354-1359, Sept., 1988.
[7] F. J. MacWilliams and N. J. A. Sloane, The Theory of Error-Correcting Codes, North-Holland: New York, NY, 1977.
[8] W.W. Peterson and E.J. Weldon, Jr., Error-Correcting Codes, The MIT Press: Cambridge, MA, 1972.
[9] Pless, V., Introduction to the Theory of Error-Correcting Codes, John Wiley and Sons: New York, NY,1982.
[10] G. Promhouse and S. E. Tavares, "The minimum distance of all binary cyclic codes of odd lengths from 69 to 99," IEEE Trans. on Inf. Th. vol. IT-24 pp. 438-442, July, 1978.
[11] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms, Theory and Practice, Prentice Hall: Englewood Cliffs, NJ, 1977.
[12] R.E. Sabin, "A Simple Algorithm for Generating Subsets of k or Fewer Elements of an n-Set," submitted to Journ. of Comb. Math. and Comb. Computing.
[13] R.E. Sabin, "On Row-Cyclic Codes with Algebraic Structure," Designs, Codes and Cryptography vol. 4 pp. 145-155, 1994.
[14] D. Schomaker and M. Wirtz, " On binary cyclic codes of odd lengths from 101 to 127," IEEE Trans. on Inf. The. vol. 38 pp. 516-518, Mar., 1992.
[15] B. D. Sharma and R. K. Khanna, "On m-ary Gray codes," Inf. Sc. vol. 15 pp. 31-43, 1978.

APPENDIX: Non-binary codes

For non-binary linear codes, a generalized Gray code [3] can be used to produce codewords. Let F = GF(q) = {0, 1, . . . , q-1}; for simplicity, let the word "bit" refer to any element of F. Let R(i) to be a sequence of i bit strings and let jR(i) represent the elements of R(i) with j affixed to each as a leading symbol. With : interpreted as "followed by", a recursive definition of a generalized Gray code of length k, R(k), is stated:

$R(1) = 0 : 1 : 2 : \ldots : q-1$

$R(k) = 0\,R(n-1) : 1\,R\,(n-1) : 2\,R\,(n-1) : \ldots : (q-1)\,R^{q-1}(n-1)$

where R (n-1) is produced by cyclically shifting by i positions to the right (with wrap-around) the order of the listing of bit-sequences in R(n-1)

As in the binary case, the transition sequence [15] is of interest. We generalize algorithm 2 in two steps. First, we generate the Gray Bit-sequences, b, by using a stack of integers and two auxiliary arrays: Used, where Used[i] indicates the number of field elements that have been used in position i since a change in position i+1, and Next, where Next[i] indicates the next field element that should be used at position i in b. The stack is popped only if the maximum number of alterations, q-1, (stored in Used) have been made to the position under consideration, t; at each iteration all j, $1 \le j < t$ are pushed onto the stack.

Algorithm 2* - Stack-based generation of q-ary Gray code

```
for j = 1 to k do                                  /* initialize data structures */
    Stack[j-1] ← j
    Used[j] ← 0
    Next[j] ← 1
max ← q-1
b ← 0      t ← 0
while t ≤ k do
    t ← Stack[0]
    b[t] ← Next[t]                  /* replacethis line to generate the code */
    Used[t] ← Used[t]+1
    Next[t] ← Next[t] mod q+1
    If Used[t] = max                                /* pop top element */
        Stack[t-1] ← Stack[t]
        Stack[t] ← t+1
        Used[t] ← 0
    else if t > 1                                    /* push all j < t */
        Stack[t-1] ← t
    Stack[0] ← 1
```

For linear code generation, elements of F may be represented by $m = \lceil \log_2 q \rceil$ bits; an element of F^n can be represented by a sequence of $\lceil mn/b \rceil$ or more integers where b is the number of bits used to store a single integer. The elements of GF(q) can be mapped to m-length bit vectors in a variety of ways, dependent upon the choice of generating polynomial for GF(q). If q is a power of 2, addition can be accomplished by use of the exclusive OR; otherwise addition is accomplished by table look-up.

To alter Algorithm 2* to generate a non-binary linear code, an augmented generating array G with k rows and q columns is used. If the original generating array, G, is a one-dimensional array of k integers, g[i], $1 \le i \le k$, G is embedded in G′ as column 1. Interior elements, g[i,j], $1 \le i \le k$, $1 < j < q$, must satisfy $g[i,j] = j \cdot g[i,1] - (j-1) \cdot g[i,1]$ and $g[i,q] = -(q-1) \cdot g[i,1]$, where · indicates scalar multiplication in GF(q). With | indicating concatenation of matrices, G′ is :

$$G' = [\ G \mid 2\cdot G - G \mid 3\cdot G - 2\cdot G \mid \ldots \mid (q-1)\cdot G - (q-2)\cdot G \mid -(q-1)\cdot G\ ]$$

Once G' is created, the linear code may be generated by Algorithm 2*, replacing the assignment b[t] ← Next[t] with c ← c + G[t,Next[t]], with addition accomplished by either exclusive OR or table look-up.

The Hamming weight of a codeword can be found by either table look-up or by counting the number of non-zero field elements in the vector. Since, each non-zero codeword may be multiplied by any non-zero element of F to produce another codeword of identical Hamming weight, the entire code need not be generated to determine the weight table. It is sufficient to generate those codewords whose leading non-zero bit is a certain element of F. To generate those codewords whose leading non-zero is a "1", we alter Algorithm 2* slightly but maintain the minimum change property. Algorithm 3* "pops" the stack immediately after the position on the stack's top, i at Stack[0], has first been altered. It produces 1/(q-1) of the entire code. Entries in the weight table for non-zero weights are multiplied by (q-1) to produce the final table.

Algorithm 3* - Generating 1/(q-1) Non-Binary Linear Codewords

```
for j = 1 to k do
        Stack[j - 1] ← j
        Used[j] ← 0
        Next[j] ← 1
        Disturbed[j] ← false
max ← q- 1
c ← 0    t ← 0                                  /* initialize the codeword */
while t ≤ k do
        t ← Stack[0]
        c ← c + G[t,Next[t]]
        If not Disturbed[t]
                Disturbed[t]← true
                Used[t] ← q- 1                  /* to force a pop */
        else
                Used[t] ← Used[t] + 1
                Next[t] ← Next[t] mod q + 1
        If Used[t] = max                        /* pop top element */
                Stack[t-1] ← Stack[t]
                Stack[t] ← t + 1
                Used[t] ← 0
        else if t > 1                           /* push all j < t */
                Stack[t1] ← Stack[0]
        Stack[0] ← 1
```