Symbol-by-Symbol APP Decoding of the Golay Code and Iterative Decoding of Concatenated Golay Codes

Li Ping, Member, IEEE, and Kwan L. Yeung, Member, IEEE

Abstract—An efficient coset based symbol-by-symbol soft-in/soft-out
APP decoding algorithm is presented for the Golay code. Its application
in the iterative decoding of concatenated Golay codes is examined.

Index Terms—Concatenated codes, coset decoding, Golay code, iterative
decoding, turbo codes, turbo decoding.

I. INTRODUCTION

The symbol-by-symbol soft-in/soft-out APP (a posteriori prob-
ability), also known as MAP (maximum a posteriori), decoding algorithm [1] plays an important role in the iterative decoding of
concatenated codes [2]–[4]. The exact APP decoding of the Golay
code can be accomplished by applying the BCJR algorithm to the
256-state minimum conventional trellis of C [5] or the eigenvector
algorithm [6] to the 16-state minimum tail-biting trellis of C [7].
The complexities of both methods are quite high. A lower cost
alternative is the approximate method of [6] applied to the 16-state
tail-biting trellis [7], resulting in complexity (normalized to operations
per information bit) of about twice the BCJR algorithm for a rate 1/2,
16-state conventional convolutional code.1

This correspondence presents an efficient, exact APP decoding algorithm
for C based on the coset decoding principle [8]–[12]. The complexity of the algorithm is comparable to the BCJR algorithm
for a rate 1/2, 16-state conventional convolutional code. We will examine the application of the proposed algorithm in the iterative
decoding of the concatenated Golay codes. We will show that, for
short interleaver lengths, the concatenated Golay codes can achieve performance similar to or better than the turbo codes.

II. PRELIMINARIES

In [8], Pless introduced an elegant 4 × 6 matrix construction of
C. In this section, we will express this method in a product form of
the hexacode and the SPC (single parity check) codes. We will then
consider the APP decoding problem for C and establish a starting
point for developing the new algorithm in the next section. For
simplicity, we will express C over {+1, −1}. It is related to C
over GF(2) by the standard mapping {+1 ↔ 0, −1 ↔ 1}.

A. Pless’ Construction of C

Denote by 0, 1, ω, ¯ω the four elements of GF(4), referred to as
characters. The hexacode is a [6, 3, 4] linear code over GF(4) with
the generator matrix [8], [9]

\[ \begin{bmatrix} 1 & 0 & 0 & 1 & \omega & \bar{\omega} \\ 0 & 1 & 0 & 1 & \bar{\omega} & \omega \end{bmatrix} \]  

The following are two mappings from the characters to 4 × 1 vectors
over {+1, −1}. The parity refers to the number of −1 in a column.

\[
\begin{align*}
\text{even}\; & : 1 
\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 
\end{bmatrix} \\
\text{odd}\; & : 1 
\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 
\end{bmatrix} \\
0 & : 
\begin{bmatrix} 0 & 1 & \omega & \bar{\omega} \\ 0 & 1 & \bar{\omega} & \omega \end{bmatrix}
\end{align*}
\]

Applying the even (resp., odd) interpretations to the 64 codewords in
the hexacode results in 64 4 × 6 binary arrays, collectively referred to
as H+ (resp., H−). Let P+ (resp., P−) be the length-6 even and odd
SPC codes over {+1, −1} containing an even (resp., odd) number
of −1, each with 32 codewords. Let

\[ h = \{ h[i, j] \} = [ h_1, h_2, h_3, h_4, h_5, h_6 ] \]

be a 4 × 6 array, where \{ h_i \} are 4 × 1 vectors. Denote by “∪”
the union of two nonoverlapping sets. It is straightforward to verify
the equivalence between the construction in [8] and the following
definition of C:

\[
C = C^+ \uplus C^-
\]  

\[
C^+ \uplus \{ h \in H^+, p \in P^+ \}
\]  

\[
C^- \uplus \{ h \in H^-, p \in P^- \}
\]  

\[
h \uplus p \triangleq [ h_{1,1} P_1, h_{2,2} P_2, h_{3,3} P_3, h_{4,4} P_4, h_{5,5} P_5, h_{6,6} P_6 ].
\]

B. The APP Decoding Problem of C

Let the transmitted codeword be a 4 × 6 array u = \{ u[i, j] \} ∈ C
and its noisy observation be x = \{ x[i, j] \}. The output of a soft-
in/soft-out APP decoder is [1], [2]

\[
L[i, j] = \frac{1}{2} \log \left( \frac{\Pr \{ u[i, j] = +1 \mid x \} }{\Pr \{ u[i, j] = -1 \mid x \} } \right).
\]

Here a factor of 1/2 is included for convenience. Let \v = \{ v[i, j] \}
be a 4 × 6 array of the bit confidence values conditioned on individual
received symbol, i.e.,

\[
v[i, j] = \frac{1}{2} \log \left( \frac{\Pr \{ u[i, j] = +1 \mid x[i, j] \} }{\Pr \{ u[i, j] = -1 \mid x[i, j] \} } \right).
\]

For example, v[i, j] = x[i, j]/σ² for an additive white Gaussian noise
(AWGN) channel of variance σ². Assuming that the entries of \v
are independent and all the codewords have equal probability of
occurrence, then (4) can be evaluated as [3]

\[
L[i, j] = \frac{1}{2} \log \sum_{e(x, w) \in C} \sum_{|e(x, w)|=1} e(x, w) \left( e(x, w) \right) \]  

\[
= \frac{1}{2} \log \sum_{e(x, w) \in C} \sum_{|e(x, w)|=1} e(x, w) \left( e(x, w) \right) \]  

\[
= \frac{1}{2} \log \sum_{e(x, w) \in C} \sum_{|e(x, w)|=1} e(x, w) \left( e(x, w) \right) \]  

\[
= \frac{1}{2} \log \sum_{e(x, w) \in C} \sum_{|e(x, w)|=1} e(x, w) \left( e(x, w) \right) \]  

\[
n\]  

0018-9448/99$10.00 © 1999 IEEE
where
\[ \langle c, v \rangle = \sum_{j=1}^{6} (c_j, v_j) = \sum_{j=1}^{4} \sum_{i=1}^{3} c[i, j] \cdot v[i, j]. \]

In (7), \( c_j \) and \( v_j \) are the \( j \)-th columns of \( c \) and \( v \), respectively. Substitute (3) into (7)
\[ \langle c, v \rangle = \langle \tilde{h} \cdot p, v \rangle = \sum_{j=1}^{6} (h_j \cdot p_j, v_j) = \sum_{j=1}^{6} p_j (h_j, v_j) = \langle p, w \rangle \]
with \( w = [w_1, \ldots, w_6] = [(h_1, v_1), \ldots, (h_6, v_6)] \).

The summations in (6) then become
\[ \sum_{e \in C^x} e^{(x, w)} = \sum_{e \in C^x} e^{(x, w)}, \quad x = e, o. \]

Equation (9) is over 4096 codewords for every \( e[i, j] \), which represents the bulk of the computation involved in (6). A straightforward summation is apparently very costly. We will explore an improved solution below.

III. EFFICIENT APP DECODING METHOD FOR C

In this section we will first introduce an \( h \)-coset partitioning of \( C \). We will show that (9) can be evaluated partially over each \( h \)-coset using a simple rule. The partial results can be efficiently combined through a set partitioning hierarchy of \( C \). These form the core parts of the new algorithm.

A. Partition of C using h-Cosets

Fixing any \( h \) in \( H^* \), we can obtain a unique subset of 32 codewords \( \{h \cdot p; p \in P^*\} \) according to (3b). We will call them collectively as an \( h \)-coset generated by \( h \). For all the elements in \( H^* \), we obtain 64 \( h \)-cosets. Similarly, another 64 \( h \)-cosets in the form of \( \{h \cdot p; p \in P^*\} \), each containing 32 codewords, can be generated by \( h \in H^* \). It can be shown that these 128 \( h \)-cosets form a coset partition of \( C \). Notice that \( h \cdot p = h \) only if \( p = [1, 1, 1, 1, 1, 1] \). Thus \( \{e \in C; e \cdot p \in P^*\} \) does not include its generator \( h \) since \([1, 1, 1, 1, 1, 1] \in P^* \). This distinguishes an \( h \)-coset generator from a coset leader.

Based on the above definition, (9) can be rewritten as (recall that \( e[i, j] = h[i, j] \cdot p_j, \) see (3))
\[ \sum_{e \in C^x} e^{(x, w)} = \sum_{e \in C^x} \sum_{p \in P^*} e^{(p, w)} \sum_{p_j \in E[i, j]/h[i, j]} \]
with \( x = e, o. \)

The inner summation above is over the 32 codewords in an \( h \)-coset and the outer one is over the 64 \( h \)-cosets for each parity. The following is a two-stage technique for evaluating (10).

B. Efficient Method for the Inner Summation in (10)

Recall that \( p \in P^* \) (resp., \( p \in P^* \)) must contain an even (resp., odd) number of \(-1\). It leads to a simple rule for evaluating the inner

2 Let \( \tilde{h} \) be the all-one \( 4 \times 6 \) array. Then \( \tilde{C} = \{\tilde{h} \cdot p; p \in P^*\} \) is a subcode in \( C \). Denoted by \( a \circ b \) the bit-by-bit multiplication between two arrays \( a \) and \( b \) over \([+1, -1]\), which is equivalent to \( a + b \) over GF (2). Fixing any \( c = h \cdot p \) (with \( h \) and \( p \) having the same parity), then \( c \circ \tilde{c} \in \tilde{C} \) is a coset of \( C \) led by \( c \). This coset is homomorphic to the \( h \)-coset generated by \( h \) since for any \( p \in P^* \)
\[ c \circ \tilde{c} = (h \cdot p') \circ (\tilde{h} \cdot p) = (h \circ \tilde{h}) \cdot (p' \circ p) = h \cdot (p' \circ p) \]
and \( p' \circ p \) has the same parity as \( p' \) (and so \( h \)).

The inner summation in (10)
\[ \sum_{p \in P^*} \sum_{p_j \in \tilde{C}[i,j]} e^{(p, w)} = e^{x \cdot w_j} \left( \frac{A^+}{a_j^+} + \frac{A^-}{a_j^-} \right) \]
\[ j = 1, 2, \ldots, 6 \]
\[ \sum_{p \in P^*} \sum_{p_j \in \tilde{C}[i,j]} e^{(p, w)} = e^{x \cdot w_j} \left( \frac{A^+}{a_j^+} + \frac{A^-}{a_j^-} \right) \]
\[ j = 1, 2, \ldots, 6 \]
with \( a_j^+ = e^{x \cdot w_j} \) and \( A^\pm = \prod_{j=1}^{6} a_j^\pm \).

Equations (11a) and (11b) are equivalent to the APP decodings for the SPC codes \( P^* \) and \( P^o \), respectively. They can be verified by expanding \( A^\pm \) and canceling out redundant terms.

C. Efficient Method for the Outer Summation in (10)

After completing (11) for 128 \( h \)-cosets, the results can be substituted into the outer summation in (10). A straightforward method is to sum over all the \( h \)-cosets for every output bit. In this way, some of the partial summations involve common \( h \)-cosets, which are duplicated for different output bits. The following technique can be used to avoid such unnecessary operations.

Denote by \( J(h_j) \) all the \( h \)-cosets whose generators have the \( j \)-th column fixed by \( h_j \). Similarly, denote by \( J(h_j, h_j') \) and \( J(h_j, h_j', h_j') \) the collection of \( h \)-cosets with two and three columns fixed, respectively (with \( j, j', j'' \) different). Then \( C^e \) and \( C^o \) can be partitioned progressively as shown below and (10) can be evaluated accordingly.

- For \( j \) fixed, \( C^e \) can be partitioned into four nonoverlapping subsets \( \{J(h_j): h_j = 0, 1, \omega, \omega^2\} \) using even interpretations. This is also true for \( C^o \) using odd interpretations. Thus (10) can be rewritten as
\[ \sum_{h_j \in H^2} \sum_{p \in P^*} e^{(p, w)} \]
\[ x = e, o. \]
Consider the braced part in (12) with \( e[i, j] \) and \( h_j \) fixed. Then \( p_j, \ e^{x \cdot w_j} \) and \( a_j^\pm \) (see (11)) are also fixed. Substituting (11) into the braced part in (12) we have
\[ \sum_{h_j \in H^2} \sum_{p \in P^*} e^{(p, w)} \]
\[ = \frac{e^{x \cdot w_j}}{2} \left( \frac{A^+}{a_j^+} + \frac{1}{a_j} \sum_{j \neq j} A^- \right) \]
\[ \sum_{h_j \in H^2} \sum_{p \in P^*} e^{(p, w)} \]
\[ = \frac{e^{x \cdot w_j}}{2} \left( \frac{A^+}{a_j^+} + \frac{1}{a_j} \sum_{j \neq j} A^- \right) \]
Furthermore, for a fixed \( j' \neq j \), \( J(h_j) \) can also be partitioned into four nonoverlapping subsets \( \{J(h_j, h_j') \} \) and so
\[ \sum_{J(h_j, h_j')} A^\pm = \sum_{J(h_j, h_j')} \]
\[ = 4 \text{ choices of } h_j \sum_{J(h_j, h_j')} A^\pm. \]
For a fixed \(j'' \neq j, j'\), \(J(h_j, h_{j'})\) can again be partitioned into four nonoverlapping subsets \(\{J(h_j, h_{j'}; h_{\mu} = 0, 1, \omega, \omega^2)\}\). Since the hexacode is MDS (maximum-distance separable, see [13]) with minimum distance \(d = 4\), any three columns, together with the parity, specify a unique \(h\) and so \(J(h_j, h_{j'}, h_{\mu})\) represents a unique \(h\)-coset. As different \(h\)-cosets cannot overlap, this partition of \(J(h_j, h_{j'})\) is invariant with respect to the choices of \(j''\). Thus the inner summation in (13c) can be uniquely expressed as

\[
\sum_{J(h_j, h_{j'})} A_{\pm} = \sum_{h_{\mu} \text{coset of } h} A_{\pm}.
\]

The key to efficiency improvement is avoiding the duplication of the summations in (13) and (14) for different output bits. Some details in evaluating (12)–(14) are given below.

i) We limit the \((j, j')\) pair in (14) to \((j, j + 1), j = 1, 3, 5\), referred to as blocks [9]. We first evaluate (14) for (16 character pairs) \(\times 3\) (blocks) \(\times 2\) (parities) = 96 possibilities of \((h_j, h_{j+1})\) pair. Each result of (14) is then used twice in (13c), where we set \(j' = j + 1\) for \(j = 1, 3, 5\) and \(j' = j - 1\) for \(j = 2, 4, 6\).

ii) We evaluate (13) for \(6\) (columns) \(\times 4\) (characters) \(\times 2\) (parities) = 48 possibilities of \(h_j\). Each result of (13) can be repeatedly used in (12) for up to three information bits in a column. The information positions for \(C\) defined in (3) can be chosen as \((i, j) = (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 1), (4, 1), (2, 2), (4, 2), (2, 3), \) and \((4, 3)\).

The above discussions are summarized below, with costs listed in brackets.

### D. APP Decoding Algorithm for \(C\)

**Preparation:**

i) Generate 96 possibilities of \(\{e^{\pm \omega^j}; j = 1, 2, \ldots, 6\}\), see the Appendix (24 exponentials and 66 multiplications).

ii) Generate 96 possibilities of \(\{a_j^{\pm}; j = 1, 2, \ldots, 6\}\) (96 additions).

iii) Generate 256 possibilities of \(\{A_j^{\pm}\}\) for 128 \(h\)-cosets. This can be done by first generating all the partial products \(a_j^{\pm}a_{j+1}^{\pm}\) and \(a_j^{-}\) and then multiplying three partial products together for every \(h\) (4 \(\times 4 \times 3 \times 2 \times 2 + 128 \times 2 \times 2 = 704\) multiplications).

**Step 1.** Evaluate (14) for 96 possibilities of \((h_j, h_{j+1})\) pair, with \(j = 1, 3, 5\) (96 \(\times 2 \times 3 = 576\) additions).

**Step 2.** Evaluate (13), with \(j = 1, 2, \ldots, 6\) and \(j' = j + 1\) if \(j = 1, 3, 5\) and \(j' = j - 1\) otherwise, for 48 possibilities of \(h_j\) (2 \(\times 3 \times 48 + 2 \times 48 = 384\) additions and \(4 \times 48 = 192\) multiplications, divide-by-2 ignored).

**Step 3.** For 12 information bits, evaluate (12) and then complete (6) (2 \(\times 2 \times 3 \times 12 + 2 \times 12 = 168\) additions, 12 divisions and 12 logarithms).

The total cost of the above algorithm is 974 multiplications, 1224 additions, 24 exponentials, and 12 logarithms (divisions counted as multiplications). The normalized cost is about 81 multiplications, 102 additions, 2 exponentials, and 1 logarithms per information bit.

For the purpose of comparison, the BCJR algorithm applied to a rate \(1/2\), \(N\)-state conventional trellis (non-tail-biting) requires about \(6N\) multiplications, \(4N\) additions, 2 exponentials, and 1 logarithm per information bit. The complexity of the proposed algorithm is thus approximately comparable to the BCJR algorithm applied to a rate \(1/2\), 16-state conventional trellis (about 96 multiplications and 64 additions per information bit).

It is interesting to note that the minimum trellis of \(C\) has 16-state [5], [7], [12]. However, this can only be achieved by the tail-biting trellis, to which the standard BCJR algorithm is not directly applicable. To the best of our knowledge, the most efficient APP algorithm for the tail-biting trellis is the approximate algorithm of [6, Algorithm A3]. For the same state number, the complexity of [6, Algorithm A3] is higher than the standard BCJR algorithm by a factor of \((W + L)/L\), where \(W\) is the wrapping length and \(L\) the trellis length. For short \(L\), we assume \(W \approx L\) (as adopted in the example in [6]). Applying the above to the 16-state tail-biting trellis of \(C\) developed in [7], the resultant cost is about twice of the proposed algorithm. Other alternatives, such as the eigenvector method in [6] (which is an exact method) applied to the 16-state tail-biting trellis of \(C\) or the BCJR algorithm applied to the minimal 256-state conventional trellises of \(C\), appear much more costly.

### IV. APPLICATIONS

Consider the application of the proposed algorithm to the concatenated Golay codes in Fig. 1. Let \(D\) be an \(I \times 12\) binary information array. Let \(D_1\) and \(D_2\) be two \(I \times 12\) arrays obtained from \(D\) using appropriate interlavings. Apply the systematic [24, 12, 8] Golay code row-wise to \(D_1\) and \(D_2\), producing two \(I \times 12\) parity-check
can be calculated with 96 exponentials. The following is an
alternative approach that is used for the complexity analysis in Section
III-D. Define \( t(x) = x \) if \( x \geq 0 \) and \( t(x) = 0 \) if \( x < 0 \) and let \( T_j = \sum_{i=1}^{4} t(v[i, j]) \). Then (see (8))
\[
E^+_{ij} = \exp \left( \sum_{j=1}^{4} (\pm h[i, j]v[i, j] - t(v[i, j])) \right).  \tag{16}
\]
Since \( h[i, j] \in \{+1, -1\} \), \( \pm h[i, j]v[i, j] - t(v[i, j]) \) results in either 0 or \(-2v[i, j]\). Fixing \( j \), all the 16 possible values of \( E^+_{ij} \) can be found as
\[
1, e^{-2v[1, j]} \quad e^{-2v[2, j]} \quad e^{-2v[3, j]} \quad e^{-2v[4, j]},
\]
\[
e^{-2v[1, j]+v[2, j]} \quad e^{-2v[1, j]+v[3, j]} \quad e^{-2v[1, j]+v[4, j]} \quad e^{-2v[2, j]+v[3, j]},
\]
\[
\]
\[
\]
which can be generated with 4 exponentials and 11 multiplications (multiply-by-2 ignored). For \( j = 1, 2, \ldots, 6 \), this amounts to 24 exponentials and 66 multiplications. Instead of calculating \( \{e^{h[i, j]}\} \) from \( E^+_{ij} \), we can also simply replace \( e^{h[i, j]} \) in (11) by \( E^+_{ij} \). This will not affect the final result of the algorithm in Section III-D since (11) is equivalent to (with \( T = T_1 + T_2 + \cdots + T_6 \))
\[
\begin{align}
&\text{for } h \text{ even: } e^{-T} \sum_{p \in F^+} e^{p[0, w]} \\
&= \frac{E^+_{ij}}{2} \left( \sum_{k=1}^{6} (E^+_{kj} + E^-_{kj}) \right) + \frac{6}{2} \left( \sum_{k=1}^{6} (E^+_{kj} - E^-_{kj}) \right) \tag{18a}
\end{align}
\]
\[
\begin{align}
&\text{for } h \text{ odd: } e^{-T} \sum_{p \in F^-} e^{p[0, w]} \\
&= \frac{E^+_{ij}}{2} \left( \sum_{k=1}^{6} (E^+_{kj} - E^-_{kj}) \right) + \frac{6}{2} \left( \sum_{k=1}^{6} (E^+_{kj} + E^-_{kj}) \right) \tag{18b}
\end{align}
\]
The extra factor \( e^{-T} \) above will be canceled out during the division in (6).

V. CONCLUSION

The trellis-based BCJR APP decoding algorithm [1] is most suitable for convolutional codes [2] or block codes with simple trellis structures [3]. When applied to block codes, the trellis-based approach may not be the best choice. This is due to the fact that the minimum trellises of block codes can be tail-biting [5, 7, 13], which are more difficult to decode than conventional trellises. Alternative methods have been explored [16, 17] for some block codes. This correspondence has shown that the coset-based technique provides a cost-effective approach to the APP decoding of the Golay code.

APPENDIX

All the 48 possible values of \( \{v[i, j] = h[k, v[j]]\} \) can be computed using the Gray code technique of [9] with 60 additions and then \( e^{\pm v[j]} \) can be calculated with 96 exponentials. The following is an anonymous reviewers.

ACKNOWLEDGMENT

The authors are very grateful for the detailed comments from the anonymous reviewers.

REFERENCES

Optimal and Near-Optimal Encoders for Short and Moderate-Length Tail-Biting Trellises

Per Stähl, Student Member, IEEE, John B. Anderson, Fellow, IEEE, and Rolf Johannesson, Fellow, IEEE

Abstract—The results of an extensive search for short and moderate-length polynomial convolutional encoders for time-invariant tail-biting representations of block codes at rates $R = 1/4, 1/3, 1/2$, and $2/3$ are reported. The tail-biting representations found are typically as good as the best known block codes.

Index Terms—Convolutional codes, error-correction coding, tail-biting encoders, trellis codes.

I. INTRODUCTION

There exist two main principles to terminate convolutional codes into block codes. Assume for simplicity that the generator matrix for

Fig. 1. Circular trellis of length $L = 6$ for a four-state convolutional encoder.

a rate $R = b/c$ convolutional code of memory $m$ is polynomial and realized in controller canonical form.

In the first method we start in the zero state and encode the $K$ information symbols followed by $m$ 6-tuples of zeros. Hence, we reach the zero state and the convolutional code has been terminated into a block code by the so-called zero-tail (ZT) method at the cost of a rate loss by a factor $K/(K + mb)$. If the trellis is short this rate loss might not be acceptable.

A termination method that does not suffer from any rate loss is tail-biting, which can be used to construct very powerful regular trellis representations of block codes. Assuming a trellis of $L$ sections (for simplicity we assume here that $m \leq L$), the tail-biting condition is the restriction that the convolutional encoder state $\sigma$ at time $t = 0$ is equal to the encoder state at time $L$, i.e., $\sigma_0 = \sigma_L$. A tail-biting trellis of length $L$ corresponds to a total of $K = bL$ information symbols, $c$ symbols per branch, block length $N = Lc = 2^b$ branches per trellis node; the number of codewords is

$$M = 2^K = 2^{bl},$$

and its rate is

$$R = K/N = b/c.$$  

Let $u_{[0,L)} = u_0 u_1 \ldots u_{L-1}$ denote the input (information) sequence, $v_{[0,L)} = v_0 w_1 \ldots v_{L-1}$ the output sequence (codeword), and

$$G(D) = G_0 + G_1 D + \cdots + G_m D^m$$

the generator matrix. The codewords of the tail-biting representation of the block code $C_{tb}$ that is obtained from the convolutional code $C$ encoded by the generator matrix $G(D)$ can be compactly written as

$$v_{[0,L)} = u_{[0,L)} G^{tb}_{L}$$

where

$$G^{tb}_{L} = \begin{pmatrix}
G_0 & G_1 & \cdots & G_m \\
G_0 & G_1 & \cdots & G_m \\
& \ddots & \ddots & \vdots \\
& & \ddots & \ddots \\
& & & \ddots \\
G_{m-1} & G_m & \cdots & G_1 \\
G_1 & G_2 & \cdots & G_m \\
G_0 & \cdots & \cdots & G_0
\end{pmatrix}$$