LDPC–CPM Systems for Aeronautical Telemetry

Erik Perrins

Department of Electrical Engineering & Computer Science
University of Kansas
Lawrence, KS 66049
E-mail: esp@ieee.org
February 28, 2025

Abstract | This report gives a detailed description of a channel capacity approaching forward error correction (FEC) system for use in aeronautical telemetry. The system is based on low density parity check (LDPC) codes that are designed for (matched to) the specific continuous phase modulation (CPM) schemes used in telemetry. The LDPC–CPM system is described in full detail herein. The step-by-step method by which the LDPC codes were designed and constructed is summarized and referenced. Additional details are given on the sparse representations of these quasi-cyclic LDPC codes. A check-node splitting/puncturing scheme is outlined, which can be used to make the LDPC–CPM decoder more robust against undetected errors. The LDPC decoder and CPM soft-input soft-output (SISO) decoder are described in detail, with additional comments on applicable parallelization techniques. And finally, numerical results are presented on the bit error rate/frame error rate performance of the various LDPC–CPM combinations and their average and maximum iterative behavior, with comparisons to the AR4JA codes currently adopted in telemetry and a design alternative (serially concatenated convolutional codes, or SCCCs) that was abandoned early in this study. The appendix provides additional background information on the log-domain processing that takes place in the decoders. The appendix also lists each LDPC generator and parity check matrix along with the random interleavers used in the system.

1 | INTRODUCTION

Forward error correction (FEC) codes are a technology that has become standard in nearly all digital communication settings, althought it was absent for the first few decades of aeronautical telemetry operations. The first FEC option was added to the IRIG-106 telemetry standard [1] about a decade ago. This option was based on the family of accumulate repeat-4 jagged accumulate (AR4JA) low density parity check (LDPC) codes. These codes were originally designed by researchers at the Jet Propulsion Laboratory (JPL) [2], [3] for use in deep-space communication. In terms of modulation type, the AR4JA codes are matched to the information theoretic characteristics of binary phase shift keying (BPSK).

In aeronautical telemetry, the modulations that are used almost exclusively are the family of three continuous phase modulation (CPM) waveforms specified in IRIG-106 [1]. These CPM waveforms have progressive degrees of spectrum efficiency relative to each other. The least efficient of these (spectrum wise) is original (legacy) CPM waveform commonly known as pulse code modulation/frequency modulation (PCM/FM), but for convenience herein we will adopt the name "ARTM0." This is in reference to the Advanced Range TeleMetry (ARTM) program of the early 2000s, which developed two "tiers" of spectrum efficiency relative to the ARTM0 (PCM/FM) baseline. The first of these is known as the telemetry group version of shaped offset quadrature phase shift keying (SOQPSK-TG), hereafter referred to as "ARTM1." The second and most spectrum-efficient is known as ARTM CPM (or multi-h CPM), which we call "ARTM2." ARTM1 and ARTM2 were adopted into IRIG-106 shortly after they were developed.

ARTM1 has the fortunate distinction of being a member of the most widely-used family of CPMs, known as "MSK-type" (MSK refers to minimum shift keying). Any CPM with a binary information alphabet and modulation index $h = 1/2$ belongs to this family. MSK-type CPMs have the unique property of allowing either a *recursive* or a *non-recursive* formulation [4], whereas CPM in general is recursive-only. In a non-recursive scenario, information is transmitted by phase *position* (a +1 or a −1, as in BPSK), as opposed to a recursive scenario where information is transmitted by phase *changes* (a phase shift of ±180°, as in differential BPSK). Thus, with ARTM1 it is possible for its information theoretic characteristics to be made similar to BPSK [5], which in turn allows it to be used with codes

DISTRIBUTION STATEMENT A. Approved for public release; Distribution is unlimited 412TW-PA-24182.

This work was supported by the Spectrum Relocation Fund (SRF) under the project "Forward Error Correction Codes for IRIG-106 CPM Waveforms." The author would like to thank Kip Temple, Bob Selbrede, and Program Manager Bobbie Wheaton.

designed for BPSK, such as the AR4JA codes. The pairing of the AR4JA codes with *non-recursive* ARTM1 (a system we call AR4JA–ARTM1-NR) was explored in [6], along with serially concatenated convolutional codes (SCCCs) paired with *recursive* ARTM1. Shortly thereafter, the AR4JA–ARTM1-NR option was adopted into IRIG-106 [1], making it the first standardized use of FEC in aeronautical telemetry, as noted above.

The AR4JA–ARTM1-NR system has proved its value on test ranges over the past decade. This has motivated the fundamental question contemplated by the present study, namely: Does a comparable option exist for ARTM0 and ARTM2? As implied above, the answer to this question cannot sidestep the recursive information theoretic properties that are inherent to CPM. This issue was tackled head-on in a companion paper [7], which fully documents the "design phase" of this study. The main contribution of [7] is the development of a step-by-step design process for matching a capacity-approaching LDPC code with the information-theoretic properties of a given CPM scheme. As design case studies, [7] focused on ARTM0, ARTM1, and ARTM2.

In this report, we build on [7] by documenting the inner workings of the LDPC–CPM system in fine detail. In Section 2 we describe the LDPC–CPM system model, including details on the transmitter, CPM waveforms, and random interleavers. In Section 3 we summarize the necessary aspects of LDPC codes, including the design and construction of the proposed LDPC–CPM codes, a sparse (compact) representation format for these codes, and a novel check-node splitting/puncturing technique that improves the performance of the LDPC–CPM system. In Section 4 we pivot toward the operation of the receiver. We give a complete description of a quasi-cyclic LDPC decoder in Section 5 and do likewise for the CPM soft-input soft-output (SISO) module in in Section 6; Sections 5 and 6 include comments on parallelization strategies for these decoders. The main portion of this report concludes in Section 7 where we provide numerical results on the performance of the proposed LDPC–CPM schemes. This includes a characterization of the average and maximum iterations required by the proposed system, with expanded results and real-time decoder architectures presented in another companion paper [8]. It also includes direct comparison of the proposed LDPC–CPM system for ARTM1 vs. the existing AR4JA–ARTM1-NR system, where we find that AR4JA–ARTM1-NR is superior by a mere 0.4 dB. The closeness of this advantage underscores the effectiveness of the design methodology in [7] and extends confidence to the results presented for ARTM0 and ARTM2. We also compare the proposed LDPC–CPM codes with a SCCC–CPM system, which received early consideration in this study. This comparison highlights the fact that the proposed LDPC–CPM codes do not exhibit "error floors," which is a highly-desirable property shared by the AR4JA codes. As such, the LDPC–CPM codes proposed herein represent a FEC solution for ARTM0 and ARTM2 that is complementary to the AR4JA–ARTM1-NR system in many ways.

This report also contains an extensive Appendix. The first part of the Appendix provides background mathematical development on log-based probabilities and log-based soft processing, which is helpful in understanding the LDPC and SISO decoders.

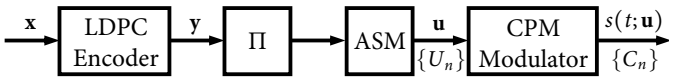


Figure 1 | LDPC–CPM Transmitter model.

The second half of the Appendix give a comprehensive (but sparse) listing of the random interleaving tables, parity check matrixes, and generator matrixes that are proposed for ARTM0 and ARTM2 (ARTM1 is included, but these codes are not recommended as replacements for the AR4JA codes).

2 | LDPC–CPM SYSTEM MODEL

2.1 | Transmitter Model

The transmitter model is shown in Figure 1. Without loss of generality, we will assume the transmission of a single code word. The information word (sequence) is denoted as $\mathbf{x} \triangleq \{x_i\}_{i=0}^{K-1}$, where K is the number of bits contained in the information word and each bit has a duration of T_b seconds. The LDPC encoder accepts $\mathbf{x}$ as its input and returns the code word (sequence) $\mathbf{y} \triangleq \{y_i\}_{i=0}^{N-1}$ as its output, where N is the number of bits contained in the code word and each coded bit has a duration of T_c seconds. The LDPC encoder has a rate $R \triangleq K/N$ and we have the relationship $T_c = RT_b$. We assume a *systematic* LDPC code, and thus the first K bits in $\mathbf{y}$ are the information bits, $y_i = x_i$, $0 \le i \le K-1$, and the last $(N-K)$ bits are the parity (redundant) bits generated by the encoder.

The code word $\mathbf{y}$ is fed to an interleaver (denoted by the symbol Π) that permutes the order of bits within the input word like so:

$$u_i = y_{\Pi(i)} \qquad 0 \le i \le N-1 \tag{1}$$

The interleaver is defined by the *interleaving table*, $\Pi(i)$, which is simply a look-up table (LUT) that contains a one-to-one mapping (permutation) between integers in the range $0, 1, \ldots, N-1$. The output of the interleaver is concatenated with a known data sequence called the *attached sync marker* (ASM), which is N_{ASM} bits in duration. Over time, code words and ASM sequences form an alternating pattern, which means there is an ASM sequence on either side of (i.e. before and after) the single code word that is assumed herein. Thus, the sequence $\mathbf{u}$ can be indexed to a certain extent on either side of the strict code word boundaries, i.e.

$$u_i = \begin{cases} \mathrm{ASM}_{i+N_{ASM}}, & -N_{ASM} \le i \le -1 \\ y_{\Pi(i)}, & 0 \le i \le N-1 \\ \mathrm{ASM}_{i-N}, & N \le i \le N+N_{ASM}-1 \end{cases} \tag{2}$$

where $\{\mathrm{ASM}_i\}_{i=0}^{N_{ASM}-1}$ is the ASM bit sequence. The sequence $\mathbf{u}$ is fed to the CPM modulator, which produces the transmitted signal $s(t;\mathbf{u})$.

2.2 | CPM Signal Model

The CPM modulator is shown in block diagram form in Figure 2. The bits in the CPM input sequence, $\mathbf{u}$, are grouped into n_0-tuples (the CPM schemes in this report use $n_0 = 1$ and $n_0 = 2$), and these n_0-tuples assume an M-ary alphabet, where $M = 2^{n_0}$. The n_0-tuples can be expressed as individual bits or as

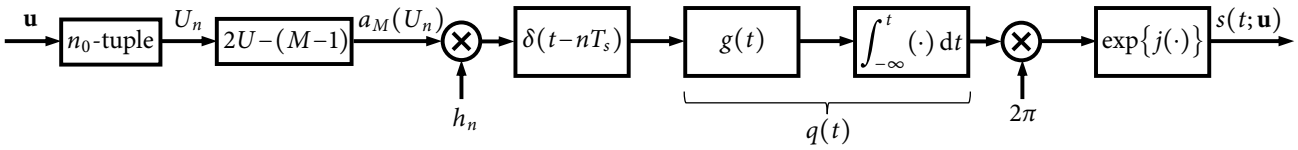
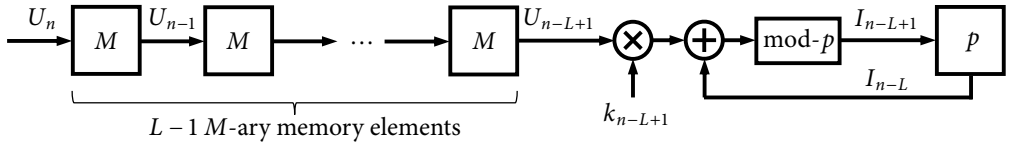


Figure 2 | CPM Modulator.



$$\mathbf{c}_n = [I_{n-L}, U_{n-L+1}, \ldots, U_{n-1}, U_n] \leftrightarrow C_n$$

Figure 3 | The finite state machine that comprises the CPE.

symbols drawn from an integer alphabet, $U_i \in \{0, 1, \ldots, M-1\}$ (when $n_0 = 1$ the bit and symbol formats are the same). The symbol format can be converted to the antipodal pulse amplitude modulation (PAM) alphabet via the antipodal function

$$a_M(U) = 2U-(M-1), \quad a_M(U) \in \{\pm 1, \pm 3, \ldots, \pm(M-1)\} \quad (3)$$

The complex envelope of the CPM signal can be expressed as

$$s(t;\mathbf{u}) = \exp\Big\{j2\pi \sum_i h_i a_M(U_i) q(t - iT_s)\Big\} \quad (4)$$

where $T_s = n_0 T_c = n_0 R T_b$ is the duration of each U_i (which ignores the additional overhead of the ASM). The set of N_h modulation indexes, $\{h_i\}_{i=0}^{N_h-1}$, are constrained to be rational numbers of the form $h_i = k_i/p$, where p is the least common denominator. The subscript for the modulation indexes is understood to be taken modulo-N_h. The phase response $q(t)$ is the time-integral of a frequency pulse $g(t)$ that is normalized to have area 1/2 and is constrained to be non-zero only during the interval $[0, LT_s]$. Due to the limited time support of $g(t)$ and its normalized area, $q(t)$ is understood to have an initial value of zero for $t < 0$ and a terminal value of 1/2 for $t > LT_s$. The block containing $\delta(t-nT_s)$ in Figure 2 provides the interface between the symbol domain (discrete time) and the signal domain (continuous time), where the current symbol time, n, is defined as as $nT_s \le t < (n+1)T_s$.

During the current symbol interval, the CPM signal can be factored into three terms by applying the initial and terminal values of $q(t)$ and also expanding (3)

$$\begin{aligned} s(t;\mathbf{u}) &= \exp\left\{j2\pi \sum_{i=n-L+1}^{n} h_i a_M(U_i) q(t - iT_s)\right\} \\ &\quad \times \exp\left\{j\frac{2\pi}{p} \sum_{i \le n-L} k_i U_i\right\} \exp\left\{-j\frac{\pi}{p} \sum_{i \le n-L} k_i (M-1)\right\} \\ &= \theta(t;\mathbf{U}_n) \exp\big\{j[\theta_{n-L} + v_{n-L}]\big\} \end{aligned} \quad (5)$$

and these three terms are functions of, respectively, the correlative state vector, $\mathbf{U}_n$, which has an alphabet of M^L unique values; the phase state, $\theta_{n-L+1} = \theta_{n-L} + 2\pi h_{n-L+1} U_{n-L+1}$, which has an alphabet of p unique values when taken modulo-(2π); and the data-independent phase tilt, $v_{n-L+1} = v_{n-L} - \pi h_{n-L+1}(M-1)$ [9], which has an alphabet of $2p$ unique values when taken modulo-(2π). It is convenient to express the phase state in terms of a phase state index, $\theta_{n-L} = 2\pi/p \cdot I_{n-L}$, where $I_{n-L+1} = [I_{n-L} + k_{n-L+1} U_{n-L+1}]$ mod-p. Thus, during the current symbol time, the CPM signal can be fully described by a continuous phase encoder (CPE) whose output "code vector" is defined as

$$\begin{aligned} \mathbf{c}_n &= [I_{n-L}, U_{n-L+1}, \ldots, U_{n-1}, U_n] && (6) \\ &= [I_{n-L}, \mathbf{U}_n] && (7) \\ &= [\mathbf{s}_n, U_n] && (8) \end{aligned}$$

which also encompasses the definition of the state vector, $\mathbf{s}_n$. The CPE state machine is shown in block diagram form in Figure 3.

The code vector is somewhat cumbersome to work with as an $(L+1)$-tuple. It is more convenient to devise a one-to-one mapping between the code *vector* and an integer, C_n, which we call the code *symbol*. The code vector can assume $N_E = pM^L$ possible values in its $L+1$ positions, and so $C_n \in \{0, 1, \ldots, N_E - 1\}$. If we view the CPM modulator as a CPE, the sequence $\{U_n\}$ is the input to this encoder and the sequence $\{C_n\}$ is its output. The $\{U_n\} \to \{C_n\}$ viewpoint is also used in Figure 1. The CPE is a finite state machine, which can be demodulated/decoded using trellis processing.

At the receiver, a matched filter (MF) is required for each possible value of $\mathbf{U}_n$, and in fact, the MF responses are defined by the time-reversed complex conjugate of $\theta(t;\mathbf{U}_n)$ in (5) [10]. Therefore, the numbers of matched filters, N_{MF}, trellis states, N_S, and trellis edges (branches), N_E, respectively, are given by

$$\begin{aligned} N_{MF} &= M^L && (9) \\ N_S &= pM^{L-1} && (10) \\ N_E &= pM^L = M \cdot N_S && (11) \end{aligned}$$

Each edge in the trellis is labeled with a unique value of $\mathbf{c}_n$ (or, more conveniently, C_n). When referring to edges in the trellis, we will use e as the index, where $0 \le e \le N_E - 1$. The various subcomponents of $\mathbf{c}_n$ can be collected from each edge and stored in various trellis LUTs, indexed by e, as follows: $s^S(e)$ is the starting state for a given edge, $s^E(e)$ is the ending state for a given edge, $\mathbf{U}(e)$ is the correlative state for a given edge, $U_i(e)$ is the i-th symbol in the correlative state for a given edge ($0 \le i \le L-1$ with $i = 0$ belonging to the "least significant symbol"), and $\theta(e)$ is the phase state for a given edge; we use vector notation for $\mathbf{U}(e)$ to distinguish it from $U_i(e)$, however, as with the other LUTs, it returns an integer that represents its variable. The LUT $C(e)$ can be viewed as the code symbol for a given edge, but such a LUT needs only to exist conceptually (and not in actual

storage) because the code symbol and the edge index have the same values, i.e. $C(e) = e$.

As indicated above, the parameters M, L, and p are what determine the state processing complexity of the demodulator/decoder. In [6], a variety of complexity reducing techniques were applied to one of the CPMs of interest in this report (but with general applicability to other CPMs as well). In this present study, the sole complexity-reducing technique we will draw from [6] is known as frequency pulse truncation (PT), which approximates the CPM signal at the receiver using a shortened value of L, i.e. $L_{\mathrm{R}x} < L$. When applied to a moderate degree, this technique does not drastically alter the minimum distance and near-minimum distance properties of the signal (its "distance spectrum") nor does it require implementation strategies such as decision feedback [6]. The same cannot be said for techniques that reduce the value of M or p [6]. For our coded application, the low operating SNRs exacerbate the penalties faced by techniques that drastically alter the distance spectrum of the signal or that require decision feedback [11].

With the PT approximation, the *transmitter* uses the CPM model as described above; however, the *demodulator/decoder* uses the smaller value of $L_{\mathrm{R}x}$ in place of L in the expressions given in (5)–(11). The new MFs are obtained from the $\theta(t;\mathbf{U}_n)$ that results, which has a $\mathbf{U}_n$ that is shortened by $(L - L_{\mathrm{R}x})$ elements. The receiver also uses a shifted definition of the current symbol time, $(n+(L-L_{\mathrm{R}x})/2)T_s \le t < (n+1+(L-L_{\mathrm{R}x})/2)T_s$, when forming the MFs and also (looking ahead in our development) when processing the received signal in (20).

2.3 | CPM Schemes

IRIG-106 [1] specifies three CPM schemes that are used in aeronautical telemetry. The most recent two were developed in the early 2000s under the Advanced Range TeleMetry (ARTM) program in order to provide progressive "tiers" of spectral efficiency relative to the original/legacy CPM scheme. We will present them in the same spectral efficiency ordering as in [1].

ARTM "Tier 0" (ARTM0). The least spectrally efficient CPM scheme we will use as a case study has the following parameters: $M = 2$, $h = 7/10$, 2RC. At the receiver, we will apply the PT approximation to model the signal as having $L_{\mathrm{R}x} = 1$. This results in a demodulator that requires $N_{\mathrm{MF}} = 2$ MFs and a trellis with $N_{\mathrm{S}} = 10$ states and $N_{\mathrm{E}} = 20$ edges. This CPM scheme is a close approximation to the predominant waveform used in aeronautical telemetry since the early 1960s: pulse coded modulation/frequency modulation, or PCM/FM. There is a slight mis-match between the 2RC pulse shape assumed here and the analog specification in [1]; however, such a mismatch is negligible relative to the mis-match introduced by the $L_{\mathrm{R}x} = 1$ approximation, which itself results in a performance degradation that is so small it is difficult to measure.

ARTM "Tier I" (ARTM1). The next CPM scheme we will use as a case study has the following parameters: $M = 2$, $h = 1/2$, 9TG. At the receiver, we will apply the PT approximation to model the signal as having $L_{\mathrm{R}x} = 2$. This results in a demodulator that requires $N_{\mathrm{MF}} = 4$ MFs and a trellis with $N_{\mathrm{S}} = 4$ states and $N_{\mathrm{E}} = 8$ edges. This waveform is known in IRIG-106 [1] as the telemetry group version of shaped-offset quadrature phase shift keying (SOQPSK-TG). In this report, we are using the SOQPSK signal model first presented by Othman et al. [12] and further discussed in [13]; the 9TG pulse shape is defined in either reference [12], [13]. The advantage of this relatively new model for SOQPSK is that strips away much of the description complexity of SOQPSK and results in simple and conventional CPM description. As with any CPM waveform, there is an inherent "differential encoding" aspect to this new signal model, which was proven in [13] to fit precisely into the differential encoding specification for SOQPSK-TG in IRIG-106 [1]. There already exists an LDPC specification in IRIG-106 [1] for SOQPSK-TG with differential encoding turned *off*, which essentially treats SOQPSK as an OQPSK-type modulation. As such, the purpose of SOQPSK-TG's inclusion in the current study is to provide an LDPC solution for SOQPSK-TG when it is treated as an ordinary CPM. The findings in [5] serve to foreshadow our findings in Section 7, which is that the existing LDPC schemes in IRIG-106 [1] with differential encoding turned *off* outperform the LDPC schemes proposed herein with differential encoding *on*.

ARTM "Tier II" (ARTM2). The most spectrally efficient CPM scheme we will use as a case study has the following parameters: $M = 4$, $\{h_0, h_1\} = \{4/16, 5/16\}$, 3RC. At the receiver, we will apply the PT approximation to model the signal as having $L_{\mathrm{R}x} = 2$. This results in a demodulator that requires $N_{\mathrm{MF}} = 16$ MFs and a trellis with $N_{\mathrm{S}} = 64$ states and $N_{\mathrm{E}} = 256$ edges. This waveform is also known as ARTM CPM in the literature (e.g. [6]) and in IRIG-106 [1]; it is an instance of multi-h CPM.

2.4 | Interleaver Operation

There are three desired code rates (R) and two desired information block sizes (K), as listed in Table 1. Thus, a total of six different interleaving tables (Π) must be specified, one for each possible code word length (N).

We design our interleaving tables as S-random interleavers [14], where the parameter S denotes the fact that elements that were adjacent prior to interleaving are at least S spaces apart in either direction after interleaving, but aside from this constraint the connections are randomly chosen. An approximate upper bound on this spacing is $S < \sqrt{N/2}$. We were able to achieve $S = 26$, $S = 28$, and $S = 32$, respectively, for $N = 1280$, $N = 1536$, and $N = 2048$. These are the values of N that correspond to the shorter block length $K = 1024$ ("1k"). We denote these 1k interleaving tables as $\pi(i)$, to differentiate them from the actual interleaving table that will be used in (1), which is $\Pi(i)$. This distinction is unnecessary for $K = 1024$ because $\Pi(i) = \pi(i)$. However, we will show how these 1k interleaving tables can be reused for the $K = 4096$ ("4k") case.

Before doing so, we illustrate the operation of $\pi(i)$ in Figure 5. The input to the interleaver is the length-N word (sequence) $\mathbf{y}$, which appears at the top of the figure. The interleaver output is the length-N word (sequence) $\mathbf{u}$, which appears at the bottom of the figure. As stated in (1), $\mathbf{u}$ is formed sequentially for $0 \le i < N$, by accessing $\mathbf{y}$ at the "random" locations indicated by $\pi(i)$. We have depicted only the first three time steps, $i = 0, 1, 2$.

Figure 4 shows the shorter 1k interleaving table $\pi(i)$ being used to operate a longer 4k interleaver. One advantage of this arrangement is that only the shorter 1k interleaving tables need

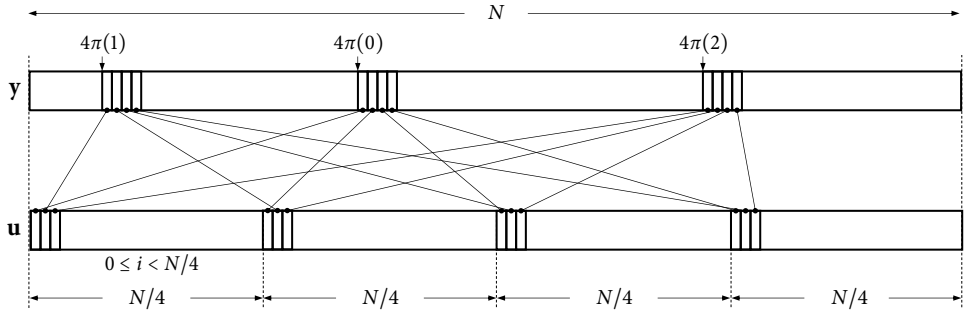


Figure 4 | Operation of a "4k" interleaver, with input **y** and output **u**.

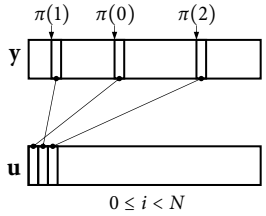


Figure 5 | Operation of a "1k" interleaver, with input **y** and output **u**.

to be tabulated in a document such as this! An additional advantage of this arrangement is that it allows the longer 4k interleaver to be parallelized by a factor of 4. In the lower-left corner of Figure 4, the first quarter-section of **u** is formed sequentially for $0 \le i < N/4$, while the three remaining quarter-sections of **u** are simultaneously formed in their respective sequential orders. Along the top of Figure 4, $\pi(i)$ is used to "randomly" access *4-tuples* of **y**, the four elements of which are "broadcast" to the quarter-sections of **u**. We have depicted only the first three time steps, $i = 0, 1, 2$, which fill twelve elements of the output **u**.

An alternative form of the arrangement in Figure 4 is to fill a $4 \times N/4$ array *column-by-column* with the input **y**. Once filled, the *column order* of this array is permuted by $\pi(i)$. The output **u** is then formed by emptying the array *row-by-row*. The load-by-column, empty-by-row arrangement is similar to a so-called "block interleaver."

Figure 4 takes a "hardware" point of view, and the actual 4k interleaving table $\Pi(i)$ is never referenced. If such a table is desired, a 4k $\Pi(i)$ can be generated from a 1k $\pi(\cdot)$ by

$$\Pi(i) = \begin{cases} 4\pi(i), & 0 \le i < N/4 \\ 4\pi(i - N/4) + 1, & N/4 \le i < N/2 \\ 4\pi(i - N/2) + 2, & N/2 \le i < 3N/4 \\ 4\pi(i - 3N/4) + 3, & 3N/4 \le i < N \end{cases} \tag{12}$$

This interleaving table, $\Pi(i)$, can be used directly in (1) and it follows the "ordinary" interleaver behavior depicted in Figure 5. We note that the S-random property of $\pi(i)$ is not preserved in $\Pi(i)$ in (12). However, in extensive simulations of (12) vs. true length-N interleavers generated specifically for the $K = 4096$ cases, we observed no performance degradation when (12) was used. As such, we use 1k interleavers exclusively going forward.

The contents of the 1k interleaving tables, $\pi(i)$, for $R = 4/5$, $R = 2/3$, and $R = 1/2$ are listed in the Appendix. These interleaving tables have lengths of $N = 1280$, $N = 1536$, and $N = 2048$, respectively.

3 | LDPC CODES

3.1 | Design and Construction

The process undertaken to design and construct the LDPC codes is thoroughly documented in our companion paper [7]. A set of six codes were designed for (matched to) each of the three CPM schemes (ARTM0, ARTM1, and ARTM2), i.e. 18 codes were designed in total. The six codes are divided into the different rates (R) and information block sizes (K) listed in Table 1.

The code construction process begins with the *protomatrix* **B**, which is relatively small and has M_B rows and N_B columns. The columns of **B** correspond to "variable nodes" and we index the columns with $0 \le n \le N_B - 1$. The rows of **B** correspond to "check nodes" and we index the rows with $0 \le m \le M_B - 1$. The design process allows the elements of **B** to be drawn from a 4-ary alphabet, i.e. $b_{m,n} \in \{0, 1, 2, 3\}$. The sum, or "weight," of the n-th column is referred to as the variable node degree, d_n; likewise, d_m is the degree of the m-th row, or check node degree.[1] The aim of the design process is simple: assign values to the elements of **B** in such a way that the "SNR decoding threshold" is minimized given the information theoretic characteristics of each CPM scheme [7]. This "matches" **B** to the given CPM scheme and minimizes the anticipated SNR at which the eventual LDPC code will yield good performance. As discussed in [7], the **B** matrixes that emerged from this design process have SNR thresholds within 1 dB of the theoretical limits predicted by channel capacity, thus it is fair to say that these are *capacity-approaching* codes. Although **B** is relatively small and dense, it serves as a template for the *distribution* of variable and check node degrees that yields the best performance, and the way in which these nodes are interconnected.

Throughout this section, we will use the $R = 1/2$, $K = 4096$ LDPC code that was designed in [7] for ARTM2 as our running example, which has the protomatrix

$$\mathbf{B} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 2 & 0 & 1 & 0 \\ 3 & 3 & 3 & 2 & 1 & 2 & 1 & 1 \end{pmatrix} \tag{13}$$

[1] For d_n and d_m, their subscripts (n and m, respectively) serve a dual purpose: (1) they help us *distinguish* which d we are referring to, and (2) they serve as an *index* from one variable (or check) node to the next. We will always use n when indexing d_n, and m when indexing d_m, and thus it will always possible to distinguish which type of d (variable or check node) we are referring to.

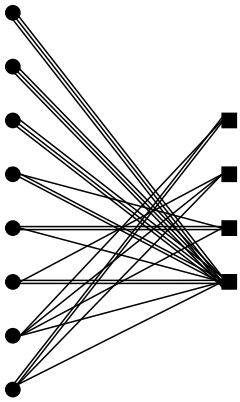


Figure 6 | Tanner graph of the example protomatrix **B** in (13).

Figure 6 shows the Tanner graph depiction of this protomatrix. The $N_B = 8$ variable nodes are drawn as circles on the left, and the $M_B = 4$ check nodes are drawn as squares on the right. Every non-zero element $b_{m,n}$ means that an edge (or edges) must be drawn to connect the n-th variable node with the m-th check node. There are a total of $N_E = 26$ edges[2] in Figure 6, and this number corresponds to the sum of all row (or column) weights of this **B**. There are many *parallel edges* in this Tanner graph due to the entries of $b_{m,n} = 2$ and $b_{m,n} = 3$ in this **B**.

Once **B** has been designed, the next steps in the construction process produce progressively larger and less-dense matrixes that preserve the degree distribution and interconnections of **B**, a process known as *lifting* in the literature. These larger matrixes are *quasi-cyclic*, meaning that they are composed of square sub-matrixes known as *circulants*, where each row is a right-hand circular (barrel) shift of the row above. As such, an entire circulant can be specified by providing only the first row. The "weight" of a circulant is the sum of the elements in the first row. Thus, if element $b_{m,n}$ is replaced by a circulant whose weight is equal to $b_{m,n}$, and likewise for all elements in **B**, then the degree distribution of **B** is preserved in the larger matrix that results.

The best results were achieved with a two-step lifting process [2], [7], [15]–[17], where a smaller "pre lift" was followed by a much larger final lift. This was the same approach taken with the AR4JA codes [2], although a pre-lift by 4 was used for the AR4JA codes and we used a pre-lift by 2 [7]. In the pre-lift by 2, each element of **B** is replaced by the following 2×2 expansions depending on the value of $b_{m,n}$:

$$0 \to \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad 1 \to \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad 2 \to \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad 3 \to \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Parallel edges may still be present after the pre-lift because the 2×2 expansion that replaces $b_{m,n} = 3$ still contains values of 2. However, any parallel edges that remain will be completely eliminated by the final lift.

In order to yield a final LDPC code that is systematic, the parity check matrix must be of the form $\mathbf{H} = [\,\mathbf{Q} \mid \mathbf{P}\,]$, where **P** is a square matrix that is full rank (invertible). In terms of the intermediate (pre-lifted) matrix **D**, whose dimensions are $M_D \times N_D$, a systematic code is achieved if the last M_D columns are full-rank (using modulo-2 arithmetic). In other words, the columns of **D** must be arranged such that a $M_D \times M_D$ square sub-matrix partition on the far-right is invertible. This rearranging of columns does not affect the overall performance of the code. In fact, the interleaver and deinterleaver already introduce a random permutation between the SISO module and the LDPC decoder. Thus, this step can be viewed as an aspect of the overall interleaving function, with the side benefit of ensuring that the code is systematic. The procedure followed in the design process is simple: (1) randomly permute the order of the columns of **D**; (2) proceed from the far right and collect columns one-by-one if they add to the rank (modulo-2) of the collection, and skip those columns that do not, and stop when a rank of M_D is achieved; and (3) the final form of **D** is with the columns in the rank M_D collection placed on the far-right end of **D** with the remaining columns to the left. The reordering of columns erases the quasi-cyclic structure of **D**, which is not essential.

In our running example, the pre-lift by 2 on **B** in (13), followed by the column permutation procedure that was just described, results in the intermediate protomatrix

$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 2 & 2 & 1 & 0 & 1 & 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 0 & 2 & 1 & 2 & 1 & 1 & 1 & 1 & 2 & 1 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{14}$$

A Tanner graph for **D** is not provided, but the pre-lift by 2 essentially doubles everything, i.e. there are $N_D = 16$ variable nodes and $M_D = 8$ check nodes, connected by $N_E = 52$ edges. Although **D** has lower density than **B**, it remains quite dense.

In the second and final lifting stage of the design process, **D** is lifted by a factor of M_L by assigning random *phases* to $M_L \times M_L$ circulants using the ACE algorithm [18]. Because of the pre-lift, the entries of **D** are drawn only from the alphabet $\{0, 1, 2\}$, and so this second lifting stage results in weight-2, weight-1, or weight-0 (all-zeros) circulants. A weight-1 circulant can be specified simply by its *phase*, $\phi \in \{0, 1, \ldots, M_L - 1\}$, which is the location of the non-zero element (i.e. the 1) in the first row. We simplify the discussion by considering a weight-2 circulant to be the superposition of two weight-1 circulants with *different* phases, where each phase is assigned one at a time via the ACE algorithm. Thus, the final *low-density* parity check matrix that emerges, **H**, contains entries drawn only from the binary alphabet $\{0, 1\}$ and is completely free of parallel edges.

Figure 7 shows the low-density parity check matrix **H** that was constructed from the intermediate protomatrix **D** in (14) using $M_L = 512$. Each 512×512 weight-1 circulant is displayed as a diagonal "stripe" that wraps around in a modulo-512 fashion (the different color stripes will be explained later). A weight-2 circulant has two such stripes (with different phases) superimposed in the same 512×512 cell.

The dimensions of **H** are $M_H = M_L M_D$ rows and $N = M_L N_D$ columns. The companion to this is the generator matrix, **G**, with dimensions $K = N - M_H$ and N (i.e., both **G** and **H** have the same

[2] We permit ourselves to use overlapping notation for the internal descriptions of the trellis diagram and the LDPC code (for example, the use of N_E in both settings), because there is a clear differentiation due to context.

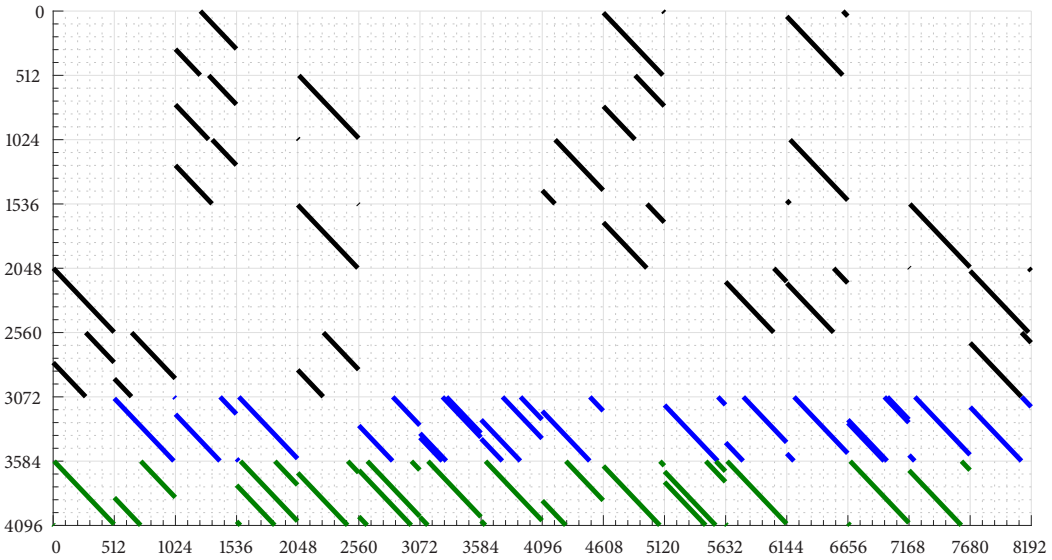


Figure 7 | Low-density parity check matrix for the ARTM2, $R = 1/2$, $K = 4096$ code. It is constructed from $\mathbf{D}$ in (14) using $M_{\mathrm{L}} = 512$.

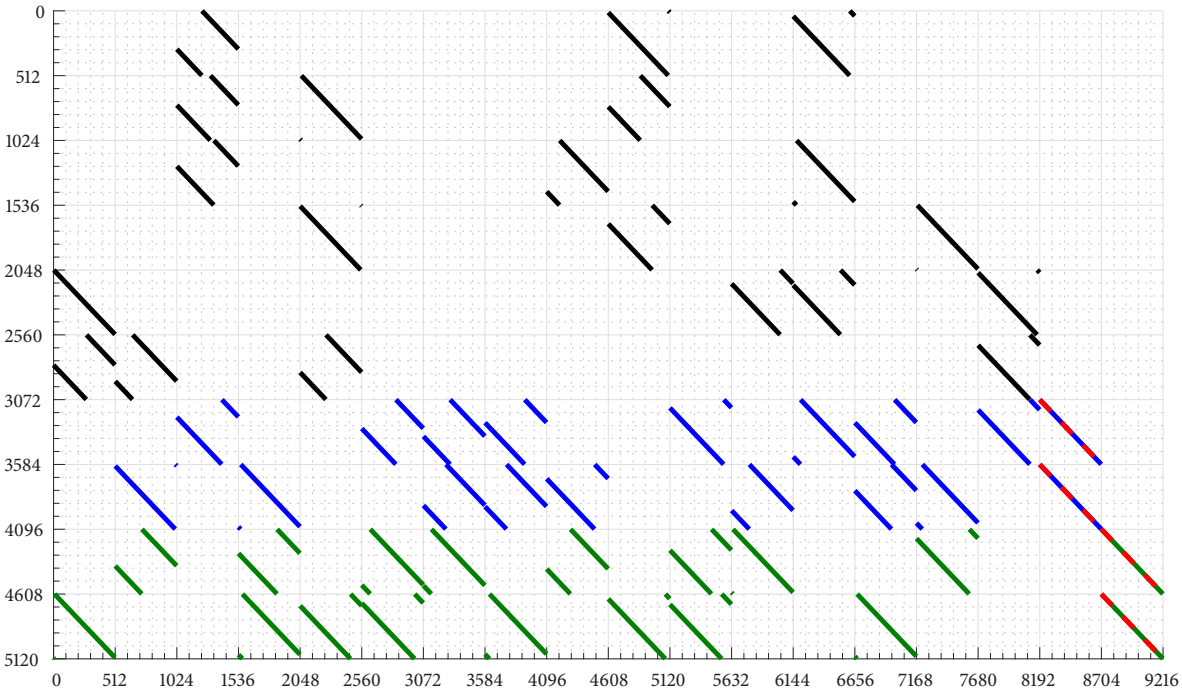


Figure 8 | Low-density parity check matrix *with puncturing* for the ARTM2, $R = 1/2$, $K = 4096$ code.

Table 1 | Dimensions of the Final LDPC Codes and Coding Gains Realized [in dB].

				Native				Punctured				Coding Gains			
K	R	$d^*_{\min}$	M_{L}	M_{D}	N_{D}	N	It_{loc}	M_{D}	N_{D}	N_{P}	It^*_{loc}	Δ_0	Δ_1	$\Delta_{1\mathrm{A}}$	Δ_2
1024	4/5	6	32	8	40	1280	1	10	42	1344	1	7.6	7.9	8.2	7.1
1024	2/3	8	64	8	24	1536	1	10	26	1664	2	8.6	9.1	9.5	8.6
1024	1/2	12	128	8	16	2048	1	10	18	2304	4	9.1	9.9	10.5	9.3
4096	4/5	12	128	8	40	5120	1	10	42	5376	4	8.7	9.1	9.5	8.1
4096	2/3	12	256	8	24	6144	1	10	26	6656	4	9.6	10.1	10.5	9.5
4096	1/2	12	512	8	16	8192	1	10	18	9216	4	9.9	10.9	11.3	10.2

number of columns, N). We point out that K as was just defined is the same as the number of bits contained in the information word. To obtain the generator matrix, we first partition $\mathbf{H}$ as [19]

$$\mathbf{H} = [\ \mathbf{Q}\ |\ \mathbf{P}\ ] \tag{15}$$

where $\mathbf{P}$ is an $M_{\mathrm{H}} \times M_{\mathrm{H}}$ square matrix and $\mathbf{Q}$ has dimensions $M_{\mathrm{H}} \times K$. We next compute

$$\mathbf{W} = \left(\mathbf{P}^{-1}\mathbf{Q}\right)^{T} \tag{16}$$

using modulo-2 arithmetic, where $(\cdot)^T$ denotes the matrix transpose and $(\cdot)^{-1}$ the matrix inverse; $\mathbf{P}$ is invertible because of our intermediate steps with $\mathbf{D}$. We then construct $\mathbf{G}$ as

$$\mathbf{G} = [\ \mathbf{I}_K\ |\ \mathbf{W}\ ] \tag{17}$$

where $\mathbf{I}_K$ is the $K \times K$ identity matrix and $\mathbf{W}$ is a dense matrix of high-weight size-M_{L} circulants and has dimensions $K \times M_{\mathrm{H}}$.

3.2 | Sparse Representations of G and H

A sparse representation of $\mathbf{G}$ focuses on a sparse representation of $\mathbf{W}$, which is a $(K/M_{\mathrm{L}}) \times M_{\mathrm{D}}$ "grid" of dense circulants. The top row of each circulant (M_{L} bits in total) is all that must be stored in advance or written in a document such as this. These can be expressed as length-$(M_{\mathrm{L}}/4)$ strings of hexadecimal characters. As listed in Table 1, all codes discussed herein have $M_{\mathrm{D}} = 8$, and thus each group of 8 such character strings represents one row of circulants of $\mathbf{W}$. Figure 3-2 of [19] shows how such a sparse representationof $\mathbf{G}$ can be used to realize an efficient encoder in hardware.

Although $\mathbf{H}$ in Figure 7 is enormous—4096×8192—the structural similarities it shares with $\mathbf{D}$ in (14) are readily apparent. We now describe a sparse representation that can be used to communicate $\mathbf{H}$ in a document such as this or in a computer program. We then show how this sparse representation can be converted into a form that is most useful to the quasi-cyclic LDPC decoder we describe in Section 5.

The essential elements needed to fully describe $\mathbf{H}$ are the locations of the non-zero elements (i.e. the *ones*) along the rows at the top of each block of circulants. These bit locations are integers drawn from the set $\{0, 1, \ldots, N-1\}$. Such a location can be decomposed into a column index belonging to $\mathbf{D}$, $n = \lfloor \text{location}/M_{\mathrm{L}} \rfloor$, and a circulant phase $\phi = \text{mod}(\text{location}, M_{\mathrm{L}})$, where $\lfloor\cdot\rfloor$ and $\text{mod}(\cdot, M_{\mathrm{L}})$ are the floor and modulo-M_{L} operators, respectively.

In $\mathbf{H}$, the row indexes of interest are multiples of M_{L}, i.e. mM_{L}, where $0 \le m \le M_{\mathrm{D}} - 1$. The m-th such row is completely specified by the $(d_m + 1)$-tuple $\{d_m, \text{location}, \text{location}, \ldots, \text{location}\}$, which is the weight d_m of that row in $\mathbf{D}$ followed by that many non-zero element locations in $\mathbf{H}$. Therefore, the entire $\mathbf{H}$ matrix can be specified by a long list of numbers beginning with M_{L}, and followed by the $(d_m + 1)$-tuple for each row. Concatenated together, this results in a comma-delineated list, $\mathcal{L}$, with individual elements $\mathcal{L}(l)$, for $0 \le l \le \mathcal{L}_{\text{len}} - 1$. The length of the list is $\mathcal{L}_{\text{len}} = 1 + M_{\mathrm{D}} + N_{\mathrm{E}}$, where N_{E} belongs to $\mathbf{D}$.

For $\mathbf{H}$ in Figure 7, whose $\mathbf{D}$ in (14) has $M_{\mathrm{D}} = 8$ and $N_{\mathrm{E}} = 52$, the length-61 comma-delineated list $\mathcal{L}$ is:

```
512,  // M_L
3,1233,5107,6614,  // row = 0
3,1303,2058,4874,  // row = 1
3,1332,4204,6172,  // row = 2
3,2553,4973,7175,  // row = 3
4,2,6036,6537,8171,  // row = 4
4,273,657,2263,8110,  // row = 5
16,1012,1398,1555,2844,3260,3294,3763,3914,4495,5567,5780,
   6205,6961,6986,7217,8111,  // row = 6
16,8,734,1569,1857,2467,2631,3000,3139,3621,4294,5082,5466,
   5550,5644,6675,7607,  // row = 7
```
(18)

where we have added "comments" and line breaks for ease of reading, but these formatting elements are not necessary (the colored text will be explained shortly).

Table 1 lists 2 information block sizes ($K \in \{1024, 4096\}$) and 3 code rates ($R \in \{1/2, 2/3, 4/5\}$), plus there are 3 CPM schemes of interest in this study (ARTM0, ARTM1, and ARTM2). This amounts to 18 distinct LDPC–CPM pairings. The Appendix gives a comprehensive listing of the sparse representations of the 18 parity check matrixes, $\mathbf{H}$, and the 18 generator matrixes, $\mathbf{G}$, that were designed in [7] for these LDPC–CPM pairings.

3.3 | Sparse Representation Needed by the LDPC Decoder

The quasi-cyclic LDPC decoder we describe in Section 5 needs the following information:

- The variable node degrees for the intermediate protomatrix $\mathbf{D}$, d_n, $0 \le n \le N_{\mathrm{D}} - 1$.
- The set of *edge indexes* that are associated with the n-th variable node of $\mathbf{D}$, $\mathcal{E}_n = \{e_0, e_1, \ldots, e_{d_n - 1}\}$, $0 \le n \le N_{\mathrm{D}} - 1$. The cardinality of the n-th set is $d_n = |\mathcal{E}_n|$.
- The check node degrees for the intermediate protomatrix $\mathbf{D}$, d_m, $0 \le m \le M_{\mathrm{D}} - 1$.
- The set of *edge indexes* that are associated with the m-th check node[3] of $\mathbf{D}$, $\mathcal{E}_m = \{e_0, e_1, \ldots, e_{d_m - 1}\}$, $0 \le m \le M_{\mathrm{D}} - 1$. The cardinality of the m-th set is $d_m = |\mathcal{E}_m|$.
- The set of *circulant phases* that are associated with edges in the m-th check node, $\Phi_m = \{\phi_0, \phi_1, \ldots, \phi_{d_m - 1}\}$, $0 \le m \le M_{\mathrm{D}} - 1$. The cardinality of the m-th set is $d_m = |\Phi_m|$. These phases belong to the circulants of $\mathbf{H}$, but as with the other variables, their groupings are driven by $\mathbf{D}$. The LDPC decoder is formulated such that phases are needed only for the *check node* update. For the variable node update, the decoder views all edges as having a phase of *zero*.

Although parallel edges may be present in $\mathbf{D}$, the LDPC decoder as it is formulated in Section 5 is unaware of such a distinction. It merely sees sets of edges associated with each variable and check node.

The simple comma-delineated list, $\mathcal{L}$, can be parsed to generate all of this information using Algorithm 1. Because $\mathcal{L}$ is organized row-by-row, the row-based information (d_m, $\mathcal{E}_m$, and Φ_m) can be fully determined one m at a time, and M_{D} is simply the final tally of the index m (plus one). The column-based information (d_n and $\mathcal{E}_n$), on the other hand, evolves one edge at a

[3] As with d_n and d_m, the subscripts for $\mathcal{E}_n$ and $\mathcal{E}_m$ will be used both to differentiate between the two and as an index.

Algorithm 1 List Parsing Routine for Sparse **H**.

Input: A comma-delineated list, $\mathcal{L}$, with individual elements $\mathcal{L}(l)$, for $0 \le l \le \mathcal{L}_{\text{len}} - 1$.
Outputs: M_{L}; d_n and $\mathcal{E}_n$ for $0 \le n \le N_{\text{D}} - 1$; d_m, $\mathcal{E}_m$, and Φ_m for $0 \le m \le M_{\text{D}} - 1$; N_{E}.
Initializations: $M_{\text{L}} = \mathcal{L}(0)$;$N_{\text{D}} = \lfloor \max_l\{\mathcal{L}(l)\}/M_{\text{L}} \rfloor + 1$; $d_n = 0$ and $\mathcal{E}_n = \{\varnothing\}$ for $0 \le n \le N_{\text{D}} - 1$; $e = -1$, $m = -1$, and $l_{\text{next}} = 1$.
while $l_{\text{next}} < \mathcal{L}_{\text{len}}$ **do**
    $m = m + 1$;
    $d_m = \mathcal{L}(l_{\text{next}})$;
    $l_{\text{next}} = l_{\text{next}} + 1$;
    for $i = 0, 1, \ldots, d_m - 1$ **do**
        $n = \lfloor \mathcal{L}(l_{\text{next}})/M_{\text{L}} \rfloor$;
        $\phi = \text{mod}(\mathcal{L}(l_{\text{next}}, M_{\text{L}})$;
        $l_{\text{next}} = l_{\text{next}} + 1$;
        $e = e + 1$;
        Add e to $\mathcal{E}_n$ and $\mathcal{E}_m$;
        Add ϕ to Φ_m;
        $d_n = d_n + 1$;
    end for
end while
$M_{\text{D}} = m + 1$;
$N_{\text{E}} = e + 1$;

Table 3 | Column-based information, d_n and $\mathcal{E}_n$, for **H** in Figure 7.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d_n	3	3	4	3	4	3	3	3	3	4	3	3	4	3	3	3
$\mathcal{E}_n$	12	17	0	22	4	23	24	26	7	1	29	13	2	32	11	15
	16	20	3	38	9	41	25	27	28	5	47	30	8	33	34	19
	36	37	6	39	18	42	43	44	45	10	48	49	14	50	51	35
			21		40					46			31			

time, anywhere in the range $0 \le n \le N_{\text{D}} - 1$, and thus it is helpful to know N_{D} at the very beginning, which can be determined as $N_{\text{D}} = \lfloor \max_l\{\mathcal{L}(l)\}/M_{\text{L}} \rfloor + 1$.

The output of Algorithm 1 for the comma-delineated list in (18) is shown in Tables 2–4. This is the information needed to describe the **H** in Figure 7 to the LDPC decoder. While unimportant, a careful inspection of the sets $\mathcal{E}_n$ in Table 3 reveals the harmless parallel edges that are present, which appear as consecutive edge indexes in a given set (e.g. see $\mathcal{E}_n$ for $n = 5$ and $n = 6$).

Table 4 | Row-based information, d_m and $\mathcal{E}_m$, for **H** in Figure 7.

m	d_m	$\mathcal{E}_m$
0	3	0 1 2
1	3	3 4 5
2	3	6 7 8
3	3	9 10 11
4	4	12 13 14 15
5	4	16 17 18 19
6	16	20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
7	16	36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51

3.4 | Puncturing Via Check-Node Splitting

A check node can be *split* into an equivalent graph with two check nodes that are connected by a non-transmitted (punctured) variable node. Check node splitting is one of the techniques that was used to construct the AR4JA codes in [2]. Although the new graph (code) is equivalent to the original, the iterative behavior of its decoder is slightly different. In [7], this technique was shown to improve the robustness of the LDPC–CPM decoder against *undetected errors*. These are instances where the decoder converges on a valid code word, albeit a different one than was transmitted. When this occurs, the parity check *passes* (because a valid code word was found) and thus the decoder is *unaware* that bit errors are present.

The splitting procedure itself is straightforward. In terms of the original protomatrix **B**, we split the check node with the highest degree, i.e. the bottom row of (13) or the bottom check node in Figure 6. The edges connected to this check node are divided in alternating fashion into two groups and assigned to two nodes, i.e a new check node is created (a new row in **B** is created). The two check nodes are then connected to each other via one edge each to a new non-transmitted (punctured) variable node (a new column in **B** is created). Figure 9 shows the new Tanner graph that emerges from the original Tanner graph in Figure 6 after the check node splitting procedure.

These arguments can be translated to the intermediate protomatrix **D**, i.e. the bottom two rows of (14) are split and two new columns are created. Likewise, the two highest-degree rows of circulants in **H** are split and two new columns of circulants are created. For the example **H** in Figure 7, the bottom two rows of circulants are the ones involved in the check node splitting and are displayed with blue and green stripes. Figure 8 shows how the blue stripes are divided in alternating fashion into two rows,

Table 2 | Row-based phase information, Φ_m, for **H** in Figure 7.

m	d_m	Φ_m
0	3	209 499 470
1	3	279 10 266
2	3	308 108 28
3	3	505 365 7
4	4	2 404 393 491
5	4	273 145 215 430
6	16	500 374 19 284 188 222 179 330 399 447 148 61 305 330 49 431
7	16	8 222 33 321 419 71 440 67 37 198 474 346 430 12 19 439

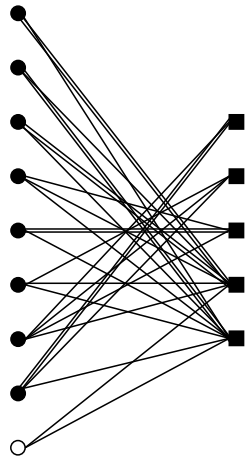


FIGURE 9 | Tanner graph of the example protomatrix **B** in (13) after the check node splitting procedure (compare with the original Tanner graph in Figure 6). The non-transmitted (punctured) variable node is drawn with an open circle.

and likewise for the green stripes. At the far right of Figure 8, two new circulants (with zero phase) are added to **H** to connect the blue rows (displayed in blue and red) and likewise for the green stripes (displayed in green and red).

And finally, these arguments can be translated into some simple editing steps on the comma-delineated list $\mathcal{L}$. To split a check node, the bit locations for that node are divided in alternating fashion into two row-lists, a new bit location is added to each row-list (with zero phase), and the row-degree is updated accordingly. For the original comma-delineated list in (18), the edited list for the equivalent split-and-punctured code is

$$
\begin{aligned}
&512, \quad // \; M_{\mathrm{L}} \\
&3, 1233, 5107, 6614, \quad // \text{ row} = 0 \\
&3, 1303, 2058, 4874, \quad // \text{ row} = 1 \\
&3, 1332, 4204, 6172, \quad // \text{ row} = 2 \\
&3, 2553, 4973, 7175, \quad // \text{ row} = 3 \\
&4, 2, 6036, 6537, 8171, \quad // \text{ row} = 4 \\
&4, 273, 657, 2263, 8110, \quad // \text{ row} = 5 \\
&9, 1398, 2844, 3294, 3914, 5567, 6205, 6986, 8111, 8192, \quad // \text{ row} = 6 \\
&9, 1012, 1555, 3260, 3763, 4495, 5780, 6961, 7217, 8192, \quad // \text{ row} = 7 \\
&9, 734, 1857, 2631, 3139, 4294, 5466, 5644, 7607, 8704, \quad // \text{ row} = 8 \\
&9, 8, 1569, 2467, 3000, 3621, 5082, 5550, 6675, 8704, \quad // \text{ row} = 9
\end{aligned}
\tag{19}
$$

where two row-lists are shown each in blue and green, the new zero-phase edges are shown in red, and the updated row-degrees are also displayed in red.

The original ("native") code has an **H** with dimensions $M_{\mathrm{H}} = M_{\mathrm{L}}M_{\mathrm{D}}$ and $N = M_{\mathrm{L}}N_{\mathrm{D}}$. The **H** for the punctured code has dimensions $M_{\mathrm{H}} = M_{\mathrm{L}}(M_{\mathrm{D}}+2)$ and $N_{\mathrm{P}} = M_{\mathrm{L}}(N_{\mathrm{D}}+2)$, using the dimensions belonging to the original **D**. We note the difference between the punctured code length, N_{P}, and the transmitted code length N. A new generator matrix, **G**, could be created with dimensions $K = N_{\mathrm{P}} - M_{\mathrm{H}}$ and N_{P}. However, the right-most $2M_{\mathrm{L}}$ columns of this larger **G** matrix could simply be discarded because they correspond to the punctured information. Thus, the original **G** is used regardless of puncturing. For completeness, Table 1 lists the dimensions of the native and punctured codes for all code rates and information block lengths.

4 | LDPC–CPM RECEIVER MODEL

4.1| Block Diagram

We now switch our focus to developing the receiver that is the counterpart to the transmitter in Figure 1. The receiver model is shown in Figure 10. The various modules exchange vectors (sequences) of *soft information* in the form of *L-values* whenever the underlying variable is binary (i.e. **u** and **y**, but not $\{C_n\}$). An L-value is defined as the log of the ratio of the two elements in a binary probability mass function (PMF). It is the most convenient soft format for binary sequences because only a single value (a scalar) is required for each time step. Because $\{C_n\}$ is drawn from an N_{E}-ary alphabet, a full N_{E}-ary PMF (a "sub-vector") is required for each time step n [i.e. the soft information for $\{C_n\}$ is a sequence (vector) of PMFs (sub-vectors)]. As explained further in the Appendix, all soft information is formulated in the log domain due to its greater numerical stability. The input vectors are denoted with "I" and contain *a priori* information. The output vectors are denoted with "O" and contain *a posteriori* information. In cases where an output is later connected to the input of another module, we compute the *extrinsic* version of such outputs, which will be defined later.

The received signal is modeled as

$$r(t) = \sqrt{\frac{E_s}{T_s}}s(t;\mathbf{u}) + w(t) \tag{20}$$

where E_s is the energy per symbol and $w(t)$ is complex-valued additive white Gaussian noise (AWGN) with zero mean and power spectral density N_0. The energy per symbol is subject to the identity $E_s = n_0E_c = n_0RE_b$, where E_c is the energy per coded bit (y_i) and E_b is the energy per information bit (x_i) (as before, these relationships ignore the additional overhead of the ASM).

The receiver processing for CPM can be subdivided into a traditional CPM *demodulator* that is positioned *outside* the iterative decoding loop, and a CPM *decoder* that is located *inside* the iterative decoding loop. Both of these are responsible for generating a soft output for **u** and thus they are labeled as soft-input soft-output (SISO) modules.

The traditional CPM demodulation process is well understood and entails familiar tasks such as matched filtering and synchronization. We assume that symbol, carrier, and frame/ASM synchronization [20] are perfectly known, which are reasonable assumptions given the similarities between the transmission model in Figures 1 and 10 and the existing AR4JA–ARTM1-NR system in IRIG-106 [1], which has been in operation for over a decade. It is important to emphasize that this more-intensive demodulation process takes place only *once* and is positioned *outside* the iterative loop. The MF bank within the CPM demodulator produces a sample, $z_n(\mathbf{U}(e))$, for each symbol time n and each correlative state vector $\mathbf{U}(e)$. The phase state and phase tilt are applied next, yielding

$$p_n(C(e);\mathrm{I}) = \frac{\sqrt{E_s}L_c}{2}\mathrm{Re}\left\{\exp\left\{-j[\theta(e) + v_{n-L_{\mathrm{Rx}}}]\right\}z_n(\mathbf{U}(e))\right\} \tag{21}$$

which is a set (sub-vector) of N_{E} values indexed by $C(e)$ for each time step, n. The complex conjugation of the MF responses, the

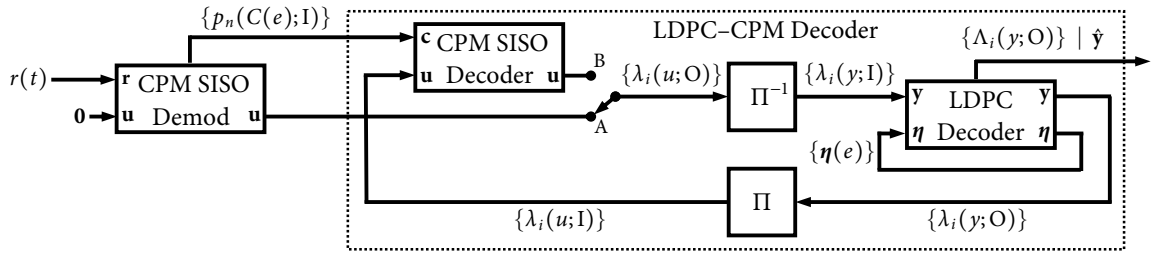


Figure 10 | LDPC–CPM Receiver model.

phase state, and the phase tilt essentially "un-do" the phase modulation of the single symbol model in (5). The precise scaling in (21), including the factor $L_c = 4/N_0$, means that $p_n(C(e);\mathrm{I})$ is exactly a log-domain PMF for the code vector $\mathbf{c}_n \leftrightarrow C_n$ for each time step.

The remaining task of the CPM demodulator is identical to the sole task of the CPM decoder, which is to generate the soft output for $\mathbf{u}$. Because of the presence of the known ASM sequence on either side of the code word, the CPM demodulator/decoder operations can take place over the extended bit interval $-N_{WU} \le i \le N + N_{WU} - 1$, where $N_{WU} \le N_{ASM}$ is a "warm-up" period for the forward–backward trellis processing. Extending (21) to cover the warm-up period is straightforward due to the availability of $r(t)$ during those intervals. To extend the time intervals for the other SISO soft input, $\{\lambda_i(u;\mathrm{I})\}$, we directly exploit the known values of the ASM bits, like so:

$$\lambda_i(u;\mathrm{I}) = \begin{cases} (-\infty)a_2(\mathrm{ASM}_{i+N_{ASM}}), & -N_{ASM} \le i \le -1 \\ \lambda_{\Pi(i)}(y;\mathrm{O}), & 0 \le i \le N-1 \\ (-\infty)a_2(\mathrm{ASM}_{i-N}), & N \le i \le N + N_{ASM} - 1 \end{cases} \tag{22}$$

where $a_2(\cdot)$ is the binary antipodal function in (3), which is then scaled to represent an L-value with infinite confidence [the minus sign is due to our definition of the L-value in (56) in the Appendix]. As will be explained shortly, $\lambda_i(y;\mathrm{O})$ is the extrinsic *a posteriori* output of the LDPC decoder. This output is not available for the outside-the-loop operations of the CPM demodulator, and so the "middle" portion of $\lambda_i(u;\mathrm{I})$, i.e. $\{\lambda_i(u;\mathrm{I})\}_{i=0}^{N-1}$, is initialized to zero for that step as shown in Figure 10 (an L-value value of zero corresponds to equiprobable, or "no *a priori* information"). Additional details of the inner workings of the log-based SISO module are given in Section 6.

As shown in Figure 10, the SISO operations take place within a larger "global" iterative decoding loop that performs a defined maximum number of iterations, $It_{\max}$. For the first global iteration, the SISO task was completed outside the loop by the CPM demodulator and that result is passed forward via the switch in position "A." For successive global iterations, the switch is placed in position "B" so that the CPM SISO decoder's updates are used. For these operations, the CPM SISO decoder accepts the nonzero input sequence $\{\lambda_i(u;\mathrm{I})\}$ in (22) and returns the *extrinsic* update for $\mathbf{u}$, $\{\lambda_i(u;\mathrm{O})\}$. The SISO module does not need to compute outputs corresponding to the warm-up intervals.

When the SISO operations are completed, $\{\lambda_i(u;\mathrm{O})\}_{i=0}^{N-1}$ (i.e. the portion excluding the warm-up intervals) is passed to a deinterleaver (denoted by the symbol Π^{-1}), whose output is obtained as

$$\lambda_{\Pi(i)}(y;\mathrm{I}) = \lambda_i(u;\mathrm{O}) \qquad 0 \le i \le N-1 \tag{23}$$

At this point, the focus in the receiver shifts to the LDPC decoder. As depicted in Figure 10, the LDPC decoder has two soft inputs. The first is the sequence of *a priori* L-values for $\mathbf{y}$, which arrives from the deinterleaver, i.e. (23). In general, the LDPC decoder requires this input to exist out to a length of N_{P}, where N_{P} may be slightly greater than N if puncturing is used. In such a case the additional values, $\{\lambda_i(y;\mathrm{I})\}_{i=N}^{N_{\mathrm{P}}-1}$, are set to zero because the punctured bits were not transmitted and thus no information was received.

The second input to the LDPC decoder is the set of "edge memory buffers," $\{\boldsymbol{\eta}(e)\}$, which comprises the internal state memory of the LDPC decoder. These buffers are initialized to zero (no information) at the commencement of the first global iteration. The LDPC decoder performs It_{loc} iterations in a "local" loop between *variable node* and *check node* updates, where the edge memory buffers store the results of these updates. When the It_{loc} local iterations are completed, $\{\boldsymbol{\eta}(e)\}$ is output so it can be preserved until the LDPC decoding portion of the next *global* iteration.

The $\mathbf{y}$ output of the LDPC decoder is computed in two formats. The first is an extrinsic version of the *a posteriori* L-value sequence, $\{\lambda_i(y;\mathrm{O})\}_{i=0}^{N-1}$, which does not need to be computed for any punctured bits that may exist. This is re-interleaved and fed to the SISO module for the next global iteration [as described in (22)]. The second is the full (non-extrinsic) *a posteriori* probability (APP) (in L-value format), which we denote as $\{\Lambda_i(y;\mathrm{O})\}_{i=0}^{N_{\mathrm{P}}-1}$. The APP output is the basis for the output hard decisions, $\hat{\mathbf{y}}$, which are used in the parity check that takes place at the end of each global iteration. No further global iterations are needed if this parity check passes.

LDPC decoders are known to have a high "peak to average" ratio when it comes to the number of iterations needed in order to pass the parity check, cf. e.g. [3]. As with any setting where such a ratio is large, this can result in a system that is overdesigned and inefficient for average use (to accommodate the peak), or one that is underdesigned so that it performs poorly for peak use (but is efficient on average). Approaches to dealing with this problem were studied in our companion paper [8] for the LDPC–CPM decoder in Figure 10, and some results from [8] are presented in Sections 7.2 and 7.3.

4.2 | Native vs. Punctured Receiver Configurations

The studies in [7], [8] also explored how the selection of $It_{\mathrm{loc}} > 1$ has some advantages and disadvantages. The advantage

of $It_{\text{loc}} > 1$ is that it can help "balance" the complexity of the LDPC and CPM halves of the global iteration, especially if the CPM half has higher complexity (as it does for ARTM2). The disadvantage of $It_{\text{loc}} > 1$ is that it *isolates* the LDPC decoder from the CPM SISO, making the LDPC decoder more prone to *undetected errors* due to the relatively small values of minimum distance for the LDPC codes designed in this study (see Table 1 for the minimum distance values we have directly observed in our simulations, which we denote as d^*_{min}). The solution to the problem of undetected errors is the puncturing/check node splitting technique that was described in Section 3.4. In [7] it was shown that the interactions between the CPM SISO and the check node splitting provide protection against undetected errors. Thus, if larger values of It_{loc} are of interest, the study in [7] recommended the upper bound $It_{\text{loc}} \leq It^*_{\text{loc}}$ along with the use of the punctured version of the parity check matrix, $\mathbf{H}$. In simulations exceeding 10^9 code words in each case, [7] reported that no undetected errors were observed in these configurations.

In Table 1, we have listed the "native" codes as being inclusive of the parameter selection $It_{\text{loc}} = 1$, and we have listed the "punctured" codes as being inclusive of the parameter selection $It_{\text{loc}} = It^*_{\text{loc}}$. The coupling of the native/punctured $\mathbf{H}$ with these parameter selections will be maintained throughout this document. We also emphasize that these are design options that affect the *receiver* only. The transmitter in Figure 1 is completely unaware of and unaffected by these options, and the original (native) generator matrix $\mathbf{G}$ is always used in Figure 1.

We now transition to the important subject of the decoding algorithms themselves; specifically, the LDPC decoder, the SISO algorithms, and ways of implementing these in a high-throughput/parallelized manner.

5 | QUASI-CYCLIC LDPC DECODER

5.1| Motivating Example

We begin with an example to motivate the structure of the LDPC decoder. As discussed in Section 3, when reduced to its essential elements, an LDPC code consists of variable nodes, check nodes, and the edges that connect them.

We consider a node for variable y that is connected to five check nodes via the edges $\mathcal{E} = \{e_0, e_1, e_2, e_3, e_4\}$ where $d = |\mathcal{E}| = 5$. Although not shown in the Tanner graphs of Figures 6 and 9, variable nodes in the decoder have an additional edge, which is the connection of y to the outside. For our example node, these six edges each have an "input" message (or a "vote" on the value of y) that arrives in the form of an L-value: $\eta(e_0)$, $\eta(e_1)$, $\eta(e_2)$, $\eta(e_3)$, and $\eta(e_4)$ are the messages arriving from the check nodes, and λ_{I} is the *a priori* L-value for y arriving from the outside. Our task is simply to update the messages back to each check node, as suggested by the name *variable node update*. The five update equations are

$$\begin{aligned}
\eta(e_0) &= \eta(e_1) + \eta(e_2) + \eta(e_3) + \eta(e_4) + \lambda_{\text{I}} \\
\eta(e_1) &= \eta(e_0) + \eta(e_2) + \eta(e_3) + \eta(e_4) + \lambda_{\text{I}} \\
\eta(e_2) &= \eta(e_0) + \eta(e_1) + \eta(e_3) + \eta(e_4) + \lambda_{\text{I}} \\
\eta(e_3) &= \eta(e_0) + \eta(e_1) + \eta(e_2) + \eta(e_4) + \lambda_{\text{I}} \\
\eta(e_4) &= \eta(e_0) + \eta(e_1) + \eta(e_2) + \eta(e_3) + \lambda_{\text{I}}
\end{aligned} \tag{24}$$

which is a simple combining (or fusing) of the "votes" coming from each source. The primary feature of each update is that it is *extrinsic*, meaning that it combines inputs from all available sources *outside of itself*, i.e. the update for edge e_0 is missing $\eta(e_0)$, and likewise for the other updates. The missing term represents what was known on that edge prior (*a priori*) to the update, and the extrinsic update represents the *value added* to that edge by the structure of the code. If the *a priori* values were included in the updates, then positive feedback would *over-accumulate* these values from one iteration to the next. At the commencement of decoding, all values of $\eta(\cdot)$ are zero and thus the outside input λ_{I} is the only information available for y.

The reader will notice that we *reuse* the incoming memory when forming the outputs in (24), i.e e_0 is found on both the left- and right-hand sides of the equal sign in (24). The apparent problem of overwriting and confusing inputs and outputs will be resolved momentarily when we introduce some working memory.

Each update requires 4 additions and thus 20 additions are required to implement (24). In theory, all inputs could be summed first and then the *a priori* value for each edge could be "backed out" (subtracted), which would require only 5 additions and 5 subtractions. This extrinsic-by-inverse approach requires the existence of inverse operator, and although addition is far from being exotic, subtraction is problematic in a fixed-point/integer implementation when values become *saturated*. Thus, we favor the extrinsic-by-exclusion approach in (24) for its numerical stability.

In [2], a more efficient approach to computing (24) was outlined that consists of a forward recursion, a backward recursion, and a completion step. The forward and backward recursion are

$$\begin{aligned}
z_{\text{F}}(0) &= \eta(e_0) + \lambda_{\text{I}} & z_{\text{B}}(4) &= \eta(e_4) \\
z_{\text{F}}(1) &= z_{\text{F}}(0) + \eta(e_1) & z_{\text{B}}(3) &= z_{\text{B}}(4) + \eta(e_3) \\
z_{\text{F}}(2) &= z_{\text{F}}(1) + \eta(e_2) & z_{\text{B}}(2) &= z_{\text{B}}(3) + \eta(e_2) \\
z_{\text{F}}(3) &= z_{\text{F}}(2) + \eta(e_3) & z_{\text{B}}(1) &= z_{\text{B}}(2) + \eta(e_1)
\end{aligned}$$

where $z_{\text{F}}(\cdot)$ and $z_{\text{B}}(\cdot)$ are the working memory for the recursions and the values of $\eta(\cdot)$ are accessed only to read. The completion step fuses the results of these recursions

$$\begin{aligned}
\eta(e_0) &= z_{\text{B}}(1) + \lambda_{\text{I}} \\
\eta(e_1) &= z_{\text{F}}(0) + z_{\text{B}}(2) \\
\eta(e_2) &= z_{\text{F}}(1) + z_{\text{B}}(3) \\
\eta(e_3) &= z_{\text{F}}(2) + z_{\text{B}}(4) \\
\eta(e_4) &= z_{\text{F}}(3)
\end{aligned}$$

where the values of $\eta(\cdot)$ are accessed only to write. This arrangement requires only 11 additions to compute the extrinsic updates.

We now turn our attention to a check node that is connected to five variable nodes via the edges $\mathcal{E} = \{e_0, e_1, e_2, e_3, e_4\}$ where $d = |\mathcal{E}| = 5$. The very meaning of a "check" is that the following equation must be true:

$$0 = y_0 \oplus y_1 \oplus y_2 \oplus y_3 \oplus y_4 \tag{25}$$

where $\oplus$ is the XOR operator and y_0, y_1, y_2, y_3, and y_4 are the transmitted (correct) values of the variables that belong to $\mathcal{E}$ for

this check. Although it is mathematical nonsense, we will take turns solving this single equation for each of the five unknowns:

$$\begin{aligned} y_0 &= y_1 \oplus y_2 \oplus y_3 \oplus y_4 \\ y_1 &= y_0 \oplus y_2 \oplus y_3 \oplus y_4 \\ y_2 &= y_0 \oplus y_1 \oplus y_3 \oplus y_4 \\ y_3 &= y_0 \oplus y_1 \oplus y_2 \oplus y_4 \\ y_4 &= y_0 \oplus y_1 \oplus y_2 \oplus y_3 \end{aligned} \tag{26}$$

The genius of LDPC codes is that the Tanner graph is so sparse (*low density*) that this nonsense can be used to formulate the messages sent from a particular check node. It takes a number of iterations for anything to propagate back to this same node (which relates to the *girth* of the code), at which point the extrinsic information gained in the process can be viewed approximately as being independent.

The similarities between (26) and (24) are already apparent. If the binary variables a, b, and c have the relationship $c = a \oplus b$, their respective L-values have the relationship [2] $\eta_c = \eta_a \boxplus \eta_b$, where it is convenient at the moment to use the notation $\boxplus$ but later we will use $\eta_a \boxplus \eta_b = \min\star(\eta_a, \eta_a)$ as defined in (54). Rewriting (26) with η and $\boxplus$ completes its similarity with (24).

The five edges in our check node have "input" messages $\eta(e_0)$, $\eta(e_1)$, $\eta(e_2)$, $\eta(e_3)$, and $\eta(e_4)$ arriving from the connected variable nodes. The five check node update equations are (26) except rewritten with η and $\boxplus$. The inverse "$\boxminus$" operator does not exist and thus extrinsic-by-exclusion is the only option. The more-efficient forward/backward/completion implementation is

$$\begin{aligned} z_{\mathrm{F}}(0) &= \eta(e_0) & z_{\mathrm{B}}(4) &= \eta(e_4) \\ z_{\mathrm{F}}(1) &= z_{\mathrm{F}}(0) \boxplus \eta(e_1) & z_{\mathrm{B}}(3) &= z_{\mathrm{B}}(4) \boxplus \eta(e_3) \\ z_{\mathrm{F}}(2) &= z_{\mathrm{F}}(1) \boxplus \eta(e_2) & z_{\mathrm{B}}(2) &= z_{\mathrm{B}}(3) \boxplus \eta(e_2) \\ z_{\mathrm{F}}(3) &= z_{\mathrm{F}}(2) \boxplus \eta(e_3) & z_{\mathrm{B}}(1) &= z_{\mathrm{B}}(2) \boxplus \eta(e_1) \end{aligned}$$

and

$$\begin{aligned} \eta(e_0) &= z_{\mathrm{B}}(1) \\ \eta(e_1) &= z_{\mathrm{F}}(0) \boxplus z_{\mathrm{B}}(2) \\ \eta(e_2) &= z_{\mathrm{F}}(1) \boxplus z_{\mathrm{B}}(3) \\ \eta(e_3) &= z_{\mathrm{F}}(2) \boxplus z_{\mathrm{B}}(4) \\ \eta(e_4) &= z_{\mathrm{F}}(3) \end{aligned}$$

This arrangement requires only 9 $\boxplus$ operations because λ_{I} does not need to be accounted for (or it can be left in these equations and set to a value of $\lambda_{\mathrm{I}} = +\infty$, which is the identity value of the $\boxplus$ operator).

The last step in our motivating example is to return to the variable node y (*after* the check node update) and update the message on its connection to the outside. The *extrinsic* form of this output message is

$$\lambda_{\mathrm{O}} = \eta(e_0) + \eta(e_1) + \eta(e_2) + \eta(e_3) + \eta(e_4) \tag{27}$$

which excludes the *a priori* value λ_{I}. Once again, the extrinsic output represents the *value added* by the structure of the code and thus the extrinsic output should be connected to any subsequent decoder to avoid positive feedback/over accumulation. The full (non-extrinsic) APP output is $\Lambda_{\mathrm{O}} = \lambda_{\mathrm{I}} + \lambda_{\mathrm{O}}$, and the $\pm$ sign of the APP output is the basis for the hard decision $\hat{y}$. Once hard decisions are made, we can return to our example check node to see if (25) is satisfied using the hard decisions.

5.2 | Quasi-Cyclic Decoder Formulation

The quasi-cyclic LDPC decoder exploits the fact that $\mathbf{H}$ is composed of $M_{\mathrm{L}} \times M_{\mathrm{L}}$ circulants. The decoder is organized around the sparse representation of $\mathbf{H}$ that was developed in Section 3.3, which in turn is based almost entirely on the much-smaller intermediate protomatrix, $\mathbf{D}$ (see the dimensions listed in Table 1). The principal quantities in the sparse representation are M_{L}, d_n, $\mathcal{E}_n$, d_m, $\mathcal{E}_m$, and Φ_m, with implied variables M_{D}, N_{D}, and N_{E}.

We allocate a length-M_{L} memory buffer (*vector*) for each edge, $\boldsymbol{\eta}(e)$, which requires a total of $N_{\mathrm{E}} M_{\mathrm{L}}$ discrete memory elements for the entire set $\{\boldsymbol{\eta}(e)\}_{e=0}^{N_{\mathrm{E}}-1}$. The processing/updating of these memory buffers (vectors) is described in a completely parallel fashion (vector-wise), i.e. M_{L} elements at a time. Although each element in the buffer can be accessed and processed in parallel, "element zero" is considered to be the element at position zero in the buffer. If we apply a phase of ϕ when accessing a buffer, as in $\boldsymbol{\eta}(e, \phi)$, then the element originally at position zero is right-shifted to position ϕ in the buffer. This could be implemented in actuality [i.e. a right-hand circular (barrel) shift ϕ times] or by use of a pointer, etc. We formulate everything such that phase is dealt with only during *check node* processing. From the perspective of the *variable nodes*, the edges have no (zero) phase associated with them. The following are a few useful properties of the phase shift: $\boldsymbol{\eta}(e, 0)$ is the same as $\boldsymbol{\eta}(e)$; ϕ can assume any integer value (including negative integers), because it is understood that phase is applied in a modulo-M_{L} fashion; a phase shift of ϕ is "undone" by another phase shift of $-\phi$.

Variable Node Update. The first half of an LDPC decoding iteration consists of the variable node updates. Generically speaking, each update involves a set of edges $\mathcal{E}$, $d = |\mathcal{E}|$, and an *a priori* input vector $\boldsymbol{\lambda}_{\mathrm{I}}$ from the outside. To perform the update for the n-th variable node, we initialize these generic quantities with the ones specific to that node: $\mathcal{E} = \mathcal{E}_n$ and $\boldsymbol{\lambda}_{\mathrm{I}} = \boldsymbol{\lambda}(n; \mathrm{I})$, where $\boldsymbol{\lambda}(n; \mathrm{I})$ is the length-M_{L} segment of information received from the SISO module belonging to the n-th circulant of $\mathbf{y}$. Motivated by (24), the brute-force extrinsic update is performed for each edge in the set:

$$\boldsymbol{\eta}'(e_j) = \boldsymbol{\lambda}_{\mathrm{I}} + \sum_{\substack{e_i \in \mathcal{E} \\ e_i \neq e_j}} \boldsymbol{\eta}(e_i), \qquad 0 \leq j \leq d-1 \tag{28}$$

where $\boldsymbol{\eta}'$ denotes temporary "working" memory that is needed only until all updates in the set have been made, after which we can overwrite the input values with the updates: $\boldsymbol{\eta}(e_j) = \boldsymbol{\eta}'(e_j)$, $\forall\, j$.

The more-efficient forward/backward/completion approach of computing extrinsic-by-exclusion is summarized as a generic processing unit in Algorithm 2. This unit has forward and backward buffers (vectors) as working memory, $\mathbf{z}_{\mathrm{F}}(\cdot)$ and $\mathbf{z}_{\mathrm{B}}(\cdot)$, respectively, and can be adapted for use as either the variable node or check node update.

The particulars of Algorithm 2 unique to the variable node update are the following: the generic "update operator" is simple vector addition $\mathrm{w}(\mathbf{a}, \mathbf{b}) = \mathbf{a} + \mathbf{b}$ (Algorithm 2 Line 2); the edges

have no (zero) phase and all references to phase can be ignored; the *a priori* optional input is passed as $\boldsymbol{\lambda}_{\mathrm{I}}$ (Algorithm 2 Line 3); and the "completion factor" is not needed, i.e. $\kappa = 1$ and can be ignored (Algorithm 2 Line 4). With simple addition, we include the caveat that in a finite precision implementation it becomes addition with saturation.

Check Node Update. The second half of an LDPC decoding iteration consists of the check node updates. Generically speaking, each update involves a set of edges $\mathcal{E}$, $d = |\mathcal{E}|$, and their associated phases Φ. To perform the update for the m-th check node, we initialize these generic quantities with the ones specific to that node: $\mathcal{E} = \mathcal{E}_m$ and $\Phi = \Phi_m$. Motivated by (26) after rewriting with η and $\min\star$ and incorporating phase, the brute-force extrinsic update is performed for each edge in the set:

$$\boldsymbol{\eta}'(e_j) = \min_{\substack{e_i \in \mathcal{E} \\ e_i \neq e_j}} \star \{\boldsymbol{\eta}(e_i, -\phi_i)\} \tag{29}$$

$$\boldsymbol{\eta}'(e_j) = \kappa \cdot \boldsymbol{\eta}'(e_j, \phi_j), \qquad 0 \leq j \leq d-1 \tag{30}$$

which is a two-step procedure where the result from the first step is then phase-shifted and scaled by the "completion factor," κ. Once again, when all the updates have been made, they are transferred from the working memory to the actual edge memory: $\boldsymbol{\eta}(e_j) = \boldsymbol{\eta}'(e_j)$, $\forall\, j$.

The particulars of Algorithm 2 unique to the check node update are the following: the generic "update operator" is $\mathrm{w}(\mathbf{a}, \mathbf{b}) = \min\star(\mathbf{a}, \mathbf{b})$ (Algorithm 2 Line 2); the *a priori* optional input is not needed and is thus assumed to be $+\infty$, i.e. the identity value of $\min\star(a, b)$ (Algorithm 2 Line 3); and the "completion factor" is not needed, i.e. $\kappa = 1$ and can be ignored (Algorithm 2 Line 4). The $\min\star(a, b)$ operator is described for scalar input/output in (54) in the Appendix. For vectors, it is applied on an element-by-element basis.

A low-complexity version of the check node update is when we drop the "correction" terms, $\mathrm{f}(|a|+|b|) - \mathrm{f}(|a|-|b|)$, within the $\min\star(a, b)$ definition in (54). This is referred to as the "simple min" approximation. In addition to simplifying the computations, this approximation also eliminates the requirement for the precise scale factor $\sqrt{E_s} L_c/2$ in (21). These modifications result in "relative reliabilities" (RRs) being exchanged between the SISO module and the LDPC decoder, instead of true L-values (As discussed in the Appendix). The simplified version of $\min\star(\mathbf{a}, \mathbf{b})$ tends to "over estimate" the RRs, and thus $\kappa = 3/4$ is recommended to attenuate this effect.

LDPC Decoder Operations. When a new received code word arrives, the "global" iterative loop commences operation. The LDPC decoder initializes all edge memory buffers to zero before these iterations begin: $\boldsymbol{\eta}(e) = \mathbf{0}$ for $0 \leq e \leq N_{\mathrm{E}} - 1$. The first half of each global iteration involves the SISO module, and the second half involves the LDPC decoder (with interleaving/deinterleaving taking place inbetween). The two decoders exchange buffers (vectors) of L-values that have slightly different lengths. On the SISO side, there are "warm up" values for the forward–backward trellis processing. On the LDPC side, if puncturing is used then there is zero-padding so the decoder is provided the length-N_{P} code word it expects. These details have been covered above.

As described in Algorithm 3, the LDPC decoder has three basic operations: the variable node update; the check node update, and output computation. The variable and check node updates take place within a "local" iterative loop that executes a total of It_{loc} iterations. These node updates use the generic processing unit described in Algorithm 2, with some customization to each case as detailed above. As shown in Algorithm 3, these node updates stride through the columns and rows of the intermediate protomatrix, **D**, with repeated read/write access to the edge memory buffers specified in each node set $\mathcal{E}_n$ and $\mathcal{E}_m$.

When It_{loc} local iterations are complete, the decoder computes two distinct outputs. The first is the extrinsic output that will be passed back to the SISO module for the next global iteration. The second is the full APP output, which constitutes the final output of the global iterative decoder. The specifics of these output computations are described next.

The LDPC decoder also "outputs" the values contained in the edge memory buffers, $\{\boldsymbol{\eta}(e)\}$, to be preserved for the next global iteration. It is not essential that these are true outputs, it is only essential that they are preserved for the next global iteration. The values in these buffers were updated in the last check node update of the current global iteration, and they will be used in the first variable node update of the next global iteration, along with an updated *a priori* input that will come from the next execution of the SISO module.

LDPC Output Computation. The LDPC output computation is performed from one variable node to the next, and is thus somewhat similar to the variable node update. We are given

Algorithm 2 Generic Extrinsic Processing Unit.

1: **Input/Output:** $\boldsymbol{\eta}(e_0), \boldsymbol{\eta}(e_1), \ldots, \boldsymbol{\eta}(e_{d-1})$, the set of edge memory buffers belonging to $\mathcal{E} = \{e_0, e_1, \ldots, e_{d-1}\}$, $d = |\mathcal{E}|$, with phases $\Phi = \{\phi_0, \phi_1, \ldots, \phi_{d-1}\}$ (assumed to be all zeros for variable node processing);
2: **To be specified:** "update operator" $\mathrm{w}(\mathbf{a}, \mathbf{b})$;
3: **Optional Input**: $\boldsymbol{\lambda}_{\mathrm{I}}$, the external *a priori* value. If not specified, it is assumed to be the identity value of the update operator;
4: **Optional Parameter:** "completion factor" κ. Assumed to be $\kappa = 1$ if not specified;
5: **Initialization:** $\mathbf{z}_{\mathrm{F}}(0) = \mathrm{w}(\boldsymbol{\eta}(e_0, -\phi_0), \boldsymbol{\lambda}_{\mathrm{I}})$;
6: **Initialization:** $\mathbf{z}_{\mathrm{B}}(d-1) = \boldsymbol{\eta}(e_{d-1}, -\phi_{d-1})$;
Forward and Backward "accumulations":
7: **for** $i = 1, 2, \ldots, d-2$ **do**
8: $\quad \mathbf{z}_{\mathrm{F}}(i) = \mathrm{w}(\mathbf{z}_{\mathrm{F}}(i-1), \boldsymbol{\eta}(e_i, -\phi_i))$;
9: **end for**
10: **for** $i = d-2, \ldots, 2, 1$ **do**
11: $\quad \mathbf{z}_{\mathrm{B}}(i) = \mathrm{w}(\mathbf{z}_{\mathrm{B}}(i+1), \boldsymbol{\eta}(e_i, -\phi_i))$;
12: **end for**
Extrinsic Outputs:
13: $\boldsymbol{\eta}(e_0) = \mathrm{w}(\mathbf{z}_{\mathrm{B}}(1), \boldsymbol{\lambda}_{\mathrm{I}})$;
14: $\boldsymbol{\eta}(e_{d-1}) = \mathbf{z}_{\mathrm{F}}(d-2)$;
15: **for** $i = 1, 2, \ldots, d-2$ **do**
16: $\quad \boldsymbol{\eta}(e_i) = \mathrm{w}(\mathbf{z}_{\mathrm{F}}(d-1), \mathbf{z}_{\mathrm{B}}(d+1))$;
17: **end for**
Inverse Phase Shift and Completion:
18: **for** $i = 0, 1, \ldots, d-1$ **do**
19: $\quad \boldsymbol{\eta}(e_i) = \kappa \cdot \boldsymbol{\eta}(e_i, \phi_i)$;
20: **end for**

Algorithm 3 LDPC Decoder Operation.

Initialization: Set $\boldsymbol{\eta}(e) = \mathbf{0}$ for $0 \le e \le N_{\mathrm{E}} - 1$ prior to the first global iteration;
Input: $\{\lambda_i(y;\mathrm{I})\}_{i=0}^{N_{\mathrm{P}}-1} = \{\boldsymbol{\lambda}(0;\mathrm{I}), \boldsymbol{\lambda}(1;\mathrm{I}), \ldots, \boldsymbol{\lambda}(N_{\mathrm{D}}-1;\mathrm{I})\}$, the *a priori* L-values from the SISO module, Eq. (22);
Input: $\{\boldsymbol{\eta}(0), \boldsymbol{\eta}(1), \ldots, \boldsymbol{\eta}(N_{\mathrm{E}}-1)\}$, the edge memory buffers, preserved from the previous global iteration;
Output: $\{\lambda_i(y;\mathrm{O})\}_{i=0}^{N_{\mathrm{P}}-1} = \{\boldsymbol{\lambda}(0;\mathrm{O}), \boldsymbol{\lambda}(1;\mathrm{O}), \ldots, \boldsymbol{\lambda}(N_{\mathrm{D}}-1;\mathrm{O})\}$, the extrinsic *a posteriori* L-values;
Output: $\{\Lambda_i(y;\mathrm{O})\}_{i=0}^{N_{\mathrm{P}}-1} = \{\boldsymbol{\Lambda}(0;\mathrm{O}), \boldsymbol{\Lambda}(1;\mathrm{O}), \ldots, \boldsymbol{\Lambda}(N_{\mathrm{D}}-1;\mathrm{O})\}$, the full *a posteriori* probability (APP) L-values;
Output: $\{\boldsymbol{\eta}(0), \boldsymbol{\eta}(1), \ldots, \boldsymbol{\eta}(N_{\mathrm{E}}-1)\}$, the edge memory buffers, to be preserved for the next global iteration;
Local/LDPC Iterative Loop:
for $i = 0, 1, \ldots, It_{\mathrm{loc}} - 1$ **do**
  for $n = 0, 1, \ldots, N_{\mathrm{D}} - 1$ **do**
    Variable node update with $\mathcal{E} = \mathcal{E}_n$ and $\boldsymbol{\lambda}_{\mathrm{I}} = \boldsymbol{\lambda}(n;\mathrm{I})$;
  end for
  for $m = 0, 1, \ldots, M_{\mathrm{D}} - 1$ **do**
    Check node update with $\mathcal{E} = \mathcal{E}_m$ and $\Phi = \Phi_m$;
  end for
end for
Compute Outputs:
for $n = 0, 1, \ldots, N_{\mathrm{D}} - 1$ **do**
  Compute $\boldsymbol{\lambda}_{\mathrm{O}}$ and $\boldsymbol{\Lambda}_{\mathrm{O}}$ with $\mathcal{E} = \mathcal{E}_n$ and $\boldsymbol{\lambda}_{\mathrm{I}} = \boldsymbol{\lambda}(n;\mathrm{I})$;
  Set $\boldsymbol{\lambda}(n;\mathrm{O}) = \boldsymbol{\lambda}_{\mathrm{O}}$ and $\boldsymbol{\Lambda}(n;\mathrm{O}) = \boldsymbol{\Lambda}_{\mathrm{O}}$;
end for

the set of edges that belong to the n-th variable node, $\mathcal{E} = \mathcal{E}_n$, along with the *a priori* input, $\boldsymbol{\lambda}_{\mathrm{I}} = \boldsymbol{\lambda}(n;\mathrm{I})$, which is the length-$M_{\mathrm{L}}$ segment of information received from the SISO module belonging to the n-th circulant of $\mathbf{y}$ Our task is to form a length-M_{L} *extrinsic* output, $\boldsymbol{\lambda}_{\mathrm{O}}$, and a length-$M_{\mathrm{L}}$ full APP output, $\boldsymbol{\Lambda}_{\mathrm{O}}$. Motivated by (27), the extrinsic output is

$$\boldsymbol{\lambda}_{\mathrm{O}} = \sum_{e_i \in \mathcal{E}} \boldsymbol{\eta}(e_i) \tag{31}$$

which excludes $\boldsymbol{\lambda}_{\mathrm{I}}$. The full APP output includes this missing term and is

$$\boldsymbol{\Lambda}_{\mathrm{O}} = \boldsymbol{\lambda}_{\mathrm{I}} + \sum_{e_i \in \mathcal{E}} \boldsymbol{\eta}(e_i) = \boldsymbol{\lambda}_{\mathrm{I}} + \boldsymbol{\lambda}_{\mathrm{O}}. \tag{32}$$

The final parity check operation begins by making hard decisions on the full APP output with the step

$$\hat{y}_i = \begin{cases} 1, & \Lambda_i(y;\mathrm{O}) < 0 \\ 0, & \Lambda_i(y;\mathrm{O}) > 0 \end{cases} \qquad 0 \le i \le N_{\mathrm{P}} - 1 \tag{33}$$

where the negative "sign" of the hard decisions is due to our definition of the L-value in (56) in the Appendix. The hard decisions are divided into length-M_{L} segments, $\{\hat{y}_i\}_{i=0}^{N_{\mathrm{P}}-1} = \{\hat{\mathbf{y}}(0), \hat{\mathbf{y}}(1), \ldots, \hat{\mathbf{y}}(N_{\mathrm{D}}-1)\}$, and the parity check operation is performed by processing these segments.

To do this, we define $\mathcal{N}_m = \{n_0, n_1, \ldots, n_{d-1}\}$ as the set of *variable nodes* that participate in (are non-zero in) the m-th check node (row) of $\mathbf{D}$. The comma-delineated list, $\mathcal{L}$, defined in Section 3.2 already contains this information in the exact order that is needed for this purpose. Recall that $\mathcal{L}$ begins with the value of M_{L}, which is already known and can be skipped. Thereafter, $\mathcal{L}$ is composed of $(d_m + 1)$-tuples of the form $\{d_m, \text{location}, \text{location}, \ldots, \text{location}\}$. Removing (skipping over) d_m leaves the set of bit locations belonging to the m-th check node, and applying the operation $\lfloor \text{location}/M_{\mathrm{L}} \rfloor$ to each location in this set yields the n values for that row of $\mathbf{D}$, which are the desired contents of $\mathcal{N}_m$.

For the m-th check node of $\mathbf{D}$, we set $\mathcal{N} = \mathcal{N}_m$, $\Phi = \Phi_m$, and perform the operation

$$\mathbf{v} = \hat{\mathbf{y}}(n_0, -\phi_0) \oplus \hat{\mathbf{y}}(n_1, -\phi_1) \oplus \cdots \oplus \hat{\mathbf{y}}(n_{d-1}, -\phi_{d-1}) \tag{34}$$

which is motivated by (25) after incorporating phase. A value of $\mathbf{v} = \mathbf{0}$ for all M_{D} check nodes means that $\hat{\mathbf{y}}$ passes the parity check. If the parity check passes, the global loop does not need to iterate further and can stop. If the parity check fails, then the global iterations continue until the maximum number of global iterations is reached, $It_{\max}$, at which point a *decoder failure* occurs and the current $\hat{\mathbf{y}}$ is returned, even though it is known to contain errors.

Another outcome that can occur is known as an *undetected error*, which is highly undesirable in some applications. This is an instance where the decoder converges on a $\hat{\mathbf{y}}$ that is a valid code word, albeit a different one than was transmitted (denoted as $\mathbf{y}$). When this occurs, the parity check *passes* (because a valid code word was found) and thus the decoder is *unaware* that bit errors are present.[4] The probability of undetected error is larger for codes that have smaller values of *minimum distance*. While [7] observed that the codes herein have the relatively small values of $d_{\min}^{*}$ listed in Table 1, it also demonstrated that the regular influx of fresh extrinsic information generated in the global iterations by the CPM SISO module serves to protect the LDPC decoder against undetected errors. Furthermore, [7] demonstrated that the check node splitting technique described in Section 3.4 enhances this protection.

6 | CPM SISO MODULE

6.1 | Basic Operation

The original development of the SISO module goes back to the BCJR algorithm in [21], although our formulation more closely resembles the notation of [22]. We apply several adaptations and customizations that are relevant to our application, including: (1) only one soft output is required, that belonging to $\{u_i\}$; (2) the soft input for the "code" actually belongs to a CPM, thus the metric increments must be formulated accordingly [this has already received some attention in (21)]; and (3) the entire algorithm is formulated using *log-based* probabilities, the additional background details for which are provided in the Appendix. The basic operations of the SISO module are given in Algorithms 4 and 5, where the former is specialized to the case of a binary ($M = 2$, $n_0 = 1$) CPM (such as ARTM0 and ARTM1) and the latter is specialized to the case of a $M = 4$ ($n_0 = 2$) CPM with a time-varying trellis (such as ARTM2). We will describe both

[4] In a simulation environment, the transmitted codeword $\mathbf{y}$ is available at the decoder and can be compared with $\hat{\mathbf{y}}$, thus exposing the undetected errors. However, in practice $\mathbf{y}$ is never present at the decoder, because that is the very point transmitting it in the first place!

Algorithm 4 Log-Based SISO APP Algorithm for $M=2$ ($n_0=1$).

Input: $\{p_n(C(e);\mathrm{I})\}$, for $-N'_{\mathrm{WU}} \le n \le N' + N'_{\mathrm{WU}} - 1$, the received *a priori* PMFs from the CPM MF bank, Eq. (21);
Input: $\{\lambda_i(u;\mathrm{I})\}$, for $-N_{\mathrm{WU}} \le i \le N + N_{\mathrm{WU}} - 1$, the *a priori* L-values from the LDPC decoder, Eq. (22);
Output: $\{\lambda_i(u;\mathrm{O})\}$, for $0 \le i \le N-1$, the extrinsic *a posteriori* L-values;
Optional Parameter: "completion factor" κ. Assumed to be $\kappa = 1$ if not specified;
Initialization: $A_{-N'_{\mathrm{WU}}-1}(s) = 0$ for $0 \le s \le N_{\mathrm{S}} - 1$;
Initialization: $B_{N'+N'_{\mathrm{WU}}-1}(s) = 0$ for $0 \le s \le N_{\mathrm{S}} - 1$;
Metric Increment:
$\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I}) = [\lambda_n(u;\mathrm{I})]^T$, for all n;
$\gamma_n(e) = -[\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
Forward and Backward Recursions:
for $n = -N'_{\mathrm{WU}}, \ldots, N'-3, N'-2$ **do**
  $A_n(s^{\mathrm{E}}) = \max^{\star}_{e \in \mathbf{e}(s^{\mathrm{E}})} \left\{ A_{n-1}(s^{\mathrm{S}}(e)) + \gamma_n(e) \right\}$, for all s^{E};
  Normalize $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
end for
for $n = N' + N'_{\mathrm{WU}} - 2, \ldots, 1, 0$ **do**
  $B_n(s^{\mathrm{S}}) = \max^{\star}_{e \in \mathbf{e}(s^{\mathrm{S}})} \left\{ B_{n+1}(s^{\mathrm{E}}(e)) + \gamma_{n+1}(e) \right\}$, for all s^{S};
  Normalize $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
end for
Extrinsic Increment:
$\chi_n(e) = p_n(C(e);\mathrm{I})$;
Completion Step:
for $n = 0, 1, \ldots, N'-1$ **do**
  $\lambda_n(u;\mathrm{O}) = \max^{\star}_{e:u(e)=0} \left\{ A_{n-1}(s^{\mathrm{S}}(e)) + \chi_n(e) + B_n(s^{\mathrm{E}}(e)) \right\} - \max^{\star}_{e:u(e)=1} \left\{ A_{n-1}(s^{\mathrm{S}}(e)) + \chi_n(e) + B_n(s^{\mathrm{E}}(e)) \right\}$;
end for
$\lambda_i(u;\mathrm{O}) = \kappa \cdot \lambda_i(u;\mathrm{O})$, for all i;

Algorithm 5 Log-Based SISO APP Algorithm for $M=4$ ($n_0=2$) and a time-varying trellis.

Input: $\{p_n(C(e);\mathrm{I})\}$, for $-N'_{\mathrm{WU}} \le n \le N' + N'_{\mathrm{WU}} - 1$, the received *a priori* PMFs from the CPM MF bank, Eq. (21);
Input: $\{\lambda_i(u;\mathrm{I})\}$, for $-N_{\mathrm{WU}} \le i \le N + N_{\mathrm{WU}} - 1$, the *a priori* L-values from the LDPC decoder, Eq. (22);
Output: $\{\lambda_i(u;\mathrm{O})\}$, for $0 \le i \le N-1$, the extrinsic *a posteriori* L-values;
Optional Parameter: "completion factor" κ. Assumed to be $\kappa = 1$ if not specified;
Initialization: $A_{-N'_{\mathrm{WU}}-1}(s) = 0$ for $0 \le s \le N_{\mathrm{S}} - 1$;
Initialization: $B_{N'+N'_{\mathrm{WU}}-1}(s) = 0$ for $0 \le s \le N_{\mathrm{S}} - 1$;
Metric Increment:
$\boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I}) = [\lambda_{2n}(u;\mathrm{I}), \quad 0 \quad]^T$, for all n;
$\boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I}) = [\quad 0, \quad \lambda_{2n+1}(u;\mathrm{I})]^T$, for all n;
$\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I}) = \boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I}) + \boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I})$;
$\gamma_n(e) = -[\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
Forward and Backward Recursions:
for $n = -N'_{\mathrm{WU}}, \ldots, N'-3, N'-2$ **do**
  $A_n(s^{\mathrm{E}}) = \max^{\star}_{e \in \mathbf{e}(s^{\mathrm{E}})} \left\{ A_{n-1}(s_n^{\mathrm{S}}(e)) + \gamma_n(e) \right\}$, for all s^{E};
  Normalize $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
end for
for $n = N' + N'_{\mathrm{WU}} - 2, \ldots, 1, 0$ **do**
  $B_n(s^{\mathrm{S}}) = \max^{\star}_{e \in \mathbf{e}(s^{\mathrm{S}})} \left\{ B_{n+1}(s_n^{\mathrm{E}}(e)) + \gamma_{n+1}(e) \right\}$, for all s^{S};
  Normalize $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$
end for
Extrinsic Increments:
$\chi_n^{(0)}(e) = -[\boldsymbol{\lambda}_n^{(1)}(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
$\chi_n^{(1)}(e) = -[\boldsymbol{\lambda}_n^{(0)}(\mathbf{u};\mathrm{I})]^T a_2(\mathbf{u}(e))/2 + p_n(C(e);\mathrm{I})$;
Completion Step:
for $n = 0, 1, \ldots, N'-1$ **do**
  $\lambda_{2n}(u;\mathrm{O}) = \max^{\star}_{e:u_0(e)=0} \left\{ A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(0)}(e) + B_n(s_n^{\mathrm{E}}(e)) \right\} - \max^{\star}_{e:u_0(e)=1} \left\{ A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(0)}(e) + B_n(s_n^{\mathrm{E}}(e)) \right\}$;
  $\lambda_{2n+1}(u;\mathrm{O}) = \max^{\star}_{e:u_1(e)=0} \left\{ A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(1)}(e) + B_n(s_n^{\mathrm{E}}(e)) \right\} - \max^{\star}_{e:u_1(e)=1} \left\{ A_{n-1}(s_n^{\mathrm{S}}(e)) + \chi_n^{(1)}(e) + B_n(s_n^{\mathrm{E}}(e)) \right\}$;
end for
$\lambda_i(u;\mathrm{O}) = \kappa \cdot \lambda_i(u;\mathrm{O})$, for all i;

formulations simultaneously, which will allow us to compare and contrast their similarities and differences.

The CPM trellis is formed by the "code" variable defined in (6), which has an integer or vector representation $C_n \leftrightarrow \mathbf{c}_n$. Each edge in the trellis has a unique value of $C_n \leftrightarrow \mathbf{c}_n$, and the various sub-components of $C_n \leftrightarrow \mathbf{c}_n$ can be stored in separate LUTs that are indexed by a given edge, e. The first two LUTs are $s^{\mathrm{S}}(e)$ and $s^{\mathrm{E}}(e)$, which are the starting state and the ending state, respectively. When there are multiple modulation indexes ($N_h > 1$), as in Algorithm 5, these two LUTs become time varying and thus we add a time subscript, $s_n^{\mathrm{S}}(e)$ and $s_n^{\mathrm{E}}(e)$, which is understood to be taken modulo-N_h. The remaining LUTs are: $C(e)$, the code symbol[5] for a given edge; $\mathbf{u}(e)$, which is the length-n_0 bit vector for a given edge, which corresponds to $U_0(e)$, the "least significant" symbol in (6) [the relationship between $U_0(e)$ and $\mathbf{u}(e)$ is given by (60)]; $u(e)$, $u_0(e)$, and $u_1(e)$, which are the elements (bits) of $\mathbf{u}(e)$ that are needed for the $n_0 = 1$ and $n_0 = 2$ cases.

The SISO module accepts two *a priori* soft inputs: one for the "code" sequence $\{C_n\}$ and one for the "information" sequence[6] $\{u_i\}$; it returns an extrinsic *a posteriori* soft output for the information sequence. The $\{C_n\}$ input is based on CPM *symbols* and uses a symbol-based index, n. The $\{u_i\}$ input and output are based on LDPC encoded *bits* and use a bit-based index, i. These indexes cover the following "time" ranges

$$\begin{aligned} -N'_{\mathrm{WU}} &\le n \le N' + N'_{\mathrm{WU}} - 1 \\ -N_{\mathrm{WU}} &\le i \le N + N_{\mathrm{WU}} - 1 \end{aligned} \tag{35}$$

where $N_{\mathrm{WU}} \le N_{ASM}$ is the "warm-up" period for the forward and backward recursions, $N' = N/n_0$, and $N'_{\mathrm{WU}} = N_{\mathrm{WU}}/n_0$. At

[5] As was mentioned earlier, $C(e) = e$, and thus this LUT is mentioned only for its conceptual value.

[6] In our system, the true information sequence is $\{x_i\}$ which is the input to the LDPC encoder. However, from the local perspective of the CPM modulator and the SISO module, $\{u_i\}$ fills the role of the information sequence.

the beginning of the algorithm, the sole interface between the bit and symbol indexes takes place at Algorithm 4 Line 7 (which is trivial for $n_0 = 1$) and Algorithm 5 Lines 7–9 (which is more involved due to $n_0 = 2$).

Once the received bit-based L-values are grouped into symbol-based n_0-tuples, $\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})$, the middle portions of Algorithms 4 and 5 are entirely symbol based, i.e. the formulation of the metric increment, $\gamma_n(e)$, and the forward and backward recursions that generate $A_n(s)$ and $B_n(s)$ involve variables that are based exclusively on n. The forward recursion touches each ending state and performs an update over the edges that merge into that ending state. For this, we define the "ending set" as

$$\mathbf{e}(s^{\mathrm{E}}) \triangleq \{e : s^{\mathrm{E}}(e) = s^{\mathrm{E}}\} \tag{36}$$

which contains all edges such that the ending state of the edge is s^{E} (M edge indexes in total). Likewise, the backward recursion touches each starting state and performs an update over the M edges that merge into that starting state, using the "starting set"

$$\mathbf{e}(s^{\mathrm{S}}) \triangleq \{e : s^{\mathrm{S}}(e) = s^{\mathrm{S}}\} \tag{37}$$

(Extending these definitions to a time-varying trellis is straightforward.) The results of a single forward or backward time step, $\{A_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$ or $\{B_n(s)\}_{s=0}^{N_{\mathrm{S}}-1}$, respectively, represent an *unnormalized* log-based PMF. As discussed in connection with (68) in the Appendix, the absolute normalization of these PMFs (i.e. that they are made to sum to unity) is not necessary because of the subtraction in the completion step of the SISO module. As such, the normalization called for in Algorithms 4 and 5 is for the narrow purpose of *preventing numerical overflow* in a finite-precision implementation. This normalization can be as simple as determining the maximum value over the set, and then subtracting this value from all values in the set. In a floating-point implementation, normalization is unnecessary because the finite block length (N) is short enough that overflow is unlikely.

The final step in the SISO module is the completion step, which forms the extrinsic output L-values. This step makes use of $A_n(s)$ and $B_n(s)$ from the forward and backward recursions, and an *extrinsic* version of the metric increment. Similar to the LDPC decoder, the SISO algorithm can be formulated as extrinsic-by-inverse or extrinsic-by-exclusion, where we favor the latter due to the non-existence of a suitable inverse for addition with saturation in a fixed-point/integer implementation. For $n_0 = 1$ (Algorithm 4) the key step for extrinsic-by-exclusion is the use of $\chi_n(e)$ on Line 17 for the completion step that follows. This variable, $\chi_n(e)$, excludes the *a priori* input term $\boldsymbol{\lambda}_n(\mathbf{u};\mathrm{I})$ that is present in the earlier metric computations that use $\gamma_n(e)$ on Line 8. For $n_0 = 2$ (Algorithm 5), two separate extrinsic increments are formed, $\chi_n^{(0)}(e)$ and $\chi_n^{(1)}(e)$, where each takes a turn excluding one of the input values from the current time step (See Lines 19–20). Compare these with $\gamma_n(e)$ on Line 10 which has contributions from both input bits.

Each extrinsic output L-value value is formed by marginalizing over the appropriate sets of edges for $u = 0$ and $u = 1$. This is done once ($n_0 = 1$) each time step for Algorithm 4 and twice ($n_0 = 2$) each time step for Algorithm 5. Again, as was pointed out in connection with (68), this subtraction in the log domain is "self normalizing" and yields accurate L-values regardless of the normalization scale of the input arguments.

Similar to the LDPC decoder, a low-complexity version of the SISO module is obtained when we drop the "correction" term, $\mathrm{f}(a-b)$, within the $\max\star(a,b)$ definition in (49). This is the "max–log" SISO, where the simple $\max(a,b)$ function replaces all instances of the $\max\star(a,b)$ function in Algorithms 4 and 5. This results in RRs being exchanged between the SISO module and the LDPC decoder, instead of true L-values. In the last line of Algorithms 4 and 5, we recommend $\kappa = 3/4$ when operating in the "max–log" configuration to attenuate the RRs due to the "over estimating" phenomenon. In the optimal $\max\star(a,b)$ configuration, this step is not necessary due to $\kappa = 1$.

6.2 | A-type Parallelization

The SISO module can be envisioned as having a *time* dimension and a *state/edge* dimension. The three main steps of the algorithm (i.e. forward, backward, and completion) each consist of an outer "for" loop that steps through time, and an inner "for" loop that steps through the states/edges. We now outline two distinct parallelization strategies that can be applied to the SISO module. The first offers a way to parallelize the time dimension, and the second offers a way to parallelize the state/edge dimension.

In [23], several possible parallelization schemes are presented for the Viterbi algorithm, which is similar to the SISO module. We select the one described as *algorithmic* parallelization and refer to it as *A-type*. The basic idea is to divide the received code word into P_{A} segments (that overlap to a small degree), and then process these segments in parallel; thus achieving a parallelization factor of approximately P_{A}.

Our received code word has a length of N bits and we allow a SISO warm-up period of N_{WU} bits on either side of this word, for a total length of $N + 2N_{\mathrm{WU}}$ bits. In terms of CPM *symbols*, we have $N' = N/n_0$, $N'_{\mathrm{WU}} = N_{\mathrm{WU}}/n_0$, and a total length of $N' + 2N'_{\mathrm{WU}}$ symbols. Figure 11 depicts the length-N code word at the very top with a shaded warm-up interval on either side. To form the overlapping segments, we divide the code word portion by P_{A}, i.e. $N_{\mathrm{Par}} = N/P_{\mathrm{A}}$ and $N'_{\mathrm{Par}} = N'/P_{\mathrm{A}}$, and allow each segment to have the full warm-up period of N_{WU} bits (N'_{WU} symbols). The symbol and bit indexes that comprise the "base" segment are, respectively,

$$\begin{aligned} -N'_{\mathrm{WU}} \le n_{\mathrm{Par}} &\le N'_{\mathrm{Par}} + N'_{\mathrm{WU}} - 1 \\ -N_{\mathrm{WU}} \le i_{\mathrm{Par}} &\le N_{\mathrm{Par}} + N_{\mathrm{WU}} - 1 \end{aligned} \tag{38}$$

and the indexes for the p-th segment are obtained with the offset

$$\begin{aligned} n_{\mathrm{Par}} + pN'_{\mathrm{Par}} \\ i_{\mathrm{Par}} + pN_{\mathrm{Par}} \end{aligned} \tag{39}$$

where $p \in \{0, 1, \ldots, P_{\mathrm{A}} - 1\}$. Figure 11 gives a graphical depiction of $P_{\mathrm{A}} = 4$ overlapping segments, each with its own warm-up intervals.

A "vectorized" version of the metric increment is formed by stacking values from each segment together as in

$$\boldsymbol{\gamma}_{n_{\mathrm{Par}}}(e) = \left[\gamma_{n_{\mathrm{Par}}}(e), \gamma_{n_{\mathrm{Par}}+N'_{\mathrm{Par}}}(e), \ldots, \gamma_{n_{\mathrm{Par}}+(P_{\mathrm{A}}-1)N'_{\mathrm{Par}}}(e)\right]^T \tag{40}$$

This is depicted in Figure 12. From there, the forward/backward/completion steps take place over the

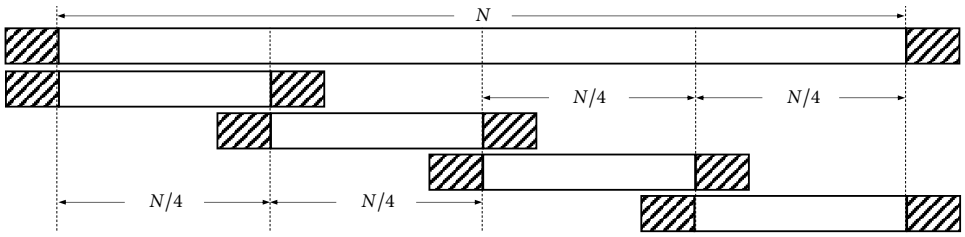


Figure 11 | Graphical depiction of A-type parallelization with $P_{\mathrm{A}} = 4$. The original length-N code word (shown at the top) has a warm up interval on either side (shaded region). The code word is divided into $P_{\mathrm{A}} = 4$ overlapping segments of length $N/4$, each with its own warm-up intervals.

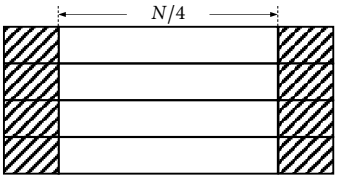


Figure 12 | The $P_{\mathrm{A}} = 4$ segments are "vectorized" by stacking them together. When these length-$N/4$ segments are processed in parallel, the number of time steps is reduced by a factor of approximately $P_{\mathrm{A}} = 4$.

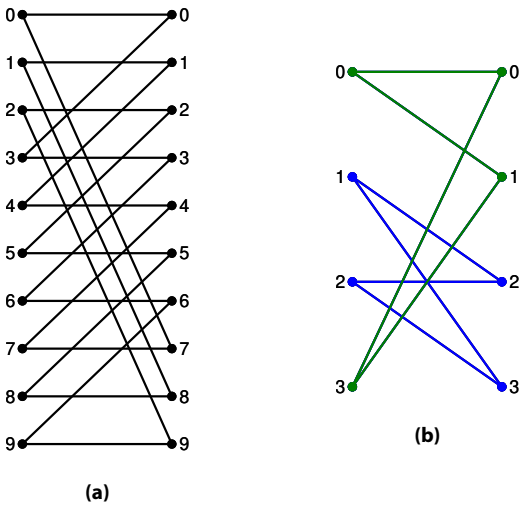


Figure 13 | Trellis diagrams for (a) ARTM0 ($L_{\mathrm{R}x} = 1$), and (b) ARTM1 ($L_{\mathrm{R}x} = 2$). The values of the starting state, s^{S}, are shown on the left-hand side of each trellis, and the values of the ending state, s^{E}, are shown on the right-hand side. The 4-state ARTM1 trellis can be decomposed into two separate 2-state "butterflies," which are shown in green and blue.

shortened index interval in (38) and generate "vectorized" quantities, $\mathbf{A}_{n_{\mathrm{Par}}}(s^{\mathrm{E}})$, $\mathbf{B}_{n_{\mathrm{Par}}}(s^{\mathrm{S}})$, and $\boldsymbol{\lambda}_{i_{\mathrm{Par}}}(u;\mathrm{O})$. The vector operator $\max\star(\mathbf{a},\mathbf{b})$ is applied on an element-by-element basis, which is a segment-by-segment basis in this context. After completion, the values in $\{\boldsymbol{\lambda}_{i_{\mathrm{Par}}}(u;\mathrm{O})\}_{i_{\mathrm{Par}}=0}^{n_{\mathrm{Par}}-1}$ can be "unstacked" to yield $\{\lambda_i(u;\mathrm{O})\}_{i=0}^{N-1}$.

The number of time steps required by the completion step is reduced by a factor of exactly P_{A}. The forward and backward steps each have a warm-up interval of the original length, N'_{WU}, and so these are reduced by a factor of

$$\frac{N + N_{\mathrm{WU}}}{N/P_{\mathrm{A}} + N_{\mathrm{WU}}} \tag{41}$$

which approaches P_{A} as N becomes very large relative to N_{WU}.

6.3 | B-type Parallelization

The state/edge processing can be parallelized by organizing the arguments of the forward/backward/completion steps in groups that can be processed together. One trellis structure that can be exploited is a "butterfly," which is defined as a sub-trellis of starting states, ending states, and connecting edges that is disjoint from the rest of the trellis. For example, the ARTM1 trellis in Figure 13 (b) can be decomposed into two separate 2-state butterflies (shown in green and blue), and the ARTM2 trellis in Figure 14 can be decomposed into 16 separate 4-state butterflies (two of which are shown in green and blue). We refer to this as *B-type* parallelization.

Because of the size of the ARTM2 trellis and the impact that parallelization can have on its throughput, we use this as a detailed case study. In our implementation, we format $A_n(\cdot)$, $B_n(\cdot)$, and $\gamma_n(\cdot)$ as large arrays of contiguous memory. We are able to stride through these arrays in either a *scalar* or *vector* fashion. For example, $A_n(s^{\mathrm{E}})$, $B_n(s^{\mathrm{S}})$, and $\gamma_n(e)$, use n as the time index, and within each time step, s^{E}, s^{S}, and e, respectively, index the scalar values. If we treat these arrays as a collection of length-P_{B} vectors, then $\mathbf{A}_n(v)$, $\mathbf{B}_n(v)$, and $\boldsymbol{\gamma}_n(v)$ use n as the time index and v as the generic index for each length-P_{B} vector within each time step.

Using the first butterfly in Figure 14 (a) as an example (shown in green), the forward recursion for $s^{\mathrm{E}} \in \{0,1,2,3\}$, which is vector index $v = 0$ using $P_{\mathrm{B}} = 4$, can be formulated as

$$\begin{aligned}\mathbf{A}_n(0) = \max\star\big\{&\mathrm{rep}_4(A_{n-1}(0))+\boldsymbol{\gamma}_n(0), \mathrm{rep}_4(A_{n-1}(7))+\boldsymbol{\gamma}_n(7),\\ &\mathrm{rep}_4(A_{n-1}(26))+\boldsymbol{\gamma}_n(26), \mathrm{rep}_4(A_{n-1}(45))+\boldsymbol{\gamma}_n(45)\big\}\end{aligned} \tag{42}$$

where $\mathbf{x} = \mathrm{rep}_{P_{\mathrm{B}}}(x)$ replicates the scalar quantity x a total of P_{B} times, forming a length-P_{B} vector $\mathbf{x}$. Similar formulas can be derived for the 15 remaining butterflies. Likewise, a similar formulation is available for the backward recursion. The end result is that $P_{\mathrm{B}} = 4$ states are updated in parallel and the forward/backward recursions require only 16 vector-based updates at each time step, instead of the original $N_{\mathrm{S}} = 64$ scalar-based updates.

As was implied in (42), our edge metrics are stored in sequential edge order as viewed from the left-hand side of the trellis, e.g. $\boldsymbol{\gamma}_n(0)$ accesses the first P_{B} edge metrics. For the completion step, we assemble the input arguments (one per edge) in this same edge order, with the caveat that we *interleave* length-P_{B} vectors pertaining to Algorithm 5 Lines 22 and 22, like so for the trellis

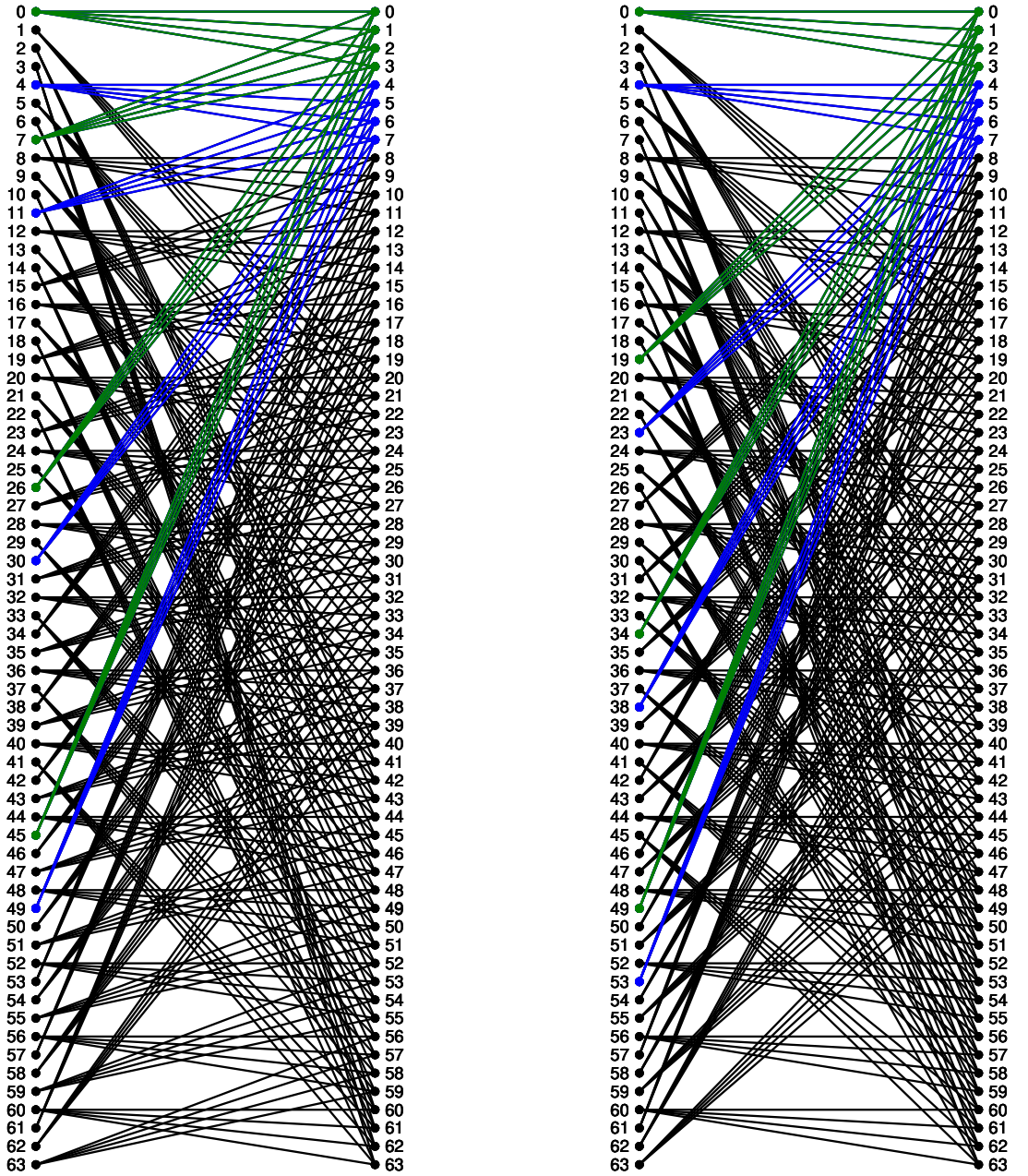


Figure 14 | Time-varying trellis diagrams for ARTM2 ($L_{Rx} = 2$) for (a) n-even, and (b) n-odd. The values of the starting state, s^S, are shown on the left-hand side of each trellis, and the values of the ending state, s^E, are shown on the right-hand side. For each case, the 64-state trellis can be decomposed into 16 separate 4-state "butterflies," the first two of which are shown in green and blue.

in Figure 14 (a):

$$\mathbf{M}(0) = \text{rep}_4(A_{n-1}(0)) + \boldsymbol{\chi}_n^{(0)}(0) + \mathbf{B}_n(0) \quad (43)$$

$$\mathbf{M}(1) = \text{rep}_4(A_{n-1}(0)) + \boldsymbol{\chi}_n^{(1)}(0) + \mathbf{B}_n(0) \quad (44)$$

$$\mathbf{M}(2) = \text{rep}_4(A_{n-1}(1)) + \boldsymbol{\chi}_n^{(0)}(1) + \mathbf{B}_n(5) \quad (45)$$

$$\mathbf{M}(3) = \text{rep}_4(A_{n-1}(1)) + \boldsymbol{\chi}_n^{(1)}(1) + \mathbf{B}_n(5) \quad (46)$$

$$\vdots$$

$$\mathbf{M}(126) = \text{rep}_4(A_{n-1}(63)) + \boldsymbol{\chi}_n^{(0)}(63) + \mathbf{B}_n(14) \quad (47)$$

$$\mathbf{M}(127) = \text{rep}_4(A_{n-1}(63)) + \boldsymbol{\chi}_n^{(1)}(63) + \mathbf{B}_n(14) \quad (48)$$

where $\mathbf{M}(\cdot)$ is an array with 128 length-4 vector terms, which has a scalar length of $2 \cdot N_E = 512$ terms. This array is divided in half, and the two halves are processed in a massive element-by-element $\max\star\{\cdot,\cdot\}$ vector operation to yield a length-256 scalar result. This halving and processing can be repeated until a length-8 scalar result is obtained (a total of six halving and processing steps). This marginalization yields 8 scalars that constitute two separate 4-ary log-domain extrinsic PMFs for U_n. Using (68), the first of these extrinsic 4-ary PMFs is marginalized into the L-value $\lambda_{2n}(u;\mathrm{O})$ (the step required by Algorithm 5 Line 22) and the second extrinsic 4-ary PMF is marginalized into the L-value $\lambda_{2n+1}(u;\mathrm{O})$ (the step required by Algorithm 5 Line 23).

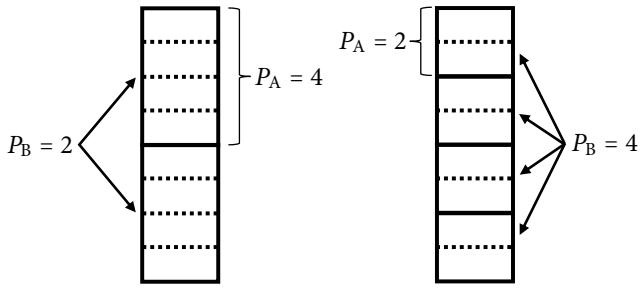


Figure 15 | Vector formats for 8-way combined parallelization. The format on the left is used for ARTM0 and ARTM1 ($P_A = 4$ and $P_B = 2$). The format on the right is used for ARTM2 ($P_A = 2$ and $P_B = 4$).

This detailed example shows how B-type parallelization by a factor of $P_B = 4$ has a natural application to ARTM2, which has $M = 4$. Similar techniques can be applied to parallelize ARTM0 and ARTM1 with $P_B = 2$ to match their $M = 2$ structure. Even though the ARTM0 trellis in Figure 14 (a) does not have a butterfly structure, it is still possible to group terms in ways that can be processed in parallel.

6.4 | Combined Parallelization

As we have just outlined, A-type parallelization is applied external to the SISO module with near-negligible impact on the algorithm architecture, while B-type parallelization is applied internal to the SISO module by specializing the processing steps to the particular trellis structure at hand.

Our prototyping platform offers a simple means of single instruction, multiple data (SIMD) parallel processing using 128-bit words. This is via the architecture known as streaming SIMD extensions (SSE), which is available on Intel processors and a similar architecture is available on Apple processors. By selecting a 16-bit fixed-point integer (i.e. "short int") scalar data type, we are able to achieve 8-way parallelization with a 128-bit wide vector. Our 8-way vector formats are shown in Figure 15.

For ARTM0 and ARTM1, A-type parallelization is applied externally by a factor of $P_A = 4$. Internal to the SISO module, this means that when "scalars" are handled (replicated, copied, etc) they are 64 bits wide, whereas vectors are 128 bits wide with $P_B = 2$ "scalars" each. Any mathematical operations (max⋆, addition, subtraction) are performed element-by-element on 16-bit elements in a 128-bit vector. When the SISO execution is finished, the $P_A = 4$ formatting is undone externally, and the results are sent to the LDPC decoder. Parallelization by a factor of 8 in the LDPC decoder is straightforward because all circulant sizes in Table 1 are multiples of 8.

For ARTM2, A-type parallelization is applied externally by a factor of $P_A = 2$. Internal to the SISO module, this means that when "scalars" are handled (replicated, copied, etc) they are 32 bits wide, whereas vectors are 128 bits wide with $P_B = 4$ "scalars" each. However, as before, any mathematical operations are performed element-by-element on 16-bit elements in a 128-bit vector.

The combined parallelization factor, F_P, is P_B times the term in (41), which is a value that is close to (but less than) 8 due to the warm-up overhead. Our observation is that a trellis of many states requires a longer warm-up interval to reach the "steady state." The values of N_{WU} we use in our system are 24, 8, and 64, respectively, for ARTM0, ARTM1, and ARTM2. In Table 5, we summarize the combined parallelization factor, F_P, we achieved for the smallest and largest code words in our study.

Table 5 | Parallelization Results for the smallest and largest code words.

Scheme	P_A	P_B	N_{WU}	N	F_P
ARTM0	4	2	24	1280	7.58
ARTM1	4	2	8	1280	7.85
ARTM2	2	4	64	1280	7.63
ARTM0	4	2	24	8192	7.93
ARTM1	4	2	8	8192	7.98
ARTM2	2	4	64	8192	7.94

7 | SIMULATION RESULTS

In this design study we have considered 3 CPM schemes (ARTM0, ARTM1, and ARTM2), 3 code rates ($R \in \{1/2, 2/3, 4/5)\}$), and 2 information block sizes ($K \in \{1024, 4096)\}$). This is a total of 18 distinct LDPC–CPM pairings.

Furthermore, the receiver implementation has the option of using the native code (always with $It_{loc} = 1$) or the punctured code (where $It_{loc} \leq It^*_{loc}$, and we always select $It_{loc} = It^*_{loc}$ herein). This increases the number of configurations to 36.

And finally, IRIG-106 [1] specifies the AR4JA–ARTM1-NR system for the six different code rates and block sizes in Table 1. This brings the number of configurations to a final tally of 42. AR4JA–ARTM1-NR is fundamentally different from the LDPC–CPM system in Figure 10, because the decoder for the AR4JA–ARTM1-NR codes does not feature a global iterative loop. Instead, the CPM SISO demodulator does a single demodulating pass on the received signal (as in Figure 10), after which the LDPC decoder iterates by itself, as presented in [6].

We now examine some performance characteristics of the various configurations.

7.1 | BER/FER Performance

We first quantify the bit error rate (BER) and frame error rate (FER). These simulation results are shown in Figures 16, 17, and 18 for the 6 LDPC codes belonging to, respectively, ARTM0, ARTM1, and ARTM2. Sub-figure (a) in each case displays results for the $It_{loc} = 1$/native option and Sub-figure (b) displays results for the $It_{loc} = It^*_{loc}$/punctured option. The BER/FER improves slightly when there are multiple local iterations per global iteration, i.e. $It_{loc} > 1$. In the case of the $K = 1024$, $R = 4/5$ codes, which have $It^*_{loc} = 1$, the BER/FER is slightly worse for the punctured option, but again this option has the advantage of being completely free of undetected errors.

The BER/FER curves for the longer block lengths ($K = 4096$) are quite steep, typically decreasing an order of magnitude for every SNR increment of 0.1 dB. The BER/FER slope is shallower for the shorter block lengths ($K = 1024$), requiring anywhere from 0.1–0.4 dB in SNR to decrease by an order of magnitude.

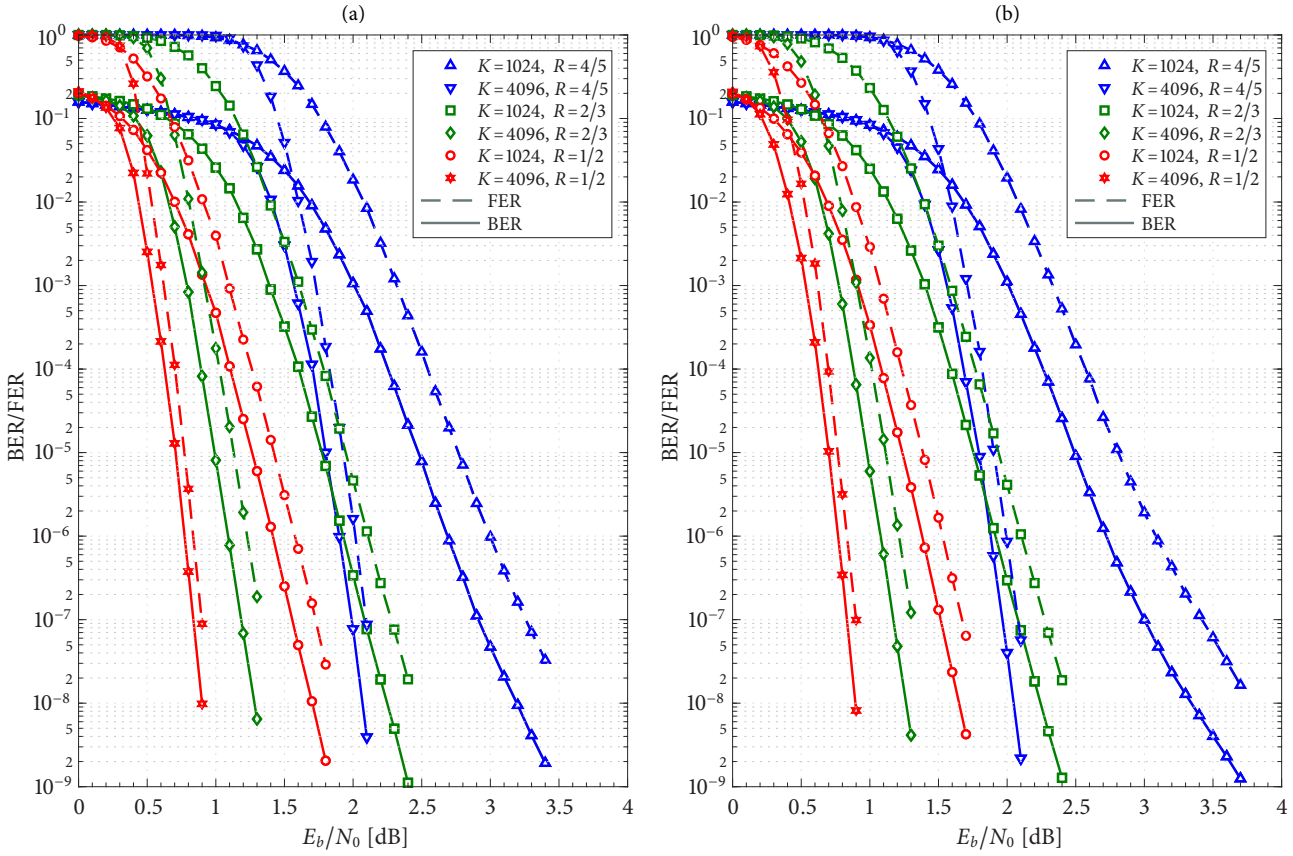


Figure 16 | BER/FER curves for the 6 codes in Table 1 paired with ARTM0 (PCM/FM) and (a) $It_{\text{loc}} = 1$/native option, and (b) $It_{\text{loc}} = It^*_{\text{loc}}$/punctured option.

Figure 17 (c) displays BER/FER curves for the 6 AR4JA–ARTM1-NR configurations. A careful comparison of the LDPC–CPM codes in Figure 17 (a) vs. the AR4JA–ARTM1-NR codes in Figure 17 (c) reveals that the AR4JA–ARTM1-NR codes remain the best option for ARTM1. This result was foreshadowed in [5] where codes designed for non-recursive MSK were shown to always outperform codes designed for recursive MSK.

Table 1 lists the coding gains of the 18 LDPC–CPM pairs, which range from 7.1 to 10.9 dB. The coding gains are denoted as Δ_0, Δ_1, and Δ_2 [in dB] for ARTM0, ARTM1, and ARTM2, respectively, and use as a reference the *uncoded* BER $= 10^{-8}$ crossing points for each CPM waveform, $E_b/N_0 \in \{10.8, 12.9, 13.3\}$ dB. The coding gains for the AR4JA–ARTM1-NR codes are denoted as Δ_{1A} and exceed those listed for Δ_1 by about 0.4 dB on average. For all modulation types, the coding gains are comparable across the information block sizes and code rates, which validates the consistency of the LDPC–CPM design approach presented in [7]. The BER extends below 10^{-8} in all cases with no evidence of an error floor.

The simulations in Figures 16–18 were conducted with a maximum limit on the number of global iterations of $It_{\max} = 512$, which is, of course, impractical. The next series of results will address this issue head on.

7.2 | Average Global Iterations Per Code Word

We now quantify the average number of global iterations needed in order to pass the parity check, which we denote as It_{avg}. For AR4JA–ARTM1-NR the iterative behavior is different, but there is only one type of iteration so It_{avg} is at least unambiguous. Figure 19 shows It_{avg} for all 36 LDPC–CPM design configurations, plus the 6 AR4JA–ARTM1-NR configurations. It_{avg} is demonstrated to decrease monotonically as E_b/N_0 increases. The BER and FER behave similarly (i.e. monotonically decreasing with increasing E_b/N_0), and so it can be said that It_{avg} and BER/FER move together.

The curves are grouped according to block size and code rate, i.e. the $K = 4096$, $R = 4/5$ configurations of ARTM0, ARTM1, ARTM2, and ARTM1-NR are plotted together in Figure 19 (a), and so forth. In each sub-figure, there is a side-by-side comparison of the native/punctured options, where we see that using $It_{\text{loc}} = It^*_{\text{loc}}$ (punctured option) can cut It_{avg} by as much as a half. The motivation for grouping by block size and code rate is that the values of It_{avg} are comparable across all three modulation types when the code rate, block size, *and* (surprisingly) BER/FER are held constant. For example, with the $K = 4096$, $R = 1/2$ codes [Figure 19 (e)] for ARTM0, ARTM1, and ARTM2, respectively, we observed $It_{\text{avg}} = \{13.0, 14.5, 12.5\}$ when operating with a FER on the order of 10^{-8} under the native option. At this same FER operating point for the $K = 1024$, $R = 4/5$ codes [Figure 19 (b)], we observed $It_{\text{avg}} = \{2.70, 2.75, 2.80\}$ for the respective modulations under the native option. In all cases, the AR4JA–ARTM1-NR configurations (orange curves) result in larger values of It_{avg}, and some cases many times the respective punctured option (red curves).

The impact of $It_{\text{loc}} = It^*_{\text{loc}}$ on It_{avg} is of particular interest for ARTM2 due to the complexity of the 64-state trellis used in the

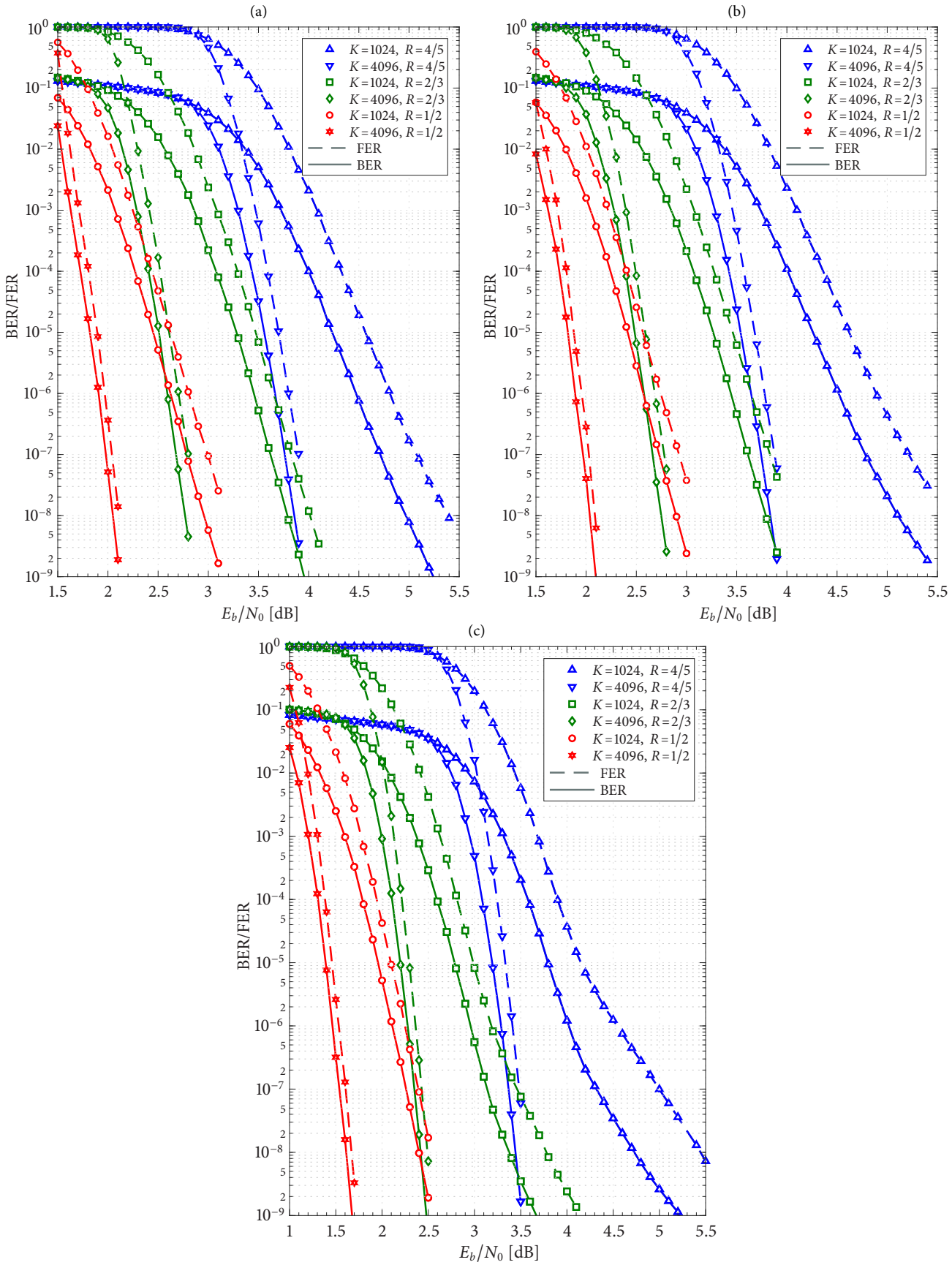


Figure 17 | BER/FER curves for the 6 codes in Table 1 paired with ARTM1 (SOQPSK-TG) and (a) $It_{\text{loc}} = 1$/native option, and (b) $It_{\text{loc}} = It^*_{\text{loc}}$/punctured option. The BER/FER curves in (c) belong to the existing AR4JA–SOQPSK-TG pairing in the IRIG-106 standard [1].

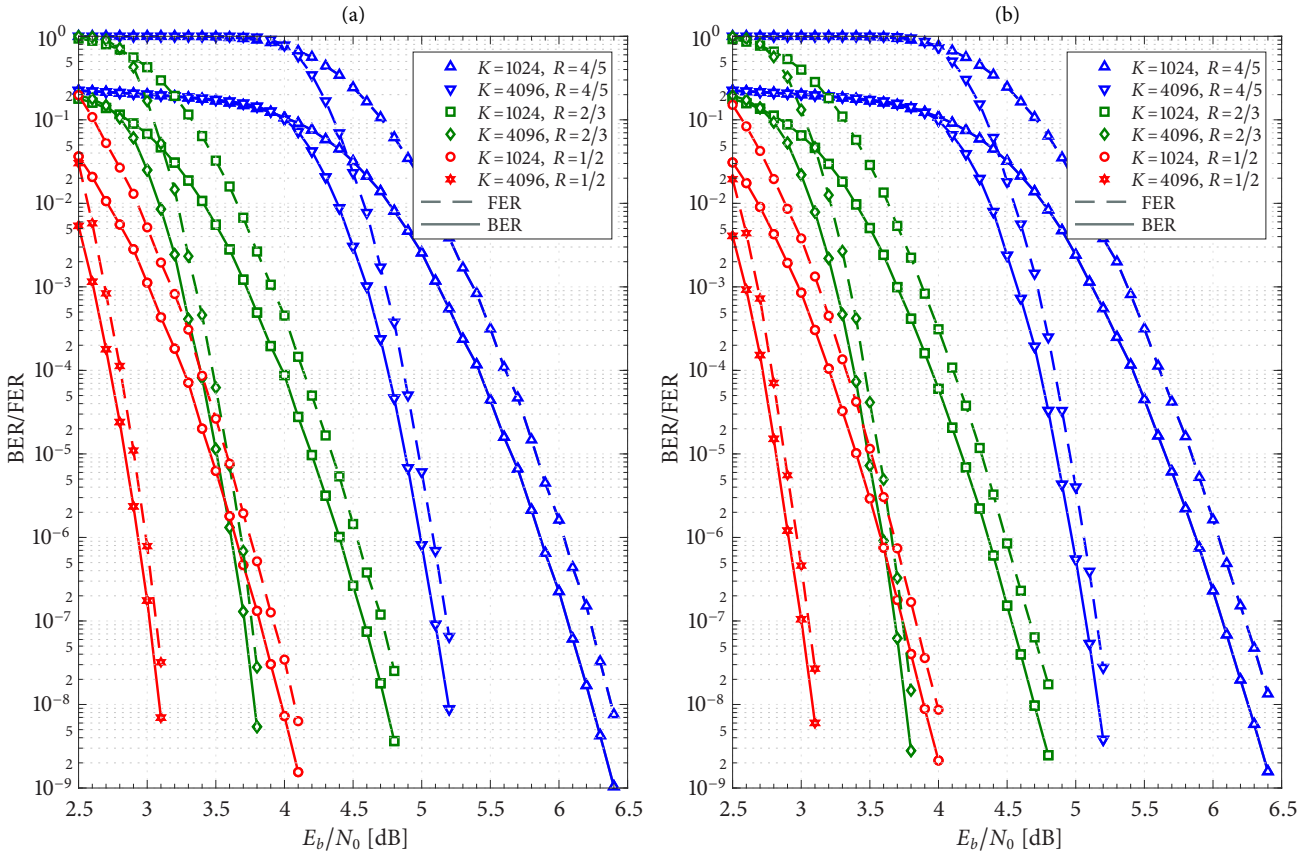


Figure 18 | BER/FER curves for the 6 codes in Table 1 paired with ARTM2 (ARTM CPM) and (a) It_{loc} = 1/native option, and (b) It_{loc} = It^{*}_{loc}/punctured option.

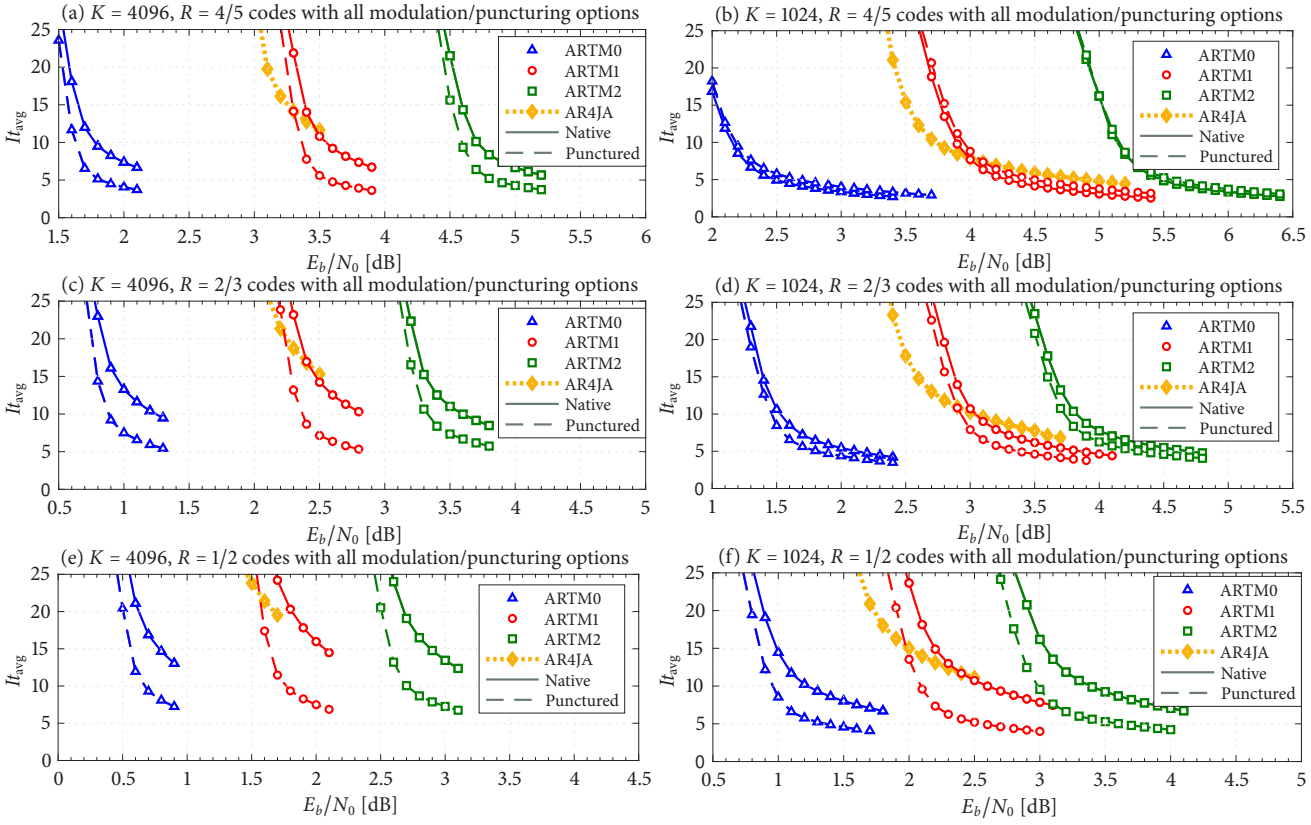


Figure 19 | It_{avg} for all 36 LDPC–CPM design configurations and the 6 AR4JA–ARTM1-NR configurations. The results are grouped according to the six combinations of block size (K) and code rate (R), as listed in the title of each sub-figure. Curves for the native and punctured options (see Table 1) are displayed in each sub-figure for easy comparison.

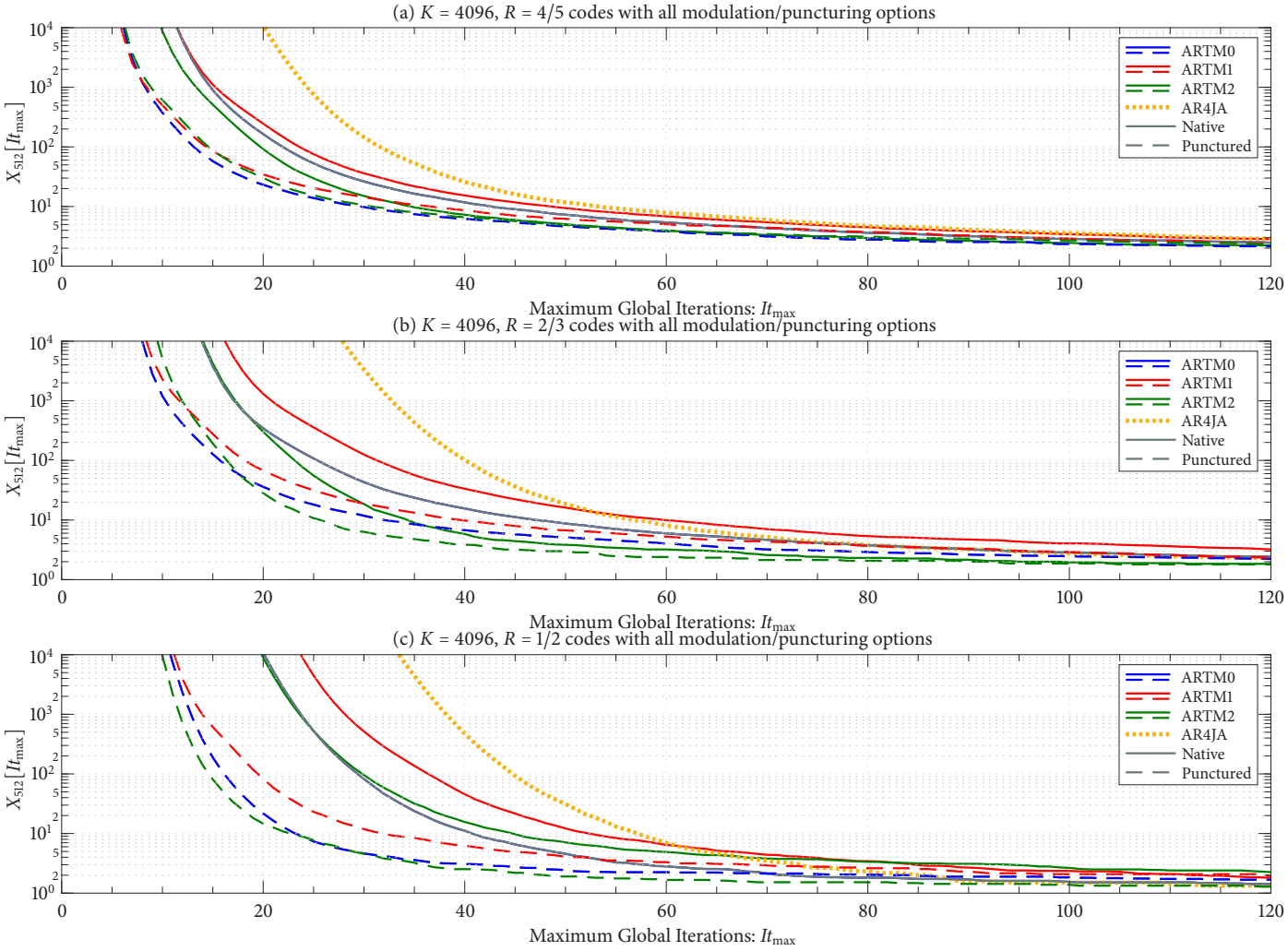


Figure 20 | FER Penalty Factor, $X_{512}[It_{\max}]$, for the $K = 4096$ codes with (a) $R = 4/5$, (b) $R = 2/3$, and (c) $R = 1/2$. Curves for the native/punctured options are displayed in each sub-figure for easy comparison.

global SISO update. Thus, $1 \le It_{\text{loc}} \le It^*_{\text{loc}}$ can be viewed as a means of addressing any complexity imbalance that might exist between the SISO and LDPC updates. In [7] the use of $It_{\text{loc}} = It^*_{\text{loc}}$ was shown to almost double the execution speed of the software decoder for ARTM2. The check-node splitting puncturing technique does introduce a slight "drag" on the global iterations because the messages must propagate through the punctured (non-transmitted) variable node in the decoder. This increases It_{avg} by about 0.5 iterations over what it would be if the native code were used with $It_{\text{loc}} = It^*_{\text{loc}}$ on It_{avg} (a pairing that is not recommended).

To sum up, the primary benefits of the punctured option are a (possible) speed-up in execution, coupled with the complete *elimination* of undetected errors. If only $It_{\text{loc}} = 1$ is of interest, it is worth pointing out that Table 1 still recommends $It^*_{\text{loc}} = 1$/puncturing for the case of the $K = 1024$, $R = 4/5$ codes. This is due to their small value of $d^*_{\min}$, which may still lead to infrequent undetected errors if puncturing is not used.

7.3 | Maximum Global Iterations Per Code Word

We now turn to an important practical question: What is the performance penalty if a hardware implementation is unable to deliver a maximum number of global iterations of $It_{\max} = 512$? This question relates to the "peak to average" iteration problem for LDPC codes and was explored in greater detail in our companion paper [8], along with a real-time decoder architecture that introduces a design tradeoff where a large value of $It_{\max}$ can be achieved at the expense of decoder latency.

The analysis in [8] showed that our (impractical) simulations using $It_{\max} = 512$ can be used as a reference in answering the "what if" question regarding the performance of the system in Figure 10 for any value of $It_{\max} \le 512$. The key is simply to maintain a *histogram* that counts the number of iterations required to pass the parity check for each code word simulated. We have already used this histogram data to plot It_{avg} in Figure 19. The terminal value in the length-$(512 + 1)$ histogram, the *endpoint*, counts the number of instances where the decoder "timed out" (i.e. $It_{\max} = 512$ was reached) and a frame error occurred. To answer the "what if" question for a smaller value of $It_{\max}$, we simply shorten the histogram to the desired length of $It_{\max} + 1$, where the values in the reference histogram that exceed our new $It_{\max}$ are lumped into the *shortened endpoint*. This is because these instances *would have been* frame errors

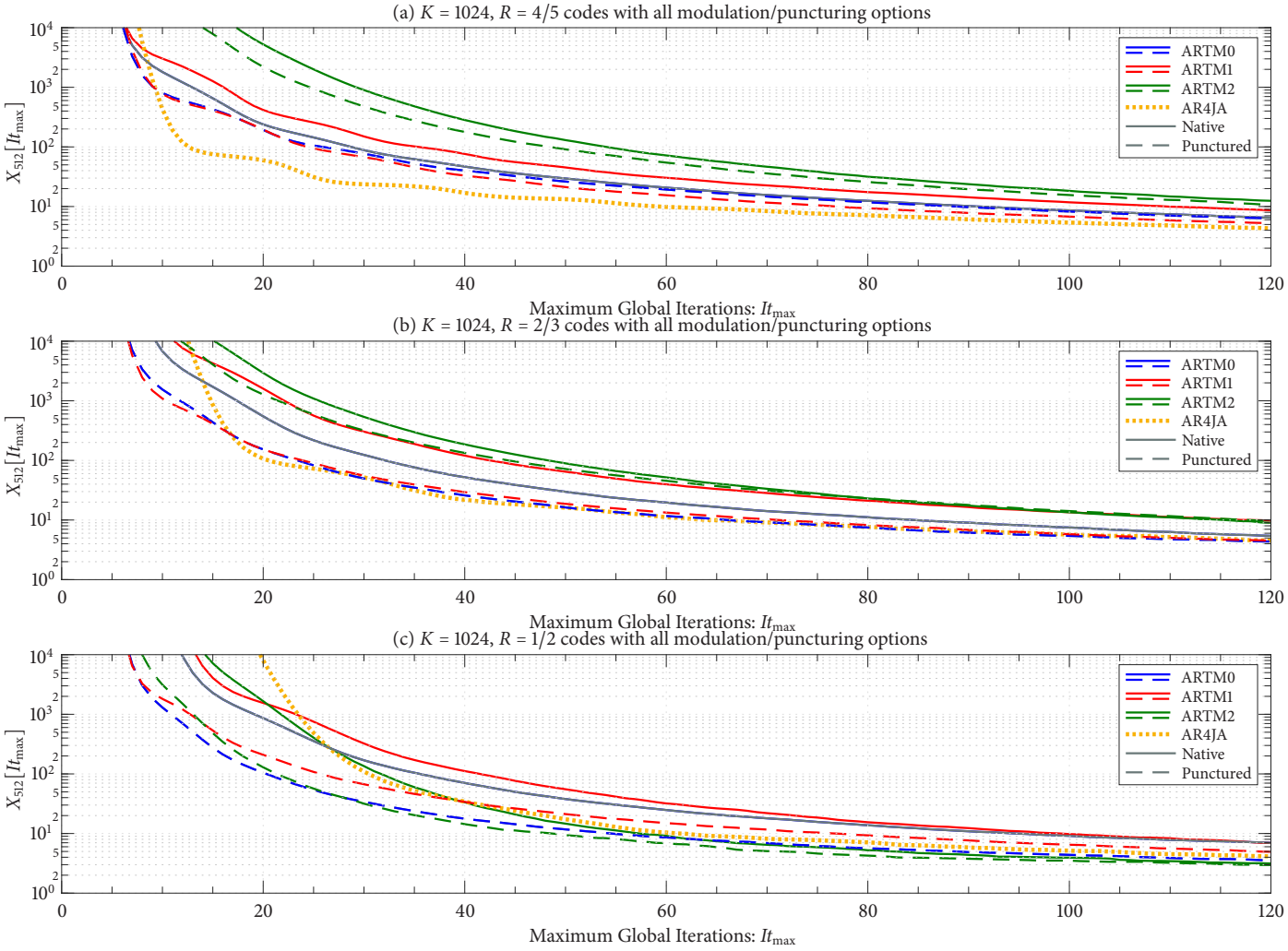


Figure 21 | FER Penalty Factor, $X_{512}[It_{\max}]$, for the $K = 1024$ codes with (a) $R = 4/5$, (b) $R = 2/3$, and (c) $R = 1/2$. Curves for the native/punctured options are displayed in each sub-figure for easy comparison.

for the smaller value of $It_{\max}$. Thus, the growth of the *shortened* endpoint relative to the *reference* endpoint corresponds exactly to the FER increase (or *penalty factor*) relative to the reference FER. We denote the FER penalty factor as [8] $X_{512}[It_{\max}]$, where $1 \le It_{\max} \le 512$ is now treated as the independent variable or as a design parameter.

Figure 20 shows $X_{512}[It_{\max}]$ as a function of $It_{\max}$ for the $K = 4096$ configurations, and Figure 21 does likewise for the $K = 1024$ configurations; all 42 configurations are represented in Figures 20 and 21, including the AR4JA–ARTM1-NR cases. The penalty factor varies as FER diminishes with increasing E_b/N_0, therefore it must be stated that the data in Figures 20 and 21 belongs to a reference FER operating on the order of 10^{-8}, i.e. *low FER*.

For the longer block lengths (Figure 20), a value of $It_{\max} < 25$ is sufficient to bring the curves for all LDPC–CPM configurations into the picture. Although the range of penalty factors displayed is quite large, maxing out at $X_{512}[It_{\max}] = 10^4$ (four orders of magnitude), the FER curves in Figures 16–18 are quite steep for the longer block lengths. Therefore, this translates to a *SNR penalty* of around 0.5 dB.

A similar informal analysis can be undertaken for the shorter block lengths (Figure 21). A modest value of $It_{\max} < 20$ brings the penalty factor curves into the picture for all LDPC–CPM configurations. However, because the FER slope in Figures 16–18 is shallower for these cases, the *SNR penalties* will be larger: approximately 0.5–1.5 dB. Thus, larger values of $It_{\max}$ may be of interest for the shorter block lengths, which likely can be supported in hardware.

In all cases, puncturing/It^*_{loc} reduces the value of $It_{\max}$ needed to achieve a given penalty factor $X_{512}[It_{\max}]$, just as it reduces It_{avg} in Figure 19.

And finally, the AR4JA–ARTM1-NR system also suffers from the "peak to average" iteration problem. Although the nature of the iterative loop is different, the AR4JA–ARTM1-NR configurations generally require greater values of $It_{\max}$ than the LDPC–CPM configurations in order to achieve a given penalty factor $X_{512}[It_{\max}]$.

7.4 | Comparison with SCCC–CPM

Serially concatenated convolutional codes (SCCCs) were initially considered in this study. This was primarily due to their strong performance for the $K = 4096$, $R = 2/3$ pairing with

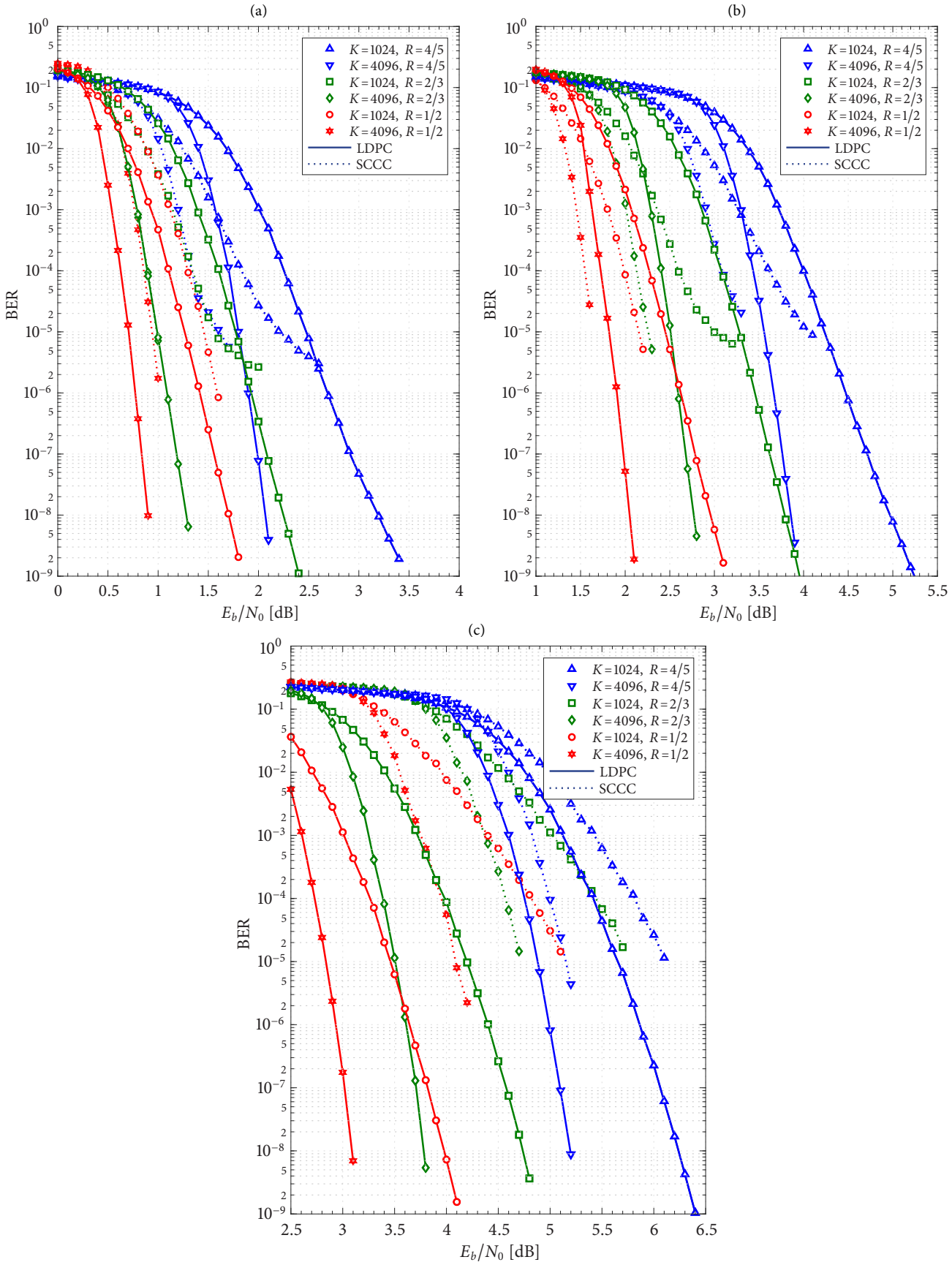


Figure 22 | BER curves comparing serially concatenated convolutional codes (SCCCs) with the proposed LDPC codes for (a) ARTM0 (PCM/FM), (b) ARTM1 (SOQPSK-TG), and (c) ARTM2 (ARTM CPM).

ARTM1 that was explored over a decade ago in [6]. The SCCC–CPM system is essentially the same as Figures 1 and 10 where the LDPC encoder (or decoder) is replaced by a convolutional encoder (or decoder). We explored the use of the same $R = 1/2$, 4-state (5,7) convolutional code that was studied in [6], where puncturing is used to achieve the desired rate (R).

The SCCC–CPM approach was explored for all six combinations of information block size and code rate in Table 1, paired with all three modulation types of interest herein (ARTM0, ARTM1, and ARTM2). The BER/FER curves for these 18 SCCC–CPM schemes are shown in Figure 22, where they are directly compared with the 18 LDPC–CPM schemes we have described herein. Sub-figures (a), (b), and (c) pertain to ARTM0, ARTM1, and ARTM2, respectively. The primary drawback with the SCCC–CPM schemes is the error floor that is clearly visible for the $R = 2/3$ and $R = 4/5$ cases with $K = 1024$. This starts to manifest itself at BER $\approx 10^{-5}$. There are some cases where the SCCC–CPM option is superior to the LDPC–CPM option, e.g. $R = 1/2$ for ARTM1. However, for ARTM2, the SCCC–CPM option is significantly worse in all cases. Different from LDPC codes, a step-by-step design procedure to address these deficiencies is lacking for SCCC–CPM. Based on these results, SCCC–CPM was abandoned at an early stage in this study.

8 | CONCLUSION

This report gives a detailed description of a capacity approaching LDPC–CPM system for use in aeronautical telemetry. This system is particularly applicable to ARTM0 (PCM/FM) and ARTM2 (ARTM CPM), although it was also studied for ARTM1 (SOQPSK-TG). A set of six distinct LDPC codes were designed for (matched to) each of these CPM schemes, which is documented in greater detail in a companion paper [7]. The proposed LDPC–CPM codes were shown to achieve coding gains on the order of the existing AR4JA option in IRIG-106. Numerical results quantifying the iterative behavior of the system were also given, with expanded results available in another companion paper [8]. In addition to describing these codes, this report focused on all aspects of the receiver implementation, including the LDPC decoder, the CPM SISO module, and parallelization strategies for both.

The first half of the Appendix that follows contains additional background information on log-based soft processing. The second half of the Appendix gives a comprehensive listing of the random interleavers, parity check matrixes, and generator matrixes referenced above.

APPENDIX
LOG-DOMAIN METRIC FUNCTIONS

The SISO module and the LDPC decoder in Figure 10 operate on input/output data that can be interpreted as belonging to the log domain. This section defines the exact and approximate formulation of the max$\star$ operator, which is central to the operation of the SISO module; and the min$\star$ operator, which is central to the operation of the LDPC decoder.

We define the "log-based addition," or "max star" operator as

$$\max\star(a,b) \triangleq \ln(e^a + e^b) = \max(a,b) + \mathrm{f}(a-b) \tag{49}$$

where $\max(a,b)$ is the simple maximum value between a and b and the "correction term" is

$$\mathrm{f}(x) = \ln(1 + e^{-|x|}) \tag{50}$$

This operator is commutative: $\max\star(a,b) = \max\star(b,a)$; associative: $\max\star(a,\max\star(b,c)) = \max\star(\max\star(a,b),c)$; and its identity value is $-\infty$: $\max\star(-\infty,a) = a$. Because of these properties, it can also be defined without ambiguity as operating on more than two terms, i.e.

$$\max_{i\in\mathcal{I}}\star\{a_i\} \tag{51}$$

When this computation is executed sequentially, i.e. in a loop, the "running result" is initialized with the identity value and then "accumulated" sequentially with each input value, a_i.

Several reduced-complexity strategies are available for the correction term, $\mathrm{f}(x)$. We will introduce these simplifications along with a simultaneous discussion on scaling in order to facilitate integer (fixed point) implementation. Let $c = \max\star(a,b)$, let S be a scale factor, and let $C = Sc$, $A = Sa$, and $B = Sb$. It follows that $C = S\max\star(a,b) = \max(A,B) + S\cdot\mathrm{f}((A-B)/S)$.

The first reduced-complexity strategy is to discard the correction term and use only the simple max in the place of max$\star$. In addition to its computational simplicity, it is also insensitive to scale, including any fixed-point scale (i.e. S) and also the scale factor $\sqrt{E_s}L_c/2$ in (21). Therefore, the "simple max" strategy avoids the requirement of estimating the signal energy (E_s) and the noise power spectral density (N_0).

The second reduced-complexity strategy is to pre-compute the values of $S\cdot\mathrm{f}(X/S)$ for integer values $X = Sx$, round the resulting values to the nearest integers, and store these in a LUT indexed by X. For example, for $S = 8$, a length-22 LUT is all that is required to store the quantized values of $S\cdot\mathrm{f}(X/S)$.

The third reduced-complexity strategy is to approximate $S\cdot\mathrm{f}(X/S)$ with the piecewise function [2, Fig. 19]

$$\mathrm{f}_S(X) = \begin{cases} -0.375|X| + 0.6875S & |X| \le 1.0S \\ -0.1875|X| + 0.5S & 1.0S < |X| < 2.625S \\ 0 & \text{otherwise} \end{cases} \tag{52}$$

which is easy to implement using combinatorial logic and integer input/output. When the arguments to max$\star$ are quantized, and/or when $S\cdot\mathrm{f}(X/S)$ is quantized or approximated with (52), then max$\star$ remains commutative but is no longer associative, and so the order of operations in (51) can be significant.

We define the "log-hyperbolic-tangent," or "min star" operator as

$$\min\star(a,b) \triangleq 2\tanh^{-1}\left(\tanh\frac{a}{2}\cdot\tanh\frac{b}{2}\right) \tag{53}$$

$$= \mathrm{sgn}(a)\mathrm{sgn}(b)\big[\min(|a|,|b|) + \mathrm{f}(|a|+|b|) - \mathrm{f}(|a|-|b|)\big] \tag{54}$$

where $\min(a,b)$ is the simple minimum value between a and b and $\mathrm{sgn}(\cdot)$ returns +1 when its argument is non-negative and −1 otherwise. The formulation in (53) has severe practical limitations in a hardware implementation and is thus provided only as a definition and *should never be used*. As with max$\star$, min$\star$ is commutative and associative, so

$$\min_{i\in\mathcal{I}}\star\{a_i\} \tag{55}$$

can be defined without ambiguity, where the identity value is $+\infty$. The above discussion on reduced-complexity and quantization strategies for (50) are equally applicable for the two "correction terms" in (54). Notably, the "simple min" strategy is insensitive to the scaling of the inputs. Also, quantization/approximation wipes out the strict associativity of $\min\star$.

APPENDIX
LOG-DOMAIN PROBABILITY MASS FUNCTIONS (PMFs)

Consider a random variable u that is drawn from a binary alphabet, $u \in \{0,1\}$. The probability mass function (PMF) of u, $P(u)$, consists of two non-negative values $P(u=0)$ and $P(u=1)$ that satisfy the normalization constraint $P(u=0)+P(u=1)=1$, which means that the PMF is fully understood if one value or the other is given. As such, a convenient way of expressing a binary PMF is with a single value called an *L-value*, which we define as

$$\lambda(u) = \ln \frac{P(u=0)}{P(u=1)} = p(u=0) - p(u=1) \tag{56}$$

where the lower-case $p(u)$ denotes the "log-domain" PMF that is obtained by taking the natural log of each element of the original "linear-domain" PMF, $P(u)$. We define the L-value with $P(u=0)$ in the numerator because it avoids frequent negation operations when using the $\min\star$ operator (54) in the LDPC decoder. Although we have followed the universal convention that a linear-domain PMF must be normalized so that it sums to unity, the L-value would arrive at the same value even if an arbitrary normalization were chosen (e.g., "the PMF must sum to the positive value C"), because such a scale factor simply cancels when the ratio (or subtraction) is taken in (56). The L-value notation includes u, but u does not index anything because the L-value is a single value, and so it serves only to designate the binary random variable to which the L-value belongs (for the PMF, u is needed as an index for the multiple terms in the PMF). In other contexts, we will add a sequence index, and/or a designation to indicate whether the L-value is an *a priori* (designated as "I" for input) or *a posteriori* (designated as "O" for output). For example, $\lambda_i(u;\mathrm{I})$ is the *a priori* L-value for u_i, and the corresponding log-domain PMF is $\{p_i(u;\mathrm{I})\}_{u=0}^{1}$

As (56) indicates, an L-value can be obtained if the PMF is known (regardless of the log or linear domain of the PMF). The reverse is also true, i.e. if an L-value is known, then the log-domain PMF can be "tentatively" expressed as

$$p(u=0) = +\lambda(u)/2, \qquad p(u=1) = -\lambda(u)/2 \tag{57}$$

if the normalization does not matter, because such a formulation preserves the "spread" (or difference) between the two PMF points in the log domain (or the ratio in the linear domain). This tentative "unpacking" of the L-value works just fine in the SISO module because there are other points in the SISO algorithm where normalization can be performed if needed. If the normalization matters, then the "final" log-domain PMF is obtained as

$$p(u=0) = +\lambda(u)/2 - c, \qquad p(u=1) = -\lambda(u)/2 - c \tag{58}$$

which sums to unity when expressed in the linear domain, where the normalization term $c = \max\star(+\lambda(u)/2, -\lambda(u)/2)$ makes use of the $\max\star$ operator defined in (49). Inserting (58) back into (56) emphasizes the fact that L-values are "self normalized."

Although an L-value is merely a single number, it has great intuitive value. Its sign, or $\mathrm{sgn}(\lambda(u))$, serves as a "hard decision" (maximum likelihood decision) on the underlying random variable u. Its magnitude, or $|\lambda(u)|$, characterizes the reliability, or confidence, of this decision. Consider the variable

$$\mu(u) = \alpha\lambda(u) \tag{59}$$

where α is a positive and perhaps *unknown* constant. The sign, $\mathrm{sgn}(\mu(u))$, or "hard decision," of this variable is identical to that of the L-value it is derived from. Although α may be unknown, if the underlying random variable u is part of a larger sequence, then the magnitude $|\mu(u)|$ remains useful in identifying the relative reliability of the hard decisions over this larger sequence. As such, we refer to (59) as a *relative reliability* (RR).

A fundamental difference between L-values and RRs is that L-values are directly linked to PMFs, and this link is broken for RRs because of the scale factor α and the nonlinearity of converting to and from back and forth from the log domain. We have already discussed the insensitivity to scale of the "simple max" and "simple min" approximations to (49) and (54), respectively. If the precise scale factor $\sqrt{E_s}L_c/2$ in (21) is not known, then the demodulator/decoder can still operate effectively using RRs by employing the "simple max" and "simple min" approximations. Such a reduced-complexity architecture is attractive due to its streamlined metric computations *and* its avoidance of estimators for the signal energy (E_s) and noise power spectral density (N_0). The performance penalty for such an architecture is surprisingly modest, on the order 1.0 dB or less depending on the modulation scheme, code length, and block rate.

We will no longer make a major distinction between L-values and RRs, because in terms of our development, they are more similar than they are different. Thus, $\lambda(u)$ and $\mu(u)$ are used interchangeably in almost every instance in the demodulator/decoder. The only exception to this rule is when the "correction terms" are used in (49) and (54), which require that the precise scale factor $\sqrt{E_s}L_c/2$ in (21) is both estimated and applied prior to decoding. As such, Figure 10 is labeled exclusively using $\lambda(u)$. A version of Figure 10 could be drawn with all instances of $\lambda(u)$ being replaced by $\mu(u)$, with the understanding that the "simple max" and "simple min" approximations are used within the SISO module and LDPC decoder, respectively.

We now consider the case of a random variable U that is drawn from an M-ary alphabet, $U \in \{0,1,\ldots,M-1\}$, where $M = 2^{n_0}$. There is a one-to-one relationship between the integer value U and a binary n_0-tuple (vector) $\mathbf{u} = [u_0,\ldots,u_{n_0-1}]^T$. This $U \leftrightarrow \mathbf{u}$ relationship is given by

$$U = U(\mathbf{u}) \triangleq \sum_{i=0}^{n_0-1} 2^{n_0-1-i} u_i \tag{60}$$

which places u_0 in the role of most-significant bit (MSB) and u_{n_0-1} in the role of least-significant bit (LSB). This relationship also defines a set of "inverse" functions, $u_i(U)$, $0 \le i \le n_0-1$, that return a binary value $u_i(\cdot) \in \{0,1\}$ for the i-th bit position for given integer value U.

The PMF of U in the linear domain is $\{P(U)\}_{U=0}^{M-1}$ and is normalized so that it sums to unity. The PMF of the binary

element u_i, $0 \le i \le n_0 - 1$, denoted as $P_i(u)$, can be obtained by marginalizing $P(U)$ like so:

$$P_i(u) = \sum_{U:u_i(U)=u} P(U), \quad 0 \le i \le n_0 - 1,\ u \in \{0,1\} \tag{61}$$

where $\{U : u_i(U) = u\}$ denotes the set of U values such that $u_i(U) = u$. Similarly, the PMF of U can be obtained from the binary PMFs belonging to $\mathbf{u}$ via the operation

$$P(U) = \prod_{i=0}^{n_0-1} P_i(u_i(U)), \quad U \in \{0,1,\ldots,M-1\} \tag{62}$$

As was done previously, the log-domain PMFs use lower-case notation, and so $\{p(U)\}_{U=0}^{M-1}$ denotes the log-domain PMF of U, and $p_i(u)$, $0 \le i \le n_0 - 1$, denotes the log-domain PMF of the binary element u_i. The unnormalized (or "tentative," to use our previous terminology) log-domain counterparts to (61) and (62) are, respectively,

$$p_i(u) = \max_{U:u_i(U)=u}{}^{\star} \{p(U)\}, \quad 0 \le i \le n_0 - 1,\ u \in \{0,1\} \tag{63}$$

and

$$p(U) = \sum_{i=0}^{n_0-1} p_i(u_i(U)), \qquad U \in \{0,1,\ldots,M-1\} \tag{64}$$

$$= -\sum_{i=0}^{n_0-1} a_2(u_i(U))\lambda_i(u)/2, \tag{65}$$

$$= -[\boldsymbol{\lambda}(\mathbf{u})]^T a_2(\mathbf{u})/2 \tag{66}$$

where $a_2(\cdot)$ is the binary antipodal function from (3), which operates element-by-element on its vector input and returns an output of the same size, and $\boldsymbol{\lambda}(\mathbf{u}) = [\lambda_0,\ldots,\lambda_{n_0-1}]^T$ is an n_0-tuple (vector) of L-values belonging to $\mathbf{u}$. In the event that normalized versions of these log-domain PMFs are desired, the max$\star$ can be taken of the entire unnormalized PMF and the result can then be subtracted from each PMF value, as shown in (58). The L-value of the binary element u_i is

$$\lambda_i(u) = p_i(u=0) - p_i(u=1) \tag{67}$$

$$= \max_{U:u_i(U)=0}{}^{\star} \{p(U)\} - \max_{U:u_i(U)=1}{}^{\star} \{p(U)\} \tag{68}$$

and, again, this operation "self normalizes" such that it is insensitive to the normalization status of the log-domain PMFs that feed into it. The above relationships for M-ary PMFs are written in general terms and are thus valid for the special case of $M = 2$ (binary), where $U = u$ and previous equations were given.

We have developed notation that distinguishes between L-values, $\lambda(u)$, and RRs, $\mu(u)$, but we will not bother to do so for log-based PMFs and their "relative reliability" counterparts. Suffice it to say that when the log-domain scaling is unknown, we can use $\mu(u)$ in place of $\lambda(u)$ in (63) through (68) but we must also use max in place of max$\star$. The resulting log-domain "PMFs" are no longer exact in their relationship to probability (i.e. they should not be converted to the linear domain); however, as with $\mu(u)$, these "PMFs" retain much of their intuitive value. In fact, these are the very differences between the BCJR algorithm and the Viterbi algorithm (VA). Whether formulated in the linear or log domain, the original BCJR algorithm and the more-general SISO algorithm work with exact *a priori* PMFs and/or L-values as inputs, use sum–product operations or max$\star$–sum operations (depending on the domain), and produce exact *a posteriori* probabilities (APPs) as outputs. On the other hand, the VA, "max log" SISO, and soft output VA (SOVA) [24] work with log-domain inputs where the scaling is mostly irrelevant (aside from quantization considerations), use max–sum operations, and, in the case of the SISO and SOVA, produce soft outputs that are consistent with the (possibly unknown) scale of the inputs and these soft outputs are very useful in downstream processing.

REFERENCES

[1] Range Commanders Council Telemetry Group, Range Commanders Council, White Sands Missile Range, New Mexico, *IRIG Standard 106-2024: Telemetry Standards*, 2024. (Available on-line at https://www.trmc.osd.mil/wiki/display/publicRCC/106+Telemetry+Standards).

[2] D. Divsalar, S. Dolinar, C. R. Jones, and K. Andrews, "Capacity-approaching protograph codes," *IEEE J. Select. Areas Commun.*, vol. 27, pp. 876–888, Aug. 2009.

[3] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, "The development of turbo and LDPC codes for deep-space applications," *Proc. IEEE*, vol. 95, pp. 2142–2156, Nov. 2007.

[4] P. Galko and S. Pasupathy, "Linear receivers for correlatively coded MSK," *IEEE Trans. Commun.*, vol. 33, pp. 338–347, Apr. 1985.

[5] K. R. Narayanan, İ. Altunbas, and R. S. Narayanaswami, "Design of serial concatenated MSK schemes based on density evolution," *IEEE Trans. Commun.*, vol. 51, pp. 1283–1295, Aug. 2003.

[6] E. Perrins and M. Rice, "Reduced complexity detectors for multi-h CPM in aeronautical telemetry," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 43, pp. 286–300, Jan. 2007.

[7] E. Perrins, "Protomatrix-based LDPC codes for continuous phase modulation," *IEEE Trans. Commun. (in review, revised version submitted January 2025.*

[8] E. Perrins, "Real-time decoder architecture for LDPC–CPM," *Entropy*, vol. 27, no. 3, 2025.

[9] B. E. Rimoldi, "A decomposition approach to CPM," *IEEE Trans. Inform. Theory*, vol. 34, pp. 260–270, Mar. 1988.

[10] J. B. Anderson, T. Aulin, and C.-E. Sundberg, *Digital Phase Modulation*. New York: Plenum Press, 1986.

[11] P. Moqvist and T. Aulin, "Serially concatenated continuous phase modulation with iterative decoding," *IEEE Trans. Commun.*, vol. 49, pp. 1901–1915, Nov. 2001.

[12] R. Othman, A. Skrzypczak, and Y. Louët, "PAM decomposition of ternary CPM with duobinary encoding," *IEEE Trans. Commun.*, vol. 65, pp. 4274–4284, Oct. 2017.

[13] E. Perrins and M. Rice, "Unification of signal models for SOQPSK," in *Proc. Int. Telemetering Conf.*, (Glendale, AZ), Oct. 2018.

[14] D. Divsalar and F. Pollara, (1995, May), "Multiple turbo codes for deep-space communications," *Telecommunications and Data Acquisition Progress Report*, vol. [Online]. Available: http://tmo.jpl.nasa.gov/tmo/progress_report/42-121/121T.pdf.

[15] R. Smarandache and P. O. Vontobel, "Quasi-cyclic LDPC codes: Influence of proto- and Tanner-graph structure on minimum Hamming distance upper bounds," *IEEE Trans. Inform. Theory*, vol. 58, pp. 585–607, Feb. 2012.

[16] B. K. Butler and P. H. Siegel, "Bounds on the minimum distance of punctured quasi-cyclic LDPC codes," *IEEE Trans. Inform. Theory*, vol. 59, pp. 4584–4597, Jul. 2013.

[17] D. G. M. Mitchell, R. Smarandache, and D. J. Costello, "Quasi-cyclic LDPC codes based on pre-lifted protographs," *IEEE Trans. Inform. Theory*, vol. 60, pp. 5856–5874, Oct. 2014.

[18] T. Tian, C. R. Jones, J. Villasenor, and R. D. Wesel, "Selective avoidance of cycles in irregular LDPC code construction," *IEEE Trans. Commun.*, vol. 52, pp. 1242–1247, Aug. 2004.

[19] Consultive Committee for Space Data Systems (CCSDS), "Low density parity check codes for use in near-Earth and deep space applications (131.1-O-2 Orange Book)," Sep. 2007.

[20] E. Hosseini and E. Perrins, "Timing, carrier, and frame synchronization of burst-mode CPM," *IEEE Trans. Commun.*, vol. 61, pp. 5125–5138, Dec. 2013.

[21] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, pp. 284–287, Mar. 1974.

[22] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output APP module for iterative decoding of concatenated codes," *IEEE Commun. Lett.*, vol. 1, pp. 22–24, Jan. 1997.
[23] G. Fettweis and H. Meyr, "High-speed parallel Viterbi decoding: Algorithm and VLSI-architecture," *IEEE Commun. Mag.*, pp. 46–55, May 1991.
[24] M. P. C. Fossorier, F. Burkert, S. Lin, and J. Hagenauer, "On the equivalence between SOVA and max–log–MAP decodings," *IEEE Commun. Lett.*, vol. 2, pp. 137–139, May 1998.

APPENDIX
INTERLEAVING TABLES FOR THE $K = 1024$ ("1K") CASE

Interleaving Table for $K = 1024$, $R = 4/5$, $N = 1280$

The following list of numbers represents the interleaving table for the $K = 1024$, $R = 4/5$ case, where the length of the interleaving table is $N = 1280$. This "1k" interleaving table contains every index in the set $\{0, 1, \ldots, 1278, 1279\}$, in random order, with the constraint that no index is within $S = 26$ positions in either direction of its sequentially adjacent indexes. The interleaving table is indexed by i and is $\pi(i) =$

1072, 327, 112, 213, 413, 261, 156, 561, 361, 729, 973, 1182, 1136, 21, 815, 923, 761, 652, 500, 619, 73, 297, 1257, 877, 1107, 692, 1227, 1068, 843, 443, 1001, 388, 203, 131, 269, 576, 1034, 338, 1167, 240, 968, 24, 471, 906, 934, 653, 525, 604, 98, 301, 783, 743, 1260, 1098, 713, 69, 873, 820, 433, 389, 172, 125, 685, 1131, 569, 1014, 242, 204, 982, 1194, 907, 463, 1046, 1162, 11, 640, 42, 298, 600, 508, 343, 951, 1277, 850, 791, 1233, 879, 76, 370, 538, 754, 700, 129, 1099, 1128, 161, 1013, 397, 259, 214, 1198, 1042, 917, 430, 46, 1070, 1157, 306, 3, 466, 605, 819, 496, 568, 965, 864, 635, 88, 691, 356, 662, 731, 143, 1003, 1123, 529, 260, 1250, 1216, 181, 925, 390, 50, 893, 1037, 429, 17, 1089, 212, 1172, 302, 832, 563, 470, 593, 694, 329, 97, 793, 665, 758, 971, 1145, 359, 630, 1118, 248, 1237, 510, 861, 173, 275, 64, 136, 393, 1010, 728, 1086, 915, 1278, 219, 480, 698, 453, 316, 1202, 10, 785, 545, 668, 954, 1160, 423, 366, 1116, 92, 518, 1039, 589, 1245, 634, 869, 153, 120, 1009, 262, 53, 826, 1272, 220, 755, 926, 455, 706, 1087, 184, 25, 799, 897, 1208, 667, 323, 422, 375, 294, 1165, 586, 1058, 528, 953, 133, 1008, 866, 263, 67, 618, 1135, 1259, 759, 481, 452, 722, 171, 33, 817, 210, 1197, 647, 99, 788, 1093, 674, 1225, 358, 412, 537, 928, 137, 313, 580, 273, 958, 616, 1040, 875, 990, 1263, 1148, 245, 462, 60, 740, 704, 196, 168, 30, 493, 796, 677, 1187, 1119, 106, 420, 1071, 1223, 836, 557, 523, 962, 637, 280, 322, 1011, 140, 603, 372, 450, 863, 757, 709, 58, 252, 1151, 485, 901, 807, 1038, 4, 206, 416, 1186, 86, 673, 1269, 1121, 957, 633, 1219, 291, 341, 995, 565, 834, 385, 532, 592, 727, 766, 874, 257, 157, 502, 795, 1094, 19, 185, 473, 1173, 1036, 116, 1273, 664, 223, 946, 80, 292, 349, 991, 427, 1244, 1065, 824, 1126, 562, 711, 750, 878, 148, 911, 527, 251, 320, 28, 607, 468, 387, 1174, 790, 187, 1033, 663, 57, 107, 499, 969, 1201, 352, 1077, 1246, 1130, 942, 822, 414, 1276, 726, 1000, 862, 548, 697, 145, 27, 225, 278, 305, 613, 583, 178, 891, 786, 648, 498, 1159, 1207, 54, 101, 1045, 1241, 1096, 952, 365, 1268, 459, 398, 821, 725, 979, 142, 12, 859, 267, 526, 752, 596, 310, 192, 1018, 690, 912, 1156, 646, 230, 426, 789, 495, 1192, 944, 65, 1230, 339, 1258, 1081, 827, 1053, 458, 150, 984, 37, 282, 885, 718, 578, 856, 109, 183, 396, 617, 309, 651, 913, 222, 792, 1026, 255, 1163, 1215, 519, 546, 1095, 751, 1242, 70, 490, 941, 454, 1125, 7, 139, 588, 707, 1271, 180, 421, 985, 1060, 848, 377, 876, 621, 284, 1025, 231, 654, 332, 1176, 539, 760, 36, 66, 1235, 810, 919, 111, 460, 1146, 579, 1090, 147, 1, 683, 190, 966, 1204, 487, 381, 998, 264, 411, 1117, 860, 351, 1055, 1267, 631, 542, 319, 733, 75, 221, 787, 1239, 104, 889, 35, 1169, 582, 162, 682, 439, 943, 1199, 193, 828, 504, 250, 135, 286, 999, 399, 353, 1137, 1108, 533, 1031, 632, 738, 0, 63, 781, 108, 1062, 1252, 1171, 899, 577, 318, 442, 705, 1217, 967, 868, 241, 475, 166, 823, 938, 285, 505, 31, 195, 666, 550, 391, 1142, 1028, 772, 115, 745, 622, 1075, 1251, 77, 333, 903, 587, 696, 988, 1220, 1106, 425, 233, 855, 1177, 947, 163, 489, 517, 202, 277, 394, 1133, 364, 794, 16, 765, 122, 1030, 1074, 615, 457, 555, 79, 916, 649, 1265, 699, 326, 1228, 829, 239, 1002, 1190, 948, 491, 735, 279, 407, 200, 1134, 872, 371, 777, 158, 114, 45, 585, 6, 1052, 81, 620, 456, 904, 1092, 547, 702, 809, 337, 837, 234, 1015, 1234, 736, 655, 1196, 970, 520, 486, 410, 1270, 272, 113, 40, 775, 870, 1161, 167, 574, 68, 2, 308, 446, 1054, 1097, 900, 380, 814, 601, 237, 1127, 1024, 350, 716, 940, 543, 1214, 409, 207, 477, 124, 1243, 629, 276, 41, 981, 769, 179, 684, 572, 1056, 85, 867, 1185, 315, 808, 511, 8, 1144, 835, 1110, 945, 918, 1027, 355, 400, 440, 474, 232, 734, 644, 138, 44, 611, 287, 176, 1232, 997, 556, 686, 83, 884, 1057, 1264, 506, 1193, 324, 1138, 774, 960, 110, 9, 406, 806, 932, 1085, 379, 847, 476, 739, 434, 283, 164, 246, 656, 1221, 209, 78, 612, 886, 541, 689, 507, 336, 48, 1122, 575, 1041, 959, 123, 797, 13, 1175, 930, 1274, 1084, 464, 744, 839, 363, 152, 717, 989, 201, 424, 627, 902, 1222, 660, 271, 94, 512, 236, 62, 1048, 307, 560, 1111, 395, 871, 1021, 591, 1076, 1155, 779, 1183, 1262, 748, 844, 344, 127, 949, 445, 693, 170, 1229, 5, 908, 650, 509, 623, 228, 478, 61, 551, 281, 373, 314, 90, 994, 1154, 1029, 1101, 1188, 753, 880, 418, 1067, 34, 121, 811, 838, 584, 1279, 695, 346, 909, 661, 224, 253, 784, 515, 1238, 194, 544, 723, 624, 975, 84, 1005, 296, 1205, 1120, 451, 1150, 1079, 405, 882, 26, 846, 1044, 595, 376, 117, 328, 676, 244, 151, 805, 773, 931, 483, 1254, 514, 963, 559, 712, 56, 993, 1189, 642, 289, 436, 746, 1104, 191, 894, 1218, 841, 87, 608, 1147, 119, 1047, 218, 15, 146, 802, 680, 348, 247, 392, 492, 536, 929, 317, 1017, 983, 55, 715, 1253, 419, 1091, 581, 770, 890, 1195, 742, 842, 186, 1049, 118, 614, 449, 645, 149, 91, 1153, 216, 384, 345, 23, 243, 482, 679, 986, 939, 295, 552, 1240, 1083, 417, 771, 883, 1112, 801, 1275, 833, 1020, 599, 444, 1050, 1184, 132, 732, 165, 1213, 199, 20, 258, 472, 378, 74, 501, 321, 570, 914, 961, 639, 288, 408, 1082, 1132, 678, 782, 1023, 853, 1247, 441, 610, 1181, 1051, 155, 747, 126, 229, 189, 710, 38, 256, 369, 887, 534, 497, 71, 1209, 976, 564, 818, 936, 669, 303, 342, 1143, 1006, 1113, 465, 432, 1261, 1180, 849, 1080, 638, 737, 105, 141, 1035, 780, 217, 602, 513, 14, 895, 401, 978, 540, 266, 174, 293, 933, 571, 367, 1007, 812, 49, 1124, 708, 1224, 335, 1255, 857, 1191, 1064, 103, 672, 467, 643, 431, 130, 609, 208, 1164, 756, 905, 404, 254, 535, 159, 980, 937, 803, 299, 1129, 368, 47, 567, 330, 1032, 1102, 1236, 503, 93, 701, 1066, 1206, 845, 598, 461, 1168, 227, 197, 671, 415, 776, 910, 749, 626, 955, 813, 1141, 383, 987, 154, 881, 51, 334, 1105, 270, 18, 96, 1069, 524, 300, 558, 597, 719, 469, 235, 1203, 854, 687, 1019, 1248, 428, 658, 182, 1140, 825, 764, 1170, 935, 898, 382, 128, 1103, 974, 625, 354, 1063, 52, 304, 22, 89, 724, 531, 211, 566, 865, 1210, 688, 798, 484, 594, 265, 177, 1012, 830, 1178, 922, 386, 659, 762, 1114, 892, 357, 1043, 964, 628, 39, 82, 448, 521, 134, 205, 1249, 1073, 554, 721, 325, 479, 1149, 290, 1212, 1004, 831, 924, 169, 675, 590, 778, 1109, 362, 238, 896, 972, 858, 636, 43, 102, 516, 437, 1179, 72, 1078, 703, 553, 331, 730, 268, 1152, 1266, 1022, 403, 175, 1226, 670, 927, 767, 360, 804, 226, 992, 852, 1115, 144, 641, 956, 438, 522, 488, 95, 1061, 1088, 549, 29, 311, 888, 1158, 606, 714, 59, 274, 1211, 920, 198, 768, 741, 340, 996, 1256, 816, 851, 681, 950, 435, 160, 374, 494, 1059, 402, 530, 100, 573, 312, 1139, 1100, 32, 1166, 249, 1200, 921, 188, 215, 763, 720, 977, 1016, 347, 840, 1231, 800, 447, 657.

Interleaving Table for $K = 1024$, $R = 2/3$, $N = 1536$

The following list of numbers represents the interleaving table for the $K = 1024$, $R = 2/3$ case, where the length of the interleaving table is $N = 1536$. This "1k" interleaving table contains every index in the set $\{0, 1, \ldots, 1534, 1535\}$, in random order, with the constraint that no index is within $S = 28$ positions in either direction of its sequentially adjacent indexes. The interleaving table is indexed by i and is $\pi(i) =$

1387, 953, 1255, 635, 1050, 157, 1183, 422, 1316, 844, 917, 781, 669, 197, 1492, 235, 374, 315, 1000, 1142, 1420, 1534, 455, 60, 1108, 486, 97, 1218, 597, 282, 1450, 6, 882, 1357, 1057, 564, 425, 812, 1266, 168, 702, 778, 666, 344, 214, 852, 636, 129, 1146, 527, 947, 1407, 246, 1097, 46, 485, 313, 1022, 1182, 1488, 988, 1443, 382, 1362, 733, 602, 83, 281, 412, 1289, 795, 1252, 453, 690, 184, 557, 909, 152, 122, 858, 1211, 1517, 251, 54, 504, 347, 1144, 764, 1408, 949, 1176, 645, 0, 1052, 989, 93, 297, 1471, 1368, 1102, 613, 218, 1277, 467, 807, 687, 896, 399, 576, 1322, 847, 1504, 140, 53, 721, 343, 501, 1133, 259, 763, 1417, 436, 1196, 1069, 110, 1032, 22, 994, 1371, 616, 172, 928, 1533, 308, 530, 1465, 899, 559, 1292, 964, 1099, 211, 1338, 373, 689, 859, 1140, 792, 757, 469, 428, 1246, 254, 1425, 1047, 56, 125, 656, 1001, 161, 23, 934, 626, 589, 1466, 825, 1171, 1514, 1303, 1110, 1204, 1375, 215, 330, 855, 536, 288, 396, 696, 885, 1337, 493, 1271, 1415, 92, 1081, 761, 459, 1004, 63, 365, 962, 1041, 164, 1477, 26, 1139, 803, 427, 921, 1238, 1377, 598, 840, 199, 265, 1199, 525, 683, 302, 1308, 1511, 558, 647, 725, 773, 1412,

98, 886, 358, 983, 496, 1017, 951, 138, 388, 1273, 1072, 58, 429, 1126, 1164, 857, 1229, 220, 821, 1376, 187, 15, 277, 1305, 1513, 922, 632, 729, 699, 1339, 568, 1424, 667, 91, 491, 355, 309, 968, 1195, 1479, 1002, 789, 393, 531, 1147, 1114, 851, 236, 818, 1256, 441, 1033, 1370, 202, 927, 170, 1306, 1519, 605, 134, 1080, 45, 724, 470, 641, 1444, 888, 96, 973, 1210, 267, 566, 387, 1489, 326, 672, 754, 788, 843, 1276, 1178, 1046, 1012, 1352, 1, 191, 1410, 1143, 432, 512, 1524, 923, 611, 153, 40, 1451, 1112, 974, 71, 107, 221, 273, 874, 1222, 351, 316, 1490, 692, 1297, 556, 796, 663, 463, 380, 1003, 765, 1331, 1398, 1166, 1082, 409, 1040, 507, 1264, 20, 908, 185, 1436, 1366, 120, 223, 869, 727, 606, 336, 279, 1135, 1226, 694, 573, 79, 838, 447, 651, 1494, 1008, 1404, 1193, 1075, 965, 156, 533, 1260, 50, 931, 786, 498, 1349, 415, 219, 1453, 3, 372, 124, 1319, 612, 1149, 250, 341, 89, 877, 280, 679, 1118, 1042, 1414, 1203, 188, 577, 963, 732, 1240, 33, 454, 1290, 528, 311, 766, 993, 836, 413, 1369, 131, 378, 638, 607, 484, 932, 1087, 72, 1502, 1462, 887, 686, 1401, 1054, 230, 1165, 797, 264, 189, 723, 1329, 1430, 349, 1234, 444, 539, 1286, 317, 39, 996, 758, 1132, 1359, 592, 625, 160, 480, 414, 384, 90, 1092, 842, 1509, 659, 893, 1198, 130, 1030, 510, 811, 260, 1326, 193, 1447, 1418, 715, 9, 935, 1480, 1281, 971, 769, 579, 1169, 43, 159, 1355, 1250, 306, 431, 356, 543, 1528, 658, 870, 1116, 477, 1039, 1386, 247, 123, 819, 1313, 210, 1441, 624, 392, 506, 94, 706, 1491, 580, 1284, 1191, 2, 1006, 740, 938, 977, 1345, 547, 1236, 154, 278, 350, 472, 1125, 1091, 55, 1389, 677, 831, 203, 1442, 634, 782, 865, 1523, 1062, 394, 905, 1160, 99, 437, 243, 581, 1483, 987, 950, 1354, 1291, 1247, 1016, 17, 532, 149, 299, 747, 716, 1194, 473, 820, 181, 1119, 1323, 64, 1452, 1405, 860, 395, 360, 655, 1049, 1521, 119, 596, 981, 919, 233, 502, 1078, 1484, 12, 952, 438, 1013, 1361, 774, 1237, 468, 270, 1170, 685, 830, 186, 720, 42, 563, 151, 1269, 1409, 1445, 637, 307, 77, 603, 872, 342, 106, 499, 1530, 1084, 1320, 1495, 1128, 216, 1200, 4, 449, 800, 1031, 377, 984, 955, 671, 1353, 920, 719, 567, 1249, 756, 1382, 1463, 292, 262, 408, 889, 1162, 328, 513, 117, 839, 1413, 37, 80, 1130, 196, 147, 478, 1522, 614, 1068, 1321, 1019, 375, 231, 1217, 940, 793, 713, 1282, 574, 7, 644, 1363, 261, 982, 759, 544, 678, 301, 1478, 907, 68, 1400, 863, 1106, 445, 116, 1448, 1187, 1518, 1067, 182, 346, 1151, 1314, 146, 390, 1248, 38, 591, 718, 1350, 503, 213, 985, 767, 271, 643, 1481, 936, 898, 1394, 808, 1018, 300, 1098, 434, 861, 538, 1058, 1208, 109, 464, 70, 359, 1174, 1526, 24, 673, 141, 728, 1426, 402, 570, 500, 209, 1340, 772, 1287, 978, 897, 1396, 1496, 310, 817, 268, 1253, 1090, 930, 174, 867, 1044, 1216, 448, 1158, 364, 239, 540, 108, 631, 44, 1434, 401, 703, 1532, 1365, 771, 737, 1324, 1122, 1293, 992, 303, 1487, 492, 668, 961, 1251, 1085, 5, 171, 1220, 881, 814, 1181, 926, 337, 553, 442, 257, 35, 1429, 1048, 587, 76, 115, 206, 709, 1372, 621, 752, 366, 1310, 1476, 471, 1137, 289, 505, 1005, 400, 176, 1261, 1228, 822, 876, 1077, 332, 144, 242, 552, 1419, 32, 913, 790, 82, 1506, 1184, 664, 967, 608, 695, 738, 430, 1035, 1449, 1155, 1302, 514, 1346, 369, 1257, 1225, 293, 1383, 177, 338, 1086, 833, 114, 551, 217, 901, 143, 249, 791, 1117, 1188, 954, 25, 615, 868, 1510, 750, 1023, 991, 435, 711, 404, 1298, 371, 57, 1437, 1472, 670, 1328, 1268, 291, 476, 95, 522, 1381, 1079, 322, 155, 1123, 126, 1192, 1221, 204, 1154, 639, 751, 593, 809, 555, 8, 710, 446, 942, 1516, 376, 1034, 853, 1344, 1458, 975, 1278, 681, 272, 1392, 903, 520, 241, 163, 327, 84, 1241, 1120, 1209, 483, 51, 617, 753, 1156, 13, 813, 646, 1422, 208, 370, 1500, 550, 416, 1029, 1335, 1070, 941, 680, 1384, 912, 990, 1288, 121, 582, 85, 1454, 1104, 519, 878, 150, 457, 240, 1254, 1189, 334, 810, 41, 628, 1416, 741, 296, 1503, 179, 383, 1026, 770, 420, 707, 490, 958, 1299, 1152, 554, 101, 594, 1061, 1111, 1379, 1474, 136, 892, 846, 1341, 353, 1223, 229, 524, 19, 657, 305, 1427, 59, 165, 1190, 1015, 768, 269, 385, 705, 426, 1505, 804, 1259, 466, 622, 1145, 572, 104, 135, 1045, 736, 1468, 1364, 854, 1230, 910, 1317, 511, 660, 1115, 222, 969, 542, 27, 1185, 339, 298, 397, 66, 368, 1439, 1262, 1395, 698, 175, 609, 1525, 780, 1059, 1150, 111, 482, 824, 742, 929, 1325, 875, 451, 1109, 253, 1360, 561, 648, 529, 14, 1007, 1205, 335, 285, 142, 74, 389, 1245, 1406, 1473, 600, 1515, 712, 200, 105, 418, 1141, 966, 495, 1071, 1036, 458, 1103, 862, 252, 1283, 1312, 654, 745, 894, 937, 321, 826, 537, 47, 1173, 290, 367, 1467, 585, 620, 684, 1529, 1438, 1380, 1136, 1224, 784, 976, 714, 461, 1055, 1021, 1105, 112, 173, 244, 406, 1311, 890, 943, 832, 744, 28, 1497, 320, 653, 62, 588, 1179, 546, 516, 276, 1356, 1390, 1258, 354, 1219, 1432, 456, 1073, 618, 1535, 1121, 801, 1461, 1011, 979, 212, 133, 849, 100, 391, 883, 700, 760, 424, 169, 1186, 1318, 730, 578, 312, 509, 248, 1499, 662, 1242, 16, 1388, 1089, 924, 49, 1285, 545, 1351, 1134, 980, 361, 610, 834, 1446, 1014, 1051, 78, 783, 475, 423, 1206, 183, 735, 323, 884, 255, 1501, 1177, 649, 697, 145, 113, 945, 11, 286, 1397, 1332, 1088, 916, 381, 569, 1263, 1531, 1440, 619, 65, 226, 835, 1294, 419, 460, 1470, 743, 1231, 777, 806, 521, 324, 652, 1129, 871, 1009, 1180, 21, 1385, 1333, 287, 128, 190, 1065, 1100, 1508, 933, 590, 1265, 224, 693, 900, 1296, 363, 52, 465, 731, 1213, 87, 403, 497, 1456, 841, 1148, 258, 333, 1024, 1403, 1347, 10, 661, 166, 799, 1056, 1485, 1520, 956, 548, 584, 1101, 762, 925, 895, 132, 362, 630, 433, 717, 1227, 1275, 225, 479, 1455, 1309, 61, 856, 329, 1131, 1172, 1399, 398, 518, 294, 1027, 195, 18, 794, 1064, 997, 565, 676, 1343, 911, 1095, 946, 1498, 755, 1232, 708, 227, 450, 162, 1307, 102, 823, 48, 633, 1201, 1421, 866, 256, 340, 535, 604, 407, 1272, 1159, 304, 1460, 494, 1066, 1373, 904, 1028, 1336, 1113, 674, 722, 995, 201, 944, 439, 127, 1304, 1243, 34, 1197, 829, 86, 640, 167, 560, 595, 1431, 348, 1527, 873, 1475, 481, 295, 526, 902, 245, 1161, 1037, 1074, 1348, 734, 205, 675, 972, 1378, 137, 1233, 704, 787, 1315, 827, 1274, 103, 73, 1107, 440, 586, 325, 939, 1457, 489, 1428, 284, 36, 237, 357, 1038, 1163, 627, 1486, 405, 192, 1202, 1076, 739, 1374, 534, 775, 701, 1235, 1295, 1334, 891, 805, 67, 1127, 998, 575, 845, 462, 1423, 960, 266, 30, 234, 318, 158, 1157, 118, 417, 1507, 194, 642, 386, 1060, 541, 776, 746, 1391, 1279, 1215, 906, 1459, 682, 816, 1020, 81, 601, 850, 1124, 474, 274, 352, 986, 1342, 232, 948, 1167, 139, 421, 180, 1093, 508, 1493, 571, 785, 29, 1411, 749, 1053, 1270, 1207, 880, 650, 1239, 319, 815, 918, 691, 1138, 1464, 1301, 75, 1010, 283, 957, 848, 1330, 1168, 238, 1083, 487, 411, 523, 779, 1512, 583, 726, 207, 1435, 31, 1367, 623, 1267, 452, 1402, 314, 914, 379, 148, 1214, 88, 688, 345, 1025, 864, 970, 1482, 275, 178, 1300, 1063, 1153, 1096, 828, 562, 517, 488, 798, 748, 228, 1433, 629, 1244, 1358, 443, 1393, 599, 915, 1212, 69, 410, 331, 999, 879, 665, 959, 1469, 198, 1280, 1043, 1327, 1175, 1094, 549, 837, 515, 802, 263.

Interleaving Table for $K = 1024$, $R = 1/2$, $N = 2048$

The following list of numbers represents the interleaving table for the $K = 1024$, $R = 1/2$ case, where the length of the interleaving table is $N = 2048$. This "1k" interleaving table contains every index in the set $\{0, 1, \ldots, 2046, 2047\}$, in random order, with the constraint that no index is within $S = 32$ positions in either direction of its sequentially adjacent indexes. The interleaving table is indexed by i and is $\pi(i) =$

481, 1871, 649, 430, 1099, 24, 964, 1054, 1659, 1156, 151, 216, 1829, 1234, 1612, 553, 325, 1559, 1289, 1991, 908, 1411, 1706, 1921, 1322, 1783, 1492, 807, 284, 74, 740, 694, 616, 1452, 872, 1357, 394, 491, 16, 436, 2044, 1749, 985, 1658, 217, 1181, 1089, 1887, 1230, 182, 110, 1850, 1038, 582, 1287, 1413, 1817, 654, 907, 947, 1621, 1569, 324, 1494, 1998, 531, 254, 1784, 818, 717, 289, 1460, 452, 1952, 1712, 863, 2037, 410, 1340, 1132, 61, 1187, 498, 1679, 14, 112, 1233, 1867, 1277, 1001, 1036, 196, 591, 957, 1630, 1079, 756, 354, 1410, 1514, 1826, 677, 919, 544, 1551, 713, 1765, 285, 236, 857, 1377, 403, 790, 1971, 160, 493, 1929, 1170, 1724, 1221, 1870, 1473, 459, 1312, 0, 1024, 124, 1649, 956, 2029, 1683, 1108, 598, 326, 1064, 644, 560, 1566, 195, 86, 913, 722, 1778, 685, 275, 1412, 873, 234, 1532, 816, 1359, 161, 1922, 1238, 1202, 1603, 1161, 1982, 41, 783, 477, 1462, 1304, 992, 1667, 2042, 596, 1858, 1078, 396, 359, 536, 1125, 909, 118, 437, 664, 1042, 948, 288, 219, 1701, 1517, 1567, 847, 323, 159, 1792, 706, 1378, 1928, 33, 1167, 1211, 771, 1748, 487, 1338, 1476, 1614, 1830, 77, 570, 370, 521, 1962, 625, 1864, 1009, 1100, 914, 126, 1419, 290, 1051, 1247, 1298, 974, 1527, 215, 822, 867, 666, 2009, 1789, 39, 418, 726, 1656, 1379, 467, 1332, 759, 1186, 1591, 257, 1756, 368, 1695, 1941, 1893, 532, 595, 1486, 1976, 628, 1117, 1152, 921, 1219, 109, 1018, 144, 1533, 214, 1842, 1418, 805, 1279, 2046, 426, 40, 293, 1075, 723, 860, 1631, 1372, 671, 1797, 1576, 260, 1703, 1914, 489, 1326, 366, 961, 623, 1484, 1746, 588, 2011, 1, 1154, 1964, 898, 530, 1519, 1239, 206, 1020, 1421, 1197, 408, 111, 449, 1856, 328, 53, 1069, 167, 1613, 853, 657, 1363, 1823, 817, 739, 1660, 1114, 1572, 969, 282, 1896, 363, 1283, 1702, 2001, 1457, 1330, 1939, 1515, 484, 903, 213, 551, 1180, 701, 1250, 11, 609, 60, 406, 145, 1031, 1402, 442, 1368, 1068, 96, 1824, 1773, 1618, 1138, 988, 855, 939, 794, 1582, 371, 327, 750, 1735, 659, 265, 1889, 1214, 479, 1699, 1934, 1999, 218, 549, 1448, 181, 1315, 30, 899, 405, 1102, 147, 708, 1404, 1535, 613, 1852, 1364, 1500, 842, 1779, 1022, 1579, 1255, 792, 2033, 746, 82, 1065, 1165, 283, 360, 497, 1736, 1205, 1901, 1973, 249, 1625, 952, 193, 448, 1309, 891, 49, 320, 1693, 1439, 556, 615, 1820, 653, 1386, 1522, 1937, 133,

993, 1275, 1026, 1343, 725, 413, 1116, 1483, 798, 2034, 374, 760, 1242, 850, 1750, 266, 1866, 929, 1578, 1059, 886, 1652, 44, 307, 191, 99, 1685, 483, 674, 587, 340, 1521, 1444, 1171, 1967, 1828, 528, 1300, 712, 1017, 1401, 1367, 636, 132, 1121, 782, 2028, 843, 965, 243, 1884, 1222, 1723, 1790, 402, 8, 1055, 883, 1257, 917, 48, 462, 1629, 316, 1926, 577, 749, 1562, 679, 508, 1474, 1596, 87, 1844, 361, 1395, 1324, 614, 130, 1126, 841, 2036, 1428, 967, 1528, 165, 1006, 251, 1738, 1074, 1677, 1039, 791, 890, 1994, 1891, 1777, 1217, 466, 548, 1176, 933, 12, 1930, 697, 661, 748, 1580, 93, 1301, 353, 1264, 1468, 397, 1341, 1633, 593, 506, 298, 1124, 155, 1546, 856, 1719, 1833, 1019, 238, 1513, 203, 1076, 897, 2035, 1755, 1425, 1980, 1381, 786, 546, 977, 454, 691, 1868, 1213, 80, 1917, 25, 1465, 646, 421, 584, 1615, 369, 1164, 821, 1265, 139, 297, 1671, 1111, 1308, 1498, 336, 1704, 1793, 1021, 202, 1757, 1950, 1058, 1835, 1414, 767, 474, 693, 1989, 904, 1573, 76, 983, 519, 1454, 2038, 432, 23, 655, 589, 1624, 1872, 241, 1268, 946, 277, 376, 1319, 728, 815, 1490, 1159, 1711, 1534, 1113, 343, 174, 1668, 1935, 1375, 1791, 1988, 1228, 1041, 773, 906, 470, 552, 69, 864, 1570, 991, 1431, 113, 1886, 1840, 239, 673, 1272, 1192, 592, 35, 825, 385, 1323, 1741, 718, 2027, 1081, 339, 188, 505, 1663, 1609, 1464, 1136, 1979, 635, 1373, 295, 1037, 1923, 787, 1705, 1544, 557, 1236, 868, 1781, 1423, 949, 1832, 440, 128, 9, 910, 1003, 63, 1295, 754, 2026, 594, 824, 1885, 695, 179, 1610, 252, 503, 381, 1983, 1745, 1491, 304, 1365, 662, 1131, 1456, 1083, 1040, 337, 1676, 1331, 1568, 1208, 433, 1408, 141, 1940, 986, 540, 1282, 1809, 755, 1175, 1846, 806, 597, 45, 928, 1643, 246, 1894, 1710, 486, 707, 2024, 1518, 7, 1768, 1441, 645, 1034, 344, 1975, 303, 378, 1120, 1325, 1070, 870, 1405, 180, 1597, 525, 1931, 434, 978, 769, 1163, 1821, 122, 1245, 1204, 1482, 1653, 1892, 1291, 1726, 231, 942, 492, 1857, 42, 1688, 565, 802, 710, 1449, 1531, 663, 83, 399, 1996, 310, 350, 1063, 600, 185, 2, 840, 1585, 1948, 1355, 1146, 889, 1396, 764, 1798, 1103, 1906, 439, 134, 1619, 926, 1210, 962, 1747, 1288, 1253, 2041, 696, 1845, 223, 658, 273, 526, 37, 1536, 387, 1035, 85, 617, 1443, 1002, 820, 1713, 1489, 1348, 1953, 1675, 779, 1122, 571, 338, 450, 1607, 136, 1802, 1919, 743, 1178, 1640, 1392, 187, 3, 1838, 2015, 1072, 1224, 287, 702, 1543, 488, 65, 395, 1297, 224, 524, 866, 1469, 1507, 1954, 639, 1700, 101, 1032, 349, 1763, 963, 1352, 832, 925, 143, 735, 1895, 599, 428, 1594, 1815, 28, 1084, 1198, 2012, 296, 1554, 690, 1655, 468, 186, 1426, 235, 777, 1139, 1280, 541, 1516, 877, 656, 1232, 1944, 73, 1048, 1387, 1762, 502, 1977, 345, 384, 997, 602, 1334, 844, 1477, 1837, 115, 1105, 425, 1898, 279, 1571, 1720, 1661, 704, 1604, 154, 1140, 2018, 1271, 19, 1435, 539, 1189, 222, 1512, 927, 640, 770, 1060, 1961, 1769, 388, 482, 315, 1023, 1399, 845, 808, 351, 75, 982, 1470, 1816, 1855, 1899, 737, 255, 1227, 116, 1722, 1682, 1294, 1119, 178, 4, 1574, 1157, 1611, 896, 668, 516, 937, 590, 550, 1776, 1335, 1943, 1193, 2006, 775, 309, 854, 1415, 1066, 1505, 1459, 38, 72, 810, 247, 1027, 1876, 1743, 730, 356, 1243, 142, 1669, 1818, 971, 1623, 391, 902, 1296, 1538, 465, 1147, 529, 936, 637, 1358, 2013, 1925, 1969, 1201, 579, 194, 308, 1587, 683, 1062, 803, 34, 1445, 1478, 765, 90, 1008, 1760, 852, 342, 1403, 1662, 1877, 1258, 259, 1811, 379, 1727, 727, 1628, 127, 464, 973, 423, 1112, 1307, 2030, 626, 573, 1362, 1918, 905, 1525, 1053, 1588, 1174, 171, 1997, 796, 91, 55, 761, 335, 1437, 208, 5, 1241, 1865, 830, 1764, 280, 699, 1681, 1799, 495, 944, 392, 1731, 995, 1129, 1299, 1645, 627, 427, 1339, 1946, 1907, 1550, 1182, 1045, 162, 1987, 1393, 795, 1479, 874, 1091, 762, 66, 199, 535, 834, 21, 1841, 2021, 580, 276, 240, 711, 472, 1698, 1753, 1215, 938, 1642, 1004, 129, 357, 412, 1267, 312, 1800, 641, 1162, 1602, 1430, 1383, 1530, 1346, 1882, 1942, 789, 1563, 58, 885, 168, 1485, 201, 1985, 569, 1090, 6, 1127, 1305, 2032, 267, 1694, 1728, 918, 1767, 703, 1648, 608, 1229, 533, 742, 1160, 648, 494, 1269, 460, 341, 1438, 95, 1849, 839, 51, 1196, 784, 1903, 1471, 170, 1043, 1370, 1955, 1540, 17, 1575, 996, 221, 1691, 567, 941, 688, 1333, 1077, 1780, 1733, 875, 271, 650, 1617, 1263, 407, 322, 603, 2019, 478, 104, 733, 364, 445, 1861, 70, 1212, 1151, 809, 1650, 774, 1030, 1951, 1388, 1508, 200, 1434, 153, 513, 700, 1115, 1583, 32, 1772, 1915, 1696, 953, 888, 261, 314, 1541, 846, 1344, 1810, 1259, 1990, 660, 607, 380, 79, 734, 1475, 469, 1306, 1639, 414, 2040, 1957, 1173, 564, 1729, 190, 1061, 1220, 1590, 1427, 1851, 29, 785, 1106, 149, 512, 970, 1547, 1775, 851, 901, 1687, 1913, 1380, 610, 333, 1510, 367, 227, 1463, 456, 1256, 1310, 409, 720, 94, 268, 1166, 665, 1050, 2002, 1634, 1012, 1839, 1223, 1345, 1098, 1875, 175, 517, 951, 1552, 780, 1734, 838, 1697, 559, 1599, 1429, 1771, 358, 317, 1806, 601, 1947, 138, 458, 1909, 724, 411, 1511, 59, 634, 245, 1155, 2025, 915, 105, 1262, 676, 1862, 1647, 881, 954, 1555, 1118, 1056, 799, 192, 278, 1194, 509, 1453, 26, 1011, 1394, 1316, 763, 1970, 1795, 555, 714, 443, 1752, 1351, 1504, 836, 611, 398, 1924, 135, 62, 916, 226, 2031, 1266, 318, 976, 1654, 878, 1879, 1094, 1141, 1608, 1218, 651, 272, 1442, 1407, 1690, 1029, 496, 1558, 1836, 561, 15, 1311, 176, 1766, 757, 441, 1487, 383, 1371, 1995, 1524, 814, 709, 612, 106, 68, 321, 990, 230, 1927, 871, 1158, 1600, 1880, 1235, 270, 940, 1273, 1424, 1709, 1635, 1085, 1673, 1813, 537, 572, 184, 1744, 752, 1461, 20, 1389, 463, 1984, 1349, 389, 2017, 829, 624, 1195, 980, 131, 1013, 672, 311, 869, 1313, 225, 89, 1908, 920, 1246, 499, 1529, 1592, 1638, 1859, 56, 705, 1097, 1052, 1786, 346, 1737, 543, 578, 1400, 1135, 1960, 461, 2010, 758, 390, 631, 1436, 1016, 1199, 163, 1493, 861, 955, 1321, 300, 210, 1248, 1825, 501, 103, 828, 1678, 1564, 911, 1356, 64, 1782, 716, 1920, 1725, 575, 263, 347, 1863, 429, 1616, 18, 1142, 1067, 1992, 772, 647, 1284, 1209, 681, 981, 945, 1416, 1501, 2045, 158, 212, 504, 542, 876, 1015, 1101, 1577, 117, 1674, 1458, 306, 1317, 833, 1819, 1932, 365, 256, 1881, 732, 422, 1739, 1149, 2008, 1244, 1774, 1049, 1366, 781, 31, 698, 1207, 1968, 924, 633, 1499, 71, 1627, 1005, 972, 1092, 887, 1593, 1466, 150, 1537, 518, 827, 457, 1664, 1843, 1278, 292, 237, 108, 1432, 1153, 1933, 581, 1801, 400, 1730, 1390, 1883, 27, 1047, 329, 753, 930, 1337, 1981, 669, 362, 1191, 1225, 987, 1589, 619, 148, 538, 1503, 1110, 1553, 1632, 475, 800, 2022, 1680, 98, 264, 211, 1302, 715, 859, 1804, 1144, 1916, 1261, 1398, 10, 1853, 1716, 892, 416, 1447, 586, 1972, 1071, 1770, 313, 667, 622, 1347, 547, 943, 1183, 177, 989, 1644, 140, 1548, 92, 250, 352, 1481, 47, 476, 2023, 849, 1936, 514, 1276, 1601, 1145, 1107, 1237, 1812, 744, 1902, 1433, 13, 1397, 1044, 1758, 689, 811, 882, 1978, 305, 558, 1860, 417, 605, 1360, 204, 1692, 81, 638, 1327, 1520, 1467, 248, 119, 515, 922, 2014, 1188, 1605, 473, 1254, 747, 994, 46, 1561, 169, 1028, 1785, 1422, 355, 813, 2047, 1900, 1137, 1827, 879, 1651, 1382, 435, 1082, 1958, 1689, 1292, 692, 1740, 606, 123, 220, 1329, 545, 652, 958, 401, 262, 302, 731, 1177, 36, 1502, 2007, 507, 1033, 999, 1216, 1440, 788, 183, 1822, 1128, 1788, 1549, 88, 446, 894, 1636, 1088, 1385, 1670, 1252, 1708, 1974, 1754, 574, 1888, 675, 404, 229, 1342, 629, 269, 719, 1488, 934, 2020, 848, 332, 1172, 1584, 1303, 125, 768, 173, 480, 1014, 1847, 1805, 54, 447, 527, 804, 1545, 1206, 1626, 1109, 1450, 1718, 1959, 1912, 979, 1251, 1374, 1073, 233, 568, 281, 912, 2039, 678, 1506, 2003, 1150, 865, 643, 1286, 1684, 120, 386, 156, 1761, 745, 330, 197, 1831, 604, 1417, 1598, 1185, 812, 1556, 1878, 1945, 510, 1451, 1721, 1336, 1249, 1093, 959, 1646, 78, 242, 1509, 900, 562, 419, 1057, 998, 1796, 1993, 670, 1134, 164, 121, 294, 1686, 453, 372, 862, 729, 766, 1376, 1557, 1409, 1184, 1285, 22, 522, 1897, 823, 1956, 1751, 1848, 1320, 334, 1620, 232, 630, 1480, 1096, 968, 198, 585, 923, 1794, 1446, 2016, 1010, 1523, 100, 438, 157, 1657, 1231, 1715, 67, 1369, 1169, 778, 375, 500, 884, 738, 684, 291, 835, 1314, 1904, 1966, 1595, 1281, 1133, 258, 1854, 618, 1095, 960, 205, 1455, 1759, 566, 1542, 114, 431, 1025, 172, 1807, 1496, 1707, 2005, 1354, 331, 793, 490, 1190, 393, 43, 1641, 895, 751, 858, 1406, 1606, 1963, 680, 1270, 1890, 621, 1104, 274, 1318, 975, 1143, 534, 583, 207, 444, 935, 97, 166, 1814, 1717, 1472, 2043, 1565, 819, 1526, 319, 1361, 1200, 1666, 377, 57, 741, 880, 1420, 485, 1965, 687, 1874, 1274, 1080, 1910, 1622, 2004, 1148, 253, 1007, 620, 451, 415, 931, 1240, 137, 209, 1834, 523, 286, 576, 1046, 102, 1353, 826, 966, 1732, 1787, 1560, 1203, 52, 736, 776, 348, 1873, 1290, 1391, 1086, 1665, 682, 382, 1123, 1495, 642, 455, 244, 1911, 1000, 1168, 420, 1986, 932, 511, 299, 563, 107, 146, 1350, 1949, 1742, 837, 189, 893, 1586, 1803, 801, 50, 1539, 1260, 1293, 1869, 1384, 686, 721, 1637, 1226, 373, 1087, 1497, 228, 1130, 1179, 424, 1905, 632, 984, 520, 471, 301, 84, 1672, 1328, 1714, 2000, 1938, 152, 950, 1808, 1581, 554, 797, 831.

APPENDIX
PARITY CHECK MATRIXES FOR ARTM0 (PCM/FM)

Parity Check Matrix for ARTM0 with $K = 1024$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 143$, and is $\mathcal{L}(l) =$

32, 9, 120, 137, 252, 345, 469, 540, 990, 1069, 1199, 9, 100, 139, 252, 348, 457, 516, 659, 972, 1203, 10, 127, 234, 341, 472, 529, 906, 952, 983, 1037, 1193, 10, 115, 142, 253, 341, 451, 530, 602, 1100, 1187, 1246, 23, 16, 44, 201, 271, 366, 401, 470, 497, 546, 606, 638, 685, 731, 756, 772, 816, 869, 918, 944, 1019, 1071, 1149, 1170, 23, 10, 55, 81, 171, 316, 364, 415, 444, 490, 573, 606, 611, 646, 765, 793, 829, 858, 884, 943, 1123, 1203, 1239, 1272, 25, 72, 160, 198, 237, 264, 313, 379, 443, 488, 516, 567, 614, 664, 696, 704, 833, 871, 898, 1020, 1055, 1069, 1108, 1148, 1173, 1249, 25, 30, 36, 88, 109, 177, 199, 283, 311, 322, 406, 443, 650, 702, 712, 744, 782, 802, 854, 1013, 1028, 1071, 1119, 1154, 1232, 1256.

Parity Check Matrix for ARTM0 with $K = 1024$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 83$, and is $\mathcal{L}(l) =$

64, 5, 10, 375, 1191, 1225, 1436, 5, 15, 1137, 1210, 1254, 1442, 6, 5, 317, 361, 1175, 1245, 1334, 6, 33, 831, 977, 1116, 1255, 1463, 8, 141, 509, 562, 604, 669, 934, 1030, 1363, 8, 82, 223, 439, 543, 749, 881, 1055, 1519, 18, 126, 159, 210, 292, 323, 416, 498, 620, 660, 751, 823, 833, 927, 998, 1069, 1341, 1400, 1517, 18, 64, 128, 244, 279, 392, 506, 552, 608, 643, 740, 805, 875, 948, 988, 1108, 1305, 1352, 1480.

Parity Check Matrix for ARTM0 with $K = 1024$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 65$, and is $\mathcal{L}(l) =$

128, 3, 498, 1466, 1686, 3, 455, 899, 1672, 4, 498, 551, 1161, 1698, 4, 475, 664, 1749, 1808, 5, 248, 457, 1521, 1751, 1927, 5, 450, 813, 905, 1081, 1697, 16, 38, 197, 289, 293, 474, 608, 648, 768, 1110, 1173, 1300, 1456, 1654, 1657, 1806, 2011, 16, 10, 64, 148, 357, 586, 686, 825, 1015, 1119, 1204, 1280, 1298, 1599, 1772, 1824, 1972.

Parity Check Matrix for ARTM0 with $K = 4096$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 143$, and is $\mathcal{L}(l) =$

128, 9, 752, 1417, 1705, 1971, 2880, 3772, 4172, 4274, 4899, 9, 308, 687, 1410, 1718, 1943, 2817, 3716, 4301, 4945, 10, 1449, 1728, 1955, 2224, 2456, 2816, 3785, 4295, 4727, 4943, 10, 631, 756, 822, 1410, 1747, 2019, 2590, 2890, 3760, 4967, 23, 49, 197, 483, 1140, 1267, 1331, 1595, 1813, 2539, 2664, 2734, 3020, 3185, 3361, 3544, 3821, 3926, 4092, 4108, 4434, 4532, 4718, 5077, 23, 12, 181, 294, 564, 908, 1149, 1643, 1899, 1931, 2060, 2430, 2460, 2657, 2752, 2973, 3119, 3262, 3415, 3657, 3871, 4462, 4503, 4756, 25, 181, 377, 411, 850, 994, 1256, 1365, 1539, 1700, 2109, 2180, 2360, 2694, 2824, 3255, 3398, 3481, 3634, 3847, 4010, 4105, 4545, 4630, 4818, 5034, 25, 67, 289, 402, 611, 878, 952, 1060, 1212, 1289, 1422, 1883, 2143, 2199, 2415, 2952, 3079, 3321, 3568, 3585, 4014, 4127, 4370, 4759, 4881, 5063.

Parity Check Matrix for ARTM0 with $K = 4096$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 97$, and is $\mathcal{L}(l) =$

256, 5, 997, 1958, 4156, 5143, 6053, 5, 497, 1796, 2439, 3988, 5158, 6, 366, 2757, 4105, 4948, 5170, 5888, 6, 403, 1795, 2482, 3101, 3733, 5895, 6, 205, 486, 1810, 4379, 5181, 5954, 6, 374, 2041, 3534, 5306, 5531, 5928, 27, 205, 547, 793, 1176, 1272, 1452, 1554, 1772, 2214, 2367, 2654, 2799, 2848, 2950, 3117, 3427, 3696, 3988, 4268, 4395, 4442, 4708, 5118, 5399, 5715, 5853, 5988, 27, 213, 424, 579, 672, 933, 1104, 1280, 1329, 1615, 2141, 2247, 2509, 2786, 2902, 3192, 3224, 3483, 3839, 4030, 4209, 4518, 4652, 4838, 5093, 5565, 5626, 5675.

Parity Check Matrix for ARTM0 with $K = 4096$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \le l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 65$, and is $\mathcal{L}(l) =$

512, 3, 1462, 4598, 5164, 3, 4252, 5466, 8151, 4, 2144, 4484, 5462, 7363, 4, 95, 4264, 5170, 6026, 5, 1308, 1950, 4293, 5056, 5599, 5, 531, 4232, 5272, 6599, 8146, 16, 371, 631, 1047, 1684, 2157, 2560, 2822, 3072, 3519, 3920, 4350, 4861, 6062, 6400, 7061, 7449, 16, 175, 998, 1945, 2141, 3011, 3496, 3584, 3769, 5098, 5534, 5998, 6332, 6656, 7129, 7533, 8051.

APPENDIX
PARITY CHECK MATRIXES FOR ARTM1 (SOQPSK-TG)

Parity Check Matrix for ARTM1 with $K = 1024$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 143$, and is $\mathcal{L}(l) =$

32, 11, 51, 326, 371, 647, 741, 868, 911, 955, 1035, 1090, 1173, 11, 43, 337, 374, 401, 644, 750, 890, 918, 939, 1118, 1166, 11, 43, 333, 381, 622, 670, 754, 874, 944, 1115, 1155, 1221, 11, 36, 346, 363, 647, 745, 894, 919, 994, 1093, 1122, 1157, 22, 28, 69, 119, 156, 189, 207, 232, 300, 337, 361, 445, 471, 483, 684, 712, 786, 821, 837, 1047, 1063, 1217, 1256, 22, 14, 67, 150, 177, 249, 269, 406, 454, 521, 547, 576, 735, 750, 789, 802, 834, 968, 1009, 1082, 1176, 1205, 1252, 23, 19, 127, 188, 195, 248, 266, 290, 440, 457, 493, 522, 561, 584, 615, 696, 782, 990, 1050, 1077, 1102, 1148, 1207, 1221, 23, 81, 124, 148, 222, 278, 301, 394, 434, 486, 520, 549, 576, 611, 650, 681, 732, 806, 847, 990, 1015, 1122, 1215, 1275.

Parity Check Matrix for ARTM1 with $K = 1024$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 91$, and is $\mathcal{L}(l) =$

64, 6, 172, 387, 653, 1075, 1097, 1449, 6, 168, 424, 1054, 1161, 1281, 1419, 7, 169, 397, 689, 788, 1067, 1313, 1447, 7, 184, 424, 662, 1072, 1343, 1386, 1470, 11, 78, 261, 323, 409, 545, 641, 935, 999, 1114, 1253, 1326, 11, 19, 210, 507, 611, 693, 753, 843, 1035, 1170, 1305, 1487, 17, 0, 124, 243, 287, 346, 504, 513, 577, 732, 829, 835, 937, 1009, 1125, 1252, 1365, 1511, 17, 10, 115, 229, 305, 349, 487, 520, 623, 707, 769, 841, 934, 986, 1184, 1248, 1373, 1477.

Parity Check Matrix for ARTM1 with $K = 1024$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 65$, and is $\mathcal{L}(l) =$

128, 4, 1097, 1133, 1383, 1839, 4, 12, 1068, 1812, 1867, 5, 568, 1064, 1313, 1597, 1810, 5, 41, 455, 795, 1086, 1917, 6, 384, 514, 680, 1090, 1202, 1577, 6, 189, 441, 552, 797, 1677, 1863, 13, 200, 295, 323, 710, 997, 1080, 1153, 1375, 1506, 1532, 1549, 1675, 1927, 13, 21, 146, 333, 742, 871, 949, 965, 1172, 1521, 1731, 1886, 1951, 1978.

Parity Check Matrix for ARTM1 with $K = 4096$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 143$, and is $\mathcal{L}(l) =$

128, 11, 244, 455, 542, 669, 2123, 2195, 2850, 3133, 4186, 4346, 4712, 11, 177, 424, 621, 706, 2148, 2289, 2852, 3025, 3126, 4132, 4630, 11, 201, 472, 576, 726, 2064, 2288, 2391, 2870, 3140, 3661, 4612, 11, 210, 497, 576, 684, 2163, 2228, 2765, 2889, 4211, 4654, 4804, 22, 78, 189, 834, 926, 1032, 1173, 1297, 1415, 1602, 1885, 2031, 2428, 2611, 2901, 3247, 3449, 3766, 3892, 4254, 4378, 4872, 5028, 22, 58, 326, 391, 970, 1098, 1402, 1480, 1663, 1740, 1813, 1940, 2478, 3032, 3209, 3376, 3556, 4024, 4392, 4564, 4640, 4800, 4978, 23, 361, 856, 1254, 1302, 1692, 1792, 1965, 2112, 2329, 2483, 2605, 2697, 3248, 3375, 3556, 3660, 3765, 3955, 3997, 4340, 4597, 4868, 4993, 23, 80, 361, 868, 1011, 1035, 1164, 1471, 1567, 1687, 2195, 2527, 2680, 2780, 3060, 3506, 3632, 3764, 3941, 4004, 4384, 4578, 4796, 5094.

Parity Check Matrix for ARTM1 with $K = 4096$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 101$, and is $\mathcal{L}(l) =$

256, 6, 2226, 4570, 4760, 4847, 5359, 5956, 6, 2081, 2259, 3518, 3635, 3976, 4650, 7, 1904, 3070, 4009, 4358, 4910, 5224, 5963, 7, 259, 1363, 3445, 3827, 4087, 5362, 5506, 8, 1071, 2083, 3315, 4014, 4520, 4669, 4807, 5273, 8, 911, 2102, 2173, 2474, 3346, 3935, 4847, 5252, 25, 61, 68, 459, 628, 787, 1105, 1369, 1610, 1634, 1991, 2525, 2621, 2943, 3127, 3319, 3635, 4240, 4549, 4901, 4987, 5333, 5520, 5648, 5681, 5948, 25, 246, 294, 626, 697, 946, 1218, 1361, 1497, 1624, 1867, 2334, 2529, 2715, 2743, 3066, 3318, 3343, 3799, 4058, 4201, 4222, 5096, 5594, 5787, 5982.

Parity Check Matrix for ARTM1 with $K = 4096$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 65$, and is $\mathcal{L}(l) =$

512, 4, 5494, 5610, 6840, 8080, 4, 4280, 5143, 6788, 6877, 5, 1859, 3292, 5124, 7150, 7965, 5, 2283, 4332, 5480, 6347, 6722, 6, 904, 1954, 2917, 3255, 5337, 6601, 6, 2088, 3319, 3984, 6462, 6812, 7510, 13, 491, 996, 1152, 1478, 1722, 2905, 3816, 4619, 4897, 5519, 5814, 7602, 7698, 13, 267, 500, 786, 1057, 2446, 2563, 3736, 4152, 4611, 5963, 6007, 6767, 7201.

APPENDIX
PARITY CHECK MATRIXES FOR ARTM2 (ARTM CPM)

Parity Check Matrix for ARTM2 with $K = 1024$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 139$, and is $\mathcal{L}(l) =$

32, 6, 42, 142, 983, 1135, 1187, 1273, 6, 63, 156, 189, 834, 989, 1129, 7, 42, 130, 313, 423, 960, 979, 1191, 7, 137, 145, 172, 460, 964, 1041, 1133, 22, 27, 47, 186, 202, 225, 279, 301, 363, 425, 451, 522, 588, 615, 662, 682, 890, 1009, 1091, 1153, 1213, 1226, 1263, 22, 176, 325, 389, 431, 479, 540, 552, 592, 629, 703, 737, 802, 836, 890, 932, 1011, 1043, 1084, 1117, 1126, 1178, 1194, 30, 4, 72, 74, 105, 214, 230, 268, 301, 338, 373, 389, 442, 480, 487, 542, 550, 670, 695, 730, 736, 778, 799, 824, 886, 914, 944, 1085, 1095, 1237, 1250, 30, 19, 74, 99, 109, 192, 242, 257, 326, 381, 396, 477, 485, 551, 584, 610, 667, 709, 726, 763, 791, 810, 859, 905, 907, 929, 1020, 1055, 1056, 1168, 1246.

Parity Check Matrix for ARTM2 with $K = 1024$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 91$, and is $\mathcal{L}(l) =$

64, 4, 984, 1170, 1259, 1348, 4, 999, 1050, 1141, 1234, 4, 771, 1223, 1350, 1487, 4, 447, 958, 987, 1150, 4, 374, 996, 1246, 1303, 4, 230, 851, 961, 1256, 29, 60, 68, 127, 173, 232, 313, 356, 419, 451, 463, 529, 544, 577, 597, 703, 723, 729, 777, 781, 855, 925, 1038, 1106, 1189, 1291, 1337, 1389, 1426, 1506, 29, 29, 47, 70, 169, 186, 207, 269, 290, 344, 403, 431, 486, 538, 576, 644, 649, 712, 781, 882, 891, 905, 1055, 1148, 1205, 1284, 1355, 1411, 1440, 1512.

Parity Check Matrix for ARTM2 with $K = 1024$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 61$, and is $\mathcal{L}(l) =$

128, 3, 847, 1507, 1681, 3, 445, 820, 1702, 3, 773, 1424, 1952, 3, 387, 1143, 1767, 4, 95, 600, 1338, 1442, 4, 116, 480, 592, 1619, 16, 170, 323, 360, 636, 705, 859, 971, 976, 1036, 1157, 1200, 1392, 1466, 1543, 1868, 1952, 16, 85, 183, 210, 260, 447, 700, 717, 933, 1097, 1272, 1280, 1580, 1791, 1819, 1866, 1978.

Parity Check Matrix for ARTM2 with $K = 4096$ and $R = 4/5$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 4/5$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 139$, and is $\mathcal{L}(l) =$

128, 6, 2519, 2926, 3728, 4486, 4665, 4890, 6, 2062, 2503, 2911, 2971, 3793, 4988, 7, 493, 2255, 2521, 2524, 3715, 4546, 4899, 7, 2484, 2816, 3002, 3783, 3791, 3914, 4217, 22, 1, 182, 325, 396, 710, 1380, 1578, 1750, 1869, 1986, 2284, 2428, 2586, 2945, 3435, 4057, 4150, 4306, 4587, 4608, 4736, 4989, 22, 416, 655, 827, 1230, 1456, 1536, 1678, 1847, 2007, 2138, 2579, 2805, 2884, 2967, 3155, 3256, 3405, 3919, 4127, 4536, 4844, 5099, 30, 105, 202, 267, 434, 600, 895, 922, 958, 1139, 1262, 1357, 1469, 1657, 1804, 2045, 2287, 2336, 2783, 3075, 3283, 3579, 3632, 3634, 3987, 4335, 4400, 4441, 4679, 4832, 5047, 30, 25, 227, 330, 544, 567, 695, 827, 954, 1055, 1130, 1192, 1343, 1459, 1722, 2164, 2372, 2678, 2702, 3176, 3229, 3380, 3505, 3508, 3598, 3846, 4068, 4191, 4292, 4422, 5047.

Parity Check Matrix for ARTM2 with $K = 4096$ and $R = 2/3$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 2/3$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 91$, and is $\mathcal{L}(l) =$

256, 4, 396, 3558, 4351, 4847, 4, 2419, 3378, 4629, 4920, 4, 493, 814, 4426, 4773, 4, 697, 2404, 3384, 5315, 4, 3168, 3502, 3807, 4634, 4, 3465, 4770, 5837, 5921, 29, 232, 300, 632, 851, 1085, 1169, 1383, 1469, 1647, 1958, 2258, 2287, 2361, 2581, 2784, 2844, 2979, 3136, 3617, 3827, 4003, 4120, 4440, 4578, 4866, 5249, 5576, 5661, 6084, 29, 148, 227, 280, 529, 820, 1107, 1503, 1593, 1655, 1810, 2026, 2107, 2539, 2619, 3035, 3247, 3740, 3882, 3963, 4210, 4539, 5071, 5159, 5352, 5413, 5595, 5665, 6027, 6055.

Parity Check Matrix for ARTM2 with $K = 4096$ and $R = 1/2$

The following list of numbers represents the parity check matrix, **H**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 1/2$. This list follows the sparse representation of **H** defined in Section 3.2. The list is indexed by $0 \leq l < \mathcal{L}_{\text{len}}$, where $\mathcal{L}_{\text{len}} = 61$, and is $\mathcal{L}(l) =$

512, 3, 1233, 5107, 6614, 3, 1303, 2058, 4874, 3, 1332, 4204, 6172, 3, 2553, 4973, 7175, 4, 2, 6036, 6537, 8171, 4, 273, 657, 2263, 8110, 16, 1012, 1398, 1555, 2844, 3260, 3294, 3763, 3914, 4495, 5567, 5780, 6205, 6961, 6986, 7217, 8111, 16, 8, 734, 1569, 1857, 2467, 2631, 3000, 3139, 3621, 4294, 5082, 5466, 5550, 5644, 6675, 7607.

APPENDIX
GENERATOR MATRIXES FOR ARTM0 (PCM/FM)

Generator Matrix for ARTM0 with $K = 1024$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 4/5$. Each string has a length of 8 hexadecimal characters and represents the top row of a size-32 circulant ($M_L = 32$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32 × 8 grid of (possibly) dense circulants, and so there are a total of 32 × 8 = 256 circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because the strings are so short for this code, they can be arranged below with 8 strings per line and so the listing below corresponds exactly to the 32×8 arrangement of circulants in the generator matrix:

```
00000000 00000000 5E861B80 62F7A3D2 9D1E9317 00000000 86E017A1 7E6C98C7
00000000 00000000 398EDC90 6CAD4404 68202365 00000000 B7240E63 F1B22454
00000000 00000000 2D958121 0E74EC93 A7649873 00000000 60484B65 1640BDC1
00021000 00040400 0F76EA88 8470EA40 875284A3 00010000 BA9203DD 5A1B0DCD
00080000 0A000000 44BE3BAE D17B86F4 DD77A68B 00800000 9EEB112F 13462E1C
00000000 00000000 4961A410 0ACC2686 61343056 00000000 69041258 55B2397A
00000000 00000000 BDDBA30C 9A26B8C2 358614D1 00000000 E8C32F76 41A7836A
10000010 00004400 A8590524 8E42525E 1292FCF2 00000100 01490A16 12367171
00000000 00000000 1B731B73 51E75165 2A8B2A8F 00000000 C6DCC6DC 59E549E1
00000000 00000000 15B44EE3 FAA29812 14C097D5 00000000 13B8C56D 9D1CF56B
00000090 00000C00 DE02C8D4 C12C3860 61C30789 00000100 B2751780 38F98DC1
00000000 00000000 E8BA1A95 764CAC29 6D614BB2 00000000 86A57A2E 263BF6A5
00000000 00000000 A6A9810E 4B90F883 C7C41A5C 00000000 6043A9AA B7E5CEC9
00000000 00000000 DDA3B50B 70B4E408 A7204385 00000000 ED42F768 D1813839
00020400 00800080 0E4DD78F E8D4C59A A63CD75E 00200000 71E3C383 1EDC3687
00000000 00000000 46D2BF00 2DF50E22 E871116F 00000000 AFC011B4 FF9319B5
00001008 00044000 742320CD F872B399 959C47C3 00010000 C81B5D08 F1916947
00000000 00000000 B226BE3D 2C382EFE C1777161 00000000 AF8F6C89 16B04735
00000000 00000000 A6B72685 F34D5828 6AC14F9A 00000000 C1A169AD 8A11F36C
00000000 00000000 06785F87 5B8B1A20 58D10ADC 00000000 17E1C19E 5321E613
08000000 00000002 DFA20208 5A923BCD D1DE6AD4 80000000 808237F8 D59F5128
00000000 00000000 D32E39B0 8CDD1F14 ECF8A466 00000000 8E6C34CB 2FCEBE03
00000000 00000000 BBF129E7 B65C93B0 E49D84B2 00000000 4A79EEFC C1A172CC
00000000 00000000 AB08ECB1 52D778CD BBC66A94 00000000 3B2C6AC2 A6505FE7
00000000 00000000 560EC6E2 DD36F786 B7BE36E9 00000000 B1B89583 6F607C29
00000000 00000000 7609DF67 D263EEF6 3F77B693 00000000 77D9DD82 249B5842
00000000 00000000 CD2FC38B 06C0EF2C 07796036 00000000 F0E2F34B FF081991
00000000 00000000 4135B4AA BCCB0893 58409DE6 00000000 6D2A904D E6D1D5C0
10000000 00000000 986B9059 080D0023 68011848 00000000 E416661A 1A0A2B4D
00000400 00000000 F8E1898C 7EA836A0 61B503F5 00000000 62633E38 70C49CE9
00100200 04010000 03717225 9F4987C7 4CBE1CFA 01000000 7C8940DC 72729C94
00000000 00000000 CF7C7758 681EEC5C F762E240 00000000 1DD633DF 7989FAAF
```

Generator Matrix for ARTM0 with $K = 1024$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 2/3$. Each string has a length of 16 hexadecimal characters and represents the top row of a size-64 circulant ($M_L = 64$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16 × 8 grid of (possibly) dense circulants, and so there are a total of 16 × 8 = 128 circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 4 will fit on each line below and thus the circulants on 2 lines must be concatenated to form the 16 × 8 arrangement of circulants in the generator matrix:

```
6BA7892ABF45236E C8D9C5D3714F302E A4FEDBD333242FE1 C69274DC3165711B
D4F0D22000C799BE 1B735D3C4955FA29 58FDC14B6FE48B0B 236E6BA7892ABF45
F4B95DED57220E46 BD8F352AD210C8A5 F7360EA75265D654 F0208E8490FC7DD6
DAF547D1FE2B5F06 7237A5CAEF6AB910 3FCFE4973F6235A6 0E46F4BD5DED5722
D51E3B00A8FFEE45 57602D7BC5290849 84D318C9B44866E3 B2D2F110A7AB5707
BD3B4E1E60DEB9C0 702EA8F1D80547FF F226BBF81D1508D3 EE45D51E3B00A8FF
32831FA3319969B7 9D50B26FB2A2602B 1671FAEBEDCA5F97 748C762F17B3AE51
024397946AD8E25C 4DB99418FD198CCB FA1D71FFDC395A45 69B7B2831FA33199
A42BE214109BA706 8FFCBFCF8011FE66 09B56084CC257EB8 2F5BA06FC18C2CAE
5E5CAA67BEBAF6A9 3835215F10A084DD 2E91E38A5FF8387C A706A42BE214109B
035EFF65BF4094C9 8339C019E6D5BE25 08686F2A40594CEC 30A665E3031C1A9D
0EBBD74043E22FD5 A6481AF7FB2DFA04 C02AF41602DB0020 94C9035EFF65BF40
F205E535CC6D740B 44EA142BED0781C6 6D8CEF1209E1AD6A BF6AC908CFEC4E40
E5E47A8B6E7D5A43 A05F902F29AE636B EEB087A2544A4A91 740BF205E535CCED
7E617060859E0216 755D5592B595A56A 0F639EDAC0843854 AE99BDD2590C2B26
071DE3FF5219D499 10B3F10B83042CF0 999203D71678EB5A 02167E617060859E
7FECF14CB1461C2C FB40539616D639F6 DB514B0EFE4BF00D 410CA3F48DADED38
2A617C3B30F36C5B E163FF67CA658A30 37FF7FD3CBFEFF6F 1C2CFFECF14CB146
0F39A3A0E16C8001 9D399E4FE164B196 97C3909EDFD6C24C DFD7ECB745FD8010
416D8CF6C4717A9F 000879CD1D070A64 AC86ED044B848C01 80010F39A3A0E16C
1B18F05796B23435 E37424A6DEA74597 3D8C3F9209ED786E 6F5805F8FA78A132
2F8A69EBD71A8359 A1A8D8C782BCB791 75DADDA46B7C6EA3 34351B18F05796B2
9E8455D318EDFBC2 439C019D6E1E540E 21656FC6E2D36F80 CF39D2A910B0F67B
9DDDE2E7DE238B05 DE14F422AE98C76F 35B14EF1A129231D DBC29E8455D318ED
74F242AC3961671F 919C2479E8AACE7F 2D5259894EB08F04 B3F8A38D0F31B1A1
05B9CED1A6F57595 38FBA7921561CB0B A4F87C0A9009808C 671F74F242AC3961
F0F05A3CD8B225D0 508D96E0FF1E28EF DA8008834758C21D 4E12874B59F54ADA
7414E91CBC43B4AA 2E878782D1E6C591 A91F53F896F985EA 25D0F0F05A3CD8B0
0489F061DB142B23 64AAF2E3926880AF 5FC523D281F7F49E 6C6EF4742BF2200F
FAFFEE8A88F07DB0 5918244F830AD8A1 6D1F7982612A7780 2B230489F061DB14
37BB4EE0D8AEBE2F FCFC5A3DFA679258 CD816280D604E31A 9E63C2CAC8CA0706
C90A457AAAE3A065 F179BDDA7706C575 4BF4D57C5F418F75 BE2F37BB4EE0D8AE
```

Generator Matrix for ARTM0 with $K = 1024$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 1024$ and $R = 1/2$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8 × 8 grid of (possibly) dense circulants, and so there are a total of 8 × 8 = 64 circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000 00000000000000000000000000000000
389530DE4D0FC37C8AA03BC924631F5A 00000000000000000000000000000000
CB09548AB217FE1DF7D809C281FBD3F9 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
DFCFD4B5D3909995D78CF09F2EE26A50 00000000000000000000000000000000
2BE0960D0F522ADF1A3135EF6B8F8F4C 00000000000000000000000000000000
00000000000000000000000000000000 00010000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
CA96AC7769CD9B8772C88045089B9EC9 00000000000000000000000000000000
E7B94456E7796926F69616A20FA0A14A 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000004000000100 00000100008000000000000000000000
92C2025B6299B0086E4A6E0A3F9F4B85 00000000000000000080000004000000
AEC528FE0312115B371F197B4530124B 00000000000000004000000000000000
00000000000004000000000000000008 00000000000000020000000200004010
00000000000000000000000000000000 00000000000000000000002000000000
5AD2664E284A6A42D11629DEA5E7E2DC 00000000000000000000000000000000
ECD97C825D2D132B87D58ECF02865834 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
9E8C6295B21D689A7A49E47D907FEF06 00000000000000000000000000000000
D77F0A0458CB2A6E29B555C020E9037B 00000000000000000000000000000000
00000000000000000000000000000080 00000000000000000000000000000000
00080000000000000000000000000000 00000000000000000000000000000000
292B5BF488960B2CAC48AF5B8995A64C 00000000000000000000000000000000
204D121A214A05C28DF2F3594841741C 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
00000004000080000000000000000000 00000000000000000000000000800000
5AF76676984C9AD181000BA4C4B9F9BF 00000000040000000000000000000000
5F26B286448924C506FA0E08E6153CA7 04000000000000000000000000000000
00000000000000080000000000000000 00000000000004010000000000000000
```

Generator Matrix for ARTM0 with $K = 4096$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 4/5$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32×8 grid of (possibly) dense circulants, and so there are a total of $32 \times 8 = 256$ circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 32×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000  D1EBEECCF19DEED384BEA5A4F03325F4
D3F076CEEBA41F653F0896B2D6572794  37C6A13EE27C4A4DD308DDC551390ABE
00000000000000000000000000000000  75EC556940AD93B5A3E87E571D3477AB
00000000000000000000000000000000  32E28B86CE97D373D6DF37F0B34E8A10
00000000000000000000000000000000  00000000000000000000000000000000
A000BCAA5192184E332429251B6FE816  24CE88207A44BE1B83027B9CC39F36F6
00000000000000000000000000000000  B219FEA388A5F22EB155948624352715
00000000000000000000000000000000  B5CF0CEE58550719FB46DA2962ECC0CC
00000000000010000000000000000000  00000040000000000000000000000000
B3D5DA68D7AE2FB13156CE322E34C45A  B4C5FDFF6827693E544020ACBBA067F1
00000000000000000040000000000000  6E973CEA2BF41E9B015FBDDB19C464B5
00000000000000000000000000000000  0BAA88B5C097BB0992396D596FE688D5
00000000000000000000000000000000  00000000000000000000000000000000
82F3C94F4882EEF732AFC404C26C2358  29F782F9BEBE2A7C27FC3E7B2C39C47C
00000000000000000000000000000000  5CB5A74097FA9CC402C3F862BD049CF1
00000000000000000000000000000000  86429B17FDEB9CA2390AEDBE8BB3BB56
02000040000000000000000000000000  00000000000000000000000000080000
85B6097E4A80A594926E391F4DD15F73  25F9B4CB0245D29F1097E5D27AF67D76
00000088000000000000000000000000  52FAE325DC384CDD7D952C0505FA5355
00000000000000000000000000008000  5DCDFF94DBACCED7136E2E6C41F1D906
00400008000000800000000800000000  00000002000000000000000000010000
EFC07C231FFB53308E9134D3F8EA8CFD  F0481976F76EA55E1509E3B75D5171E5
00000011000000000002000000000000  CDA361844CCA2B002430F0F628B1CBFC
00000000000000000000000000001000  7150B7769FDB62DD6A6554357199A044
00000000000000001000020000000000  00000000004000000000000000000000
04754199F8AE26E4CAA8B0A414857EFE  B70A39F8C2C86F86B72AAE7ADC8CFFB4
00000000000000000000044000000000  2803DA3DEE52A933DEB7CB4D774D5F67
00000000004000000000000000000000  AEC857E97490D820703121D7D905F8E2
00000000000000000000000000000000  00000000000000000000000000000000
F7F5F0D5AB1FD7E01B7CABE461474872  D69291A479D67A90E3A74487C18222D8
00000000000000000000000000000000  25919002056C03444B965B618C7CE469
00000000000000000000000000000000  BE3FAF76DE2AB22E3F84A834FDFB4B4D
00000000000000000000000000000000  00000000000000000000000000000000
85D4A59FA181AFAF1BA87B6FA649C7B8  AA0A3E2722FC0F2D2496343BE8002BDC
00000000000000000000000000000000  EEC8D727F5F72BFF5A37A761FC6FFF08
00000000000000000000000000000000  E4BA815DA1352AE6720B08FB268222EA
00000000000000000000000000000000  00000000000000000000000000000000
AEE6508B8BDB5DBBC18B01AB33EE6963  34AEFEF00921F7BF70537F2973F9C68C
00000000000000000000000000000000  09B84E25C7B424B9ACB4943D70C81867
00000000000000000000000000000000  CC2029576B2C14CF7CA19BEAFE5472B8
00000000000000000000000000000000  00000000000000000000000000000000
BF3765014F9DAD35D1A2FB464B549458  DE0532B38DA2B5FAAC84BA3B4EA4F937
00000000000000000000000000000000  E904C7CD0FE97241611BA76EA2614253
00000000000000000000000000000000  B0B4903BE6F12CFF7CE505959F28CF96
00020000000000001000000004000010  00000000000000000010000040000000
006FE9B3E826B20E7D14C98DB7804DF8  101F4C4DB76AB849BD36A984E1454688
00044000000000000002000000000010  B624635A5B547DEB2615D6EF33DE6766
00000000000000000000000004000000  70A75E72750974671C17C415486B55FD
00000000000000000000000000000000  00000000000000000000000000000000
7E8CDFA0A8303622F85632C5493251DE  D35AF586B90AE5E869552BE1C32246DD
00000000000000000000000000000000  8FF36572B1ED1C3DC8099FD692F36E25
00000000000000000000000000000000  2E40B8BAFCAB7C07038FB4AC9D8CDFCB
00000000102040040000000000000000  00008100000000000000000000000000
99EA77CC438678BD48EB3922709487BE  32C44871D606DE524524E573751A0119
00000000000010881000000000000000  1FE011904A515B4D182932795DF3A42D
00000800000000000000000000000000  6BB04E770010EB2F421E080D6C88F26E
00000000000000000000000000000000  00000000000000000000000000000000
F3B51C5AEE1746369D0255FA48A2C8E7  5CD1B1E9FE5DE15C5B7CBB52D6AE3D97
00000000000000000000000000000000  0FAF046C6D61DE93B05D878E6F05E5ED
00000000000000000000000000000000  9521BD40F3AA3032AC8EBA1164AA04B5
00000060000400000000800000000000  80000000000002000000000000000000
A9E9B705E9450D792D946C3D78895559  00D53FB7A1CD9C0DE393D4F84B56E4CB
00000000008800000000800020000000  F30BD8E868CB46B30B3BB3F7BE0177E8
08000000000000000000000000000000  109655C53D423E64F55043816ED8C2D7
00000000000000000000000000000000  00000000000000000000000000000000
5A0EF6E2A13354869340A82F068B4550  281A281790ED8C10435998F009A5E314
00000000000000000000000000000000  FC4B6DA9C6045B5990480AD257621A01
00000000000000000000000000000000  0F932F56402B7E9B1A1ED30DC17DA4D2
00000000000000000000000000000000  00000000000000000000000000000000
323EB09CB1B85869997867167627C209E  3F6A3889E4CD6E0548C9F8A2427EB120
00000000000000001000000000000000  66D834CD5EA38AE834892473B5906884
00000000000000000000000000000000  E7788E8389801749C149BB2E482B7842
00000000000000000000000000000000  00000000000000000000000000000000
064A0A9B0798E8A0E4E5C2030AD16FCF  B4A1980396D7D4B6570FFD7E8D1F64DA
00000000000000000000000000000000  A92787A38AA78D98D4618856EB68B1DD
00000000000000000000000000000000  A646F33E3AA3D115D7F7521B4ADDD346
00000000000000000000000000000000  00000000000000000000000000000000
1DDA232C8D404D394462FAAE77FB5216  3067075C587F41655E21C6B5C1E9BB0C
00000000000000000000000100000000  3AE7D7AA9B8D63A6433B349AEA5C530B
00000000000000000000000000000000  234F99E3899B8A0906B3FFE761CF8918
00000000000000000000000100002000  00000000000000000400000000000000
E12A9CA99F84C07511C5C5044DA6E17D  EF103D410EA0ECAEE16DB366367A2C9C
00000000000000000000000000004400  69B7718D8283DF05AD138830679673F3
00000000000000000400000000000000  A71845DB44F886BE2568CE7FF906059C
00000000000000000000000000000000  00000000000000000000000000000000
F3439BAADEDF4A8329BAAF234A78177A  568C0F07C313E08132F84E371F073B73
00000000000000000000000000000000  39499863F3520A673CB90492F49F3126
```

Generator Matrix for ARTM0 with $K = 4096$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 2/3$. Each string has a length of 64 hexadecimal characters and represents the top row of a size-256 circulant ($M_L = 256$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16 × 8 grid of (possibly) dense circulants, and so there are a total of 16 × 8 = 128 circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 16 × 8 arrangement of circulants in the generator matrix:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000020000000000000000000000000000000000000000000000
A75774F431955B663D85135AB354C349FBE71E889652FAE5E214A4C515317D30
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
61BE4CCE4B833D25FE309F12C7252C48C622158BF93F53D18BF6617E6ED06DB6
0000000000000000000000000000000000000000000000000000000000000000
0010000000000000000020000000000000000000000000000000000000000000
0000100000004000004000000000000000000000000000000000000000000000
EC5C59157943A5919CCF0D4DB48A9E9288C12EE8CBAA54D74992F84D4B735615
0000000000000000000300000000000000000004000000080000020000000000
0000000000004000000000000000000000000000000000000000000000000000
0000020004000000000000000000000000000000000000000000010000000000
62F62D6898D4D19397FCFE53C2DF200985643537FAF0FD7A0363D9EDD0ABFE8D
0000000000000000000000000000800000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
AAB4D8AEAD2B481B51CE3EFBDFD61EB792660C6471FB1635B202FFFD02591910
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
F8F9A6A2B24458FE26C9DB32CAC5672FFE52693F3D089C9ED896CF9927BDEA90
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000010000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
28CFD1D92CC318EE127E323288B671B2D022B3273D823ED50E58B12DEC516510
0000000000000000000000000000000000000000000000020000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
F9ED263B36E4BAF4A5BA1AEBD80AFBA2D2D697DFEF41CE278A3077C2E1CEDBD6
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
2208A8A02A020A88000A8A222AA0A08A80AAA20822820A88220002A0AA88A8A0
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0501515551505055010551015411514141551041545110141005504114054114
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
B24A635C86E1CEAD55FA841230A3DDE39BDF5A7E929E2B3CF5D7B50EB56DAE4C
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
833A7B117878F028CE675B3E8570B5B25D1160DDB4B94896F77190CE9E4FADE0
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
89185181F774A984862AF7F84655F6BF240D80CF97FDEF630D79CBB304B21A7D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0A5B6D5E76CCC3DAE01D2D7C831048326C2A5620EDC97173E3D3FB6875DBDB80
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000010000000000000000000021000000000000000000000000
8040000000000000000000000000000000000000000000000000000040000000
3E35EE32B54E4F4F41A2174E762E76D22E7163DAAE54561E6A4048F25E8FDB08
0000000000000000800080000000000000200000000000000000042000000000
0000000020000000000000000000000000000000000000000000000000000000
1000000000000000000000000001000000000200000000000000000000000000
CF395E22F960E846C2B505ED59F6DFDE22CDD70D2F99DF8A45F30A7A6C6D14A0
0800000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
89E670F7ED565E4F74D1425CCE0BD1993F325D0CF8A5BFDF5C62E3F156D39F12
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
E92FF6F0501A20667DEDB73E2B7D11C005C2300B173B2132713E550ACA6213DB
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000800000000080000000
0000000000020000000000000000000040000000000000000000000000000000
5F0DA5BD187E286A3FE665819F97640A200DA41FC7D1CFC9B1A7F0639F5C3A30
0001000000001000000000000000000000000000040000040000000000000000
0000000000000000000000000000000000000001000000000000000000000000
0000000000000000000000000000008000000000000000008000000000000000
D29674F92C06B305FD34A22CD7ED395111E47B13815F74B3324F8DE7C085FA5E
0000000000000000040000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
454959B3301F3F470A7B7E834FEB005BC5EB5939BA1D9F458A59DC21E549AA53
0000000000000000000000000000000000001000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
A6C07D876F1433DC8E9CB7CB0656F94DB38569876E446689CA9CE68A0743B85C
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
AA0282880000200A800828A28A2002A282A82A800A20A0A2288222002202A222
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
4541140001105504154045515440044551501445001041514511015444544140
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000040000000000000000000000000000000
0000000000000000000000000000000000000000000000000000010000000000
CF856980DB570B505641914F5AA4C48A94C4560A85F973B5D9270D6FB970B50D
0000000000000020000000000000000000000000000000000800000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000040000000000000000000000000000000000000000
31C3CB6ACB659318C27F09D2B393314FA338B09F7127D14FE6F8386F26CEA186
0000000000000000000000000000000000000000000000000000200000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
A5F576269DDDBD62744D95828856EC686B9CE7AF5145CA396D03A5ACBE7A8555
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000400000000000000
8082F1605E48AEFE45B480B42443353BC49B23F2B6C92311995BAC64067F24F8
0000000000000000000000000000000000000000000000000000000000000000
0040000000000000000000000000000000000000000000000000000000000000
0000000000000000010000000000000000000000000000000000000000000000
210659FC173F50CC374E2F11DB1DED2DBB7A0FED2B73643908D29EC6EE5456B8
0000000000000000800000000000000000000000000200000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000040000000000
8A42250E1904032DAF947CEC568220DD174E6E82D83E271B6C7E6DF2B5F33651
0000000000000000000000000200000000000000000000000000000000000000
0000000000000000000000000000000000000001000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
A56EF3B309174174147BDF113C2671791A82E98B8D929106B989647010161E11
2000000000000000000000000000000000000000000800000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
7F11AC4592AE3218A55CC9BFE00018F995A60DEF9AC41186BBC6DFAC3C40AC8B
0000000000000000000000000000000000000000000000000000000000000000
```

Generator Matrix for ARTM0 with $K = 4096$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM0 (PCM/FM) with $K = 4096$ and $R = 1/2$. Each string has a length of 128 hexadecimal characters and represents the top row of a size-512 circulant ($M_L = 512$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8×8 grid of (possibly) dense circulants, and so there are a total of $8 \times 8 = 64$ circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

19791B044CFB6B41063327AB9322DE9CA101D1D6AC7DB37B7A0FE4B4D937241A681C7AFE75D1590B668C41868347048B0E059CD6AAD9D544F8214624D7FD8112
F88D7BFC481D29F5275EB2DC0EAFC647850EA397589905C7FF4E326EE0794BF1416B5459A79894CB64D6E3BF7CD8DE92AE332CC9848FC23AF2561F6B6468CAE2
8518935FF6044865E46C1133EDAD0418CC9EAE4C8B7A728407475AB1F6CDEDE83F92D364DC9069A071EBF9D745642D9A31061A0D1C122C3816735AAB675513E0
E706D2459EF9A1976D3C291D515203E93ADFD520D5F186CC51D2FA7580444BDD0112E2C632C6628DB1DB572230D32B83AF57093AB292CAC09B49D1CA04A95234
234BBAC1C6AA5A91DAA5E0D82511160B5BE777034B281C04B08B503B99814D69A8A3B7C0958910BDD9E5EB48190617 8DBD73300BEA80997F3139C87D05EE9E36
84C9ED4CF9E6A64BBBCD02DB8B755576C12BBF7FDEDF7FF0095F4851E3CD87C08D48D0399503F26A4DF39A1B9A065E1E6AAB60CD2C9229AD9C3AABC2E7DB1B7F
867B92BA264E720E05B7439384E34E80563E678E4B8D8649D2B1128C396222B9202A6CD8CE3BB1BEE7A90CA006F2FE469699FF75656A9E59CB9CCEC5783D0203
C9CC46F3C675E28DB0158A958083BE68F94CAA2FD8C2D24EE3D3134A3652885960DDB6FEA463B6631519B506290207613AF26E5480D89E97AF65F46FA49703BE
CA6EB1C34D3AC540F2BBD2A9AA0D47C242E12AE7469E2C94368758A496B5CE2E0AB856799A20A6A14BADCFC064A36FCF604FB2C417E946F0299A729EA46DF05E
8CEA72A7997752B3003379E9E3C8AAB81379DC073F7D10880F177BB9830F42F4264A647CD7F45B2A163717A21A3778811AFAC2284CA01AA90266F4C1E25CF889
69CA7A91B7C17B29BAC70D34EB1503CAEF4AA6A8351F090B84AB9D1A78B250DA1D62925AD738B82AE159E668829A852EB73F01928DBF3D813ECB105FA51BC0A6
7981222CF9766988AB7801EEBA81B6663D427B5D5380F425339E52A08D9A3EFD6C73306EF7BEAB0F7143F7995AC2DB0E10DCA29FCAACE2C230AEB964C808BDAB
CFFA75F78D91E393A3AB71E4129C5E2E0AEE93E6F21EBF1EB23353DA52D2623EA8BF0620FB8B186D377BDF123D508AEC89216CBA3A47A3A69BEF59C812F4CB37
B3A322C8AAE11A7C54AFA88BD2170F1E1F171099FDA231C516E90D9ECC911923B5E955C550E92CD2BBB3C585F506DAF8325288F90BB9BB1C4641A5C810EFFC5E
584F19B16537B900FDA36961B291A656058D26450811D551F3DB715C06A734185FA088C4A4D216E516204B589F46C4D231FA1B588D5759BCEC650D66936B237D
69131806BB4029C8A1524DCE6CCAD11A3BEA2C812A6620A1DFCE3125A580C052491B669525464385688AD0BA62CA2ECEBC6EC7E3F86D391F56EA6FACC711BBF3
AD238FA8ED1ECE29F82FE10CF6C57D12CB1D1B4D97757DD826F4C55084376D5038CB97A8CC0A4B7240049375BD56A829373005390DFFD323D0BA3D1E73BEBC01
640AF643C6563F923D0E3C47F5171CCB6CFA82B78DCCFF54FDBC13839E5D213DCAAB556C34F694A7116AC51ED1894923CEF33BDC8EA5C91F213EEDB1F29796E0
E8F479CEFAF006B48E3EA3B47B38A5E0BF8433DB15F44B2C746D365DD5F7609BD3154210DDB540E32E5EA330292DC900124DD6F55AA0A4DCC014E437FF4C8F42
C749BEA8FD377908983531D9588E1193F63F17E6AA8DBF6EB126AF12269E16B09BE36F4A6AEFCFBC713E1DAC46BE882995DBEBF8371E852F19A83BEB8A013EFF
91BFF3D08D3E74445B13A7E2717030604B6F775A412960766E5EAC02A81DD93558F33266573219D36BE572D2C15B8CC406BFF5C08B6185A7420D804DB72A310D
4B0466FAFF477DA55D6D8B531F89E79E4521C4F87FC247B78BD4D19161566FA4804C872918AEA8E64B9E1B17CC21C29240B6F9EAE26DD5251A5689E9B5564FDF
D3312503510D79C9C37980EA9CCA0E4E908FCB417DB422A777E1431C53FF342916DEAF39C19B78A47BF8E535A02ADBF03E281599D82E8BD08B99EF5E01EF0791
4CF61DBA53E88CE119D5D4A67B2D145F4731A0C07D55308297F5BA82663F4B7FEB663D16D875D956071847F30C62C1E19D5631183505A54011567033A7C1F304
CB7E401FD157C9B7514B9C386E6BD00A5C73D0DDC1498499ACB75564EE72AB1CDE7703F997D6D38B9FA07E211189EDC2483BFA600AA690B6ED589249DCDD38E0
7491BEA4265A4ED15B9CAA80A605C895D0DF58B9B3720D5F381B7A62F50DD5C58A9E3A46C4FEC5C584D490328D8643689DA185285EC45320DBCBC1477F71F022
6249277374E3832DF9007F455F26DD452E70E1B9AF402971CF403705261266B2DD5593B9CAAC7379DC0FE65F5B4E2E7E81F8844627B70920EFE9802A9A42DBB5
25DCC52DA7DF1AAE5604C9ADC79D4752AEDADDF1B4B07BC9391930B9CA5BD620435CD2823DD2CD9459414160DE3B877793D14CDF6B5290061D41EFE9477C7951
4311817278D4091FF86EA3EDAC38D83AC403866F72E7EE0E7D4AE6C120EB587A70C3A71B3C6C0DFD95C4F12BF4EF8770A4CB9AFDAA89D785520EE901DF769B7A
222521D28D8C8DF8B094CFC92357C4C3D60E5D8405F5F9A36A7281979DBFFF04783860C881AAE49E371A8393A653EA288497182F6AEF070E9B06E470E1E684FA
54A3D0C8976B2C3804C737601A3B4DBBA28EA921B8DD3BFBA28D2EB166B92EA9F880192BF0BD0E8D97A852297F048127990C9F6CC5B8681742CD592D68BA7AB4
992E8EACF3ED2A9CB1AC7E6628CCACD8E646FFB99621A126CF7CB62BADE4FCF4D89544F6E1A6579715C9BBD31A6A037CC3DF1AC398199A856E053342F341E0C1
B31916661443 8E74D0D204DC9C31650ACFCFA121595D3A7A6D8C22ABD1AAD28411A48892681E99BACBA4347422ECBE27C84ED0C84BE439AA5D0E5DBBBE45913F
BFB36D2B31FFFB71F39A6725F99E7DC16B100C96E3C577F91BBF43ACD0F770553687C4E1C160EA4A2773EE04CE44491C20886BBBB606DFDA79F701C029E127FE
3976EEF91644FECC645998510E39D34348137270C5942B3F3E84856574E9E9B6308AAF46AB4A1046922249A07A66EB2E90D1D08BB2F89F213B43212F90E6A974
0E2C9950366C6869E15299B978BCF29958E948F8B50A3714897A4ADF3D16CE2B7D8FC2A67715AEA7C1A52D7BC51D115E12A982FAC2FB59A4CDB9F7E9171B52E1
875B919085759B693D1DCB97C357D2C88BC2FB5812108CB48756CC0BE4F08AD87964159AB5932C2A8CFF493C210688AA3E0B23297FFF317448378EAE6475C045
6FD7637C7F0F487894A56DA5EF7D503BC8B26E2DE1A8567EC6D9A9E5E024F50D91C907BA9498B46D50BB9A4468CAAF23714A355FFCDE534E27CE2E431BE215C
489E69F6FAAC512C0862B66B8D8CB720ED7D32B8C7EDED5DDD9E72A7CEBCF5DA3E412CB38582F8666F4EABF96326D75ED66CBB9ECE01E8D95D17338C28A0346F
370AD127A29EA98BEE881B3184E67CF8BF31F94CEFB0329A2B16DB810D49AF51E15024F1E4137E0F9706E7E92F3E832B9E15E1A6DC7C236D68E66928F8901639
233C3E13196B1406DAF08EF2F7CE102E19B1FF2AA5DC897C77642422E32D071DFFAECED41061AE660F97C942463ED84DFD3CD034CACA02B03F23E24ED0B06F7C
950722C47449EAF324D125FD26C6EC503BBA5AAA3ABE034C5D6EB134DE45AA423D89F99C70FF28FB9D4D1241D650814F4B8F7EAEC231461B7881E42222647E1E
8F893B42C1BDF08CF0F84C65AC501B6BC23BCBDF3840B866C7FCAA977225F1DD90908B8CB41C77FEBB3B504186B9983E5F250918FB6137F4F340D32B280AC0FC
44A00D5E7F4F6522B47E6B83EFC4B8B92EC57E173498B9150DB425802DC6177FA76D31F2F66F0894B73378840F573D0B1487AEAFE5956CA4E8D02A0544206C8D
DD8271FB6F01EC64125A2FF3D16706CF383E69A6186FCFD32B65F775FE172C4CC8A88A2861FC5329244B3ABFBBBE7729813B9F8CD49A13548FE74AA6D4077DAC
9D0706D4D0F024C176EF9B51805831B4123D154D10094C9F2E9A9A57FCF4D8A5B3B58DAB3B013D54F628D9C7D9041536910CC81F359098E62676F97A72F93CDB
B0318A40056C31DEABE60C87AE2FD848434A3267338694178759 5A55466FC4836801F941EC744FEDF213C70353254446FF7100A6BFBA79E504E9CD590E70326
15ACCC9BBEA95C186FDF43DB4C63CB258EB0F25ECA348DD8205865B43BC08D59F5A21723D4082CB2586296D8BC4D0E37DA95823B5921BA36EFE567588E5DEE7C
F667E9B1AB13318BD250BBB8B8570C6B5C585D867CE368880C1C8643E345DA21AE8A5CD76F780C9CD0B01A0C861324B43454283697750ED0D68A4FFEF05F4F33
6FD5FD0120D2F0CC306056346313E7F6249DD06A5E1AD6E31798EFC632E561B84749756876DF9FA4BA0E4EB66AD59467AD42D8A7343D7C2F698D69D17C24B9D5
293FFBC17D3CCFD99FA6C6AC4CC62F4942EEE2E15C31AD71617619F38DA2203072190F8D176886BA29735DBDE0327342C06832184C92D0D150A0DA5DD43B435A
EA7372464235F50E83A4F95F141A871ACD5BF99D46FF2E1F8E64910C95ABE220F8141D32544E733A15569933E8EDD61CA3F703BDA5C18DDC3EFE3FCA85407669
A11BE7A2E8FEAAD3284E8D090FCE1120FEE72D21FDE19A0595054D0C0A4A3AD1EEE990D842656222748061617012928CD1A2137003F94B0FF79B0780577CAFFA
4164EBD2013A5DFF7E3C17AE4C9CFFA4B6EB497D2A04432A30635EEF0381BA4A79564E5EFF58EB52E3731BE0FD0D21A2992745C0CA764ED7738A27C771D7EC85
0F33ED0AAD00BFF39288D4BFDF7389F1DD088F5D39F702F2854751FEB4FC1EE03DD6FB6B66341B613E2172B94CF72AE87AAA9E970E62E5E08DD2DD3D838CC667
80D158BE0C2DC2BF544D48034EDB894B5FFC9F9B2B82E84A77B333B08890A050C63EE8D4EA648DF5C74DF9A90F7730F9DADF0B7FB1A5B8BAC0915D209DFE8B17
87C69FB41C6112B007C776764BC8E29B471F5DEC81A2D5E3EB32ED89EFD83C0021A898BDA8ODC222BB670366DED7393F1940DFF9AD62BF75718EEA29F48BBA8A
226602BA8C872C6FAEC50CD4540BCB74B9DEC9BB92EC37AD747682E8F4AC428F48ABF6C0DD132D50E36A8BBFB71A9ED54D99C3A2B5E54F27A3235BDEE6D3ED
3BA8A7D22EEA2A1F1A7ED071844AC01F1DD9D92F238A6D1C7D77B2068B578FACCBB627BF60F00086A262F6A037088AED9C0D9B7B5CE4FC65037FE6B58AFDD5C6
E06E6CF5D96BC652E03A62847077889E9B58297F3E951ABD75A8033EAC6ADCB1E3E421AF7E1768176F27FE9D79600555CA42AA62841CCAE63136101DD1B3090C
393FF55DC3F08E1B6D3F3A29E645F50DB11C3DB11BCD95BA6F92CA4AAD7DCEEDB214FDC163569ADF331CDC089DA724B750F4778DF5151102DC2470AD2E25DD19
77827DA8689126825F70AC401BE804BDE5FF6FDA1B4557708A77BD03062B20F6580A15B4B6F7B32A87C302A4F406C377BB36EC4E32BD321425CA02763492686D
1F1CC740B75393BDD3656BC72D7DD9F44B35014DB99B6169878007B70DA99E8638C733740A3F6EEA3A21688696F3F825311F9BBDC76916C00DB2998F03D4509
86316AE71C20632C2EEC22195656EA66B1A5524AADB6771BFBED7CB98E01EE56DF138C478D2439B3136E66B7D69E2B66A55FF3327E9C575DF0FC08918C1EEB11

APPENDIX
GENERATOR MATRIXES FOR ARTM1 (SOQPSK-TG)

Generator Matrix for ARTM1 with $K = 1024$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 4/5$. Each string has a length of 8 hexadecimal characters and represents the top row of a size-32 circulant ($M_L = 32$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32×8 grid of (possibly) dense circulants, and so there are a total of $32 \times 8 = 256$ circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because the strings are so short for this code, they can be arranged below with 8 strings per line and so the listing below corresponds exactly to the 32×8 arrangement of circulants in the generator matrix:

```
88B19B9B 92B86179 44F9DD4B CD94CD94 DD4B44F9 C5CB4395 2583EF49 96D2F33A
F9269673 823AF46A 41974470 AF382F3A 54704197 ABFA1031 C90C7594 23B69E4D
96E8CB40 F067EBCD E6EF36FA 80AE80AE 36FAE6EF 63615E7D 73070276 6175B096
D6175193 FD27CA3D 973220E8 BED5BED5 20E89732 9B5E11B1 B21FE944 DF3E71C7
AFCE5E09 05FF7C7C 83EAD943 D54AD54A D94383EA 71054AC8 E6B01A4C B43E09ED
D21620F 3A30121F 4DAD2716 55DB55DB 27164DAD D04BDA05 8C56F822 E9174DF7
61ACAC7C 55B0E37E D10A55AD 253C253C 55ADD10A 24D08E15 FE1F36D7 D31A7F44
F02E3BC8 64B5EDCD D1B9CB56 D778D778 CB56D1B9 EAA9BF12 A761DA1C 16F04D45
6C6E90C8 8786CC9E 8AFB593C 9E3E9E3E 593C8AFB FC1DA940 996D9C68 DA958B40
5EEA5012 9475734A 19A00415 EDA8EDA8 041519A0 2644FC4F 04432E69 B03B1084
E9C79F6F 7EECF31C A54A8AAE 7FA17F01 8AAEA54E 4C26C092 0902FB71 12D3899B
9D89B1FC EAA8FF1E 02BB3823 D4C1D1C1 3823023B C4CCE417 C6486E40 2C26A196
89B96E9A 348B3764 B9719426 6AB96A99 9426B975 F81649C3 6CA7F238 DDD1908A
895A50FD E04AC630 1416E302 B8A7B8A7 E3021416 871277F6 784580BD 22BB2CB4
988475A5 A4D05394 FEC5D306 6E196E19 D306FEC5 2EF67552 B57E0EC5 E3D05995
E1DEB91C A70E35F0 EAF04EE4 20A520A5 4EE4EAF0 A06CC3B8 AA8386AF 545C6D1A
49A2C832 BE460706 40A6D659 B7FCB7FC D65940A6 AFD2645F 23C6799C 037DABA1
819260EA B4DC032F DF76E66D C8D9C8D9 E66DDF76 E4EBAF6E 22ECAA64 225E980A
5EB2E1E3 DEAB15D1 60ADC0CE 031D031D C0CE60AD 6A7A2FC4 43ABB35B 292372D7
B3C1C92D 32CB567C A4EFBCF2 C0E8C0E8 BCF2A4EF 920342D1 D39593D4 545459F9
B7C35BDC 22B6D65E C9EB87F1 71D270C2 87D1C9EB 763B41DE 1C86089B 8A8C3CDF
FE7487A1 66A8064F 57DABF8E 42A742A7 BF8E57DA 211BD637 4C766359 421C10D3
2BFF09D7 330B2FF5 E3B4E8D0 5B205B20 E8D0E3B4 2A35F8B8 0E0CD5D7 33721BAE
49AFD846 A4534876 D1CF7E59 7CB57CF1 FE59D1CF DBC673C7 69E217AC E6D944C3
2F3D2860 701E7F68 B8425243 500F500F 5243B842 FA39FD07 047EFE84 A437D7D7
B389A355 EA6B34DF 6F9BA144 76FE76FE A1446F9B A46D6BD6 B94AFD0E 39EDC651
87DB3808 F0D67437 860FFD93 DCE3DCE3 FD93860F CFCBE02D 9B85627C 8D587332
FD635477 D6681275 A45EC555 005B485B C555AC5E 20163A2F E43C7CB6 2CC78F72
DB52BA14 F3361E83 E90F6A79 1BA41FB4 6A79E98F 60417709 44E439B9 D618AA9B
1EA0E315 394F4AC5 6489E84E E63C663C F84E6489 F43C5213 6D8ABF58 5DF63F4C
31FAB729 5A5FDE06 8F45669F 4ED74ED7 669F8F45 80CE0943 FE443288 6747FF9B
10E4282F 3A6B3908 5366D1FB 94EC14EC D1FB5366 6ACBD36A CAAA0D6D D2B740EC
```

Generator Matrix for ARTM1 with $K = 1024$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 2/3$. Each string has a length of 16 hexadecimal characters and represents the top row of a size-64 circulant ($M_L = 64$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16×8 grid of (possibly) dense circulants, and so there are a total of $16 \times 8 = 128$ circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 4 will fit on each line below and thus the circulants on 2 lines must be concatenated to form the 16×8 arrangement of circulants in the generator matrix:

```
318D1D5872F9772B AF8A87CC049E45F0 22AC73ACD54ECBE5 C6CBC3B0555C07D5
0EBC6F1970566E4A C3DDF9B3C7BF13F3 30EB243B4A562B0A 57C7297274E0E069
D2A38608C8B39767 E71607D9E71D6F20 F5C36D59E1488D86 725BB2F98C4BC233
CF4B23A89CBBBB90 1DD081FCD7CAA0C4 69054C02294B9893 FFBFF0DD68211167C
5B7051BA49938275 718E510BAE657581 FDCEE12C191A974E F3AFE295F0EF20BD
09A1C0E9AFE8CD76 0B526BF38C077B57 6F239CA73F878462 3CC3AAC4CF785A76
C5D1C7CBAB1BC43F 3C0D51CF9172845B 821FBD36B3267A12 604170FACF606BE8
DC759378A87B4922 B411CC590F27E799 D66DAAF7531DAED8 3AED1768CEA6E7A2
BA88F6D416C06917 94D46F42E0A0A588 0D902967C96A3F89 FD5BE6363835A1EE
FD2624BE5DA834DA 68CF570031C9D695 C37BF5D3957031A8 8FBB3D25AAEE912D
B486065BBC02E03B 77CD6948FF1A1FF1 3CA2817759586FA3 7473A79CCB0F34F2
D8C52EAF18E1086B E4754E16746D667F E70371017C4D879A 4AE6D7BCF522FC79
284E8F8469AF8B7B DF2B114736D507DF E12197935D016CDC 0CFC199D0DCC9EA1
00FB87FE7F929522 ABFA232711C20F0B 0CEB16796306FDE6 5597AD9CE4C51653
97121E5828D83BFE 368882FBC9E70463 7153685690B6310E 4CC7BCE23CDE44BF
2FE050B6F931D3BC 9E5DD647CBF31A49 819E64C0566778A7 E2A8B0B5C072FB73
02D315DE6CE835C1 697A496BB207496E 7655053F5F5EEBBD 9350ACDE25CE2959
D85B4EC1CB845C34 61209DC46A999E2A FE5030DDC9AD4D60 CFBE1F890D490DD6
A21622EC6EB5848A B5AC33A6FB51A616 F24B865BBD95B21F DE63F1F55C1D2B68
AE0E5AA239946120 AB09B2761B9EDEC2 A3DA17AC63BE192E 0117D8157BCA4A0F
6F2E84960FB1C85F A3367556B3DCAF74 8B3384B83C7325BF B185294B65580606
C9AF88A7C2E3C73E AB7F308B850BBEC5 09CEF742641708EA 114C52A30C7ECA5F
71822F9796A728D6 60C26F44776B06DE 545A5E898ECE04B5 0B32D3443457C5E9B
FDE87F6584EEF959 0474BE29EAD275C6 7FD55884FC613682 643740A4B4FF30D0
D46D750A059281D7 694F2C38BCA6C6D9 976262F769435B8F 0C1A94D5B01DB30A
E96B9BDE981B8513 8C7C7576F04DCFA8 0972EECF23487369 683ABA8B8457BBC4
013B99A829F22EFE 760FAD6B2252FE31 F237351A2D74DC6F 2424D13EF023A599
E371050865AEA3F7 0681349B548B12E5 CC345E96BEB4556F BDBE30DD53B61E1D
BB743891E98A600A D2D5F3B040B4713D DAA084C2B4FCFD8C F878EB479AF30506
C56FE158FCF67D89 9FAE0FE84679AA31 9056369BFB55D95D C136066F60896D73
53E828C19FA85294 8F51819BF8323428 23F3D3C11FC28030 5A36007F4FB7B54D
84EF93F527BE6459 A1CC26166C61B1F5 589132562E1BD0D4 E4EFC81A1E7FE2BB
```

Generator Matrix for ARTM1 with $K = 1024$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 1024$ and $R = 1/2$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8×8 grid of (possibly) dense circulants, and so there are a total of $8 \times 8 = 64$ circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

```
0900A4500800004208408050090020 40 5750DC8AA76391A0217A190690C30CC8
65693512240205D2B120B518614A9150 6BCEF3778953CE04B8F59EDA757869D9
01A9154E4333C14254200D61060B91E6 0048050280400002104204028048010 2
108100A012004080120140A010000084 BFFBA04A1707F825A2DF60CAF8BF56DD
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 22E72594FCCA38DF59781D80D3EAC998
00000000000000000000000000000000 00000000000000000000800000000000
00000000000000000000000000000000 359B0AA87388A06ABC057EC54AD1BDB5
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 9D54F0BE6E6384EE86479BFBDCBC6604
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 1E7649AE33276BF566A3D4D48B49AB53
00000010800020100040001000402110 05985762241A25010010C12A30920562
0100A160004824429000A44091486440 D9301FDB05C9BEA7589873CDECFB6863
11AEC408140A82000182D423040AC40B 8000000840001008002080080020100
008000200080402000000021000040 20 800AD6F46FB8C1744A70DD12AB9B068E
00000000000000000000000000000000 00020000000080000000000000000000
00000000000000000000000000000000 BA5A37F5E68554714B38A098198AA699
04000000000000000000000000000000 00000000000000000000000040000000
00000000000000000000000000000000 13FB1672336E0315211D4DB7D95AF451
00000000000000000000000000000000 00200000000000000000000000000000
00000000000000000000000000000000 8D86EBEAC542206BF9C8AE004F359E7B
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 CD089C2EA1349F1B1B5528169FC7A3C3
01000402110000000108000201000400 12A3092056205985762241A05010010C
44091486440010 0A160004824429000A 3D5FA9B1C04A6B44A304E70177D99FDB
423040AC40B11AEC408140A82000182D 0008002010080000000840001008802 0
02100004020008002000804020000000 CE47E92E480CC5625DDE92D946EAB75B
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 380D50DA3D23ED3866357B94A037E04C
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 AC86BEA85A87CEFB76E5D9C6B1A5AD20
```

Generator Matrix for ARTM1 with $K = 4096$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 4/5$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32×8 grid of (possibly) dense circulants, and so there are a total of $32 \times 8 = 256$ circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 32×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000  00000000000000000000000000000000
5D198C9233076B9F19532BB784292CA9  D61FD9EEC0FC0F31B54E7FA3F5ABDBA4
00000000000000000000000000000000  00000000000000000000000000000000
31693A59B59E6D7029A77AA5F1B53F55  3104B330321F6441BE2081C87796A014
00000000000000000040000000001000  02001000000000000040000004000000
B3DE74CC892CE1689423125DF2A88A33  06EFAFF2EF4F998D0CB3F84B9F203273
00000000000001000000000000000000  00000020000000000800400000002000
36A1C10884C215E931DA91A901EDA816  923E76BE593E5BB0160446578BD4B0B2
00000000000000000000000000000000  00000000000000000000000000000000
4899AA79AF16F9CB9DF3D023EF3945B2  7BB6E41FF011576D940F8B72330287EA
00000000000000000000000000000000  00000000000000000000000000000000
BEED635DE6285E2EACD77BDCCE55E876  89CCA10170BDF06010629E1EC58C739D
00000000000000200000000000000008  00000008000010020000002000000000
137D0B07B6DDDFEE068B7A0645E0F9E7  496095C65AB55AA2EFEA4C32E5952D22
00000000008000000000000000000000  00100000000000002004100000000000
69D51657A4132F231311B5C97E548FBF  9C8C24B47CBCD24DB2B350EF308A24A9
00000000000010000002000000000000  00000000000000000030800002000000
6008F41E0B953817F671527B59E19588  77C2C2FCC42698C39AEA4EFEEDE7F33F
00000000000000008000000000000000  08800000100000000000002000000000
B26A52A941DD71B6728265336C2BDEB5  A0A5E9662127CFDA182A873F96F02BD3
00000000000000008000000002000000  20000000000000008000000C00000000
6827E7673D9B465931903BA91B043442  796F7C11AAF34B0425FEF3CF1C0F280D
00000000002000000000000000000000  00040080000010000800000000000000
6670C43B0FFB314ABB7725E94A34ABED  B350B72FBEC325616BFF70A56D349F99
00000000000000000000000000000000  00000000000000000000000000000000
84F89736E0F0B061E8DB5D288F69F7DD  342749CEB612CCAC58600F94300D1F69
00000000000000000000000000000000  00000000000000000000000000000000
D84E4B739ABEFD472F2A0C94AC38027C  34E119E8495C8B15B395E2A598831337
00000000000000000000000000000000  00000000000000000000000000000000
5B6C2E0B5030357883BF306413526C59  698F1DDB0386E67C0C76005B0902BF5C
00000000000000000000000000000000  00000000000000000000000000000000
7A2A12BDBFB8A3F7184D42FBB30C0D61  24960D1C207527D6A5AAD0BD9DD0FB20
00000000000000000000000000000000  00000000000000000000000000000000
0DC4E05B271BD2FFDDED5165FAF17C7D  265240795C3EBC940BBB363CE7AE5976
00000000000000000000000000000000  00000000000000000000000000000000
214E1E8CC0B7BDBA6FB9836838251E07  9D12914202282E38B48CA0F2A2E68ADC
00000000000000000000000000000000  00000000000000000000000000000000
54E994F6530F77ECB9AAA4B9670B389C  70A46C3409F233146FAED3AFACA9C59A
00000000000000000000000000000000  00000000000000000000000000000000
49897CC0ECC8721E00DDF634832F8D63  15ECEC51A5030FD337A94B492386A186
00000000000000000000000000000000  00000000000000000000000000000000
ECD55BD5ECE9F60DFE57E0E2D0B7B356  1A4E3C2C5E1D5F5C001154DB3E3085BB
00000000000000000000000000000000  00000000000000000000000000000000
7BBD8314518CDD5DADCE4826F53E6991  865DBC414EA30B389BB24022E450AF83
00000000000000000000000000000000  00000000000000000000000000000000
5C0BAD8E9774CFF57097AFC297732285  4B4DFD630FC7AD4330E9F142E63718D7
00000000000000000000000000000000  00000000000000000000000000000000
658046B6DDB229449E49B7E7A0021838  A17681693BA1B65E267EB1A2143EA2B2
00000000000000000000000000000000  00000000000000000000000000000000
ED243E9B0BF6F14C45548ACA99A1B19A  C73C9AFF731ABE0DCE5FCC25392A020A
00000000000000000000000000000000  00000000000000000000000000000000
B7B95D09AE540391AAD1B1389FBDDD28  9D669B84DD572D26B6516C383FBD7BE5
00000000000000000000000000000000  00000000000000000000000000000000
DF71C2E3404E7806E2F16329C5F128CB  A9F5E01DBFBDE9BD4ABB89F8777EDF30
00000000000000000000000000000000  00000000000000000000000000000000
DC43C09A6FA6FC3CCC007A85923E347F  11149B8D7CEC1A38FBD77F940A76D413
00000000000000000000000000000000  00000000000000000000000000000000
649E4EDE6AD2DAF820118552F44D0FFF  918812CF3C83FF86B4FFAAAF07BC0FE5
00000000000000000000000000000000  00000000000000000000000000000000
A0F83BF7129D587F0DD6D98D64641E0A  B998C739B8190BC796DD839189D08B95
00000000000000000000000000000000  00000000000000000000000000000000
72757800F09DB63D5C4E6C08847CD2D2  6ED84816979714D529D3095ADEF89663
00000000000000000000000000000000  00000000000000000000000000000000
F4B54A16B9403F6EEAC0CE08A7700206  C39CBD314B6882061B54E575AE2DFFBE
20000000000000008000000000000000  02000000200100000000000080000000
B94F9C12F228524C6C18CBB5648A6DCB  0527188932A0E1C21AA3338024CCD30A
00000000000000000000000000000800  00204000000000000050000000000000
14284A1AF9C540B7AA6DD4EF6E304414  91E212A646B82FBB0F4F0D4849846C6D
00000000200010000000000000000000  00000000020000002000100001000000
E0F665DAD090647E9282EC3EE5A8EBFD  E94E776F96877DC00AC392432A382BEF
00000800000000000000000000000000  08008000002000000000000000001000
B7440F04AE80B5848565CA915FBA6D15  04DEECC202FCD453F72FF8B359E1C32F
00000000000001000000000000000000  00000000000001000000100000000000
5327FA6772812D3C5F12A621EDB18A88  27EDAAA934BB9B0168737075A1F3765D
00000000040000000000000000000000  00080000000000010000000000000000
C8DA1DDD17CDC6C470FAA5FB3811659C  2302903214FD7D9A37CBA8EEC7D133D1
00000000000000000000000000000000  00000000000000000000000000000000
43F606F052A20326BF52B38A547B83CF  148B1FA56CC2184FF360BE3AE565D0B7
00000000000000000000000000000000  00000000000000000000000000000000
5C389670B0B9607548D89CBE7196506B  EAB5FB347F9DCF8BC239D3F86F1CD507
00000000000000000000000000000000  00000000000000000000000000000000
DE60F75C0BD7C2E449A7C77D72F5A804  0C41200878DB14E4FE83E928ACD18834
00000000000000000000000000000000  00000000000000000000000000000000
0A544921EE583D4BE25BCBAE2D0D5152  D1E28D1D724667EE5C91BD4FFAE04788
00000000000000000000000000000000  00000000000000000000000000000000
EB4992EC1A56442432255D548FC86A61  41F6A6DEA1E74DDA537391D1B34FBD56
00000000000000000000000000000000  00000000000000000000000000000100
C4327C0E52B969601FE7E50DF6C8D2D7  F4F7FF18F8215574229A1F56AB5DA2C5
80000000000000000000000080000000  00000000800000000000008080000008
C495C28BEC5BB87A2072B41C03066C85  EB99A5B647EDD7E9FBE3E86C6D6020E3
00000000000000000002000000000000  00000000000040000000400008000010
BB089D22296257423EC67AD64D74B75F  D63A508B1072C4CC88AC688CE9676CFF
00000000000000000001000000000000  00000000000000000000000000001000
6776CFDF027B1DF0F428E7ED2E8867DC  8F8FE3FA0BBF5E45AE00E1B1908047AC
00000000000000000000000000000000  00000000080000000000000000000000
3E3C14ABC5742DF1D9E4A6A54FC31801  0B285DA01401D47B2B7FD5246501C80D
00000000000000000200000020000000  00020000000000040000200000020000
F48EF4D9E63C2A5872442AB5467E438C  55812D7D868C2FB00DC0C365A54B0E91
00000000000000080000000000000000  00000001000000010000020000000000
03B08B4FFB552A66F90DDD7D017AC29C  F491027D0D713BF5334E37E53D8C4040
00000000000000000000000000000000  00000000000000000000000000000000
E11CD30131CB265DDBC599E78D58E43C  FE7848F1137FB8164D5DDFE514E087F2
00000000000000000000000000000000  00000000000000000000000000000000
C4F4EEAD800275FD42C6A76DF57B0D5D  588892E88F0298119751AEB2558924E6
00000000000000000000000000000000  00000000000000000000000000000000
F9B26E6DFB9F248423EC120800ADAAE6  965E45AA16A328F0D7DEE9106C2A274E
00000000000000000000000000000000  00000000000000000000000000000000
91A0663EDB793DC2C332CF58C4054388  F9EA5412816884249 69B2728B12CDEE6
00000000000000000000000000000000  00000000000000000000000000000000
4C20D968143AE24C38E431592B9A1E94  82E5FEC6620AC271FEF638B259A12148
00000000000000000000000000000000  00000000000000000000000000000000
F588BC2E5DDD1D9E2D3801F1A1DDEC71  B8CDCF29028ECC5D4EDFE0EE85CAF99D
00000000000000000040000000000000  00000000000000000040000004000000
1F0D8E67DD7611F1ED5C7ED3EBF59AA3  D2A63BF03BCF9A196274B425C0594406
00000000000001000000000000000000  00000200000000000004000000000000
7D5E4704FE6212A64779AB0610258D12  AA9F12444C132DCB9A5B7D7BC3704D07
00000000000000000000000000000000  00000000000000000000000000000000
FA6C27A03116B61EAEC7AE30C3710B49  E0769913D6541007EE981ADBE1B37FF8
00000000000000000000000000000000  00000000000000000000000000000000
01664E080DEAB80D8B2EDB16526C2EC2  F4BC7729FE68A52D590031333EDF6D27
00000000000000000000000000000000  00000000000000000000000000000000
A341B6DE055364DA054F71A49A47401B  DE7DCE2F61244285516C3DF36AEFBCC0
00000000000000000000000000000000  00000000000000000000000000000000
9D3FA0DF10D1A96D8279DA78F231A314  25F2E124647B59283B9BC5044EBBB65F
00000000000000000000000000000000  00000000000000000000000000000000
C3CAE11EC2A2CB67AE309A8E529DA1D4  F760A4B8279A2624B87FD2C6D188EEF8
00000000000000000000000000000000  00000000000000000000000000000000
9FC837C57E1635A8EB586A2AA64FE3A9  11CEC821BB33D4E53888D6A8161205C2
```

Generator Matrix for ARTM1 with $K = 4096$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 2/3$. Each string has a length of 64 hexadecimal characters and represents the top row of a size-256 circulant ($M_L = 256$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16 × 8 grid of (possibly) dense circulants, and so there are a total of 16 × 8 = 128 circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 16 × 8 arrangement of circulants in the generator matrix:

```
 4E3F301ABB81E45CC8B375C4A284320B4966A2C52A954B566910B415AE2DEDFF
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
5C6A04710A86535A10098C063C8C9C09C950B8FB43B7C470FC1B10161A503AE9
0000000000000000000000000000000000000000000000000000000000000000
D6CF80D6F3C8067001AEEC4BB7FCFA041AF38B070DF56DE0E3A3431E05011D89
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
F8CF6D0EF11E7304FE758F720CEB663604F53DDFE91D5E848D6AAC7A46622018
0000000000000000000000000000000000000000000000000000000000000000
CCF18ACB4C4D9E23F6FE6D839B899E95A44974AA8B3F617FA029014A5E23B53D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
B14F3E668A83256600DE220CD07B91A3472C124F45EECC4604D1A932F917E258
0000000000000000000000000000000000000000000000000000000000000000
AADE3D98FBB5D1548B2477DFB7C09E3CB47137BB5E67EEEA2A200C206892A41F
0800000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
000000000C0000000040000000000000000000000010000000000000000000
0000000000000000000000000000000000000000000000000000000000000400
0000000000000000000000000000000040000000000000000000000000000000
4F9D3AD9FFF6067050C8B4492D749C34A2F31C15841690C6BED1A0C92AFAEB8B
0000000000000000200000002000000000000000000000000000000000000000
09046652A5B9ABCC1A1A7F2CD149FD97BE24361AF30F0CC94018005E91294E7F
0000000000000000000000000000004000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000400000002000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
895DD7B29BBCF0D43459FF0EBA20FFA01584D8EF8E445F5E95FEDAC9C3F252F2
0000000000000000000000000000000000000000000000000000000010000000
F9690D6ADC72C57A8EEE294B50B12DE9C6046550AD582C9D2CB34F0AD9989F25
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
1F064F49EC81150CA0EC9CC8C62FE9E10FDBBB627B595B0453954FBF6EBDB37A
0000000000000000000000000000000000000000000000000000000000000000
7CEE931507EFFD2B93FAAD77B26AA5A921632C277338D6DD4A85B621108E0CE4
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
E9F82300644D87938BBAA8A7609F8F433808C1D3A273296352CF6A24619AE3A9
0000000000000000000000000000000000000000000000000000000000000000
8DFCF5D6435FB812309ED57455ACBE40DCAE47CECAA1AD41F222C0F4C5A83A95E
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000020000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
D7CE5C5485B00DCD2788D0466AF267851838E76DCAE2095DC4FED8A7FA3006E9
0000000000000000000000000000000000000000000000000000000000000000
DC4918B173C38F98EB7AD375F5158392B443867C911C97FA03D37056EE22A9C9
0000020000000800000000400008000400000000400002000000000200010
0040000000000000000001000000000000000000000000000000000000000000
0001280018000103408000009800040002010088002420004004260002200020
0100000000000000000020000000000000000000020000000000000000000080
0000000000020000000000000000000800000100000000000000000200000000
A500E6D648B232DD782128B33C70A97398B121C5E2FC9EF04C7A9896BB044A53
1000010000080000400008 0A40000008000200030080400010000200011000
8EAB3081C167B1689A3D61B3DF0FF2C38D7C917A0058ED9EBC6FBD367D9EE8AA
0000000000000000000000000000000000000000000000000000000000400000
0000000000000000000000000000000000000000000000000000000000000000
0000600000002000000000000000000000000000800000000000000000000000
0000000000000000000000000000000000000000000000000000002000000000
0000000000000000000000000200000000000000000000000000000000000000
03BADA33CF6327A4C71DEB1D3DB92B15A013E1DF975468AF2A99ECE3D85A46A8
0000000000010000000010000000000000000000000000000000000000000000
8DBE5F474D78E6E10053395405DB2DBE5FBED1C1C13011F98A55F13FFA3953BC
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

```
2D85636A391DD955FC16E4563B218AB4E08C6C84369919E1BDF0FF805ED1EB6A
0000000000000000000000000000000000000000000000000000000000000000
A80E0DE66D3804C9C95F4809D422F4F452161B6470D07F307D29D932ADE3974D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000008000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
19501115A5A30530EAB9BD004BC832197315D2AA6D8DCFA6005AE5B7E682D6DF
0000000000000000000000000000000000000000000000000000000000000000
8272DFDD7381DDAE79387210F9FFF7BD1A2D7585CE1C7D4056A9AC08185E29E7
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
2000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
A9F3492B99EF1E772BF9B19BBEA80EC9D37B03972884EEA52D829985AA1D75EA
0000000100000000000000000000000000000000000000000000000000000000
5312AE0649D415ED025D55C0BE1994580589BF314FF1549FCA84C5684CF8D43C
0000800000000000004000000000004000000000000000000000100000000000
0000000000000000000000000800000000000000000000000000000000000000
0000000080000400000004000004000000002008600000010200000000011
4000000000000000000000000020000000000000000000000000000000000000
2004000000000000000000000000000040000000000000000000000000000000
5E9C32DCAD9A64F6E52518111791EEB0E5B49D464959D9E80044F6020109D1E1
0000080000000400002000000002000000000000000100410000000100000
D365F4B81ABB52187A825987664BE8F9C8367AAC70C67D83C1DDB5986869A807
0000000000000000000000020000000000001000000000000000800000000000
0000000000000000000000000000000000000000000000000000000000000100
8000000040100000000200000000000220000000010000180000000800000
0000000000000000000000000000000800000000000000000000000000000000
0000008000000000000000000000000010000000000000000000000000000000
2A6B5A26F12E515D88F4166D2094D19AA47E9E4CBB8DB76E1098F0AB318D5CB4
0400000000000002008000000000000000000010000000008000040000000
0754963FDDE2FF70D1DC0512677E75B4D1FACBAB96A0CDD32A7A31D46173F611
0004000000008000200000000000001000000000000000000000000000000020
0000000000000000000000000000000000000000200000000000000000000000
0000000200044000100002C0003000440010000010000000080200010000 4
0000000040001000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010000000004001000000000000000000
87EF28D4A700F0ABC8900CFC313C1EE42A23347B52BBE02247AC8F216821441D
0000000000000000020000800012000080002000080000000000040100
```

Generator Matrix for ARTM1 with $K = 4096$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM1 (SOQPSK-TG) with $K = 4096$ and $R = 1/2$. Each string has a length of 128 hexadecimal characters and represents the top row of a size-512 circulant ($M_L = 512$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8×8 grid of (possibly) dense circulants, and so there are a total of $8 \times 8 = 64$ circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
67488F812902C865DDE17A9CD19E78E8CD50CDEE86EA2CB350084E4B3CE312FA1351B7A35A8086DDA246F0872540760C5398499E2AB8F0059E96F6D04E4951B2
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
E323EE6447E5073E62A0CD9E42C18B39C29E19B0BBF1F3C5F403CD4B3B3F8505502A5153DF555B4C7A50F1B121DC837B83EA0F581B3E637EE2287EFFD060AE52
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
76216F42899D199719552A238D4AB07C9157CCDB46476C52D174145F3DE6C9FD122E1293843C9B9005FEB451ADEEE876FE1AE54CDB03CEA3C5F2DDC7F470D951
8F2322970F502ADF20795D23140B33A218701F94C5C7059E468D824D793CC79EDC07E3765B988E1FC13365D0D7E7AA662166CEF7DC2AF2A18E9B2C1FC8C6146B
C74913DC28970A5DD40C6836668E9A9DD6BF6DD8BE63EFFAD2637BDAF2D285856FEC45372883CBC1B71B12D8EABEDF007E8D1AC40DA3DF123CB54CA75DCDD50F
DA594CAEA8A71BA35E2470966ECCCAF1304626CCC0DDFC216701B1B871D2656E590AA7F8112C04E36A8DD85539599509B77ED44BBDE4CB1FA7B53C8BA6583104
45372883CBC1B71B52D8EABEDF007E8D1AC40DA3DF123CB54CA75DCDD50FC74913DC28970A5DD40C6836668E9A9DD6BF6DD8BE63EFFAD2637BDAF2D285856FEC
B558FC9632ECA0547A2BAFFD04826D3422C8AA8C6C2A77B0A419853C827CBABD9B84DFBD3DC9D4BFD52647A620B9A48382CCA5464F184E2C2FB5EEA0A4DF3AE6
CCFE05CD1941A32B729E4AC11EF935D4ED052533F3C31A72440390C01C75D8CB852CB652FCB3DA02EFFEAF733CFB62B39BB7F04D4790CC891BE58C34A887FD70
633BC64E8359AC9DFE9F9B4824F6C639D023C14C0F8A679F9B63172EC6FE0CC53BABCE5DD14484A5B2A94B33469CEF6418B747EF85DEA9CE64EF253DDEDCA54A
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
7A5CBBE83E589FB0B21B65FB4C9CC4A90393AA64DA8C16C424C7462912251752454BFC2D16413CFD8AFD483FBA62E0070BDDC286EB590CEC3B5389B811C16311
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
E1D204409DB25F6C306B4A07A7B622E01FB8AA063DCE232574E027441C573EBFA869ABC67B34452F84FC305844C95866CE3ACF700F97E88D415F4DB7449E85CD
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
EE40A39576F718EC89E3ED92BB4451DE55473E1DECDF40DD446594F9A49B06FD28A07C558FBAEAC830121FC5AA52F08A3F8D17FA8A6AD0E6868AF398EDCCA2F0
A2EE3775EE9CAB56D2C7771C378642F1E784A1B1E2535F3DAD2716F453F47A5B31A5169BBF774D3DDC2DC56BA77C834248BC7E16E32921706DE2807AF695C28F
821C5FB9F0FCF735F99F7B9D2E811D681FCA3B7511DBF28D40C8AD12833BDD259B9B807D223B8F218D45ED8329638EAB8BF25278BC825FDEC24D145B530027DB
81D1824E6BA270E4F20ED06D045AC55E9B907EBCD6825CE95AD6DB669FD5A904F4D35988F71840CD05C204C532B64CE4252A1BDBBFB31D84CAED31E29959A26C
807D223B8E218D45ED8329638EAB8BF25278BC825FDEC24D145B530027DB821C5FB9F0FCF735F99F7B9D2E811D681FCA3B7511DBF28D40C8AD12833BDD259B9B
0299B9CD7AAC1C70825E5E0FCA61F0AEB8DD2E9EB0C93E8AB196DC24637790237E5DE081569115A6B7CAEE04BB37BF9F50DB666BFE878596D9E2F72EAD0F8C5F
D7201AAEB0CDDC5A1754DF1222C80638D33639F93B5F959F78AC3E4AF0541C074C159FAD218CD75902B238D7AF6123B10DA8C323A8F5D32D1A38E8F8B6C65270
25A65422ED86C22ADCC53DE489BBAE016DCE23AA86179DE88DF8126E2DA79E6C6E4EB7DE1691D9CB0DE16FA4F75CCA2BFC9120CC0440CB32B0C427121943EA6C
1A9560F922AF99B68C8ED8A5A2E828BE7BCD93FA245C252708793720OBFD68A35BDDD0EDFC35CA99B6079D478BE5BB8FE8E1B2A2EC42DE85133A332E32AA5447
B0BFB1D59DE8676D7DA0FA68976562BBFCFA4590490614F2A5B5310FAB88F14A8D986923F0154AA5024E052A0CE38DF3CBF0B1D2523913359227BA517CEF6B3B
CD1D353BAD7EDBB17CC7DFF5A4C6F7B5E5A50B0ADFD88A6E510797836E3625B1D57DBE00FD1A35881B47BE24796A994EBB9BAA1F8E9227B8512E14BBA818D06C
238AA72CC7524D82EF82873C2C1B32C26F9410742449275C36D7FD1D02863DF2F69E070C4E14528DD4C76B6133D75072043EB30BA7625ED2AE7A4BED22950283
BE00FD1A35881B47BE24796A994EBB9BAA1F8E9227B8512E14BBA818D06CCD1D353BAD7EDBB17CC7DFF5A4C6F7B5E5A50B0ADFD88A6E510797836E3625B1D57D
0904DA6845915518D854EF6148330A7904F9757B3709BF7A7B93A97FAA4C8F4C4173490705994A8C9E309C585F6BDD4149BE75CD6AB1F92C65D940A8F4575FFA
3DF26BA9DA0A4A67E78634E48807218038EBB1970A596CA5F967B405DFFD5EE679F6C567376DE09A8F21991237CB1869510FFAE199FC0B9A32834656E53C9502
49ED8C73A04782981F14CF3F36C62E5D8DFC198A77579CBBA289094B655296668D39DEC8316E8FDF0BBD539CC9DE4A7BBDB94A94C6778C9D06B3593BFD3F3690
FE8E1B2A2EC42DE85133A332E32AA54471A9560F922AF99B68C8ED8A5A2E828BE7BCD93FA245C252708793720OBFD68A35BDDD0EDFC35CA99B6079D478BE5BB8
855378F7DB0214417E0AC5662ABDD088E4560E213572A52099972A636B6714136CF87394AAC58A767DF48CE1FB3778707F6D7E8A09FA45BF2C3BC961EDED09A5
EBB9BAA1F8E9227B8512E14BBA818D06CCD1D353BAD7EDBB17CC7DFF5A4C6F7B5E5A50B0ADFD88A6E510797836E3625B1D57DBE00FD1A35881B47BE24796A994
47520034B945301748FBC9AE99A8FCDD9E40C88D5D9C7624FCFE26FE127ADE378125C9DE22EFB87CBC6841B143A830120E5E7797E064742328D22D3231E19917
50B0ADFD88A6E510797836E36A5B1D57DBE00FD1A35881B47BE24796A994EBB9BAA1F8E9227B8512E14BBA818D06CCD1D353BAD7EDBB17CC7DFF5A4C6F7B5E5A
149BE75CD6AB1F92C65D940A8F4575FFA0904DA6845915518D854EF6148330A7904F9757B3709BF7A7B93A97FAA4C8F4C4173490705994A8C9E309C585F6BDD4
9510FFAE199FC0B9A32834656E53C95823DF26BA9DA0A4A67E78634E48807218038EBB1970A596CA5F967B405DFFD5EE679F6C567376FE09A8F21991237CB186
BBDB94A94C6778C9D06B3593BFD3F369049ED8C73A04782981F14CF3F36C62E5D8DFC198A77579CBBA289094B655296668D39DEC8316E8FDF0BBD539CC9DE4A7
CDC2F8C8791A60968F6B5743E96EA02E0104C661E6EC57B6CA3AB90D6DD1F889D584B98B8D60E2BAE49FEFD688A0C23FC1B6329A38C2D5BBAC1DDC213750002F
8DEF60D6EF71B013349F235FDE8F61CB8AAD978A3CA3460BC3E00490654C558EE88A806FBC94B04C507F1B9B6E1B57AFEE2A1013FE4D1CB54B26F6A9FCBF4012
EC083A9851D2011971580541A4877AD6CE8647D78C13A4650434546537147322210DBCEF8F04D34894463E387D4622C2F53B4A6F76830A744BC938483A3AC1DFB
49A3E4B6BE15C7C4817B0BFC05A68D5800B744EF0C1A8F667EA74FDF3CFEBD78D3E7CFBD96FB66C3C626CDC711DF60D0D17DB11206EFE8AB1A27618C562A389F4
CEF8F04D34894463E387D4622C2F53B4A6F76830A744BC938483A3AC1DFBEC083A9871D2011971580541A4877AD6CE8647D78C13A4650434546537147322210DB
4CC398377B65F34B582E74B90817839ABF409EBF69BCCAC0EB468F087759C278D0CE8FA4EB79BC1CABB02262EE53639A9C8BD915B2C0D12F7034B4D87B4D50EA
CE1843756C28AB93ABB2268E37E029F0227E82B8243C0A34745A9E5A5ADF7ECBF07126789D9134AEC956F7DF2E963740076D445A01FC8D9056A7DD6EBB581359
94297FBBE916DDE579D3925149C8948A75D3CB0F1D6642A3AA78C7DFEB9CA1B8BFBE0B6CA5C9D5AFD96E7A734A94108EAACF7B484416954734ACE1FB8FABE416
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
9F27199D58A3E81DD9EFEEAA6FC1C337DDEF65CA5B1DDA2C1756ED649082640D15FBA5579EF6B1C2EBAD9FCE8B4EE0D29F7904D989F34A75515EF222546A821C
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
F7C8D733617DBDF81DE05ED3F9C10F5A576AE5FF9EB95814860503206F4549A7A24B6CC3AEDD71A32318C891B213F87EF882F4B5B96E55B1B5B13019472826CD
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

APPENDIX
GENERATOR MATRIXES FOR ARTM2 (ARTM CPM)

Generator Matrix for ARTM2 with $K = 1024$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 4/5$. Each string has a length of 8 hexadecimal characters and represents the top row of a size-32 circulant ($M_L = 32$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32×8 grid of (possibly) dense circulants, and so there are a total of $32 \times 8 = 256$ circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because the strings are so short for this code, they can be arranged below with 8 strings per line and so the listing below corresponds exactly to the 32×8 arrangement of circulants in the generator matrix:

```
00000000 569A613E 29ADF08E 00000000 17060088 00000000 5859D8E1 00000000
00020000 D993E182 562AB0E9 00200000 D71E0543 00000004 7344DA73 00011800
00000000 6F32DCB0 D3BB74BF 00000000 2110D73F 00000000 E037FE82 00000000
00000000 77F3C400 DA3D7CFF 00000000 0E24F7FF 00000000 00336882 00000000
80804000 B6017797 179C4C1A 00040000 25E6BA0E 04000000 313DC9C5 00100110
04008000 38F0D658 295C2E14 00080000 290A28C1 00000000 48C6FD48 00000200
00000000 9DB65D5D 2AC4F666 00000000 20FEC647 00000000 6FC5F68F 00000000
00000000 1038710A 2228E7B7 00000000 51A6F991 00000000 A6AC82B3 00000000
00000000 BE2DF07D 90E9ABEB 00000000 7FFA9EAA 00000000 821E3E18 00000000
00000000 65B8AF01 38385D5D 00000000 572056EE 00020000 CD5BE084 08000000
00000000 1BF308DE D0141BF7 00000000 EF6CF8C6 00000000 8CD79E4B 00000000
00000000 EDCFC7F2 A52BA270 00000000 9E620C4E 00000000 C6056643 00000000
00000000 D8D3FDE3 53BCD8E9 00000000 FEA934C5 00000000 B0582D2B 00000000
00000000 06585C43 0BC6290B 00000000 4740B954 80000000 FC306C0E 00000200
04000000 54F9E335 475AABF6 00000000 693AD377 00000000 563A283E 00000000
00000000 F6991541 FC41243C 00000000 386D74E0 00000000 09E4B4F2 00000000
00000000 D3BDBED0 640B3F10 00000000 6EAF03C2 00000000 41FF41FF 00000000
00000000 E2ACC12D 40160C21 00000000 8A0345BD 00000000 9C452DBA 00000000
00000000 D17C4792 560F1004 00000000 F5A99E69 00000000 3C56CAED 00000000
00000000 7929AB7F A64983A3 00000000 D2EAB0E0 00000000 2725E65F 00000000
00000000 48DB297C 0BD218E0 00000000 7DFC62F1 00000000 B8A95200 00000000
00000000 721D5CCC A0E9929E 00000000 CF39E7D4 00000000 57264FD6 00000000
00000000 6D50D3C2 EF25629A 00000000 CC685BDE 00000000 DAB87EE8 00000000
00000000 62BFB0AF 6FECB785 00000000 F59B7ED6 00000000 61A514D0 00000000
00000000 2DAA2B54 080F1E94 00000000 F6D059F0 00000000 D06B7611 00000000
00000000 013FC8E9 ABF4618C 00000000 5B5C2F41 00000000 B9FA46D4 00000000
00100000 12916D71 479DB2B4 01000000 76A80276 00000000 53BC6F64 00004000
00000000 94CD99F6 6B9F5B62 00000000 8FCCBCC9 00000000 A01058E8 00000000
00000000 8A0DF5B9 CF3AC310 00000000 51E1CE60 00000000 11B68561 00000000
00000000 44BC4105 02164531 00000000 02432F9D 00000000 AC850D1D 00000000
00048000 149A343C 7FF48CC6 00080000 25CCBEC3 01000800 A96D47C2 20200204
00000000 06E03D14 BB342DEA 00000000 C3219B05 00000000 5797F8D6 00000000
```

Generator Matrix for ARTM2 with $K = 1024$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 2/3$. Each string has a length of 16 hexadecimal characters and represents the top row of a size-64 circulant ($M_L = 64$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16×8 grid of (possibly) dense circulants, and so there are a total of $16 \times 8 = 128$ circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 4 will fit on each line below and thus the circulants on 2 lines must be concatenated to form the 16×8 arrangement of circulants in the generator matrix:

```
0000000000000000 0000000000000000 C5274DDD7C78F9DC 0000000000000000
0000000000000000 D3775F1E3E773149 24B7D752BBE5372B A4E9BBAF8F1F3B98
0000000000000000 0000000000000000 0EF720B0BF516529 0000000000000000
0000000000000000 C82C2FD4594A43BD B15A52BA6228C823 DEE41617EA2CA521
0000000000000000 0000000000000000 C4B62A94065A5382 0000000000000000
0000000000000000 8AA5019694E0B12D B287FC11841B9B78 96C55280CB4A7058
0020000000000000 0000000000000000 BD05B4222DF4F683 2000000000000000
0000000000000010 6D088B6D3DA0EF41 6CBBD6436F220CE0 A0968445B69ED077
0000000000000000 0000000000000000 BAD9B91BE9458622 0000000000000000
0000000000000000 6E46FA516188AEB6 60AE27A94630BA28 5B37237D28B0C457
0000000000000000 0000000000000000 76428135682904F6 0000000000000000
0000000040000000 A04D5A0A413D9D90 7D0FA8843CFD10B4 C85026AD05209ECE
0000000008000000 0000000000000001 092BDA7AA4CAF477 0000000000000000
0000000000000000 F69EA932BD1DC24A 0969511B7A6C996D 257B4F54995E8EE1
0000000000000000 0000000000000000 0937C4E3EF562A96 0000000000000000
0000000000000000 F138FBD58AA5824D 0736BC6669F8A898 26F89C7DEAC552C1
0000000000000000 0000000000000000 5E8EBA734CB04304 0000000000000000
0000000000000000 AE9CD32C10C117A3 BD75FB3765DDE33B D1D74E699608608B
0000000000000000 0000000000000000 168C19964A9B73C7 0000000000000000
0000000000000000 066592A6DCF1C5A3 C89CC998FEE94972 D18332C9536E78E2
0000000000000000 0000000000000000 300A9306106E5E16 0000000000000000
0000000000000000 A4C1841B97858C02 244EB2D927518D26 015260C20DCBC2C6
0000000000000000 0000000000000000 4378F1172ADAEFCA 0000000000000000
0000000000000000 3C45CAB6BBF290DE 1F83990C76F85ACD 6F1E22E55B5DF948
0000000000000000 0000000000000000 C4C60555A0495743 0000000000000000
0000000000000000 8155681255D0F131 16F78CE9DE6C15B8 98C8AAB4092AE878
0000000040000000 0000000000000000 985D9AC6185EA740 0000040000000000
0002000000000000 66B18617A9D22617 33B95C112922457F 0BB358C70BD4E913
0000000004000000 8000000000000000 DF748119EF21F5DD 0000000000000000
0000000000000000 20467BC87D7777DD E819268FFC1722EA EE90233DE43EBBBB
0080000000011000 0000000010000000 0A83F8C1C522A372 0000000001000000
0000000080001000 7E307148A8D482A0 9B702955632096CA 503F1838A4556A41
```

Generator Matrix for ARTM2 with $K = 1024$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 1024$ and $R = 1/2$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8×8 grid of (possibly) dense circulants, and so there are a total of $8 \times 8 = 64$ circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000 392CCA875CCDE4BCE7B5DB5A62456B99
00000000000000000000000010000000 00000000000000000000000000000000
00000000000000000000000100000000 00000000000000000000000000000000
5D99D2252E0823A2FEAC125B75E42E6F 00000000000000000000000000000000
00000000000000000000000000000000 67ACE31F6AAD74927674A2CF2580AD7E
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
7CE63EA3C48530AC561810E1599F761B 00000000000000000000000000000000
00000000000000000000000000000000 EBBDD0AF43952DBABA4B3ABB57E4FDC8
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
0B7B4AFBCD72C30FA885A00AF23481EE 00000000000000000000000000000000
00000000000000000000000000000840 D1C9AAAA5DB26B870D2EECF99C2A0D23
00000000000000000000100000000000 00000000000001000000000000000000
00000000000000000000000000001000 00000000000000000000000000400000
CCBA8FE4EFA3F67F9C812B258655AA4E 00000000000000000010000000000000
00000000000000000000000000000000 8748C85212575EEA4CE936FDFFD4B943
00000000000000000000000020000000 00000000000000000000000000000000
10000000000000000000000000000000 00000000000000000000000000000000
29D246558F732C51A2FBFF346AB7C162 00000000000000000000000000000000
00000000000000000000000000000000 48BD9A9EC9A969259E76258B3C3BE3E3
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
8892D27F9DA1DE2B5C07E96CDAF10A61 00000000000000000000000000000000
20000000000000000000000000000000 2BD9481A43DDA6D74701F90A0EC735B0
00000000008000000000000800000000 00000800000000000800000000000000
00000000000000000000000000000000 00000000000000000000000000002000
C27D58F0AB890F4F7C8D5DE444986BED 00000010080000000000080000000000
00000000000000000000000000000000 963552582F69D0DD385A9031140A990D
00000000000000000000000000000000 00000000000000000000000000000000
00000000000000000000000000000000 00000000000000000000000000000000
BAD925F9017D30B0EDB1677735F7D0B8 00000000000000000000000000000000
```

Generator Matrix for ARTM2 with $K = 4096$ and $R = 4/5$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 4/5$. Each string has a length of 32 hexadecimal characters and represents the top row of a size-128 circulant ($M_L = 128$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 32×8 grid of (possibly) dense circulants, and so there are a total of $32 \times 8 = 256$ circulants in this list. These dimensions correspond to $K/M_L = 32$ and $M_D = 8$. Because of the length of the strings for this code, only 2 will fit on each line below and thus the circulants on 4 lines must be concatenated to form the 32×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000  BD62016C5FA3202640DB03D0AF9F613A
A2ECBD51EE1ADBA24616D631D1D5DA2A  00000000000000000000000000000000
00000000000000000000000000000000  0F42BE7D84EAF58805B17E8C8099036D
00000000000000000000000000000000  1E857CFB09D5EB100B62FD19013206DA
00000000000000000000000000000000  14177CD65622146E14D965CF454DF5EE
C89383C1C157DBFAC4419496A3723B9A  00000000000000000000000000000000
00000000000000000000000000000000  973D1537D7B8505DF379588851B85365
00000000000000000000000000000000  2E7A2A6FAF70A0BBE6F2B110A370A6CB
00000000000000000000000000000000  D6B4AA25BEE1FE206EB7542111103FD1
9ED868987EBB829570A77753AA5BEDFA  00000000000000000000000000000000
00000000000000000000000000000000  50844440FF475AC2A896FB87F881BADD
00000000000000000000000000000000  A1088881FE8EB585512DF70FF10375BA
00000000000000000000000000000000  13D376FDA6D3FC9BF6FF054CF1372862
CB048A707AC67907C088AEA951720327  00000000000000000000004000000000
00800000000000000000000000000000  15B3C4DCA1884F4DDBF69B4FF24FD3FC
00000000000000000000000000000000  2BE789B943109E9BB7FD369FE49FA7F8
00000000000000000000000000000000  4C0C49C89AF091353C4A12AB2ED16A26
57A338C0C9AA86298FD6182364BD49DA  00000000000000000000000000000000
00000000000000000000000000000000  4AACBB45A899303127226BC244D4F128
00000000000000000000000000000000  9559768B513260624E44D78489A9E250
00000000000000000000000000000000  B8F122F2C57F8B1663D40ABC27BC7140
137DE37ECA98F6CE17672D835E07AD36  00000000000000000000000000000000
00000000000000000000000000000000  2AF09EF1C502E3E48BCB15FE2C598F50
00000000000000000000000000000000  55E13DE38A05C7C917962BF458B31EA0
00000000000000000000000000000000  F722AA2EEFBAD5B41BA6886EE1018038
87B02C65ABF6B6E374C300F543ED3B4F  00000000000000000000000000000000
00000000000000000000000000000000  21BB840600E3DC8AA8BBBEEB56D06E9A
00000000000000000000000000000000  4377080C01C7391551777DD6ADA0DD34
00000000000000000000000000000000  F22D7ECEBF9EEFA8572594A1F3EB2A2A
CEDA15AC91551945371E89C38514300F  00000000000000000000000000000000
00000000000000000000000000000000  5287CFACA8ABC8B5FB3AFE7BBEA15C96
00000000000000000000000000000000  A50F9F595157916BF675FCF77D42B92C
00000000000000000000000000000000  EBB7298225EA69FFE90869176FAAC5CC
AA6B5B1672D6C6B14130826F854C2E8D  00000000000000000000000000000000
00000000000000000000000000000000  A45DBEAB1733AEDCA60897A9A7FFA421
00000000000000000000000000000000  48BB7D562E675DB94C112F534FFF4843
00000000000000000000000000000000  CADCE9491DE53A015108C59FB3F7ECF3
E23B63AB780F81BCF98CA5F632AE5B29  00000000000000000000000000000000
00000000000000000000000000000000  167ECFDFB3CF2B73A5247794E8054423
00000000000000000000000000000000  2CFD9FBB679E56E74A48EF29D00A8846
00000000000000000000000000000000  31BEA6EDB0EF0EFB274E483AE10186AF
B98B2A157C03A94DD239FCD5B6B08E19  00000000000000000000000000000000
00000000000000000000000000000000  20EB840E1ABCC6FA9BB6C3BC3BEC9D39
00000000000000000000000000000000  41D7081C35798DF5376D877877D93A72
00000000000000000000000000000000  2F2EC4ABB4B531F6312CB9BD024A9DE5
59003AFAC984922F7E8764D40BC1B81A  00000000000000000000000000000000
00000000000000000000000000000000  E6F4092A7794BCBB12AED2D4C7D8C4B2
00000000000000000000000000000000  CDE81254EF297966255DA5A98FB18965
00000000000000000000000000000000  2907E84A4E2B04512C27CC665B6DB6F7
083D3C772551AE66DB7777BD7C6B288F  00000000000000000000000000000000
00000000000000000000000000000000  31996DB6DBDCA41FA1293AAC1144B09F
00000000000000000000000000000000  6332DB6DB7B9483F425275582299613E
00000000000000000000000000000000  1DC4BE7865EB6676AE017B0BE4B8EAE0
E76E9F2052F4771C77CBB944A7905DD2  00000000000000000000000000000000
00000000000000000000000000000000  EC2F92E3ABA07712F9E197AD99DAB805
00000000000000000000000000000000  D85F25C75740EE25F3C32F5F33B5700B
00000000000000000000000000000000  AF4BAF3A36AAA303C571058A5B567B42
FCC2B36B38CD1CEA55E6B2C1FC7FAEEC  00000000000000000000000000000000
00000000000000000000000000000000  16296D59ED0AAD2EBCE8DAAA8C0F15C4
00000000000000000000000000000000  2C52DAB3DA15525D79D1B555181E2B88
00000000000000000000000000000000  2EAFD5B2962712A24E88C169A1F85EA3
2327E3F6AA0C2E9A79576B730A0F3D8C  00000000000000000000000000000000
00000000000000000000000000000000  05A687E17A8CBABD56CA589C4A893A23
00000000000000000000000000000000  0B4D07C2F519757AAD94B13895127446
00000000000000000000000000000000  AF146B9458607BD017E0C1CF2CC55E35
15C906075A4D7DAEF4BDA9364D36EA8D  00040000000000000000000000000000
00040000000000080000000000000000  0738B31558D6BC512E516181EF401F83
00000000000000000000000000020000  0E71262AB1AD78A2DCA2C303DE803F06
00000000000000000000000000000000  E5050022384C92D7BDB5D51E94C3A6CC
E21C13E1E229447BBF3D776421F9C4C8  00000000000000000000000000001000
00000000020000000000000000000000  54FA5B0E9933941400 88E1324B5EF6D7
00000000000000000000000000000000  A9F4B61D306728280111C26496BDEDAE
00000000000000000000000000000000  BDCA257C646B24FEF59C32FE5EF8BDF0
56B186BAF691163461C55B79D10C6093  00000000000000000000000000000000
00000000000000000000000000000000  C3F97BE2F7C2F72895F191AC93FBD670
00000000000000000000000000000000  87F2F7C5EF85EE512BE3235927F7ACE1
00000000000000000040000000000000  9BE0267A4D73BB2A60AAF66554E85726
AC1F69CF5C619CA32428DA3DA201266F  00000000000000000000080002400000
01000048000000000000080020000000  D89453E95C9A6F0099E93DCECCE982B9
00000000000040000000000000000000  B028E79AB934DE0133D27B9D99D30573
00000000000000000000000000000000  87A4C892C6A1132201589B22FFAB8946
B460800B4B10D0A0FA1A2FE4D4FBD871  00000000000000000000000000000000
00000000000000000000000000000000  6C8BFEAE251A1E93224B1A844E880562
00000000000000000000000000000000  D917FD5C4A343D26449635889D100AC4
00000000000000000000000000000000  1F1A5EAFE138CF15A29BDDEEBB750EA4
2A7B49404F1ED90499031D85BD51B2F3  00000000000000000000000000000000
00000000000000000000000000000000  77BAEDD43A907C697ABF84E33C568A6F
00000000000000000000000000000000  EF75DBA87520F8D2F57F09C678AD16DE
00000000000000000000000000000040  EEAE469C07E128F015B5C24F8C16D1E3
A0F7B55FC0975602EFA951121D173B7F  00000000000000008000000000000000
00000000000000008001000000010000  093E305BC78FBAB10A605F84A3C156D7
00000004000000000000000000000000  127C60B78E1F756210C0BF094783ADAE
00000000000000001000000000000000  3A5E2E070F220A7D66F38F04328F1318
61A97AE2F939B31C9C5E03B1EE834310  80000000000000000000000000000000
80000000000010000000000000000000  BC50CE3C4C60F978B81C3C8821F59BCF
00000000000000000000000040000000  78B19C7898C1E2F17038711043EB379F
00000000000000000000000000000000  E296314530A006F06A115D42E2FDEB47
59B0587F62F6B5C233E74301B2BF15D7  00000000000000000000000000000000
00000000000000000000000000000000  750B8BF7AD1F8A58C514C2801BC1A845
00000000000000000000000000000000  EA17176F5A3F14B18A2985003783508A
00000000000000000000000000000000  8473006221FD0D22925706C6BCC3C7F5
446CAC7344033C10C3A5C793B0E023D4  00000000000000000000000000000000
00000000000000000000000000000000  1B1AF30F1FD611CC018887F4348A495C
00000000000000000000000000000000  3635E61E3FAC339803110FE8691492B8
00000000000000000000000000000000  C1AB299A744917B8B0B2598801BF499E
BB4F72E696E2DC02369E0FC544FEA9EC  00000000000000000000000000000000
00000000000000000000000000000000  662002FD267B06ACA669D1245EE2C2C9
00000000000000000000000000000000  CC4005F84CF60D594CD3A248BDC58592
00000000000000000000000000000000  FD2FE68D827D5C4E1F5FD897D1968272
4AB654DE79F7703C55299EFDD3B0647D  00000000000000000000000000000000
00000000000000000000000000000000  625F465A09CBF4BF9A3609F571387D7F
00000000000000000000000000000000  C4BE8CB41397E97F346C13EAE270FAFE
00000000000000000000000000000000  60EB7A64EF1B457C6D53ECD1C3D79567
0D04138506BF87A915E35D7A8141B29F  00000000000000000000000000000000
00000000000000000000000000000000  B3470F5E559D83ADE993BC6D15F1B54F
00000000000000000000000000000000  668E1EBCAB3B075BD32778DA2BE36A9F
00000000020200000000000000000000  4303B6FAF64EDF83D6709EE8711B9352
13F54379AC2A83CCC5A7489558AF1CE9  00000000000000010020000000000000
00000000040000000200000000 2004  7BA1C46E4D0B0C0EDBCBD13A7E0F71CE
00000000010000000000000000000000  F74388DC9A16181DB797A274FC1EC19A
00000000001000000000000000000000  B723088CE9E75C1D184269D733ADB81F
68CE5D5000B256FCB42012BB529EAFD6  00000000000000000000000000000000
00000000000000000000000000000000  A75CCEB6E07EDC8C2233A79D70746149
00000000000000000000000000000000  4EB99D65C0FDB91844674F3AE0E8C283
00000000000000000000000000000000  D1059E568F0E17C35F4FC3343B95E70B
00BD54D61A9DF14DC78A15A5DE0405F7  00000000000000000000000000000000
00000000000000000000000000000000  0CD0EE579D2F4416795A3C385F0D7D3F
00000000000000000000000000000000  19A1DCAF3A5E882CF2B47870BE1AFA7E
```

Generator Matrix for ARTM2 with $K = 4096$ and $R = 2/3$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 2/3$. Each string has a length of 64 hexadecimal characters and represents the top row of a size-256 circulant ($M_L = 256$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 16 × 8 grid of (possibly) dense circulants, and so there are a total of 16 × 8 = 128 circulants in this list. These dimensions correspond to $K/M_L = 16$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 16 × 8 arrangement of circulants in the generator matrix:

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
D58AA22686F08A09200EB33FD1A061BC4CAB8E841021C3E1B74C77432AE78334
C68F18438672420665D9991202BF3CB90BEA2B3807F7CAF5C111D1F74C470E4D
7CAF5C111D1F74C470E4DC68F18438672420665D9991202BF3CB90BEA2B3807F
0000000000000000000000000000010000000000000000000000000000000000
0000000000000000000000040000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
D96591CE66DD25515A5A811392C7342AB3E7F8D07CF9B63625 12E917970CD821
D06F2E88520E8F3DBE9A63BBD3F41CA8C670BB0B6DAE1FEF4DE3C5EB462B3734
E1FEF4DE3C5EB462B3734D06F2E88520E8F3DBE9A63BBD3F41CA8C670BB0B6DA
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0020000000000000000000000000000000000000000000000000000000000000
C6EB8FF180F51B09F83C48ADEC9F29302D84A18A07778F108AD3BF1D5225A013
B9EA6A402FFA035CA94920B25DC2D6405D39B08937D0A037AB48FEBC3FE91A72
0A037AB48FEBC3FE91A72B9EA6A402FFA035CA94920B25DC2D6405D39B08937D
0000000000000000000000000000000000000000000000000000000000000000
0000000800000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
6D2F81D4F96735AF811CF3A1D989A9012BA0D51D7C70BD0178F6A07C1E46B84C
D040D913F6B510E9ACE66AB51A5A8876DE037324811B8CCA85F7BCF5C1E58CD0
B8CCA85F7BCF5C1E58CD0D040D913F6B510E9ACE66AB51A5A8876DE037324811
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
6E24EE3DF307602B4A4754299956FFFF03E91703FEC2482AD78AAC5BEA7E93D6
1DA01FA19577EF3562CD01FC2923B802A60CD2CA7651A1CA4AF353A0B113099E
1A1CA4AF353A0B113099E1DA01FA19577EF3562CD01FC2923B802A60CD2CA765
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
D9B0984914A232DCA646100AE50B8B4ABC6E892533CCD5799B09E0D9DD304941
269B97F2F949A9D019A2A8060B6556C9ECE7337D14D961F4E3CD9FAE3AC9E48B
961F4E3CD9FAE3AC9E48B269B97F2F949A9D019A2A8060B6556C9ECE7337D14D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
911BB7774BBAD5AC31AF113E6C2A2D09F27C5BD2E182DB2DADED58C81EBF1448
C5E889F5082B47E9904CDA663C934C5DA0E38D1DD3DDDC251EC00EC9DD92156A
DDC251EC00EC9DD92156AC5E889F5082B47E9904CDA663C934C5DA0E38D1DD3D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
814E29F48F115EF8D6C254D3DAA030C6E003AA25C8A5E497258E0D9404F62404
3AA5CD8A2E79C416FD2162B02E94CBC2AC86298028D74BFB6DD744CE3DE6C74E
74BFB6DD744CE3DE6C74E3AA5CD8A2E79C416FD2162B02E94CBC2AC86298028D
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
78EAD082083F514E30D8DD646EF381CB40434BF2B460602F80411086D8434B51
63357F3BFEE2A759B2FC2C0680673ECC92337516D4F83ADC65D1AD1720920242
83ADC65D1AD1720920242633 57F3BFEE2A759B2FC2C0680673ECC92337516D4F
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000010000000000000000000000000000000000000000
C01D2B4EA750AC8BE7FBD4BD24B4F5FC6D2A1B1C0B3D4A68D38CD6A0B16F1FE8
3348D48F3AEC2C86BA4D3B2372426B8B2521B8564FA4736303D63408BCED3220
4736303D63408BCED32203348D48F3AEC2C86BA4D3B2372426B8B2521B8564FA
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
26ABD920B6F07201B070E92A25BF6AC1DBB6F34817C88A62A554658803AEA664
EAF858AD0111C86437D095C569B0F447B10FA85C769E06EF3E3E0653F4BBFE3A
E06EF3E3E0653F4BBFE3AEAF858AD0111C86437D095C569B0F447B10FA85C769
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
607784672225 30CAD6D2BB093F6BB8F6D1C1E3BADD99776F3D9CA751F8FA91C5
571AD38557F1837046C2BB94354D36596196F2499212F12DC1ED0902149FE0F9
2F12DC1ED0902149FE0F9571AD38557F1837046C2BB94354D36596196F249921
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000010000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
3C7962BBBAB194D2AEA214227F3AB1DFBA31A2E8000C8940B134B497F0DBFE03
8C08B2364C6F71325420FD0856F7FC3979D6917A1C9D73BC443D9EAE0485E187
D73BC443D9EAE0085E1878C08B2364C6F71325420FD0856F7FC3979D6917A1C9
0000004000000000000000000000008000000000000000000000000000000000
0000400000000000000000000000000000000000000000000000000000000000
0000000000000000000000000080000000000000000000000000000000000000
0200000000000000000000000000000100000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000000000000
AAEDAE2ED069B8B97BC518E9AFB25F3553D11896A3A693DA903D42E0AFE12D36
5F48708DBA116E2F30CC2D7AD481C220CD6A0635B305B73E16947815F9BD737C
5B73E16947815F9BD737C5F48708DBA116E2F38CC2D7AD481C220CD6A0735B30
0000000000000000010000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000080000000
0000000000000010000000000000000000000000000000000000000000000000
0000000000000000000000020000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
2F6AAD2810C5A0B06053182AB328D8E0E8DA07B8999287C275EE69662319210B
D011ED7E38EA5C38AF336885FDD829702DA40CC7763FD5162D36C01431F8BD5F
FD5162D36C01431F8BD5FD011ED7E38EA5C38AF336885FFD829702DA40CC7763
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
D34E03F0FAF44B477C7E185D8BEBD3F1B4E9DAD5F6C972CAA015B307DB3583DE
01FD385B82C3C25F785A294D6DAB65EB7F90DA889BFAE3B69C439BB95C2D6D67
AE3B69C439BB95C2D6D6701FD385B82C3C25F785A294D6DAB65EB7F90DA889BF
```

Generator Matrix for ARTM2 with $K = 4096$ and $R = 1/2$

The following list of hexadecimal character strings represents the generator matrix, **G**, for ARTM2 (ARTM CPM) with $K = 4096$ and $R = 1/2$. Each string has a length of 128 hexadecimal characters and represents the top row of a size-512 circulant ($M_L = 512$ for this code). This list follows the sparse representation of **G** defined in Section 3.2, which is actually the sparse representation of the submatrix **W** in (17). For this code, **W** is a 8×8 grid of (possibly) dense circulants, and so there are a total of $8 \times 8 = 64$ circulants in this list. These dimensions correspond to $K/M_L = 8$ and $M_D = 8$. Because of the length of the strings for this code, only 1 will fit on each line below and thus the circulants on 8 lines must be concatenated to form the 8×8 arrangement of circulants in the generator matrix:

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
54AFF20B67D5C09B8E6CF431754B7F9BADA55625A79BB3A7C1EAB0A63AD167AC5A12FD8352181A99B2DD08F112072EEFABF151C6905CF1D98D80F88E2D9ADB5B
00000000000000000200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
5A169F07CFFB701189C03592F4DD93569B42E912111ECA0BDC535E4F378D67761B46C5279BB09853E37D99EEAF09440CEFC89389A5253E6A81EEF8DE7152C657
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A41EACD236E8C9A6CBFAE7311EED96D692F08FE132C46A8AD994DCD945D3144934974D1D28ADACDCC98370938C0FD46A71B50A96D121FD8C6DEF10F92F14CF5E
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
7731462A64854DBF13FDED1265DE3350663F3ADABC38786B87A505ADC0F3571AF4E5BBDB2007C9AFB269683B2B91FF04232B01C40FC63F1ED7C6F037055BC630
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000020000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
85EB34A47E2E7A0A0CFDA092D86864E1FBB8A8F8316C8EAE8C13F4CAF4FD61620EBE9667CA6450E793914D2142E26F79DC66376E271FDD1421D6B2FFC9D13177
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
B99E6E49D0310979CD553E9833C1D3B5CCA80C3498FF56C66359111818A0C041BF5B2C194052EF5B8BF39713A632B3F537A947FD422C3B6E555C209EFC4B4272
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
A73180F0D5E4DCE5944411C79E480745D7B0C91B60C840EE1A05210BA869E221F80A3B2A3EE139B325B4162EFEE76487A705A69C1B764F67119B1FC00BB30D1B
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
3AD0A5044B3E9B2D828ED665661E3F8F4B20C6E2F524AD5DA4321850DCD3F482F90EA1C854D38BE38527C2A16BE026D453678C19ADDCB04D72EAAE3B4DA4D4A6
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
CB1A287CE11A70FCB9FA91FAF3CA0885B0642EAE79F5C6012E4E55E9A97433664B077B9EADC64F13FCD8CF2C720BCE99563C7576FCB60A222768C33603457C7E
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
DB82266814DC0AB2AE20909DAD7D4050D26948864771EBEA5A8A6E34A0BB5B298E875349EC7630377F5CF6A7C8BB3B4BD244C6AC25FF755CE02D912229966855
00020000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
3988480B8A443E06EB47DE4B86BB841C20F50871F354F84FBC864075DDF49F52CE6F852FED5E0F3BCB754E5FA2CEBD0100C8B8D739E41E3BACAF484E0AD349DF
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
03680F20BA3BFAC2140AF1C20BC35F960BF606F55EFED16E9811D9E5C318AB362C75820120263C13D46B9C033C0F5BF73C4ACA9724140315CADCE21EA701D823
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
82D1FEDDA5D66BF9336E6BBA40BC9CB6FCC70F48F9C432B75D28464AB56E920BF53D3D0FAD4B82390823510CB6FA6CC8CC6A6CCACA1DF32D03B7BD5EC620C188
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
47F869C2152BD98CC30763694AEF197736F01C3CD9FBB71CBAFA8A375CD3C4E3F0FBFCB5E4A7194C362CD0ECBDB0B246E949794B7F081F72FB29D4B0E3A24FE2
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
27A634C8F1B55FB94D684E76306460FA2E0CF72BD95A9AD15FC3410F84C9982B66319735D6CF046ED5BA723BE024D0298301DE71CC0218EB72B486C483976B8B
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
53590D3749FDB0F7A4F383BA5831441F1DCEC8BC01DE87E06299477F0BDFBB1FEAEDEFCEB8CE671F7A5DD92DF67CE4164F16FFE11261C622C7CABC6A2B983999
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```