LDPC code overview and testing

Comms and OBC Report

November 27, 2018

Konstantinos Kanavouras, Electra Karakosta-Amarantidou

1 Summary

After studying the performance of a variety of codes recommended for high data rate transmission in [1], we are going to focus on the operation principles and real life testing of LDPC (Low-Density Parity-Check).

2 Complete Tasks

2.1 Code operation

LDPC is a **linear parity check code**.

- **Parity** check codes rely on the addition of parity bits to the incoming message. Let us assume that we add one parity bit to the info bits. Given that one bit is flipped, we can detect this error having predefined whether the sum of 1's in the message should be even (even parity) or odd (odd parity). This is implemented by performing the exclusive OR operation to the bit stream.

- **Linear** (n,k) codes use n bits to provide protection to k bits of infromation. In order to do so, they construct a set of linear equations.

For demonstration purposes, we consider Hamming linear block code of 3 parity bits. With 3 parity bits, we have 3 **parity equations**, which can identify up to $2^3 = 8$ error conditions. One of the conditions corresponds to no error, so ultimately $8-1-3 = 4$ information bits can be protected. A parity equation is assigned to each parity bit. In a $(7,4)$ Hamming code, the parity equations are determined as follows:

- The first parity equation checks bits 4, 5, 6, 7
- The second parity equation checks bits 2, 3, 6, 7
- The third parity equation checks bit 1, 3, 5, 7

Location 5 has the binary representation of 101, thus appears in equations 1 and 3. By applying this rule, we obtain the tables bellow, where *d* and *p* denote data and parity bits respectively.

Bit number	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{d_1}$	$\mathbf{p_3}$	$\mathbf{d_2}$	$\mathbf{d_3}$	$\mathbf{d_4}$
Equation corresponds to $\mathbf{p_1}$	1	0	1	0	1	0	1
Equation corresponds to $\mathbf{p_2}$	0	1	1	0	0	1	1
Equation corresponds to $\mathbf{p_3}$	0	0	0	1	1	1	1

Figure 1: Hamming (7,4) Full table

	$\mathbf{d_1}$	$\mathbf{d_2}$	$\mathbf{d_3}$	$\mathbf{d_4}$
$\mathbf{p_1}$	1	1	0	1
$\mathbf{p_2}$	1	0	1	1
$\mathbf{p_3}$	0	1	1	1

Figure 2: Hamming (7,4) Abbreviated table

For larger dimension block codes, a matrix representation of the codes is used. Specifically, linear parity check codes encode data streams by multiplying them with *Generator matrices* $\mathbf{G}$ and decode them accordingly with *Parity Check Matrices* $\mathbf{H}$. Given a message $\mathbf{p}$, the codeword $\mathbf{c}$ will be

$$\mathbf{c} = \mathbf{G}\mathbf{p} \tag{1}$$

The received codeblock is represented as $\mathbf{y}$. If no bit is flipped during transmission, then $\mathbf{y} = \mathbf{c}$. Otherwise, $\mathbf{z}$ array, for which

$$\mathbf{z} = \mathbf{H}\mathbf{y} \tag{2}$$

returns the position of the error in binary form. From the aforementioned it is also evident that

$$\mathbf{H}\mathbf{c}^{\mathbf{T}} = \mathbf{0} \tag{3}$$

LDPC utilizes **sparse H** matrices to minimize complexity. A **sparse matrix** is one which contains a very small amount of 1's, row and column-wise (small row and column weights), and can be denoted as in the following figures.

Subsequently, we focus on the encoding procedure of the coding scheme, as in the current phase of our work we are mostly interested in the ability of LDPC to be utilized for downlink operation. In the future, we will elaborate on decoding techniques as well.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 3: Dense format

$$\mathbf{H} = \begin{bmatrix} 0 & 2 & 4 & 6 \\ 1 & 3 & 4 & 6 \\ 2 & 4 & 5 & 6 \end{bmatrix}$$

Figure 4: Sparse format

Encoding

The cost (number of operations) of utilizing the generator matrix $\mathbf{G}$ of a linear block depends on the Hamming weights (number of 1's) of the basis vectors of $\mathbf{G}$. If the vectors are dense, the cost of encoding using this method is proportional to n^2. This cost becomes linear with n if $\mathbf{G}$ is sparse. However, LDPC is given by the null space of a sparse parity-check matrix $\mathbf{H}$. It is unlikely that the generator matrix $\mathbf{G}$ will also be sparse, rendering the straightforward method of encoding LDPC proportional to n^2. This is too slow for most practical applications. Hence, it is desirable to have encoding algorithms that run in linear time.

There are different approaches to attain the above attribute. One of them is the *Lower triangular modification method*. In its essence, this method converts $\mathbf{H}$ into a low triangular form through matrix permutations, which is also sparse, and has the following form:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ \mathbf{C} & \mathbf{D} & \mathbf{E} \end{bmatrix} \tag{4}$$

By multiplying $\mathbf{H}$ from the left with

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{E}\mathbf{T}^{-1} & \mathbf{I} \end{bmatrix} \tag{5}$$

and exploiting the (3) relationship, we construct a linear system from which the parity bits can be extracted by performing back substitution, among other matrix operations. Back substitution is made possible due to the triangular form of $\mathbf{T}$. More detailed information regarding the contents of this chapter can be found in [7].

2.2 Testing

Unfortunately, there are no COTS components available that provide a **physical implementation of LDPC**. As such, LDPC can only be implemented in an FPGA, or in software within an MCU (Micro-Controller Unit).

In order to evaluate the feasibility of the implementation of LDPC aboard the CubeSat's MCUs, practical implementations of LDPC encoding and decoding were tested on STM32 boards. In particular, these open-source libraries were used:

- **freetel/codec2-dev**
  The freetel/rowetel implementation of LDPC is **optimized for STM32 microcontrollers**, bearing low memory consumption and high performance on low-end systems. The optimization process is described in [3].

- **tavildar/LDPC**.
  Saurabha Tavildar's implementation of LDPC is more flexible than Freetel's, and is accompanied by a test framework including modulation and random noise.
  The **H** matrices for the specific code are extracted from IEEE's WiFi 802.11n standard. Particularly, as elaborated in [8], three subblocks (27, 54, and 81 bits) and four code rates (1/2, 2/3, 3/4, 5/6) are comprised in the standard, making a total of 12 base matrices. Before beginning the encoding process, the base matrix is expanded to generate the **H** matrix in the following manner:

(a) Coding rate R= 1/2.

0	-	-	-	0	0	-	-	0	-	-	0	1	0	-	-	-	-	-	-	-	-	-	-
22	0	-	-	17	-	0	0	12	-	-	-	-	0	0	-	-	-	-	-	-	-	-	-
6	-	0	-	10	-	-	-	24	-	0	-	-	-	0	0	-	-	-	-	-	-	-	-
2	-	-	0	20	-	-	-	25	0	-	-	-	-	-	0	0	-	-	-	-	-	-	-
23	-	-	-	3	-	-	-	0	-	9	11	-	-	-	-	-	0	0	-	-	-	-	-
24	-	23	1	17	-	3	-	10	-	-	-	-	-	-	-	-	-	0	0	-	-	-	-
25	-	-	-	8	-	-	-	7	18	-	-	0	-	-	-	-	-	-	0	0	-	-	-
13	24	-	-	0	-	8	-	6	-	-	-	-	-	-	-	-	-	-	-	0	0	-	-
7	20	-	16	22	10	-	-	23	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
11	-	-	-	19	-	-	-	13	-	3	17	-	-	-	-	-	-	-	-	-	-	0	0
25	-	8	-	23	18	-	14	9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0
3	-	-	-	16	-	-	2	25	5	-	-	1	-	-	-	-	-	-	-	-	-	-	0

Figure 5: Matrix prototype for codeword block length n=648 bits, subblock size is Z = 27 bits

- A value of "0" corresponds to a sparse sub-matrix that is an identity matrix, exhibiting a diagonal of I.
- If the value is $n > 0$, the identity matrix is rotated clockwise n times.
- If the value is "-", the submatrix is 0.

Both implementations were tested on STM32F1, STM32L4 and STM32H7 boards, as well as on a PC. The source code corresponding to the test is available on Gitlab's `obc/obc-docs`.

2.2.1 Results

The most significant metric of the implementation is the **throughput**, i.e. the speed in which the data is processed. Ideally, the throughput should be more than the chosen transmission/reception *info data* rate. Throughputs are given in *information* bits/second[1] (higher is better).

		STM32F1	STM32L4	STM32H7
tavildar (648, 324)	1/2	*not enough memory*	*not enough memory*	6455 bit/s
freetel (224, 112)	1/2	*not enough memory*	904 bit/s	4884 bit/s
freetel (2580, 2064)	4/5	*not enough memory*	*not enough memory*	4280 bit/s

Table 1: Decoding throughput

[1]Data rates are given in *kilobits* and *Megabits*, where 1 Mbit = 1 000 000 bit = 10^6bit.

		STM32F1	STM32L4	STM32H7
tavildar (648, 324)	1/2	*not enough memory*	*not enough memory*	0.376 Mbit/s
tavildar (648, 432)	2/3	*not enough memory*	*not enough memory*	0.559 Mbit/s
tavildar (648, 486)	3/4	*not enough memory*	*not enough memory*	0.664 Mbit/s
tavildar (1296, 648)	1/2	*not enough memory*	*not enough memory*	0.347 Mbit/s
tavildar (1296, 864)	2/3	*not enough memory*	*not enough memory*	0.517 Mbit/s
tavildar (1296, 972)	3/4	*not enough memory*	*not enough memory*	0.630 Mbit/s
freetel (224, 112)	1/2	0.079 Mbit/s	1.600 Mbit/s	10.181 Mbit/s
freetel (2580, 2064)	4/5	*not enough memory*	1.564 Mbit/s	13.696 Mbit/s

Table 2: Encoding throughput

	STM32F1	STM32L4	STM32H7
Current	36 mA	11 mA	84 mA
Power	130 mW	40 mW	302 mW

Table 3: Theoretical power consumption during operation

		1-core Encoding	1-core Decoding
freetel (2560, 2054)	4/5	582 228 kbit/s	200 kbit/s
tavildar (648, 324)	1/2	106 146 kbit/s	19 kbit/s
tavildar (1944, 1458)	3/4	674 124 kbit/s	36 kbit/s

Table 4: Throughput on a personal computer

The above results show that **decoding data on the MCU is out of the question**, unless the uplink data rate is extremely slow, i.e. lower than ~ 5 kbit/s.

On the other hand, **encoding LPDC data is possible**, although some cases are marginal. More specifically, the < 0.7 Mbit/s throughputs might need to be increased, depending on the data rate requirements and our link budget. Freetel provides an extremely fast encoder that, if implemented, will ensure an easy LDPC implementation.

Note that a **high-performance** microcontroller will have to be used. The lack of a Floating Point Unit and the reduced "horsepower" of the STM32F1 series renders it impossible to use for LDPC. Thankfully, higher-grade microcontrollers from STMicroelectronics bear no significant cost or software complexity differences.

Decoding rates on PC may not be fast enough, but can be easily increased by using multiple processor cores, a GPU implementation [2], or delayed processing using a buffer.

2.2.2 Implementation considerations

In order to adapt the code for use in a resource-wise constrained environment, a few modifications had to be made to ensure maximum performance:

- STM32H7 data and instruction **caches** had to be enabled
- The code was compiled with full optimisation (`-O3`)
- There was careful memory allocation. For STM32H7, different memory areas had

to be configured in the linker script. Some variables could also be made `const` so they are not doubly stored in the RAM.
- Variable sizes were reduced as much as possible.
- Instead of generating **H**-arrays from prototype matrices on the Tavildar algorithm, they were stored in the code as constant sparse matrices.

Also note that, in contrast to decoding, the *encoding* operation takes *constant* time to run based on the block length, not dependent on the message content.

2.2.3 Optimization of Tavildar's encoding algorithm

Using callgrind and KCachegrind, Tavildar's *encoding* algorithm was profiled, in order to find the major slow points that could be improved for usage in embedded systems.

After the analysis, the following optimizations were performed:

Optimization description	Time taken
`vector.at()` calls converted to `vector[]`	94.65%
Used ^ operator instead of addition+modulo for `XOR`s	89.79%
Moved calls to `vector.size()` for each row outside of the loop	79.11%
Moved conditions for `_row_mat` to the `for` loop	92.62%
Stored start & stop points in a matrix, instead of testing on every loop	57.44%
Moved start & stop matrix accesses for each row outside the loop	88.69%

Table 5: Optimizations on Tavildar's LDPC encoding function

The total theoretical elapsed time was reduced to 31.7% of the original. When run on the MCU, the time was reduced to 39.0% of the original (corresponding to a 2.6× speed increase).

The simulation algorithm confirmed that the LDPC results after the optimization were not changed, thus proving the correctness of the analysis. Table 2 contains the results of the *optimized* code.

2.2.4 Encoding performance

While encoding with LPDC seems feasible based on the experimental measurements, we might need to implement even faster encoding, in order to accomodate possible larger data rates or block lengths. This could be done using the following (complex & power-wasting) solutions:

- Multiple MCUs performing many operations at once,
- A separate high-performance chip that will only calculate LDPC codes,
- An external RAM chip,
- An FPGA, or
- Further attempts for code optimization.

The *multiple MCU* option seems the most likely one, as it can simultaneously implement operational redundancy.

As far as **memory** is concerned, an AR4JA $N = 24576$ parity-check matrix was used to calculate memory consumption. An efficient way to store sparse arrays is to store their elements in pairs; therefore, each of the 94208 nonzero items on $\mathbf{H}$ takes up 2+2 bytes, for a total of:

$$94208 \cdot (2 + 2) = 368\,\text{kB}$$

which is quite less than the amount of available RAM or ROM on many STM microcontrollers.

2.3 Post-processing

2.3.1 Link Budget

For evaluating the results presented in the previous chapters of this report, we firstly constructed a hypothetical link budget for the **downlink** employing AMSAT's corresponding tool. The configurations made were:

- Downlink Frequency: 2.405 GHz
- Transmitter Power: 2 W
- Modulation/Demodulation: OQPS with LDPC encoding (**Required** $\frac{E_b}{N_0} = 4.5, BER = 10^{-6}$)
- Spacecraft Antenna: Dualband circular patch with maximum gain 8 dBi
- Ground Station Antenna: 22 element Yagi, gain 21.7 dBi
- Noise floor at Ground station: -105.5 dB, as measured in [6]
- Receiver bandwidth: 2.3 Mhz (from AT86RF233's datasheet)

Sat TX, 2W (dBm)	33
GS Gain (dB)	21.7
Path Loss (dB)	157
Sat Gain (dB)	8
GS RX (dBm)	-119.3
GS Noise (dBm)	-105.5
250kbps Margin	12.9
500kbps Margin	9.9
1Mbps Margin	6.9
2Mbps Margin	3.8

Table 6: Downlink Budget

Given that we want to operate on high data rates, since for a low cost system like the one in question a **10 dB margin** is recommended to make up for any losses not taken into account, the required $\frac{\mathbf{E_b}}{\mathbf{N_0}}$ **should be at most equal to the 4.5 considered for the model.**

2.3.2 Evaluation

In this section, we evaluate the performance of the aforementioned LDPC implementations regarding their suitability for our mission. The coding rates of interest are $\frac{2}{3}$, $\frac{3}{4}$ and $\frac{4}{5}$.

As we can see in Figure 6, Tavildar's code is satisfactory in nearly all available alternatives.

In contrary, Freetel's $\frac{4}{5}$ $(2580, 2064)$ requires that $\frac{E_b}{N_0} \approx 6$ in order to achieve a 10^{-6} BER (Figure 7), a value which is deemed non functional and therefore can not be utilized in our case, despite the outstanding difference in speed, when compared to Tavildar's implementation.

We also review the BER values of CCSDS recommended [5, 4] AR4JA codes, pictured in (Figure 8). It is evident that the specific set of codes operates very well even for higher coding gains and should be tested in future steps. A possible way to do so would be to infuse the correlative parity check matrices into Tavlidar's code, although further study of the differences between the various encoding techiques is required. AR4JA **H** matrices are available in the Pretty Good Codes website.

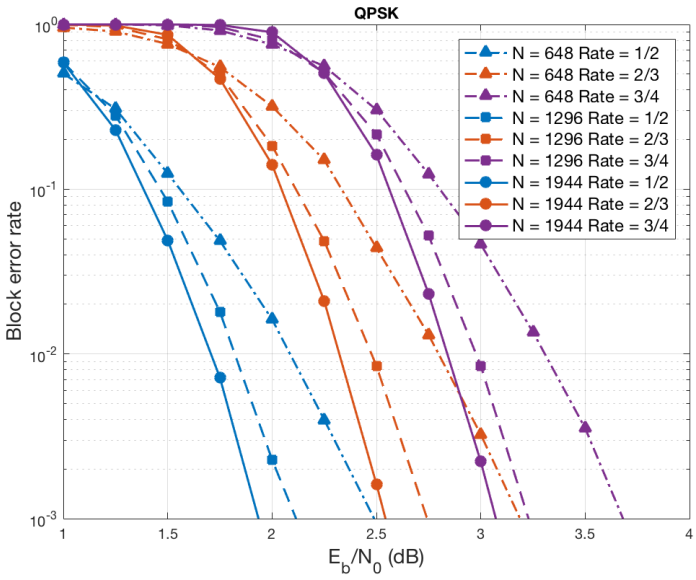

Figure 6: Tavlidar's LDPC implementation decoding performances

3 Conclusions

- The decoding procedure is not optimized for use on Microcontrollers. On the other hand, encoding is considerably faster, thus proposed for **downlink** operation.

- The amount of data required for monitoring the scientific payload needs to be at least approximately determined in the immediate future/

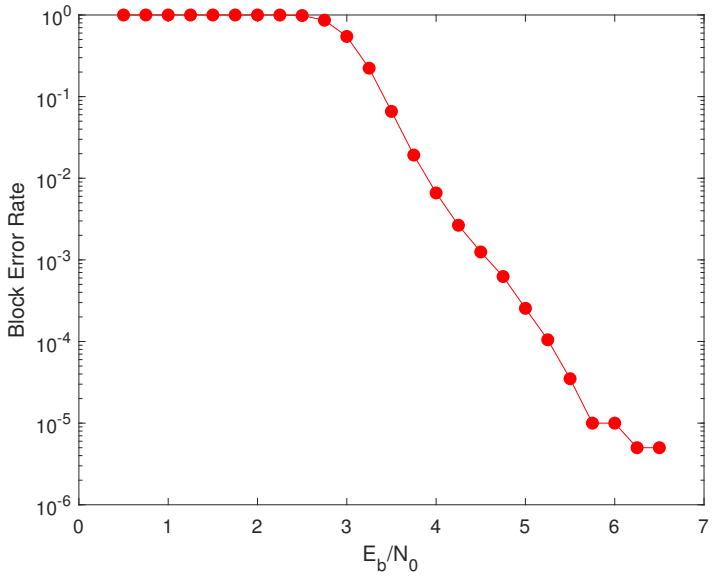

Figure 7: Rowetel's $\frac{4}{5}$ $(2580, 2064)$ decoding performance

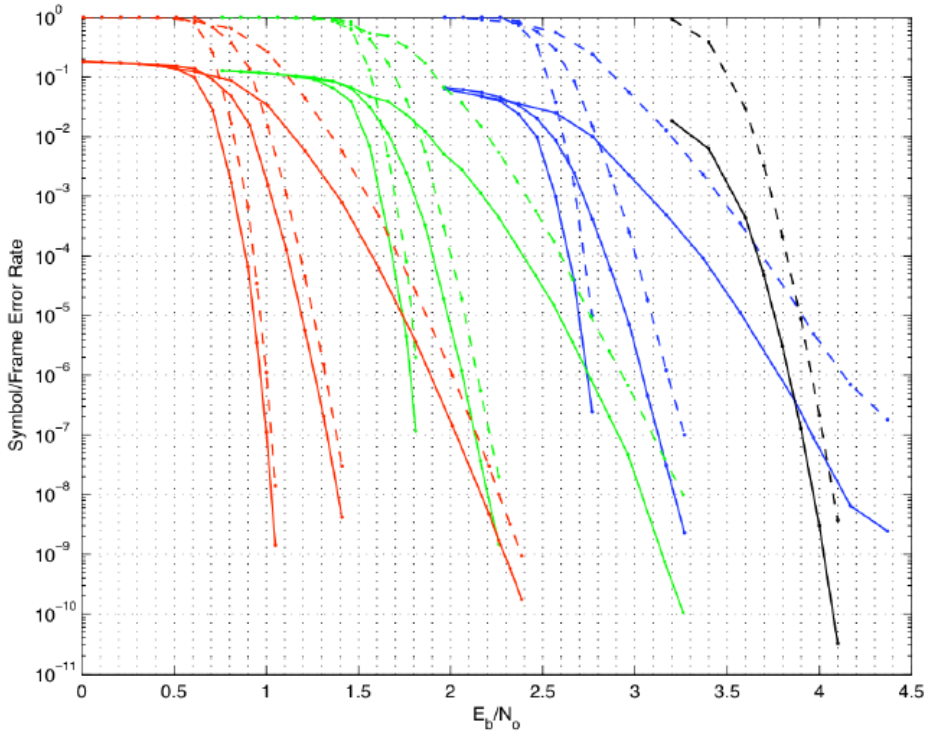

Figure 8: Bit Error Rate (Solid) and Frame Error Rate (Dashed) for Nine AR4JA Codes and C2, with Code Rates 1/2 (Red), 2/3 (Green), 4/5 (Blue), and 7/8 (Black); and Block Lengths k=16384, 4096, 1024 (Left to Right in Each Group), and 7156 (Code C2)

- There are **four** factors that affect the choice of the most appropriate coding rate and block size of the code:
  1. the useful bit rate of the selected coding rate
  2. the SNR
  3. the maximum info bit rate that the MCU can provide
  4. the complexity of the design to employ the desired block length

4 Incomplete Tasks

- Final selection of LDPC algorithm, rate and block size
- Decoding mechanisms study
- Further optimization of LDPC algorithms. Possible usage of DSP peripheral, an FPGA, a dual-MCU setup, bitsets for generator matrices, or research results.
- Test of LDPC implementation on transceivers
- LDPC Implementation on ground station: Possible multi-core or GPU utilization?
- Simulations and tests with CCSDS **H**-matrices
- Evaluation of Freetel's encoding method, and its differences to Tavildar's

References

[1] Comms subsystem report, *Enhanced Performance Error Correction Techniques*

[2] Jan Broulim, Alexander Ayriyan, Vjaceslav Georgiev, Hovik Grigorian, *OpenCL/CUDA algorithms for parallel decoding of any irregular LDPC code using GPU*

[3] Donald Reid, David Rowe, *Porting a LDPC Decoder to a STM32 Microcontroller*

[4] CCSDS 131.4-M-1, *TM CHANNEL CODING PROFILES*

[5] CCSDS 130.1-G-2, *TM SYNCHRONIZATION AND CHANNEL CODING— SUMMARY OF CONCEPT AND RATIONALE*

[6] Graig Lee Francis, *ISM S-BAND CUBESAT RADIO DESIGNED FOR THE POLYSAT SYSTEM BOARD*, Master Thesis

[7] Tuan Ta, *A Tutorial on Low Density Parity-Check Codes*

[8] Yi Hua Chen, Jue Hsuan Hsiao, Zong Yi Siao, *Wi-Fi LDPC Encoder with Approximate Lower Triangular Diverse Implementation and Verification*