Reliable Embedded Systems Design Project: LDPC Codes Simulator

Alessandro de Gennaro, Daniele Jahier Pagliari

June 30, 2014

Contents

1 Introduction

This document describes a software environment for the simulation of Low-Density Parity-Check (LDPC) codes, developed as a final project for the course in Reliable Embedded Systems Design at Politecnico di Torino, during the academic year 2013/2014. LDPC codes are a subset of the family of linear block codes characterized by a very sparse parity-check matrix. This property allows to implement efficient decoding using iterative algorithms, which in turn guarantee some very interesting statistical properties in terms of error correction [2]. For these reasons, LDPC codes are currently widely studied, especially in research related to reliability and fault tolerance for Solid State Drives [1].

The simulator that we implemented is meant to function as a *golden* model to which hardware implementations of LDPC error correction algorithms for SSDs and Flash memories can be compared for verification purposes. It covers the main aspects of an LDPC code-based system, including generation of new codes, encoding, simulation of the effects of faults occurring in the memory through an equivalent channel model, error correction and extraction of the message bits from the corrected codeword. LDPC codes being a very wide and diverse set of codes, we focused our work on building a simulation framework suitable for future extension if a new family of codes needs to be tested, rather than specializing on a particular code generation or error correction strategy. As a study case for this architecture, we implemented one of the most common code generation solution [3] and two iterative error correction algorithms [1]. Standard implementations for encoding, decoding, parity-check data storage and channel model are provided as well.

2 Simulator Internals

The simulator was written using the C++ language, in order to exploit the advantages of Object Oriented Programming in terms of abstraction and interfacing, which are key to the flexibility and extensibility of the code. As it is known, the process of storing data in a memory system subject to faults can be modeled as a transmission on an equivalent communication channel [1]. We identified the main components of a generic communication over a channel that uses linear encoding for error detection/correction, and for each of these elements we implemented an abstract class describing its interface. The result are five blocks, which represent the main core of our software:

- Parity Check Matrix: the H matrix for the target linear code. It contains the information needed to perform parity check, and via some intermediate steps, also for encoding and decoding (class name: `pcheck_mat`).
- Encoder: receives a message word and generates the corresponding codeword. It needs parity-check information at creation time (class name: `encoder`).
- Channel: implements a channel model with a certain error modality and probability. It receives as input the encoded word and produces a (possibly corrupted) output (class name: `channel`).
- Error Corrector: receives the output from the channel and the parity-check information, and attempts to perform error correction adopting a certain algorithm. It produces a codeword as a result (class name: `error_corrector`).
- Decoder: receives the output of the Error Corrector and the parity-check information, and extracts the message bits from the codeword (class name: `decoder`).

In addition to those blocks which are always present, there is another optional element in the chain: the Code Generator (class name: `generator`). This is needed if the user of the simulator wants to generate a new LDPC code parity-check matrix using one of the many existing algorithms (typically based on random approaches with specific constraints) [3]. Clearly, the Parity Check Matrix can also be inserted as external input to the simulator, and the generation part can be skipped.

The general architecture of the simulator core described above is summarized in Figure 1, where only the main methods for each interface are reported.

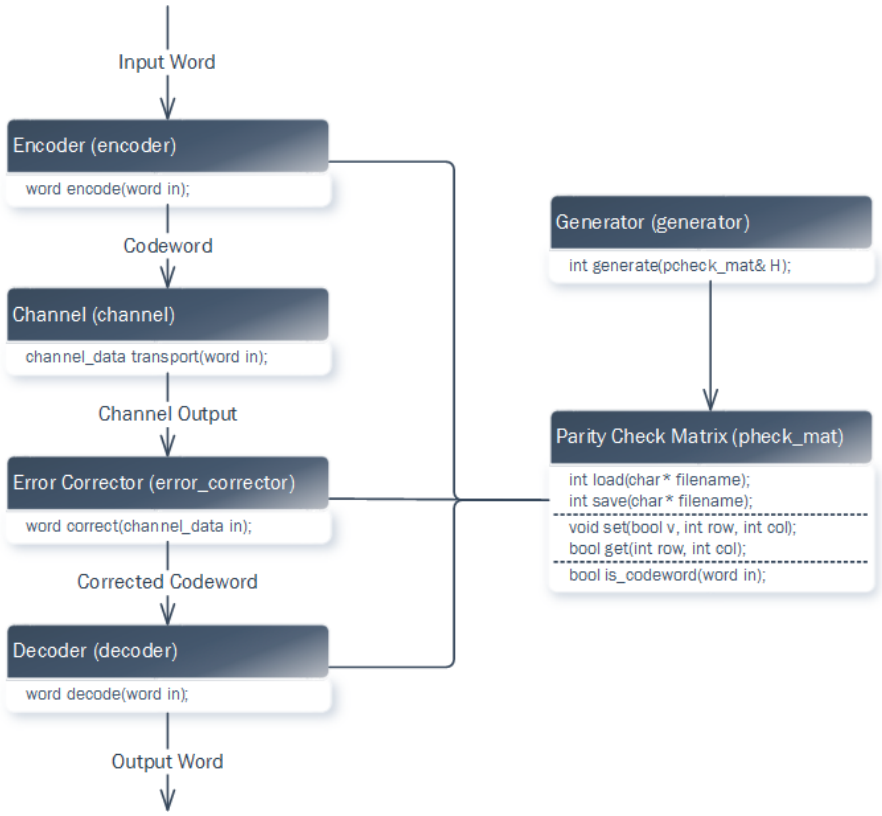

Figure 1: Internal architecture of the simulator.

The usage of an abstract class for each element of the system allows to easily extend the simulator functionality. In fact, each part of the simulator ignores implementation details of the others, and only exploits their very simple interface. Thus, if a new algorithm has to be simulated and studied, it is sufficient to inherit from the corresponding abstract class and implement the pure virtual methods defined therein. We will now discuss the details of each of these classes and some other support objects used to connect them, describing the current implementation and providing examples of how extensions can be added.

2.1 Word

Strings of bits are represented in the simulator in the class `word`. Basically, it is a wrapper for an array of boolean values, plus some additional utility members, in particular the length of the array and an error flag. A `word` object can be constructed empty by simply specifying the length. Alternatively, it can be initialized in different ways:

- specifying another boolean array containing the initialization values for each bit.
- passing an integer value containing the value to represent. This works for words of length up to 64 bits.
- passing a string of `char` elements with value '0' and '1'.
- passing a reference to an object of class `channel_data`. See Section 2.2.

Wrapping words inside a class allows some convenience features with respect to using arrays directly, such as easily determining the length, setting the error flag if initialization values are incompatible (e.g. if the string contains ASCII codes different from '0' and '1') or if there is a length mismatch between a class' method and the word passed to it as input, etc.

Besides the usual setter and getter methods, it is important to underline the presence of a friend function to compute the Hamming Distance between two words:

```
/* hamming distance */
friend int hamming_distance(word& w1, word& w2);
```

This function is used to count the number of different bits between two words (i.e. the number of errors) when computing statistics during the simulator execution.

2.2 Channel Data

In Section 2.5 we describe how we implemented the equivalent communication channel used to model the behavior of the memory under simulation. However, it is important to anticipate that all descendants of the abstract class `channel` receive a codeword (an instance of class `word` described in

Section 2.1), but return a different object, belonging to class `channel_data`. This data structure is very similar to a word, but its internal values are of type double, rather than boolean. The reason why channel simulation does not output a boolean word is that some advanced channel models (e.g. those used to model MLC Flash memories) can produce more than two discernible values, due to the decision strategy implemented in hardware to select the cell value, as described in Section 11.3 of [1]. To account for an implementation of these models to be added to the simulator in the future, we used floating point values as channel outputs, even if for simpler models (such as that for SLC Flash), a boolean array would have sufficed.

2.3 Parity Check Matrix

The parity check matrix (typically called H in literature) is the fundamental data structure of any linear code, and as a consequence, of the simulator. Theory states that this matrix has dimensions $(n-k) \times (n)$ where k is the length of the message to be transmitted (or equivalently the data to be stored in memory), and n is the length of the *codeword*, i.e. the word produced by encoding, containing message bits and parity check bits. We will use these two letters to denote those two quantities in the rest of the document.

The Parity Check Matrix (`pcheck_mat`) interface we devised is composed of the following methods:

- `int load(char* filename)`: loads the parity-check information from the path specified in filename. Following a UNIX convention, it returns 0 in case of success, and -1 otherwise.
- `int save(char* filename)`: saves the parity-check information to the file specified in filename. The return value has the same semantic as the load method.
- `void set(bool v, int row, int col)`: set the element $H(row, col)$ of the parity-check matrix to value v. The value to be set is passed in boolean format, with true meaning binary 1 and false meaning binary 0.
- `bool get(int row, int col)`: returns the binary value corresponding to $H(row, col)$.

- `int get_n(void)`: returns the codeword length n.
- `int get_k(void)`: returns the message length k.
- `bool is_codeword(word& w)`: returns true if the input word w is a codeword for the code described by H, false otherwise.
- `void print(ostream& os)`: prints the content of matrix H. A stream friend operator (<<) is also defined, so that H can be printed using the usual C++ stream notation.

In the present version of the simulator, this interface is implemented by the class `pcheck_mat_dense`, which as the name says, provides a *dense* representation for H. Basically, H is stored in the most obvious way, as a 2-dimensional array of bool elements. We named the class in that particular way to underline that this is the simplest but not the only possible way to store H. In fact, LDPC matrices are characterized by sparseness, i.e. the number of ones is generally much smaller than the number of zeros. So, another possible data structure that could be used is an array of rows, each storing the *column index* of the elements of that row that contain a binary 1. This would reduce the memory needed to represent H, both in RAM during execution and in the file system for storage. However, modern computing systems typically have huge disk space and RAM memory, so this optimization could be excessive. In any case, it works as a perfect example of how the simulator could be extended. If a new class `pcheck_mat_sparse` is defined, it simply has to inherit from `pcheck_mat` and implement the previously described virtual functions, and the rest of the system will keep working without any change. Clearly, some other components of the simulator would have been more efficient and faster if they had knowledge of the internals of the H matrix representation, but as we already discussed, we chose flexibility over efficiency.

2.4 Encoder

The second fundamental block of the simulator is the encoder. The purpose of this block is to take a word of length k (a message) and generate the corresponding codeword of length n. The only function defined by the interface is:

```
virtual word encode(word& in) = 0;
```

Systematic Encoder

Several algorithms for encoding of linear block codes exist, but the standard one is based on the product between the message and the so called *generating matrix* (G), of size $k \times n$. This is the solution implemented by class `systematic_encoder`, which inherits the encoder interface and internally stores matrix G in a 2-dimensional boolean array. The encoding function simply consists in a matrix product. The generating matrix used by the systematic encoder, is in *standard* or *systematic* form (hence the class name), meaning that all message bits are at the beginning or at the end of the codeword. Section 1.3.1 of [3] describes how to construct G in systematic form, starting from a generic H. Such transformation requires to reduce a part of the parity check matrix H to the identity I_{n-k} using the Gauss-Jordan elimination algorithm. To avoid the usage of external libraries, which typically do not deal with modulo 2 arithmetic and in general make the code less portable and more difficult to compile, we coded the Gauss-Jordan algorithm in a private method with the following prototype:

```
void systematic_encoder::gauss_reduce(pcheck_mat& H, pcheck_mat& H2);
```

A few details are important to understand the functionality of our systematic encoding class. First of all, the cited article from MacKay describes the situation in which message bits are placed in the most significant bits of the codeword (i.e. to the left). In other words, matrix G is in the form:

$$G = [I_k|P]$$

To simplify the function `gauss_reduce()` (in terms of readability, computational complexity is the same) we decided to use the reverse representation, where the message bits are in the least significant positions of the codeword, and matrix G has the form:

$$G = [P|I_k]$$

The two solutions are perfectly equivalent from any point of view, and both are referred to as *standard form* in literature. Except for this detail, the code that generates matrix G from H can be understood following the explanation in [3].

Another important consideration is that, as the article underlines, H is not a square matrix, so Gaussian elimination may require swapping some of its columns to succeed. This process transforms the original code into a so

called *equivalent code* [2]. The error correcting properties of two equivalent codes are exactly the same, so this transformation does not change the performance of the selected code. However, it must be taken into account in some situations. In particular, when an instance of class `systematic_encoder` is constructed, the constructor takes matrix H as a parameter, and for the reasons just explained, it could modify it (the only possible transformation being the exchange of two or more columns). So, when a new matrix H is generated using a generator class (see Section 2.8), there is not guarantee that the parity check information will actually remain the same, but only that the *properties* of the generated code will not change.

In any case, the simulator command line interface (CLI) proposes to the user to print and/or save the newly generated parity check matrix to file *after* the encoder has been constructed. By doing so, the matrix that will be saved to the file system is already modified selecting the leftmost $(n-k)$ columns to form an invertible sub-matrix, so it can be retrieved and reused in the future without further modifications.

The `encode()` function also checks that the input word has the correct length (k), and if this is not the case, returns an undefined codeword with the error flag set. Finally, check Section 2.7 to see how this particular implementation of the encoder must be paired with an appropriate decoder to work correctly.

2.5 Channel

The main target of the simulator are memory systems, in particular big NAND Flash memories, such as those found inside Solid State Drives (SSD). Section 11.3 of [1] explains how such a memory can be modeled, for our purposes, using an equivalent channel model. The memory is considered as a communication channel, where data is transmitted when it is written to the memory, and is received when it is read. During the period between these two events, an error might occur due to one or multiple fault modes; in the equivalent model, this corresponds to a certain error probability during the transmission. The abstract class that describes the characteristics of a generic channel model in our simulator is fancifully called `channel`. Its interface has two methods:

- `channel_data transport(word input)`: the main functionality of the channel is obtained by calling this function. It takes as input a word,

and produces as output the result of the transmission over the channel model implemented by the calling object (or in other words, the result of a read from a memory).

- `like_ratio(double x)`: This function returns the a priori likelihood ratio of the channel for a certain symbol x (a bit in the particular case of a binary channel). The meaning of this quantity is defined in Section 11.5.3 of [1].

As we discussed in Section 2.2, different channels have different types of output, so the return value of the `transport()` function is a general object of class `channel_data` rather than a `word`.

Binary Symmetric Channel

In the current version of the simulator, we modelled the most widely used channel model for SLC NAND Flash memories: the Binary Symmetric Channel (BSC). For this channel, each bit has a transition probability modelled as a Bernoulli random variable with the same success probability p. The overall channel, hence, constitutes a Binomial random variable with parameter p. The reason why an SLC memory can be modeled in such a way, and how the value of p can be calculated are issues discussed briefly in Section 11.3.1 of [1] and are beyond the scope of this document.

Class `bsc` contains the implementation of the channel interface in the case of the BSC. Its constructor takes the transition probability p as parameter, or it can also be called void, in which case a default p value is set. Note that the likelihood ratio has a slightly different formulation with respect to the one presented in [1], because in our environment, bits over the channel are represented with values 0 and 1 rather than ± 1 (it is just a matter of conventions). The `transport()` method for class `bsc` is implemented using the default `rand()` function of the C Standard Library, which returns an uniformly distributed value over the range of all integers. When reduced to the range $[0 : 1]$ and discretized, this Uniform random variable can be easily transformed into a Bernoulli. This avoids the need of including external libraries for discrete random variables in the code, which once again is an advantage for portability.

As in the case of most other components of our simulator, other descendant of the `channel` interface can be implemented. For example, this could

be the case if an user wants to take advantage of the simulator to test algorithms for MLC NAND Flash memories, which adopt a different equivalent model (Section 11.3.2 of [1]).

2.6 Error Corrector

Error corrector is probably the single most important element of the entire system. In fact, the efficiency of some probabilistic, iterative, error correcting algorithms is what distinguishes LDPC codes from the rest of linear block codes. Moreover the simulator is meant to test different error corrector strategies with different families of LDPC codes (possibly versus their hardware implementations) so this part of the chain, together with the code generator, are the most likely to be extended.

Again, we described a very simple interface (class `error_corrector`) which is common to all error correctors. This interface is composed of a single function, which prototype is the following:

```
virtual word correct(channel_data& in) = 0;
```

The semantic of this method is obvious: the output of a channel model is passed as parameter, and the corresponding codeword is returned, after attempting to correct possible errors.

Belief Propagation Error Correctors

Being the key part of the work, error correctors are where we focused our implementation effort, developing the code for two of the most used iterative strategies:

- Standard Belief Propagation Algorithm (BP)
- Logarithmic Belief Propagation Algorithm (Log-BP)

Both of these algorithms are described in Section 11.5 of [1]. For each of them we implemented a class, named respectively `bp_error_corrector` and `log_bp_error_corrector`. Both inherit from the `error_corrector` interface, but since the two algorithm share most of the private code, including members and methods, we defined Log-BP as a descendant of BP, maximizing readability and code reuse. This is yet another example of how the properties of OOP can be used to easily increase the simulator functionality.

The two iterative probabilistic algorithms we implemented need to have knowledge of the equivalent channel model for the memory, in addition to the parity check information of the code. Their execution is not guaranteed to converge to a result in a given number of steps, so a parametric maximum number of iterations has to be set too. Each algorithm will stop either when the tentative corrected word produced at the current step is a codeword of the target code (i.e. if the `is_codeword()` method of `pcheck_mat` returns true), or when the maximum number of iterations has been reached. Hence, the constructor has a similar prototype for both:

```
bp_error_corrector(channel& chn, pcheck_mat& H, int imax);
```

The implementation of the `correct()` method for BP and Log-BP is quite complex, and makes use of a number of auxiliary functions defined in the corresponding classes. We tried to keep all variables and functions names as similar as possible to the ones in [1], so using the book as a reference should make our code easy to understand.

One important point, however, requires further explanation: to improve error correction efficiency, the data structure used internally to these classes to represent the parity-check information is different from the one used in the rest of the simulator. It is important to underline that the latter can be used to build the former, so the global environment is completely transparent to this reshaping of data, which happens locally in our two error corrector classes. The reasons for this difference are explained in the following section.

Tanner Graph Data Structure

Most LDPC codes error correcting algorithms, and in particular both BP and Log-BP, can be described using a data structure called Tanner Graph (Section 11.4 of [1]). The same information contained in the Tanner Graph can be obtained from the parity-check matrix H (which is in fact the *adjacency matrix* of that graph), so having an explicit data structure to represent it is not fundamental, but it has some advantages.

In fact, BP and Log-BP behave as a sequence of *message-passing* steps over the edges of the Tanner Graph. An edge is present between Check Node i and Variable Node j (details in [1]) only if the corresponding element of the parity-check matrix $H(i, j) = 1$. As we know, LDPC matrices are by definition sparse, so 0 is by far the dominant value in H; as a consequence, it would be extremely inefficient to iterate over *all elements* of H multiple

times to check if an edge is present at every step of the iterative algorithm, especially for big matrices.

When discussing the Parity Check Matrix class in Section 2.3 we explained how we chose not to optimize memory occupation. The reason is that typically a single copy of H is contained in the simulator memory at any given time, and even if represented in a dense format, its memory footprint won't be bigger than a few KB, which are very easily dealt with by a modern workstation. Error correction, on the contrary is repeated for every test vector and a typical simulation run could receive thousands or millions of such inputs. So, while memory occupation does not represent a problem, a decrease in error correction performance could.

Because of all these considerations, we built an efficient data structure to represent the Tanner Graph, based on the *adjacency list* concept, which is better for sparse graphs. Two arrays of `node` objects are used to store Variable Nodes and Check Nodes, and each of their elements contains a list of edges. Nodes and edges don't need to change as long as the code used remains the same, so they are filled iterating over matrix H *only once* inside the constructor of the error corrector object. In particular, this is the task performed by private function:

```
void build_graph(void);
```

After that, when error correction is actually performed, the algorithm only scrolls the list of edges for each node, improving performance significantly. For convenience, both error correctors also store a reference to the original matrix H in one of their members, but the actual algorithm execution never makes use of that data structure for message passing. The only moment in which the parity check class is used is when the algorithms must check if a word is a codeword or not (calling the `is_codeword()` method) at the end of each iteration.

2.7 Decoder

The next step of the simulation chain is the decoder. This block is in charge of receiving the output of error correction and extracting message bits from the codeword. The interface that must be adopted by all implementations of a decoder is again extremely simple. The only required method is the following:

```
virtual word decode(word& in) = 0;
```

In most cases, a decoder implementation needs to be paired with the corresponding encoder. Intuitively, the reason is that the decoder needs to know the positions in which the encoder has placed message bits to extract them. However, we decided not to enforce this relation building a unified interface for the two, and left future users with maximum flexibility. Thanks to C++ multiple inheritance, a developer could very easily create a single class implementing both interfaces, if it wants to keep encoder and decoder strategies strongly connected.

Systematic Decoder

The decoder currently used in the simulator is implemented in class `systematic_decoder`, which is obviously meant to work with the encoder described in Section 2.4 (although implemented in a separate class for readability). It is perhaps the simplest object in the entire simulator. In fact, since codewords constructed with a systematic G matrix contain all message bits to the rightmost positions, what the `decode()` function must do is simply extract the last k bits of a codeword and return them. An error checking step is performed to make sure that the codeword length is the expected one.

2.8 Generator

We already mentioned that our software does not only allow to simulate a set of vectors with a preexistent LDPC code, but also to generate a completely new parity-check matrix. A significant number of different LDPC codes families exist (Section 11.4 of [1]), each with different properties and characteristics. In an effort to make the software adaptable to any of these families, we defined another abstract class with a common interface for all code generators. The method that must be implemented is the following:

```
virtual int generate(pcheck_mat& H) = 0;
```

where the parameter contains a reference to the parity check matrix that will be filled by the generator, and the integer return value is used with the usual UNIX convention to signal a failed generation. The field of LDPC codes constructions is so vaste that little could be added to that interface without risking to limit it to some families excluding others.

Even Ones Generator

We wrote an implementation of the generator interface just described, using one of the most common semi-random methods found in literature ([1], [3]). In particular, one family of LDPC codes which is known to have good properties is the one in which the parity-check matrix H is characterized by the same number of 1s (or *weight*) in each column and/or row, and the specific position for the 1s is selected randomly.

Section 1.3 of [3] describes how to generate such a matrix, and in particular it details a series of correction steps performed after the first generation to guarantee some useful properties on the generated H, such as avoiding a short girth in the corresponding Tanner Graph. Our implementation of this method can be found in class `even_ones_generator`.

The constructor for this class receives the parameters of the code to generate and the desired column weight (`ones` parameter) which must be an integer ≥ 3. The `generate()` method performs an initial control to make sure that the parity-check object received has the correct size, and then manipulates that matrix in four steps. After filling the n columns of H randomly with the specified weight of 1s, the result in analyzed to ensure that no row only contains 0s (like in any other linear systems, an empty equation is useless) and finally the guidelines of [3] are followed to improve the generated matrix. The code is extensively commented and should be quite readable; it makes use of a series of private auxiliary methods.

It is important to recall the discussion in Section 2.4 about the further modifications to H performed in the encoder constructor in order to have the corresponding generating matrix G in systematic form. These modifications are specific to that particular encoder implementation, but must be taken into account to understand the main flow of execution of the simulator.

2.9 Main

To conclude this description on the internal functionality of our LDPC codes simulator, we provide a brief description of the way in which the previously described blocks are connected in the main source file of the program (`main.cpp`).

This file defines a function `simulate()` which calls the appropriate methods on the various components of the system and performs the entire flow of encoding, transmission, error correction and decoding on a number of test

vectors (retrieved from file, as explained in Section 3). The main function prompts the user with a series of command line menus to select between available usage options and then calls `simulate()` to start the actual execution. Pointers to each of the fundamental blocks are declared at the beginning of the file as global variables, since they are used both from main and from the simulation function:

```
pcheck_mat* mat;
encoder* enc;
channel* chn;
error_corrector* erc;
decoder* dcd;
```

The generator, instead, is declared local to main, since code generation (or loading) happens just once at the beginning of each execution, before `simulate()` is called.

One important thing to notice is the fact that a new seed for pseudo-random number generation is created at every execution, using the time since Epoch:

```
/* random number generation seed */
srand(time(NULL));
```

This allows to have an unpredictable behavior of the channel, which resembles a real life system, and allows to evaluate statistically the performance of the error correcting algorithm over the selected LDPC family. An user may want to use a fixed seed instead, if he or she wants to obtain a deterministic and repeatable result from the channel model.

Most components of the system include basic parameter checking in their constructors and methods. During user input collection at the beginning of execution, these and other possible wrong parameters are checked, and meaningful messages are reported if they occur. The rest of the code in `main()` simply allocates instances of the system blocks using appropriate constructors. In particular, parity check matrix, encoder and decoder are created here, together possibly with the generator, if the user selects the option to create a new code.

Channel and error corrector, on the other hand, are constructed inside `simulate()`; this allows to perform multiple simulations with the same code but using a different error probability for the channel and/or a different correction algorithm, for sake of fast comparison between results.

The function prompts the user for the remaining parameters and settings before allocating the other two core blocks. Then, a while loop is entered, in which the entire simulation input file is scrolled, and every line is considered as an hexadecimal integer number corresponding to the vector to simulate; as we said the integer constructor for class `word` works for vectors of up to 64 bits. If longer stimula have to be used, the string constructor has to be used and the file has to be filled with binary values directly. We chose the other method because it looks more clean and readable. Each iteration of the loop:

1. Creates the test vector.
2. Encodes it.
3. Simulates transmission on the channel (memory storage).
4. Calculates the Hamming Distance between transmitted and received codewords (number of raw errors).
5. Checks if the result of transmission is still a codeword or not.
6. Launches the error correction algorithms.
7. Calculates the Hamming Distance between transmitted and corrected codewords (number of final errors).
8. Checks if the corrected bits represent a codeword or not.
9. Decodes the resulting codeword.

Obviously, checking if a word is a codeword is *not* a measure of whether it contains errors or not, only the Hamming Distance is. However, we left it as an additional information that can be collected together with the global results to better understand how the error correction algorithms work. When the entire file has been processed, the total and average number of raw errors and final errors is computed and presented to the user. The `simulate()` function returns control to main, and possibly it can be called again in the same run to perform a different simulation with the same parity check data.

3 Simulator Usage

In the previous section we described how the simulator works internally and the main blocks that compose it. This section describes the simulator from an external point of view. It aims at helping the user to start working on it, pinpointing on how to configure different settings and choosing between the available options. The interactive command line interface of the simulator will be described, and all the phases of the configuration will be covered step by step. The user can use this tutorial as a starting point in order to exploit the tool without getting stuck into any problems.

3.1 Requirements and Compilation

Before discussing how to use the simulator, it is worth mentioning how to compile and build it from the source files. The first step is to obtain the code, which is stored as a public git repository on GitHub:

```
git clone https://github.com/djah/LDPC_simulator.git
```

The tool was written in `C++`, and compiled with `g++`, which is part of the GNU Compiler Collection (GCC) suite. The source code is based on standard libraries, so compilation should complete successfully on any standard Linux distribution. A basic Makefile is provided together with the code, therefore a user just needs to type `make` command on a terminal to produce a working executable. In a similar way, the simulator can be removed from the file system by typing `make clean`.

So far, the program was tested on two x86 machines installing Linux Mint with Kernel version 3.5, therefore even tough it is very likely that the tool will work under the majority of systems, including Windows and OSX, it is guaranteed to work properly in this environment only.

3.2 Preparing Test Vectors

Before launching the simulator executable, the user should prepare the input vectors which will be tested by the tool during the simulation phase. The tool indeed, expect to find the test stimuli in a text file, written in hexadecimal format, one for each line.

For clarity's sake, an example of a simulation input file is depicted. In this example, three test vectors are introduced into the simulator, each of them composed by 8 hexadecimal digits (32 bits):

```
AAAAEEEE
2834FC00
0000FFFF
```

For reasons described in the Section 2, this file format only works for messages of less than 64 bits, but the simulator code contains support for bigger words too. Clearly, each line of the input file should not contain a number that cannot be expressed with the message length (k) selected when the simulator is started. If this happens, the number from the file will be truncated to the k least significant binary digits.

After having prepared the messages that the simulator will analyze, the user will simply specify the absolute or relative path of the file in the correct menu to instruct the simulator about the location of the inputs.

3.3 Generation/Load Phase

As soon as the simulator starts, a simple command line interface will be displayed on screen; the user can choose between the different options just by typing the number corresponding to the action he wants to perform followed by the `enter` key. We will report the main menus of this interface to help the reader follow this tutorial.

On the first screen after execution begins, the user is requested to choose between *Generating a new LDPC code* or *Loading H matrix from file*:

```
LDPC Simulator. De Gennaro/Jahier Pagliari, RESD Project, 2014
Select operation:
1) Generate new LDPC code
2) Load H matrix from file
3) Exit
>
```

Generate a New LDPC Code

This option allows the user to create a brand new LDPC code. Indeed, after having selected such option, the user is prompted to input the number

of bits of a message that will be processed by the code (k) and the number of bits of the corresponding codeword (n). Clearly, as explicitly reminded by the simulator, the length of the codeword must be higher than the length of the message: $n > k$, because of the the check bits needed by any linear code to detect and/or correct possible errors. An error message will be returned if the user tries to input parameters such that $n \leq k$. Since the simulator currently implements a random generation strategy based on constraining H to have the same number of $1s$ in each column (see Section 2.8) the number of ones per column will be asked to the user in the next prompt. This number must be $|ones| \geq 3$ as specified in [3].

```
Select operation:
1) Generate new LDPC code
2) Load H matrix from file
3) Exit
> 1
Insert message length (k): 32
Insert code length (n) (must be > k): 50
Generating new code with the even ones method.
Select number of ones: 3
```

Load H Matrix From File

This option instead, is meant for users who want to introduce a previously generated parity-check matrix into the simulator using a text file representation. After selecting this option, the path of the file containing matrix H must be inserted. The matrix could be generated using the simulator itself, using an external tool or even manually as long as the format is respected. In particular, the very simple file representation contains:

- Word length (k) in the 1st line.
- Codeword length (n) in the 2nd line.
- From the 3rd line on, each line corresponds to a row of matrix H, with each element separated by a space.

An example, for a $(9, 6)$ code (numbers are kept small for easier formatting) is reported below:

```
6
9
0 0 1 0 0 0 1 0 1
1 1 0 0 0 1 0 0 0
0 0 0 1 1 0 0 1 0
```

Because of linear codes theory, the number of rows in H is supposed to be $(n - k)$, and each of them should be composed of n elements. If the matrix has been generated previously using the same tool, this is guaranteed to be true. Otherwise, or if the file format is wrong, an error is produced.

```
Select operation:
1) Generate new LDPC code
2) Load H matrix from file
3) Exit
> 2
Insert filename: h_32_50.txt
Data loaded correctly.
```

Print H to Screen

When the parity check has been loaded in the simulator or created, the user will be asked if he or she wants it printed out on screen. Here is an example:

```
Print the parity check matrix for the code?
1) yes
2) no
> 1
k = 8, n = 12
Parity check matrix:
1 1 1 0 1 1 1 0 1 1 1 1
1 0 1 1 1 1 1 1 1 1 0 1
0 1 1 1 0 0 1 1 0 1 1 1
1 1 0 1 1 1 0 1 1 0 1 0
```

If the selection is 2 (no), this step is simply skipped.

3.4 Simulation and Save Phase

The next menu may be considered the *main menu* of the simulator. It allows to start a new simulation, and to save the parity check matrix under test to a file.

```
Select operation:
1) Start simulation
2) Save parity check matrix to file
3) Exit
```

Save H to File

Save parity check matrix to file allows to save the currently used matrix H into a file, using the simple text format explained before. The user must simply provide the path to the file and click enter. Please notice that this option will be proposed again after each simulation, so one might decide to save H later, without losing any data.

```
Select operation:
1) Start simulation
2) Save parity check matrix to file
3) Exit
> 2
Insert filename: my_H.txt
Data saved correctly.
```

Start simulation

After *Start simulation* option has been chosen, the user will be requested to define some parameters needed by the tool to set up the environment for the test. These parameters are based the currently implemented models and algorithms, which are explained in detail in Section 2.

First option to enter is the *transition probability* of the *BSC channel*. As described in Section 2, this is a measure of the number of faults that occur in the memory system, and a higher value will lead to an higher average number of bits which might flip on the word to be transmitted (i.e. read from memory). User can input a value between 0 and 1, where 0 corresponds to a perfect channel where all the bits are transmitted correctly without

any errors, and 1 means having an inverting channel where all the bits are received with the opposite logic values. Maximum level of entropy is reached using $p = 0.5$. The user may want to tune this parameter in order to verify the soundness of the algorithm currently considered for different memory conditions.

Second option to be selected is the algorithm used for the error correction. Currently, two algorithms are implemented inside the simulator, *Standard Belief Propagation* and *Logarithmic Belief Propagation*. Both of them were implemented using [1] as a reference, and are detailed in Section 2. Peculiarity of these algorithms is the possibility to choose the maximum number of iterations to perform internally for each considered codeword. Therefore, the next prompt of the interface will ask for this parameter. Clearly, more iterations make the algorithm slower but more accurate. In fact, it will try to correct the codeword as many times as selected by the user, terminating when it succeeds or when the limit is reached.

The last step before simulation actually start is to specify the path for the test vectors file, in order to feed the tool with words to be tested. The format of this file has been described in Section 3.2.

```
Select operation:
1) Start simulation
2) Save parity check matrix to file
3) Exit
> 1
Select BSC equivalent channel transition probability (between 0 and 1): 0.02
Select error correction algorithm:
1) Standard Belief Propagation
2) Logarithmic Belief Propagation
> 1
Select maximum number of iterations of the algorithm: 200
Select filename containing simulation inputs: sim_input.txt
```

During the processing phase, our simulator displays some output for each test vector. An example of the format is the following:

```
==============================================
Simulation vector #6
Input word: 0010001000100010
Encoded word: 01000010001000100010
Memory output: 01010010001000100010
Number of raw errors: 1
The received word is not a codeword
Output of error correction: 01000010001000100010
Number of errors: 0
The corrected word is a codeword
Decoded word: 0010001000100010
==============================================
```

First line shows the cardinal number of the vector currently under simulation, starting with 0 at the beginning of each run. *Input word* and *Encoded word* represent the word and the codeword before and after the encoding phase respectively. The sequence of bits labeled *Memory output* corresponds to the word read from the memory, possibly modified by one or more errors; the following line indeed shows the number of bits which flipped, i.e. the number of raw errors for the vector. It is very likely that if at least an error is present, the message received will no longer be a codeword for the code. The simulator checks this property and outputs the information on screen as well.

Then, the correction algorithm previously selected by the user is run, and the result is displayed in the line *Output of error correction*. The number of errors after correction follows, together with the result of checking if the corrected string of bits is a codeword or not (i.e. if the algorithm has terminated because it reached the maximum number of iterations or not). Finally the *Decoded word* is printed too.

Simulation Statistics

After all test vectors have been processed by the tool, some global *Simulation statistics* are displayed. Here is an example:

```
Simulation statistics:
Number of simulation vectors: 15
Channel error probability: 0.05
Number of raw errors: 41
Average raw errors per vector: 2.73333
Number of errors after correction: 30
Average errors per vector after correction: 2
```

Several information are displayed, including the total number of vectors that were simulated, and the total and average number of errors before and after correction. User may exploit these data for evaluating the algorithm in different environment conditions, e.g. with different channel error probability.

Finally, the main menu will be displayed again, allowing the user to start a new simulation with different parameters, to save matrix H to a file, or to terminate the execution.

```
Select operation:
1) Start simulation
2) Save parity check matrix to file
3) Exit
> 3
Quitting...
```

References

[1] R. Micheloni, A. Marelli, K. Eshghi *"Inside Solid State Drives (SSDs)"*, Springer.

[2] R. G. Gallager, *"Low-Density Parity-Check Codes"*, IRE Transactions on Information Theory, 1962.

[3] D. J. C. MacKay, *"Good Error-Correcting Codes based on Very Sparse Matrices"*, IEEE Transactions on Information Theory, 1999.