The Documentation of

Polar Codes in MATLAB

A MATLAB package in the field of Error Control Codes

BY

Harish Vangala
harishvictory@gmail.com

Under the supervision of

Prof. Emanuele Viterbo and Dr. Yi Hong,
Software Defined Telecommunications Lab,
Dept. of Electrical and Computer Systems Eng.,
Monash University, Clayton, VIC – 3800,
Australia.

If you find any sort of bugs in code/documentation, please report to: harishvictory@gmail.com

Contents

Preface

This is the full documentation of a MATLAB package dedicated to help simulating the polar codes in various channel models such as a binary symmetric channel (BSC), a binary erasure channel (BEC), and an additive white Gaussian noise channel (AWGN).

The package focuses to provide the most fundamental blocks related to polar codes, to aid researchers start working with polar codes right away. The package is expected to save a significant amount of valuable time required to develop modules related to polar codes such as encoding, decoding, code-construction, and performance analysis such as plotting FER/BER curves.

More importantly, the package provides an implementation of greater efficiency to the users which is tested, benchmarked, and improved over a long period of time. As we find, the only way to see any significant improvement in the speed of the routines in this package is to rewrite them in a lower level programming language such as C/C++, with a similar logic.

Due care has been taken to provide the users a code with detailed comments and help written all over so that they can easily build over the modules. During the writing of this code, a significant importance has been given to be concise and user-friendly. The result is a very small, portable package of fully readable matlab files that you can easily download from the below link.

`www.polarcodes.com`

Chapter 1

A Quick Reference

1.1 Installation

The MATLAB package is openly available at below link:

`www.polarcodes.com`

It is the enhanced online resource of our earlier site at: `http://www.ecse.monash.edu.au/staff/eviterbo/polarcodes.html`, hosting the code along side of the other tutorial videos and many other useful resources related to polar codes.

The installation of the package is achieved by probably the simplest way possible: Just copy the few source files (extracted from the zip file) to a your local directory.

More preferrably, adding the location of the extracted files to the matlab's *path* (File → Set Path → Add folder) will allow the user to use the modules from any working directory.

1.2 Quick basics

Here is a matlab package, that you can start using right away to work with polar codes, by simply copying them on to your hard drive (see Section 1.1).

More specifically, the packages allows you the following fundamental operations.

1. Choice of the three popular channels: BEC, BSC, and AWGN
2. Polar encoding
3. Polar decoding
4. Polar code construction
5. Plot the performance curves

Finally, it is made sure such that all the above will only empower the users to upgrade the package and build more advanced simulation environments for their research with polar codes.

For a quick illustration of what this package is capable of doing, we will simply show some pictures, tabular forms, and cheat sheets in this chapter. An impatient user should be able to start right away by simply reading a couple of diagrams in this chapter.

In the next chapters, we will provide a traditional and complete documentation of the package, which will unleash the full features of the package for a more advanced use.

1.3 Illustrations and Tables

This section contains several illustrations to help understand the architecture of the package quickly and start using it immediately. In the next chapter we will provide a detailed and complete documentation.

The Fig. 1.1 describes a mind-map of what constitutes this package. It also describes the functional components of polar codes. The terms 'universal' and 'non-universal' refer to the dependence of code structure on the channel conditions. If the code structure is changed everytime the channel conditions change (by running the code-construction again), then the code is called a non-universal code. If the code is not changed with channel conditions, then it is called a universal code.

The Fig. 1.2 describes a division of the package into two categories: the basic routines and the derived routines. It is a very convenient division to understand the package.

Finally, the Tables 1.1 and 1.2 quickly describe all the main routines and their arguments that a user should be interested in practice. A more detailed and complete documentation of the same along side of many more utilities will be given in the next chapter.

Figure 1.1: The overall conceptual structure of the MATLAB package (parts in gray are unavailable, may be available in future versions)

Table of Functions

(Note: Arguments in *underscored italics* are optional)

Family of Basic Utilities
initPC(N,K,channelname,channelstate,*silentflag*,*frozenbits*)
x = pencode(u,*myfrozenlookup*)
u = pdecode(y,channelname,channelstate,*myfrozenlookup*)
x = systematic_pencode(u,*myfrozenlookup*,*algorithmname*)
u = systematic_pdecode(y,channelname,channelstate,*myfrozenlookup*)
y = OutputOfChannel(x,channelname,channelstate)
myfrozenlookup = build_a_lookup(frozenbits,frozenbitindices)

Family of Plotting Utilitites
plotPC(N,K,channelname,designstate,channelrange, *MaxIters*)
plotPC_codechanging(N,K,channelname,channelrange, *MaxIters*)
plotPC_systematic(N,K,channelname,designstate,channelrange, *MaxIters*)
plotPC_systematic_codechanging(N,K,channelname,channelrange, *MaxIters*)

Table 1.1: The core utilities of the package

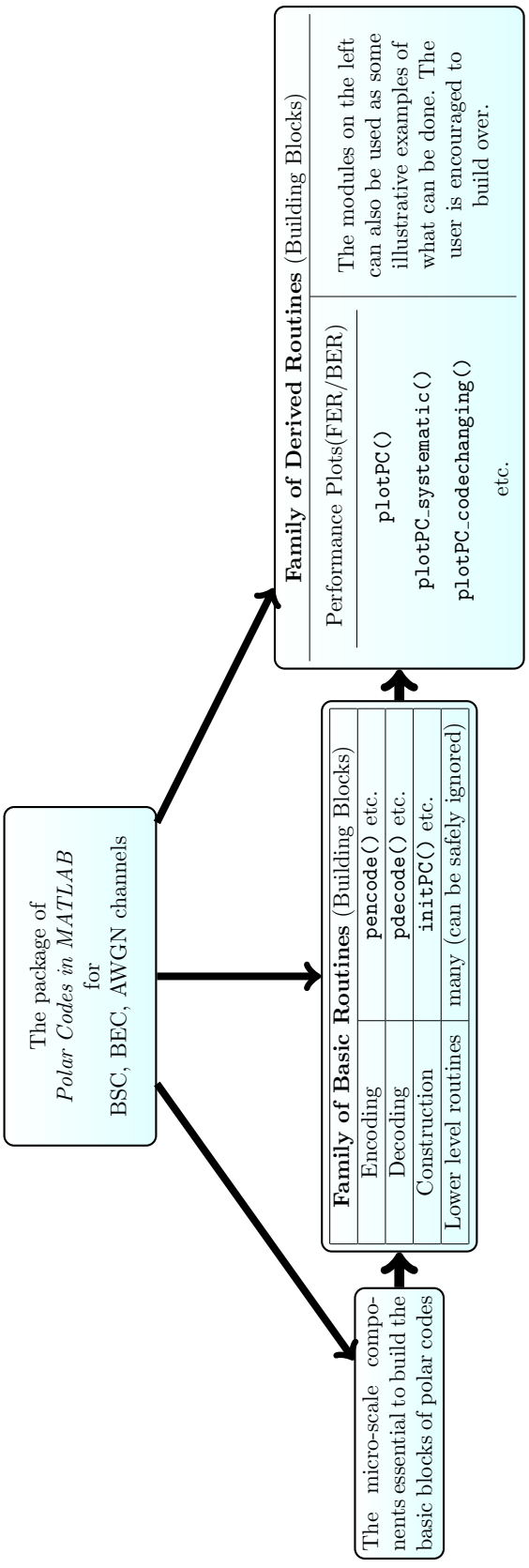

Figure 1.2: Operational division of the package contents

Argument Glossary

N — The blocklength, always a power of 2 (if not, it will be adjusted by `ceil`'ing to the immediate power-of-2)

K — The message length, a positive integer $\leq N$ ($\implies$ the rate of the code $R = K/N$)

`algorithmname` – The name of the systematic-polar-encoding algo: one of 'A' or 'B' or 'C' (in-quotes)

`channelname` – One of the strings (single-quotes) `'BSC'` or `'BEC'` or `'AWGN'` with the respective channel-state parameters p or ϵ or E_b/N_0(in dB) respectively. (`'AWGN'` assumes $\sigma^2 = \frac{N_0}{2}$ and BPSK signalling).

`channelrange` – A vector of channel-state values

`designstate` – The single value of the channel-state to be used during the polar code construction

`frozenbits` – A $(N-K) \times 1$ size binary vector of $N-K$ frozen bits (usually are all zeros by default)

`frozenbitindices` – A $(N-K) \times 1$ size integer vector of $N-K$ unique indices from $\{1, 2, \ldots, N\}$

`MaxIters` – (Always optional) Maximum number of iterations to be used with Monte-Carlo estimation (default: 10000). More its value, more the accuracy and wider the range of parameters it supports

`myfrozenlookup` – (Always optional) A $N \times 1$ size lookup vector, overriding the internal one. It has only the values $-1, 0, 1$, representing whether the corresponding element index is an information-bit (-1) or a frozen-bit (0 or 1, to which it is frozen to). One should typically use `build_a_lookup()`, to build a lookup vector of userdefined values. It is a very critical parameter for polar codes.

`silentflag` – (Always optional) A binary value (default: 0), requesting whether to run silently without printing any o/p

Table 1.2: Quick descriptions of the arguments used in Table 1.1

Chapter 2

The More Detailed Reference

As Fig. 1.2 describes, the contents of our package *Polar Codes in MATLAB* fall into three categories as below. This division follows a top-down approach, in the same respective order.

1. Plotting utilities (derivatives)
2. Encoding/Decoding functions (basics)
3. Micro-scale functions (granulars)

Due to a significant overlap in the arguments of various functions, we would like to save ourselves from a heavy repeatation. We therefore request the readers to refer back to the Table 1.2, in order to understand various arguments that are missing in the descriptions over here.

Also, users are encouraged to use the syntax `help function-name` within MATLAB environment to understand various elements of the package, where `function-name` can be the name of any MATLAB script (`.m` files) that you notice in the package. It should be noted that the descriptions in this document essentially follow the same `help` descriptions, while improving their presentation for a better readability.

2.1 Channel Selection

One of the important features of the package is to give users a flexibility among all the three popular channels of:

1. A *binary symmetric channel* (BSC) with transition probability p
2. A *binary erasure channel* (BEC) with erasure probability ϵ
3. An *additive white Gaussian noise* (AWGN) channel with signal-to-noise-ratio (SNR) in decibels defined as $10\log_{10}\left(\frac{E_b}{N_0}\right)$, which implicitly assumes two components of a BPSK modulation of $\{0,1\} \rightarrow \{\pm\sqrt{R \cdot E_b}\}$ (R is the rate of the code $\triangleq \frac{K}{N}$) and a variance of $\frac{N_0}{2}$ for the additive noise.

When required to simulate such a channel, the user is encouraged to make use of '`OutputOfChannel()`', in order avoid any mis-match.

Typically, such a channel is specified to various functions, simply as a string and a real-value pair: `channelname` and `channelstate`, which take the following special cases:

1. `'BSC'` and p, $0 \leq p \leq 0.5$
2. `'BEC'` and ϵ, $0 \leq \epsilon \leq 1$
3. `'AWGN'` and $SNRdB$, $SNRdB \triangleq 10\log_{10}\left(\frac{E_b}{N_0}\right) \in (-\infty, +\infty)$

We will now proceed to the specific details of all possible functions in the earlier three categories.

2.2 The Plotting Utilities

i.	`plotPC()`
ii.	`plotPC_codechanging()`
iii.	`plotPC_systematic()`
iv.	`plotPC_systematic_codechanging()`

These utilities are the examples of various experiments that a user is typically interested to perform, while working with polar codes. These functions essentially estimate the *frame error rate* (FER) and the *bit error rate* (BER) of polar codes in various parametric cases.

The common functionality of all the below functions is to produce the following results, while preserving the current settings of the polar coding set by an earlier run of `initPC()`.

1. Live-in status of the running simulation (which is typically longer than a few seconds) printed in a user-friendly output format, (in steps of 100 samples)

2. At the end, it outputs a figure window, containing two plots:
   - FER vs. Channel-state
   - BER vs. Channel-state

3. Creates two global variables `FER` and `BER` in the current workspace, containing the main results.

The core technique of all the functions below is to simply simulate a large number ($\leq$ `MaxIters`) of sample encoding-decoding operations under individual noisy channel conditions, in order to Monte-Carlo estimate the FER and the BER.

Supplying an explicit integer value of `MaxIters` is optional, whenever the default value of `MaxIters`= 10000 is sufficient. But, an explicit larger value must be supplied whenever the simulations produce a zero FER/BER at an essential channel-state value.

i. `plotPC`(N,K,`channelname,designstate,channelrange,` *MaxIters*) :

Obtains the performance curves **FER/BER** vs. **channelstate** of a (N,K) polar code when **channelstate** takes values in `channelrange`.

The design/construction of the polar code is performed under `channelname` channel at a channel-state value `designstate`, with blocklength of N bits, encoding K message bits at once.

e.g. `plotPC(1024,512,'BEC',0.1,0.05:0.05:0.3,20000)`

ii. `plotPC_codechanging`(N,K,`channelname,channelrange,`*MaxIters*) :

Obtains the performance curves **FER/BER** vs. **channelstate** in a similar way to `plotPC()`, but the code construction is repeated at every value of **channelstate** in `channelrange`. This way, the polar code changes at every time channel changes and such a code is called a non-universal code. (Note: polar codes are non-universal by definition)

Note the absense of the argument `designstate` as in `plotPC()`, denoting the use of a unique code designed by running the code-construction at `designstate` and keeping the code unchanged for all channel conditions.

e.g. `plotPC_codechanging(1024,512,'BEC',0.05:0.05:0.3,20000)`

iii and iv.
`plotPC_systematic`(N,K,`channelname,designstate,channelrange,` *MaxIters*) **&**
`plotPC_systematic_codechanging`(N,K,`channelname,channelrange,` *MaxIters*) **:**

Exactly same as the earlier two functions except for the use of "systematic polar codes" instead of original polar codes by Arıkan. Such a variant should exhibit the same FER as the original but is known to have a better BER under similar channel conditions.

2.3 Encoding/Decoding/Code-Construction Functions (Basic Modules)

i.	`initPC()`
ii.	`pencode()`
iii.	`pdecode()`
iv.	`systematic_pencode()`
v.	`systematic_pdecode()`

A common feature of all the last four functions is an optional argument of `myfrozenlookup` vector. It is an $N \times 1$ vector conveying a lot of information at once, about the frozen bit indices, and information bit indices within the indices of $\{1, 2, \ldots, N\}$, and also the frozen bits used at the frozen bit indices, as follows:

$$\texttt{myfrozenlookup}(i) = \begin{cases} 0, \text{ if } i^{th} \text{ bit is frozen to 0} \\ 1, \text{ if } i^{th} \text{ bit is frozen to 1} \\ -1, \text{ if } i^{th} \text{ bit is information} \end{cases} \tag{2.1}$$

Such a vector is the output of a polar code construction algorithm, and is already available to the function by a prior run of `initPC()` (is a must before running any function in this section).

By supplying an alternative `myfrozenlookup`, user will be able to bypass the default settings and analyze the consequences. This is critical for example in cryptography, where the frozenbits and their indices can carry a *secret key*, which might be partially-known/unknown to an evesdropper.

The bonus functions at the end of this section will the users help generating the lookup vector `myfrozenlookup`.

i. `initPC(N,K,channelname,designstate,`*silentflag*`,`*frozenbits*`)`

This stores all the information about the polar codes in a global structure named "`PCparams`", that we want to simulate. It runs a full run of the polar-code-construction algorithm and chooses the frozen bits indices and sets those frozen bits to `frozenbits` (optional with default: all zeros).

All this information is critical for the rest of the functions. Further, the function also pre-allocates memory for performing various decoding operations. Therefore, one must have run this function with appropriate initial values, before using any other function in this section.

> **TIP:** One can manually view/change the contents of `PCparams` in order to manipulate the behavior of the encoder and decoder blocks. This is very useful in adanced simulations related to polar codes. For example in cryptography, `frozenbits` and their indices are changed very arbitrarily. In that case, one can run `initPC()`, manipulate the resulting structure `PCparams` (especially the element `FZlookup`), and run the subsequent `pencode()` and `pdecode()` functions (similar to `plotPC()`) to investigate the change in performance. The whole set of changes can then be conveniently written, for example, as a new variant of `plotPC_xxxx()` function.

ii. `x=pencode(u,`*myfrozenlookup*`)`

Encodes the K message bits in `u`, into N bits in `x`. `myfrozenlookup` is an optional argument, used to bypass the default frozenbits/indices setting.

iii. `u=pdecode(y,channelname,channelstate,`*myfrozenlookup*`)`

Decodes original `u` from a received $N \times 1$ (noisy) vector y, assuming the underlying channel `channelname` and its channel-state parameter value `channelstate`. Optionally, it can use a user-defined frozenbit information embedded in `myfrozenlookup`, which is otherwise known to the receiver during the run of `initPC()` itself.

iv. and v.
`x=systematic_pencode(u,`*myfrozenlookup*`)` &
`u=systematic_pdecode(y,channelname,channelstate,`*myfrozenlookup*`)`

Exactly same as the earlier two functions except for the use of "systematic polar codes" instead of original polar codes by Arıkan. Such a variant should exhibit the same FER as the original but is known to have a better BER under similar channel conditions.

A Couple of Bonus Functions:

i.	`OutputOfChannel()`
ii.	`build_a_lookup()`

1. To make sure we have no mis-match in explicitly simulating various channels such as BSC,BEC, and AWGN, we provide the following function:

   ```
   y = OutputOfChannel(x,channelname,channelstate)
   ```

   It simply passes `x` via an instantiation of the noisy channel named `channelname` under its channel-state parameter value equal to `channelstate`, and returns the noisy output of the channel. (involving a few bit-flips of a BSC, or a few erasures of a BEC, or a noise-adding AWGN)

2. When required, one usually suggests the more intuitive `frozenbits` and their locations (`frozenbitindices`). But all our functions take a lookup vector (`myfrozenlookup`) that has this information embedded. Forming such a vector requires a bit of code. The following function bridges this gap, by helping to generate the vector quickly.

   ```
   myfrozenlookup = build_a_lookup(frozenbits,frozenbitindices)
   ```

2.4 Micro-scale Functions

These are the small MATLAB functions that enable building various basic blocks that are discussed in the previous section. There might not be of a great use with these functions, more than knowing their fundamental purpose. So we simply give away the following Table 2.1, containing a short description of each of the functions.

Function Name	Arguments & A Short Description
`j = bitreversed(i)`	Using the N from `initPC()`, it returns decimal equivalent `j` of the reversed the binary $n = log_2(N)$ bit representation of the index `i`, by looking-up a table of N integers $\{0, 1, \ldots, N-1\}$
`j = bitreversed_slow(i)`	An exactly similar function as above, but uses no lookup tables (and hence is slower). This is used to build the lookup table during `initPC()`.

`EncoderA()`
`EncoderB()`
`EncoderC()`
	These are the various systematic-polar-encoding algorithms optionally used by `systematic_pencode(u)`. The arguments are slightly involved. Interested readers can read the `help` descriptions of these functions.

`FN_transform(d)`	Returns the $N \times 1$ vector resulted as a product of an $N \times 1$ binary vector $\mathbf{d}$ with the n-fold kronecker product $\mathbf{F}^{\otimes n}$ of $\mathbf{F} \triangleq \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \end{smallmatrix}\right)$ in binary field
`c = logdomain_diff(a,b)` `c = logdomain_sum(a,b)`	Adding or subtracting a given two real numbers in log-domain, which is essential for avoiding overflow/underflow problems with likelihood ratios.

`lowerconv(upperbit,`l_1`,`l_2`)`
`lowerconv_BEC0()`
`lowerconv_BEC1()`
	The second likelihood-transformation of Arıkan (see [1, Eq.(76)]) his seminal paper) on two LLRs, which also uses decisions from the earlier decoding steps. In a BEC channel, these operations are optimized with a couple of very small 3×3 lookup-table. Calling the appropriate function is taken care of by `pdecode()` etc functions.

`frozenlookup = pcc(N,K,channelstring,channelstate,frozenbits)` — An independent polar code construction function, returning a lookup-vector holding everything necessary for defining a polar code	
`pdecode_BEC()`	Optimized version of `pdecode_LLRs()` for a BEC. It is implicitly called by `pdecode()` etc.
`pdecode_LLRs()`	The core utility called after computing LLRs of received symbols `y` within `pdecode()` etc., in a BSC or AWGN channel
`updateBITS(latestbit,i)`	Broadcast and update various bits after the knowledge of the latest bit decision `latestbit` at index `i`
`updateLLR(i)`	Compute the `i`-th bit-likelihood by using a computational-tree of different likelihood transformation operations applied on the N received likelihoods
`updateLLR_BEC()`	Optimized equivalent of the above function in a BEC setting
`upperconv(`l_1`,`l_2`)` `upperconv_BEC()`	The first likelihood-transformation of Arıkan (see [1, Eq.(75)]) on two LLRs. In a BEC channel, these operations are optimized with a couple of very small 3×3 lookup-table. Calling the appropriate function is taken care of by `pdecode()` etc functions.

Table 2.1: Table of micro-scale functions constituting the basic modules in Section 2.3

Chapter 3

A Discussion on the Algorithms Used

We have provided a video tutorial at the below link:

`www.polarcodes.com`

The videos explain the basic concepts of polar codes to a beginner in a detailed fashion within an hour. We will refer to them and skip the most of the algorithmic discussion.

This section will provide an additional information related to the algorithms that we used in implementing these different modules.

3.1 The Encoding

The Classic Encoding

The original polar codes are non-systematic, and being a linear code, the encoding simply needs a matrix multiplication. Due to the larger size of the code, matrix multiplication becomes computationally-expensive ($\mathcal{O}(N^2)$). An alternative implementation (similar to FFT's butterfly-circuit implementation) comes to the rescue with a significantly reduced computational-complexity of $\mathcal{O}(N \log N)$. This is available in Arıkan's seminal paper [1].

Our package provides only the efficient implementation, and the other implementation is usually not interesting in any case.

The Systematic Encoding

On the other hand, systematic variants of polar encoding do exist, but their usual formulation takes the form of solving a bunch of linear equations. Fortunately, efficient solutions are available in [2].

In our package, we provide the implementation of all the three algorithms from [2], and when not stated explicitly, our interface considers the least complexity algorithm by default (uses a bit more memory than other two algorithms, but is always manageable).

3.2 The Decoding

Our package provides an implementation of the basic successive cancellation decoder, which we believe to be the *most efficient* implementation possible as of today, and is also distributed freely for all educational and research purposes. It is an all-MATLAB implementation, and as we are aware of, the only way to improve its efficiency is via reimplementing it in a lower-level programming language such as C++, with exactly same architecture. (We noticed more than 100x improvement in its run-time)

Note that, the celebrated decoder for polar codes is successive cancellation decoder (SCD), even though it exhibits a relatively poorer performance compared to state-of-the-art codes. Being fundamental for all the later advanced decoders that exhibit superior performance, one cannot avoid having an SCD.

A quick understanding of how to implement an SCD in MATLAB is possible by reading the sample pseudocode in [3].

Eventhough we started off with the above implementation, we discovered more powerful implementations both in terms of speed and memory-efficiency. The ideas stem from the discussions in [4–6], encouraging us to use a tree-architecture for an SCD instead of a rectangular memory architecture. Eventhough this tree-based implementation ideas look slightly involved and complicated-to-implement compared to Arikan's original exposition in [1], it got significantly simplified and now it became a very natural implementation of an SCD.

Our package implements precisely the most efficient tree-based software architecture of a successive cancellation decoding. All those interested can take time to read the script files with abundant comments to easily understand the underlying concepts.

3.3 The Code Construction

There are a number of polar code construction (PCC) algorithms, one can read [7] to get a glimpse of the major construction algorithms.

One critical conclusion of [7] is that at all practical blocklengths ($N \leq 2^{16}$), the simplest of all constructions is as good as the highly complex PCC algorithm. Taking the lead of this interesting conclusion, we provide only the simplest construction and nothing more. It also avoids an additional ambiguity in the choice of a construction algorithm.

However, one issue that every user should be aware of is the non-universality part of a PCC, where any PCC algorithm assumes a specific channel condition to run the algorithm (say a *design-condition*). To obtain the best code performance for the same code at a range of channel conditions, one should try to optimize over several design-conditions, as illustrated in [7], requiring to simulate FER vs. channel-state curves for each code obtained at each design-condition.

3.4 The Use of Log-domain calculations

The SCD involves calculations using all likelihood ratio (LR) values (or Bhattacharyya parameter or probability of error values in the construction phase). The LRs/Bhattacharyya-params are usually stored directly in double-precision floating-point variables. However, it is well-known to cause an underflow or an overflow, after several likelihood/Bhattacharyya-param transformation operations. In otherwords, the available computer variables are either in-capable or in-efficient of storing the exact LR/Bhattacharyya-param values and hence the simulations fail.

A popular solution to this problem is to store log-likelihood ratios instead of likelihood ratios, i.e., a log-domain representation of all the LRs or the Bhattacharyya params. Throughout the package, we will perform our real-valued calculations in log-domain. This completely avoids the problem of an underflow/overflow.

Throughout the package we prefer to use natural (or base e) logarithms, except during the decibel (dB) calculations of SNR (E_b/N_0).

Bibliography

[1] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, Jul. 2009.

[2] H. Vangala, Y. Hong, and E. Viterbo, "Efficient algorithms for systematic polar encoding," *IEEE Communication Letters*, vol. 20, no. 1, pp. 17–20, Jan 2016.

[3] H. Vangala, E. Viterbo, and Y. Hong, "Permuted successive cancellation decoder for polar codes," in *International Symposium on Information Theory and Applications (ISITA)*, Melbourne, Oct. 2014, pp. 438–442. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6979881

[4] I. Tal and A. Vardy, "List decoding of polar codes," in *IEEE International Symposium on Information Theory (ISIT)*, St. Petersburg, Aug. 2011, pp. 1–5.

[5] ——, "How to construct polar codes," *IEEE Transactions on Information Theory*, vol. 59, no. 10, pp. 6562–6582, Oct. 2013.

[6] C. Zhang, B. Yuan, and K. K. Parhi, "Reduced-latency SC polar decoder architectures," in *International Conference on Communications (ICC)*, Ottawa, ON, Jun. 2012, pp. 3471–3475.

[7] H. Vangala, E. Viterbo, and Y. Hong, "A comparative study of polar code constructions for the AWGN channel," *arXiv:1501.02473 [cs.IT]*, 2015, (submitted). Available: http://arxiv.org/abs/1501.02473.