6.972 PRINCIPLES OF DIGITAL COMMUNICATION II

Fountain Codes

Gauri Joshi, Joong Bum Rhim, John Sun, Da Wang

December 8, 2010

Contents

Fountain Codes

Consider a setting where a large file is disseminated to a wide audience who may want to access it at various times and have transmission links of different quality. Current networks use unicast-based protocols such as the transport control protocol (TCP), which requires a transmitter to continually send the same packet until acknowledged by the receiver. It can easily be seen that this architecture does not scale well when many users access a server concurrently and is extremely inefficient when the information transmitted is always the same. In effect, TCP and other unicast protocols place strong importance on the *ordering* of packets to simplify coding at the expense of increased traffic.

An alternative approach is studied in this chapter, where packets are not ordered and the recovery of some subset of packets will allow for successful decoding. This class of codes, called *fountain codes*, was pioneered by a startup called Digital Fountain and has greatly influenced the design of codes for binary erasure channels (BECs), a well-established model for the Internet.

This chapter aims to study some of the key contributions to fountain codes. In Section 1, we discuss the idealized model for this class of codes. In Section 3, we see how the rebirth of LDPC codes give rise to the first modern fountain-like codes, called *Tornado codes*. This was further refined to *LT codes* in Section 4, the first *rateless* fountain code. Next, we present the most sophisticated fountain codes to date, called *Raptor codes*, in Section 5. In Section 6, some extensions of fountain codes to other communication frameworks are considered. Finally, in Appendix A, we discuss some practical issues in designing fountain codes with illustrations from implementations that were designed for this chapter.

1 Digital Fountain Ideal

The digital fountain was devised as the ideal protocol for transmission of a single file to many users who may have different access times and channel fidelity. The name is drawn from an analogy to water fountains, where many can fill their cups with water at any time. The output packets of digital fountains must be universal like drops of water and hence be useful independent of time or the state of a user's channel.

Consider a file that can be split into k packets or *information symbols* and must be encoded for a BEC. A digital fountain that transmits this file should have the following properties:

1. It can generate an endless supply of encoding packets with constant *encoding cost* per packet in terms of time or arithmetic operations.

2. A user can reconstruct the file using *any* k packets with constant *decoding cost* per packet, meaning the decoding is linear in k.

3. The *space* needed to store any data during encoding and decoding is linear in k.

These properties show digital fountains are as reliable and efficient as TCP systems, but also universal and tolerant, properties desired in networks.

Reed-Solomon codes are the first example of fountain-like codes because a message of k symbols can be recovered from any k encoding symbols. However, RS codes require quadratic decoding time and are limited in block length. More recently, low-density parity-check (LDPC) codes have become popular and also exhibit properties that are desired in the digital fountain. However, the restriction of fixed-degree graphs of the early LDPC codes means that significantly more than k encoding packets are needed to successfully decode.

In this chapter, we explore *fountain codes* that approximate a digital fountain well. These codes exploit the sparse graph structure that make LDPC codes effective but allow the degrees of the nodes to take on a distribution. We show that these codes require n encoding packets close to k, meaning the *overhead* $\epsilon = (n-k)/k$ is low.

2 Preliminaries

2.1 Binary Erasure Channel

The erasure channel is a memoryless channel where symbols are either transmitted perfectly or erased. Hence, the output alphabet is simply the input alphabet and the erasure symbol '?.' For an erasure probability p, the conditional probability of the channel is

$$p(y|x) = \begin{cases} 1-p, & y = x; \\ p & y = ?; \\ 0 & \text{otherwise.} \end{cases}$$

As mentioned, this is a commonly-accepted model for packet transmission on the Internet.

A binary erasure channel (BEC) corresponds to when the input can only take values 0 and 1. In this case, the channel capacity is well-known to be $C = 1 - p$. Although fountain codes can be applied to general erasure channels, the analysis in this chapter almost focuses exclusively to when the input is in $\mathbb{F}_2$ and the channel is a BEC.

2.2 Edge Degree Distributions

Fountain codes are heavily inspired by codes on graphs and iterative decoding. In fact, a critical aspect of both practical implementations and asymptotic analysis is the design of bipartite graphs, much like in LDPC codes. This section provides a primer on degree distributions, a stochastic tool for such designs.

Consider a probability mass function on $\{0, \ldots D\}$, with values $\omega_0, \ldots \omega_D$ such that $\sum_{i=1}^{D} \omega_i = 1$. A random variable y then satisfies $\Pr(y = i) = \Omega_i$. Without loss of information, we can define a *generating polynomial* of this distribution to be $\omega(x) = \sum_{i=0}^{D} \omega_i x^i$. One immediate result of this notation is that the expectation of a random variable generated from this distribution is $\omega'(x)$, the derivative of the generating polynomial with respect to x.

This notation can be used to describe the nodes and edges on a random bipartite graph. We define the *degree* d of a node to be the number of edges connected to it. Conversely, we can define an edge to have degree d if it is connected to a node of degree d. Note that for edges, directionality matters in the specification of the degree.

Consider $\Lambda(x)$ and $P(x)$ corresponding to degree distributions of the left and right nodes respectively, [1] meaning the probability that a node has degree d is Λ_d or P_d. We can describe the same graph using degree distributions on the left and right edges, denoted $\lambda(x)$ and $\rho(x)$ respectively. If there are L left nodes, R right nodes and E edges, then $\lambda_d = Ld\Lambda_d/E$ and $\rho_d = LdP_d/E$.

[1] In Chapter 13.4.4, $\Lambda(x)$ and $P(x)$ are used slightly differently. Although proportional to the distributions here, they are scaled differently and hence do not sum to 1.

Some simple manipulations show that the *average* left edge degree distribution is

$$d_\lambda = \frac{1}{\sum_i \lambda_i / i} = \frac{1}{\int_0^1 \lambda(x)} dx.$$

If we are interested in constructing a bipartite graph with L left nodes and R right nodes, the average left and right degrees must satisfy the constraint $d_\rho = d_\lambda L/R$.

3 Tornado Codes

We begin our survey of fountain codes by studying Tornado codes. These codes, first published in [1, 2], were the first steps towards achieving the digital fountain ideal described in Section 1. Tornado codes are block codes and hence not rateless. However, they do approach Shannon capacity with linear decoding complexity.

We consider a system model in which a single transmitter performs bulk data transfer to a larger number of users on an erasure channel. Our objective is to achieve complete file transfer with the minimum number of encoding symbols and low decoding complexity. For k information symbols, RS codes can achieve this with $k \log k$ encoding and quadratic decoding times. The reason for the longer decoding time is that in RS codes, every redundant symbol depends on all information symbols. By contrast, every redundant symbol depends only on a small number of information symbols in Tornado codes. Thus they achieve linear encoding and decoding complexity, with the cost that the user requires slightly more than k packets to successfully decode the transmitted symbols. Moreover, Tornado codes can achieve a rate just below the capacity $1-p$ of the BEC. Thus they are capacity-approaching codes.

Tornado codes are closely related to Gallager's LDPC codes [3], where codes are based on sparse bipartite graphs with a fixed degree d_λ for left nodes (information symbols) and d_ρ for right nodes (encoding symbols). Unlike regular LDPC codes, Tornado codes use a cascade of irregular bipartite graphs. The main contribution is the design and analysis of optimal degree distributions for the bipartite graph such that the receiver is able to recover all missing bits by a simple erasure decoding algorithm. The innovation of Tornado code has also inspired work on irregular LDPC codes [4, 5, 6].

3.1 Code Construction

A typical Tornado codes consists of a series of bipartite graphs as shown in Figure 1. The leftmost k nodes represent information bits [2] which are to be transmitted reliably across the erasure channel. The nodes in all consequent stages represent parity check bits, forming sequence of graphs $(B_0, B_1, ..B_m, C)$. Assume that each stage B_i has $k\beta^i$ input bits and $k\beta^{i+1}$ output bits, for all $0 \le i \le m$ and $0 < \beta < 1$. Thus, the number of nodes shrinks by a factor β at every stage except the last.

The graph C in last stage is a conventional erasure code of rate $1-\beta$ which maps $k\beta^{m+1}$ input bits to $k\beta^{m+2}/(1-\beta)$ output parity check bits. An LDPC code is usually used as the conventional code C in the last stage of the graph. The value m is chosen such that β^{m+1} is roughly $\sqrt{k}$. This is to ensure that the combined code will still run in linear time if the conventional code has quadratic

[2] Although, the codes are applicable to any alphabet size q for the symbols, here we assume $q = 2$. Hence the information and encoding symbols are all bits.

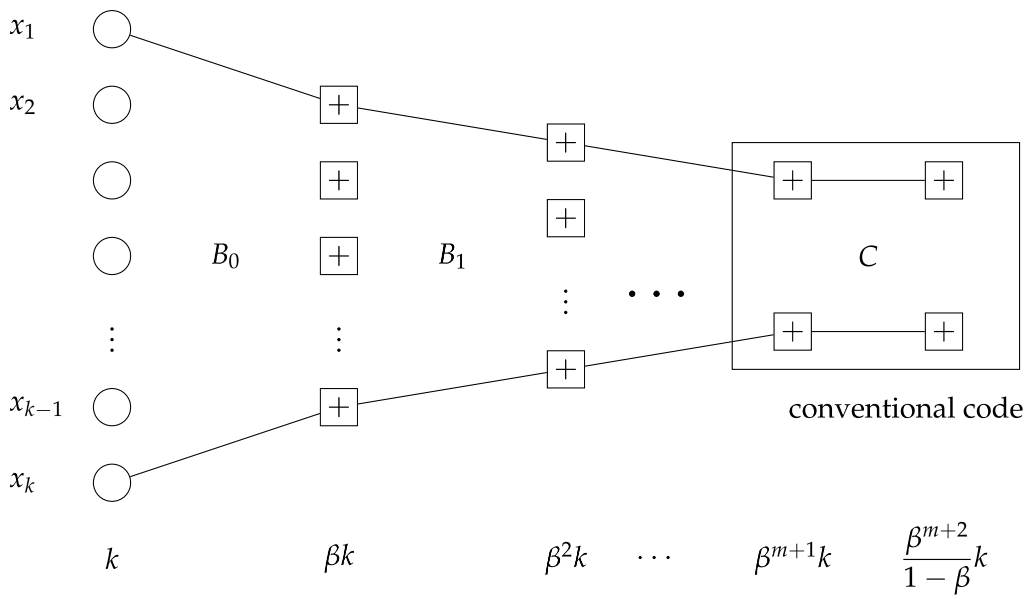


Figure 1: A visualization of a Tornado code. The k information symbols are inputted into a cascade of k sparse bipartite graphs and a conventional code. The output of the Tornado code is the k information symbols and the $n-k$ parity check symbols.

decoding time. Thus, in total the Tornado code has k information bits and $n-k$ parity check bits, where

$$n - k = \sum_{i=1}^{m+1} \beta^i k + \frac{\beta^{m+2} k}{1-\beta} \tag{1}$$

$$= \frac{\beta k - \beta^{m+2} k}{1-\beta} + \frac{\beta^{m+2} k}{1-\beta} \tag{2}$$

$$= \frac{\beta k}{1-\beta}. \tag{3}$$

The overall code has rate $k/n = 1-\beta$, corresponding to an overhead of $\epsilon = \beta/(1-\beta)$.

At every stage, the input and output bits are randomly connected by edges which are chosen from carefully designed degree distributions. This choice of edges allows the following simple decoding algorithm, which is applied to every stage of the graph to recover the information bits at the input to B_0.

Algorithm 3.1 (Decoding)**.** *Given the value of a check bit and all but one of the message bits on which it depends, set the missing message bit to be the XOR of the check bit and its known message bits.*

Figure 2 illustrates the operation of Algorithm 3.1 for a particular level of the code. The first figure shows the graph structure used to generate the check nodes. Suppose symbols a and d are unerased while b, c and e are erased. All the check nodes are unerased. In the second figure we observe that check node $a+b$ satisfies the property that all but one of its message bits are known. The dotted lines represent the connections of the check nodes to the bits that are already known. Thus, we can decode bit b successfully. Similarly bit c can be decoded using check bit $a+c+d$.

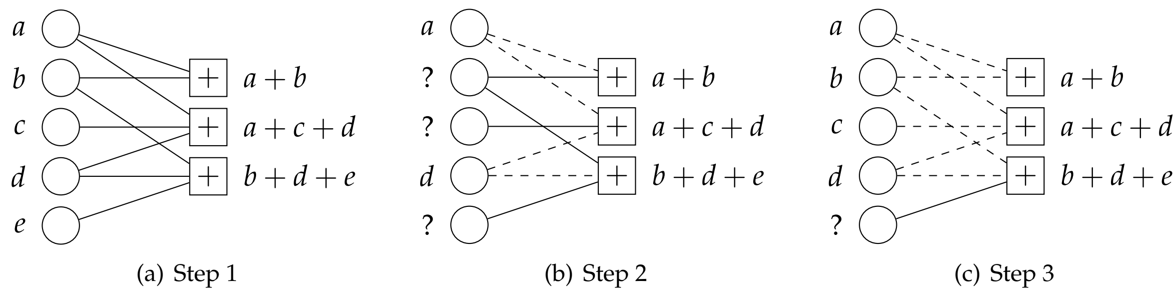


Figure 2: An illustration of Algorithm 3.1.

Since b was decoded in the first step, the check $b + d + e$ can be used to recover bit e, as shown in the third figure.

The decoding for the multi-level Tornado code proceeds as follows. First, the conventional code C is decoded using its standard algorithm to determine the $k\beta^{m+1}$ input bits. Then for all consequent stages of the graph, we apply Algorithm 3.1 in a backward recursive fashion. For stage B_i, we use the decoding algorithm to recover the $k\beta^i$ input bits from the $k\beta^{i+1}$ output bits, which are recovered from stage B_{i+1}.

3.2 Optimal Degree Sequences

By carefully designing a graph for an overhead $\epsilon > 0$, we can construct a code for any rate $1 - \beta$ with encoding time proportional to $n \ln(1/\epsilon)$. The recovery algorithm also runs in time proportional to $n \ln(1/\epsilon)$. Furthermore, a codeword can be recovered with high probability if $\beta(1 - \delta)$ of its entries are erased. In this section we concentrate on one stage of the graph, and design optimal degree distributions so that the decoding algorithm can in fact correct δ fraction of erasures with high probability.

Consider a stage B_i of the graph. The input bits are called left nodes and output bits are called right nodes. Algorithm 3.1 is equivalent to finding a node of degree one on the right, and then removing it, its neighbor, and all edges adjacent to the neighbor from the subgraph. We repeat this until no nodes of degree one are available on the right. The process is successful only if there is a degree one right node available at every step of decoding. The optimal degree distributions are designed in such a way that there are a small number of degree one right nodes available at every time step.

Assume that δ fraction of the left nodes are erased. The series of different graphs formed when removing edges of degree one are modeled as a discrete-time random process, and a system of differential equations is solved for the optimal degree distributions. Our objective is to ensure that the number of right nodes with degree one $r_1(x)$ remains positive throughout the decoding process. For the sake of simplicity, we omit the detailed analysis and present only the main results obtained by solving the set of differential equations.

- The number of right nodes with degree one is found to be $r_1(x) = \delta\lambda(x)[x - 1 + \rho(1 - \delta\lambda(x))]$. For all $\eta > 0$, there exists a k_0 such that for all $k \geq k_0$, the decoding algorithm will terminates with at most ηk message bits erased with probability at least $1 - k^{2/3}\exp(-k^{1/3}/2)$, if $r_1(x) > 0$ for all $x \in (0, 1]$, or equivalently,

$$\rho(1 - \delta\lambda(x)) > 1 - x \qquad \text{for } x \in (0, 1]. \tag{4}$$

- If the degree distribution $\lambda(x)$ is such that $\lambda_1 = \lambda_2 = 0$, then for some $\eta > 0$, the decoding process on the subgraph induced by any η-fraction of the left nodes terminates successfully with probability $1 - O(k^{-3/2})$. This is because the main contribution towards the error comes from the degree three nodes on the left, while degree two nodes lead to a constant error probability.

These results lead to the following important theorem.

Theorem 3.2. *Suppose there exists a cascade of bipartite graphs $(B_1, B_2, ..B_m, C)$ where B_1 has k left nodes. Also suppose each B_i is chosen at random with edge degree distributions specified by $\lambda(x)$ and $\rho(x)$, such that $\lambda(x)$ has $\lambda_1 = \lambda_2 = 0$, and suppose that δ is such that*

$$\rho(1 - \delta\lambda(x)) > 1 - x \qquad \text{for all } \ x \in (0, 1]. \tag{5}$$

Then, if at most a fraction δ of encoding symbols are erased independently at random, the recovery algorithm terminates successfully with probability $1 - O(k^{-3/4})$ in $O(k)$ steps.

So far, we have obtained some constraints on the degree distributions so that the decoding algorithm terminates successfully. Now, we determine the distributions so that the code can achieve the Shannon capacity $(1 - p)$ of a BEC with erasure probability p. This is achieved by bringing δ, as close as possible to β, where $(1 - \beta)$ is the rate of Tornado code. For a positive integer D, the harmonic sum $H(D)$ defined as $H(D) = \sum_{i=1}^{D} 1/i \approx ln(D)$. Then, the optimal left degree sequence is the *heavy tail distribution*

$$\lambda_i = \frac{1}{H(D)(i-1)} \qquad \text{for } \ i = 2, 3, .., D + 1. \tag{6}$$

The average left degree equals $d_\lambda = H(D)(D + 1)/D$. The optimal right degree sequence is the *Poisson distribution*,

$$\rho_i = \frac{e^{-\alpha}\alpha^{i-1}}{(i-1)!}, \tag{7}$$

where α is chosen to guarantee that the mean of the Poisson distribution $d_\rho = \alpha e^\alpha / (e^\alpha - 1)$ satisfies the constraint $d_\rho = d_\lambda/\beta$. Note that when $\delta = \beta(1 - 1/D)$, these distributions satisfy the condition in (5). Example plots of $\lambda(x)$, the truncated heavy tail distribution for various values of D are shown in Figure 3.

Note that the heavy tail distribution does not satisfy the property that $\lambda_2 = 0$. To overcome this problem, a small change to the graph structure has been suggested. The βk right nodes are divided into two sets of γk and $(\beta - \gamma)k$ nodes each, where $\gamma = \beta/D^2$. The first $(\beta - \gamma)k$ right nodes and k left nodes constitute of the first subgraph P, and the remaining nodes form the second subgraph Q. The heavy tail/Poisson distributions are used to generate edges of subgraph P, and on Q, the k left nodes have degree 3 and $3k$ edges connected randomly to the γk right nodes. The design strategy for degree distributions described above leads to the following main theorem in [2].

Theorem 3.3. *For any R with $0 < R < 1$, any ϵ with $0 < \epsilon < 1$, and sufficiently large block length n, there is a linear code and a decoding algorithm that, with probability $1 - O(n^{-3/4})$, is able to correct a random $(1 - R)(1 - \epsilon)$-fraction of erasures in time proportional to $n \ln(1/\epsilon)$.*

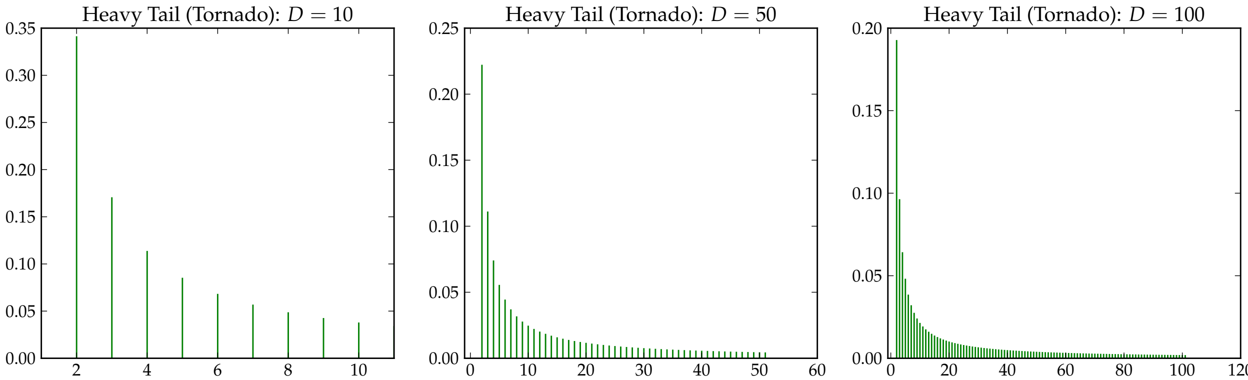


Figure 3: Heavy tail distribution under various D values.

3.3 Practical Considerations

The analysis in Section 3.2 assumes that equal fraction of message bits are lost in every level of the graph. However, in practice this is not true, and a larger number of encoding bits are required to completely recover the message. A solution to this problem is to use as few cascade levels as possible, and to use a randomly chosen graph instead of a standard erasure-correcting code for the last level. A practical Tornado code that uses this idea is the Tornado-Z code [7]. It shows a significant performance improvement in encoding and decoding times as compared to RS codes. Practical Tornado codes are also used in [8] and led to the innovation of irregular LDPC codes.

Since Tornado codes require only linear time for encoding and decoding, they offer a better approximation to digital fountains than Reed-Solomon codes for many applications. However, they still suffer from a powerful drawback in that the code is not rateless. One must determine the rate of the code ahead of time based on the erasure probability of the channel. Hence, Tornado codes do not satisfy the digital fountain ideal of being tolerant to a heterogeneous population of receivers with a variety of packet loss rates. While theoretically one could use a code with a very large number of encoding packets, this is not viable in practice because the running time and memory required are proportional to the number of encoding packets. Tornado codes became obsolete after the development of the Luby Transform (LT) codes [9], to be described in Section 4.

3.3.1 Matlab Implementation

For this report, we develop a Matlab implementation of the Tornado-Z code. For a detailed description of the code, discussion on the Tornado code implementation, and simulation results, see Appendix A.1.

4 LT Codes

LT codes are the first practical rateless codes for the binary erasure channel [9]. The encoder can generate as many encoding symbols as needed from k information symbols. The encoding and decoding algorithms of LT codes are simple; they are similar to parity-check processes. LT codes are efficient in the sense that a transmitter does not require an ACK from the receiver. This property is especially desired in multicast channels because it will significantly decrease the overhead incurred by controlling ACKs from multiple receivers.

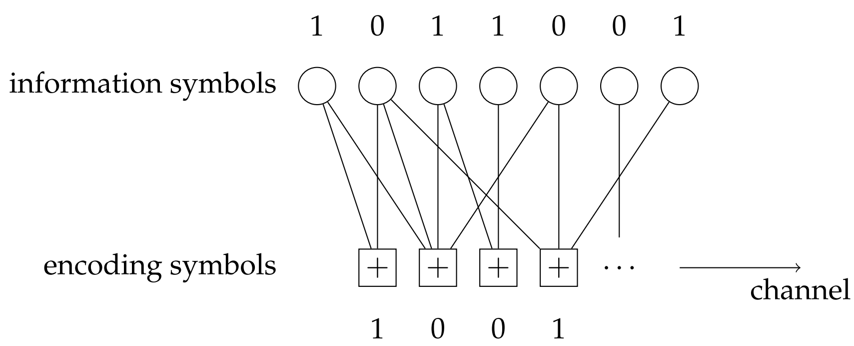


Figure 4: Generation of encoding symbols

The analysis of LT codes is based on the analysis of LT processes, which is the decoding algorithm. The importance of having a good degree distribution of encoding symbols arises from this analysis. As a result, the Robust Soliton distribution is introduced as the degree distribution.

LT codes are known to be efficient; k information symbols can be recovered from any $k + O(\sqrt{k}\ln^2(k/\delta))$ encoding symbols with probability $1-\delta$ using $O(k \cdot \ln(k/\delta))$ operations. However, their bit error rates cannot be decreased below some lower bound, meaning they suffer an error floor as discussed in Section 4.3.

4.1 Code Construction

4.1.1 Encoding Process

Any number of encoding symbols can be independently generated from k information symbols by the following encoding process:

1) Determine the degree d of an encoding symbol. The degree is chosen at random from a given node degree distribution $P(x)$.

2) Choose d distinct information symbols uniformly at random. They will be neighbors of the encoding symbol.

3) Assign the XOR of the chosen d information symbols to the encoding symbol.

This process is similar to generating parity bits except that information symbols are not transferred.

The degree distribution $P(x)$ comes from the sense that we can draw a bipartite graph, such as in Figure 4, which consists of information symbols as variable nodes and encoding symbols as factor nodes. The degree distribution determines the performance of LT codes, such as the number of encoding symbols and probability of successful decoding. The degree distribution is analyzed in Section 4.2.

4.1.2 Decoding Process

Encoding symbols are transferred through a binary erasure channel with the probability of erasure p. The special characteristic of a binary erasure channel is that receivers have either correct data or no data. In other words, no matter what information the decoder receives, there is no confusion in it. Thus, the decoder need not “guess” the original data; it either recovers the true data or gives up.

For decoding of LT codes, a decoder needs to know the neighbors of each encoding symbol. The information about neighbors can be transferred in several ways. For example, a transmitter can send a packet, which consists of an encoding symbol and the list of its neighbors. An alternative method is that the encoder and the decoder share a generator matrix and the decoder finds out the index of each encoder based on the timing of its reception or its relative position.

With the encoding symbols and the indices of their neighbors, the decoder can recover information symbols with the following three-step process, which is called *LT process*:

1) (Release) All encoding symbols of degree one, i.e., those which are connected to one information symbol, are *released* to *cover* their unique neighbor.

2) (Cover) The released encoding symbols *cover* their unique neighbor information symbols. In this step, the covered but not processed input symbols are sent to *ripple*, which is a set of covered unprocessed information symbols gathered through the previous iterations.

3) (Process) One information symbol in the ripple is chosen to be processed: the edges connecting the information symbol to its neighbor encoding symbols are removed and the value of each encoding symbol changes according to the information symbol. The processed information symbol is removed from the ripple.

The decoding process continues by iterating the above three steps. Since only encoding symbols of degree one can trigger each iteration, it is important to guarantee that there always exist encoding symbols of degree one to release during the process for successful recovery. Note that information symbols in the ripple can reduce the degrees of decoding symbols. Information symbols in the ripple keep providing the encoding symbols of degree one after each iteration and, consequently, the decoding process ends when the ripple is empty. The decoding process succeeds if all information symbols are covered by the end. The degree distribution of encoding symbols is analyzed in terms of the expected size of the ripple in Section 4.2. This decoding is very similar to Tornado decoding in Section 3.1 and belief propagation (BP) decoding in LDPC codes.

4.2 Degree Distribution

LT codes do not have a fixed rate and hence the desired property is that the probability of success recovery is as high as possible while the number of encoding symbols required is kept small. Describing the property in terminology of the LT process,

- the release rate of encoding symbols is low in order to keep the size of the ripple small and prevent waste of encoding symbols;
- the release rate of encoding symbols is high enough to keep the ripple from dying out.

Therefore, the degree distribution of encoding symbols needs to be elaborately designed so as to balance between the trade-off. This is the reason that the degree distribution plays an important role in LT codes.

For example, the *All-At-Once distribution* ($P_1 = 1$ and $P_d = 0$ for $d = 2, 3, \ldots$) generates encoding symbols that have one neighbor each. Any received encoding symbol can immediately recover the associated information symbol. Once an encoding symbol is erased, however, the associated information symbol cannot be recovered. To prevent the failure, the transmitter needs to send more encoding symbols than k; this distribution leads to waste of encoding symbols because of high release rate of encoding symbols.

Before analyzing degree distributions for LT codes, we define several concepts.

Definition 4.1 (encoding symbol release)**.** *Let us say that an encoding symbol is released when L information symbols remain unprocessed if it is released by the processing of the $(k-L)$th information symbol and covers one of the L unprocessed information symbols.*

Definition 4.2 (degree release probability)**.** *Let $q(i, L)$ be the probability that an encoding symbol of degree i is released when L information symbols remain unprocessed.*

Definition 4.3 (overall release probability)**.** *Let $r(i, L)$ be the probability that an encoding symbol has degree d and is released when L information symbols remain unprocessed, i.e., $r(i, L) = P_i q(i, L)$. Let $r(L)$ be the overall probability that an encoding symbol is released when L information symbols remain unprocessed, i.e., $r(L) = \sum_i r(i, L)$.*

Proposition 4.1 (degree release probability formula I)**.** $q(1, k) = 1$.

Since some information symbols need to be covered in order to start the ripple at the first iteration of the LT process, $q(1, k) = 1$ is required; the LT process fails otherwise.

Fact 4.2 (degree release probability formula II)**.**

- *For $i = 2, \ldots, k$ and $L = 1, \ldots, k-i+1$,*

$$q(i, L) = \frac{i(i-1) \cdot L \cdot \prod_{j=0}^{i-3}(k-(L+1)-j)}{\prod_{j=0}^{i-1}(k-j)},$$

- *for all other i and L, $q(i, L) = 0$.*

This is the probability that the encoding symbol has i neighbors, among which $i-2$ information symbols are in the first $k-(L+1)$ processed symbols, one information symbol is processed at the $(k-L)$th iteration, and the last information symbol is one of the L unprocessed symbols:

$$q(i, L) = L \cdot \frac{\binom{k-(L+1)}{i-2}}{\binom{k}{i}} = \frac{i(i-1) \cdot L \cdot \prod_{j=0}^{i-3}(k-(L+1)-j)}{\prod_{j=0}^{i-1}(k-j)}.$$

For all i and L other than $i = 2, \ldots, k$ and $L = 1, \ldots, k-i+1$, $q(i, L) = 0$ by the definition of $q(i, L)$.

4.2.1 The Ideal Soliton Distribution

The Ideal Soliton distribution displays ideal behavior in terms of the expected number of encoding symbols needed to recover the data, in contrast to the All-At-Once distribution.

Definition 4.4 (Ideal Soliton distribution)**.** *The Ideal Soliton distribution is given by*

- $\Theta_1 = 1/k$,
- $\Theta_i = 1/i(i-1)$ *for all* $i = 2, \ldots, k$.

Figure 5 shows the Ideal Soliton distribution with various k values.

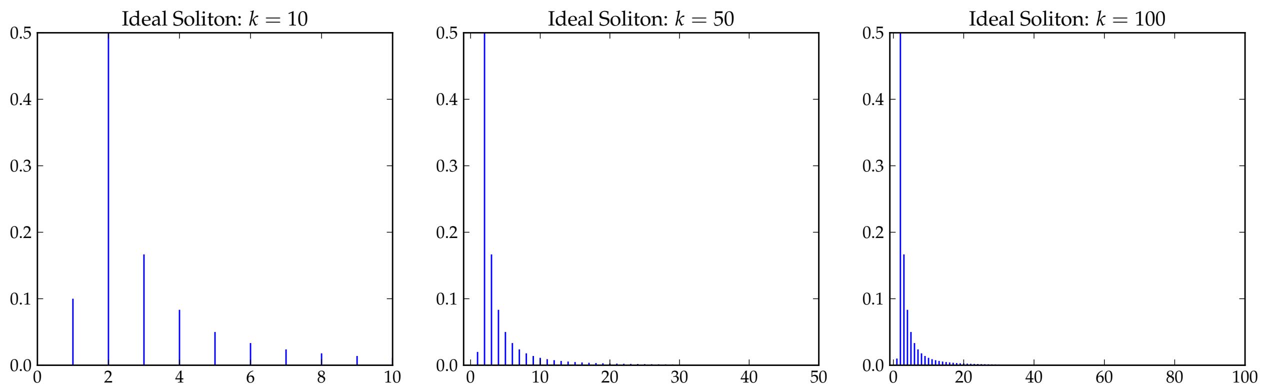


Figure 5: Ideal Soliton distribution for $k = 10, 50, 100$

Theorem 4.3 (uniform release probability)**.** *For the Ideal Soliton distribution,* $r(L) = 1/k$, $\forall L = 1, \ldots, k$.

Since $r(L)$ is the same for all L, the Ideal Soliton distribution results in the uniform release probability, i.e., the encoding symbols are released uniformly at each iteration. In fact, the Ideal Soliton distribution works perfectly in the sense that only k encoding symbols are sufficient to cover the k information symbols and exactly one encoding symbol is expected to be released each time an information symbol is processed. Also, the expected ripple size is always 1; there is neither waste of encoding symbol nor exhaustion of the ripple.

In practice, however, the Ideal Soliton distribution shows poor performance. Even a small variation can make the ripple exhausted in the middle of decoding, which leads to failure. Therefore, we need a distribution that ensures the ripple of large expected size enough to enable stable decoding as well as has the nice property of the Ideal Soliton distribution that maintains the expected ripple size constant in order not to waste encoding symbols.

4.2.2 The Robust Soliton Distribution

Definition 4.5 (Robust Soliton distribution)**.** *Let R denote the expected ripple size and δ denote the allowable failure probability. The Robust Soliton distribution M is given by the two distributions Θ and T:*

$$M_i = (\Theta_i + T_i)/\beta,$$

where Θ is the Ideal Soliton distribution, and T is given by

$$T_i = \begin{cases} \frac{R}{ik} & \text{for } i = 1, \ldots, k/R - 1 \\ \frac{R \ln(R/\delta)}{k} & \text{for } i = k/R \\ 0 & \text{for } i = k/R + 1, \ldots, k \end{cases},$$

and $\beta = \sum_i (\Theta_i + T_i)$ denotes a normalization factor.

The idea in the Robust Soliton distribution is that a distribution T that increases the expected ripple size is added to the Ideal Soliton distribution Θ so that the resulting degree distribution has larger expected ripple size than P while still maintaining approximately uniform release probability. Suppose that the number of encoding symbols is $n = k\beta = k\sum_{i=1}^{k}(\Theta_i + T_i)$. The decoding starts with a reasonable size of ripple, $k(\Theta_1 + T_1) = 1 + R$. In the middle of the decoding, each

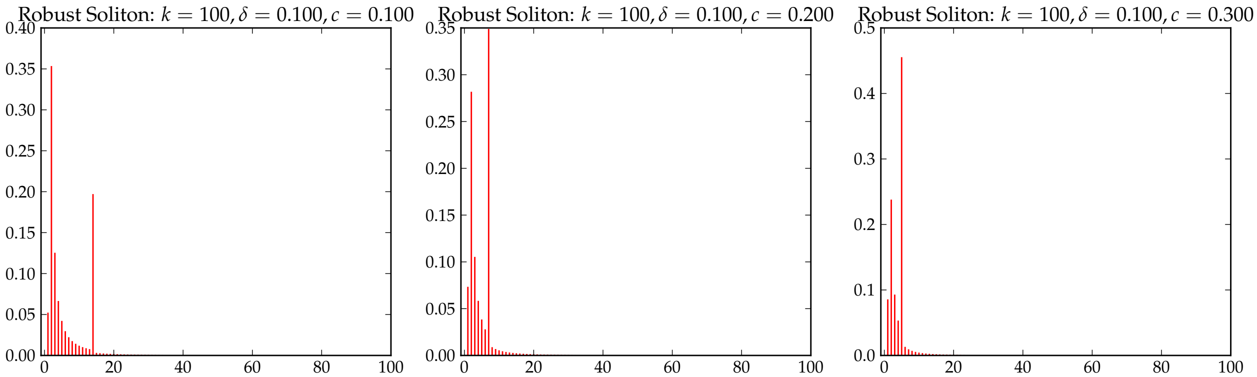


Figure 6: Robust Soliton distributions with $k = 100$, $\delta = 0.1$, and $c = 0.1, 0.2, 0.3$.

iteration processes one information symbol, which means that the ripple needs to increase by one to supplement the processed information symbol. When L information symbols remain unprocessed, it requires $L/(L-R)$ released encoding symbols to add one to the ripple on average. From **??** 4.2, the release rate of encoding symbols of degree i for $i = k/L$ makes up a constant portion of the release rate when L information symbols remain unprocessed. Thus, the density of encoding symbols of degree $i = k/L$ needs to be proportional to

$$\begin{aligned}\frac{1}{i(i-1)}\frac{L}{L-R} &= \frac{k}{i(i-1)(k-iR)} = \frac{1}{i(i-1)} + \frac{R}{(i-1)(k-iR)} \\ &\approx \Theta_i + T_i.\end{aligned}$$

For $i = k/R$, T_i ensures that all unprocessed information symbols are all covered and in the ripple, i.e. $L = R$.

From the intuition that a random walk of length k deviates from its mean by more than $\ln(k/\delta)\sqrt(k)$ with probability at most δ, we can determine the expected ripple size is

$$R = c \cdot \ln(k/\delta)\sqrt{k}$$

for some constant $c > 0$ so that the probability of successful decoding is greater than $1 - \delta$. The following theorems support this intuition when the decoder uses $n = k\beta$ encoding symbols to recover k information symbols.

Theorem 4.4. *The number of encoding symbols is $K = k + O(\sqrt{k} \cdot \ln^2(k/\delta))$.*

Theorem 4.5. *The average degree of an encoding symbol is $O(\ln(k/\delta))$. The decoding requires $O(\ln(k/\delta))$ symbol operations on average per symbol.*

Theorem 4.6. *The decoder fails to recover the data with probability at most δ from a set of any K encoding symbols.*

Figure 6 shows the Robust Soliton distribution with $k = 100$, $\delta = 0.1$ and changing constant c.

4.3 Performance Analysis

The asymptotic performance of LT codes can be analyzed by density evolution. We need to determine degree distributions of information symbols and encoding symbols that is applied to the density evolution. LT codes do not specify the degree distribution of information symbols but it is derived from the encoding algorithm.

4.3.1 Degree Distribution

Suppose that we have k information symbols. Let P_d denote the probability that an encoding symbol is of degree d and ρ_d denote the probability that an edge is connected to an encoding symbol of degree d. The Robust Soliton distribution defines $P_d = M_d$. Then, $\rho_d = \frac{ndP_d}{\sum_d ndP_d} = \frac{dP_d}{d_\rho}$, where d_ρ denotes the average degree of encoding symbols. With P_d and ρ_d, the degree distributions of encoding symbols in node perspective and in edge perspective are given by $P(x) = \sum_{d=0}^{k} P_d x^d$ and $\rho(x) = \sum_{d=1}^{k} \rho_d x^{d-1}$, respectively.

When an encoding symbol is generated, it selects its degree d with probability of P_d and chooses d information symbols as its neighbors. Thus, the probability that an information symbol is chosen by an encoding symbol of degree d is d/k, and the probability that an information symbol is chosen by an encoding symbol is $\sum_d P_d \cdot d/k = d_\rho/k$. While we generate n encoding symbols, the degree of an information symbol, which is the same as the number of times that the information symbol is chosen, follows a binomial distribution; the probability that the information symbol has degree l is $\Lambda_l = \binom{n}{l}\left(\frac{d_\rho}{k}\right)^l\left(1-\frac{d_\rho}{k}\right)^{n-l}$, $0 \geq l \geq n$. If we make $k \to \infty$ to see the asymptotic performance, the binomial distribution can be considered as a Poisson distribution with mean $n \cdot \frac{d_\rho}{k} = d_\rho/R$, where $R = k/n$; the degree distribution of information symbols in node perspective is given by

$$\Lambda(x) = \sum_l \Lambda_l x^l, \text{ where } \Lambda_l = e^{-d_\rho/R}\frac{(d_\rho/R)^l}{l!}.$$

Then we get the degree distribution in edge perspective

$$\lambda(x) = \sum_l \lambda_l x^{l-1}, \text{ where } \lambda_l = e^{-d_\rho/R}\frac{(d_\rho/R)^{l-1}}{(l-1)!}.$$

This discussion shows that the degree distribution of information symbols follows asymptotically a Poisson distribution [10].

4.3.2 Density Evolution

In a binary erasure channel, the density evolution algorithm requires only one value to transfer between input and encoding symbols, which indicates the probability of erasure. A graphical interpretation of LT process is basically the same as the message passing algorithm. It is similar to the decoding process of LDPC codes. Difference between them is due to the location of channel observation nodes: they are connected to right nodes (i.e., factor nodes or encoding symbols) in LT codes while they are connected to left nodes (i.e., variable nodes) in LDPC codes, Figure 7.

In the message passing algorithm, an information symbol is not decodable, i.e. it is '?', only when it receives '?' from every neighbor encoding symbol. On the other hand, an encoding symbol is '?' when either it is erased in the channel or it receives '?' from at least one of its neighbors. To sum up,

$$\begin{aligned} Pr(m^j_{i\to e} =?) &= \sum_d \lambda_d \left(Pr(m^j_{e\to i} =?)\right)^{d-1} = \lambda\left(Pr(m^j_{e\to i} =?)\right), \\ Pr(m^{j+1}_{e\to i} =?) &= \sum_d \rho_d \left[1 - \left(1 - Pr(m^j_{i\to e} =?)\right)^{d-1}(1-\epsilon)\right] = 1 - \rho\left(1 - Pr(m^j_{i\to e} =?)\right)(1-\epsilon), \end{aligned}$$

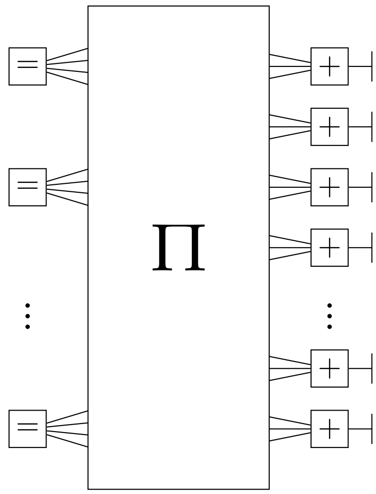


Figure 7: Normal graph of a LT code.

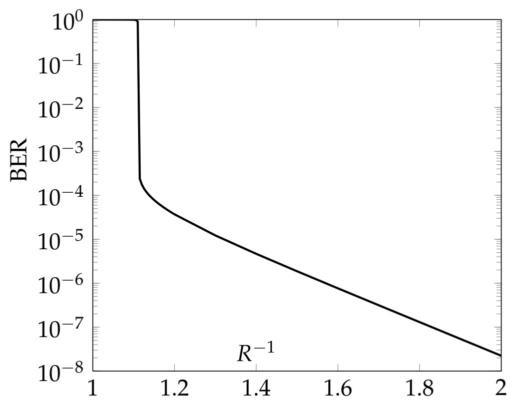


Figure 8: Asymptotic performance of an LT code using the Ideal Soliton distribution in BEC with erasure probability $p = 0.1$.

where $m^j_{i\to e}$ and $m^j_{e\to i}$ denote a message from an information symbol to an encoding symbol and a message from an encoding symbol to an information symbol at jth iteration, respectively. After a sufficient number of iterations, we get $Pr(m^{\infty}_{e\to i} =?) = \gamma$ such that γ is a solution of $\gamma = 1 - \rho(1 - \lambda(\gamma))(1 - p)$. Then the probability that an information symbol cannot be recovered is $\sum_d \Lambda_d \left(Pr(m^{\infty}_{e\to i} =?)\right)^d = \Lambda(\gamma)$, which is known as bit error rate (BER).

Figure 8 shows the BER of an LT code using the Ideal Soliton distribution for infinitely many information symbols. Note that the Robust Soliton distribution converges to the Ideal Soliton distribution as $k \to \infty$. The error floor is clearly shown in Figure 8 for $R^{-1} > 1.1$. From the EXIT chart perspective, we show that there exists a fixed-point at small $(q_{l\to r}, q_{r\to l})$ values in Figure 9, which corresponds to the error floor region.

One intuition about the error floor comes from the degree distribution of information symbols. Since the degree distribution $\Lambda(x)$ is a Poisson distribution, the probability of an information symbol being of degree 0 is $\Lambda_0 = e^{-d_\rho/R}$. Any LT codes cannot have lower bit error rate than that probability. In fact, the error floor is a property of low-density generator-matrix (LDGM) codes.

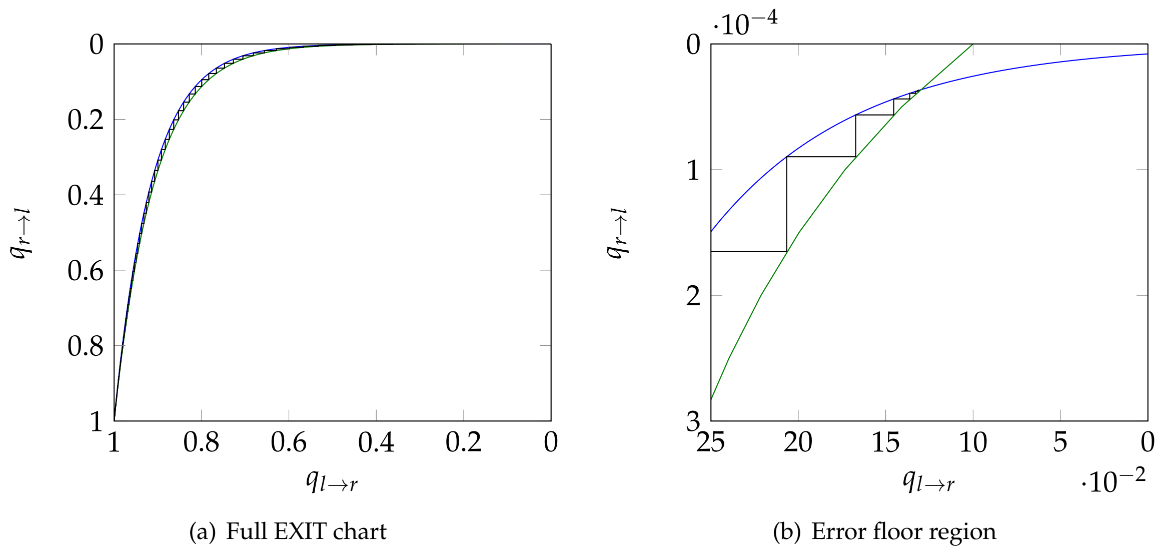


Figure 9: EXIT charts for iterative decoding of a LT code at $n/k = 1.2$ on a BEC with erasure probability $p = 0.1$.

The error floor problem can be solved by applying precoding techniques, such as Raptor codes in Section 5.

4.4 Implementation

We build an LT codes simulator with both MATLAB and Python and run with various parameter values. Apparently, choice of c and δ in the Robust Soliton distribution affects the performance of LT codes. For a more detailed discussion, see Appendix A.2.

5 Raptor Codes

LT codes illuminated the benefits of considering rateless codes in realizing a digital fountain. However, they require the decoding cost to be $O(\ln k)$ in order for every information symbol to be recovered and decoding to be successful. Raptor (rapid Tornado) codes were developed and patented in 2001 [11] as a way to reduce decoding cost to $O(1)$ by preprocessing the LT code with a standard erasure block code (such as a Tornado code). In fact, if designed properly, a Raptor code can achieve constant per-symbol encoding and decoding cost with overhead close to zero and a space proportional to k [12]. This has been shown to be the closest code to the ideal universal digital fountain. A similar vein of work was proposed in [13] under the name online codes. [3]

We have already seen two extreme cases of Raptor codes. When there is no pre-code, then we have the LT code explored in Section 4. On the other hand, we can consider a block erasure code such as the Tornado code in Section 3 as an example of a pre-code only (PCO) Raptor code for the BEC.

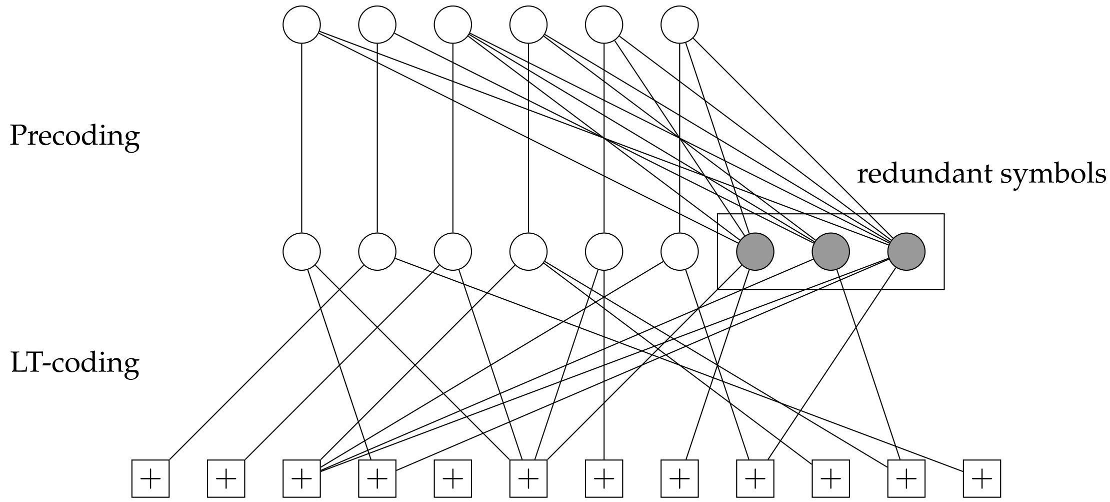


Figure 10: Two-stage structure of a Raptor code with a Tornado pre-code. In general, other block codes such as LDPC or right-accumulate codes can also be used.

5.1 Code Construction

We define a *Raptor code* to be a two-stage process with (m, k) linear block code $\mathcal{C}$ (called *pre-code*) as the outer code and an LT code specified by a node degree distribution $P(x)$ as the inner code. Common effective pre-codes include Tornado, irregular RA, or truncated LT codes. An illustration of a Raptor code is given in Figure 10.

The encoding algorithm is simply the cascade of pre-code and LT encoders. Hence, the Raptor code maps k information symbols into m *intermediate symbols* and then runs an LT encoder to generate a fountain of output symbols. The resulting encoding cost is the sum of the encoding costs of the individual codes.

Similarly, the decoding algorithm is the cascade of LT and pre-code decoding. We will call this decoder reliable for length n if the k information symbols can be recovered from any n encoding symbols with probability at most $1/k^c$, where c is a constant. The decoding cost is again the sum of the individual decoding costs.

Beyond encoding and decoding cost, there are two additional parameters that are used to gauge performance. The space of a Raptor code is the size of the buffer necessary to store the pre-code output. For a pre-code with rate $R = k/m$, the space is $1/R$. The overhead is the fraction of redundancy in the code, and corresponds to $\epsilon = (n-k)/k$ in this case.

We now formulate a Raptor code that will asymptotically have constant encoding and decoding costs, and minimum overhead and space. We assume the pre-code $\mathcal{C}^\star$ has rate $R = (1+\epsilon/2)/(1+\epsilon)$ and is able to decode up to $(1-R)/2$ fraction of erasures. Note that this is significantly less powerful than a capacity-achieving code, which can decode up to $(1-R)$ fraction of erasures. We assume $P(x)$ is close to an ideal Soliton distribution but with some weight for degree one and capped to a maximum degree D. Setting $D = \lceil 4(1+\epsilon)/\epsilon \rceil$ and $\mu = (\epsilon/2) + (\epsilon/2)^2$, the

[3] Online codes was first presented in a technical report in 2002, about a year after the Digital Fountain patent but before the release of Shokrollahi's preprint of [12].

degree distribution is

$$P_D(x) = \frac{1}{\mu+1}\left(\mu x + \sum_{i=2}^{D} \frac{x^i}{(i-1)i} + \frac{x^{D+1}}{D}\right). \tag{8}$$

As will be shown in the next section this code has space consumption $1/R$, overhead ϵ and encoding/decoding cost of $O(\ln(1/\epsilon))$.

5.2 Error Analysis

We first analyze the LT decoder under the relaxed condition of recovering a fraction $(1-\delta)$ of the encoded symbols. This is given in the following lemma:

Lemma 5.1. *Consider an LT code specified by $P_D(x)$ and constants c and ϵ. Then for $\delta = (\epsilon/4)/(1+\epsilon)$, any set of $n = (1+\epsilon/2)m + 1$ encoding symbols are sufficient to recover $(1-\delta)m$ input symbols via BP decoding with probability $1 - e^{-cn}$.*

Proof. The proof utilizes a result from [2] based on analysis of a random graph with input and output edge degree distributions $P(x)$ and $\rho(x)$ respectively. When an iterative decoder is used for a BEC model, then the probability that there are δn errors is upper-bounded by e^{cn} if $P(1-\rho(1-x)) < x$ for $x \in [\delta, 1]$, where δ and c are constants. This is in fact a generalization of Theorem 3.2 for Tornado code analysis.

The remainder of the proof is about determining the edge degree distributions and relating δ to ϵ. We can note immediately that $\rho(x) = P'(x)/a$, where $a = P'(1)$. For an overhead of ϵ, it can be shown that

$$\rho(x) = \left(1 - \frac{a(1-x)}{n}\right)^{(1+\epsilon/2)n}.$$

Further analysis will yield that $\delta = (\epsilon/4)/(1+\epsilon)$ for $P_D(x)$. The details of the proof can be found in [12, Lemma 4]. ☐

We can now combine the LT code with $\mathcal{C}^\star$ to achieve the following theorem.

Theorem 5.2. *Given k information bits and a real-valued constant $\epsilon > 0$, let $D = \lceil 4(1+\epsilon)/\epsilon \rceil$, $R = (1+\epsilon/2)/(1+\epsilon)$ and $m = \lceil k/R \rceil$. Choose the pre-code $\mathcal{C}^\star$ and degree distribution $P_D(x)$ as discussed above. Then the Raptor code with parameters $(k, \mathcal{C}^\star, P_D(x))$ has space consumption $1/R$, overhead ϵ and encoding and decoding costs of $O(\ln(1/\epsilon))$.*

Proof. From any $k(1+\epsilon)$ encoding symbols, we can use the LT decoder to recover $(1-\delta)$ fraction of intermediate symbols with bounded error probability as shown in Lemma 5.1. The pre-code decoder can then recover the k input symbols in linear time.

The encoding and decoding cost of the pre-code can be shown to be proportional to $k\ln(1/\epsilon)$. Meanwhile, the average degree of P_D is $P'_D(1)$, which is proportional to $\ln(1/\epsilon)$. ☐

5.3 Practical Considerations

Although the Raptor codes constructed above are asymptotically optimal, their performance may be quite poor for practical regimes. We now analyze two extensions that make Raptor codes better suited for applications.

5.3.1 Finite-length Raptor Codes

The upper bounds in the analysis above are very loose when the length of the information symbols is short, i.e. in the tens of thousands. For this regime, some care is needed in designing proper degree distributions for the LT code. In [12], the authors introduce a heuristic objective of keeping the expected size of the ripple of the decoding constant. A linear program is then used to find a degree distribution that minimizes this objective. An error analysis for the LT code is found in [14].

In practice, pre-codes such as Tornado or right-regular codes may perform badly for finite-length regimes. LDPC codes have been suggested as an alternative provided that there are no cyclic conditions that can lead to errors when using iterative decoding on the BEC. Moreover, an extended Hamming code as an outer code before the pre-code has been suggested as a way to improve the pre-code.

For very small lengths, i.e. on the order of a few thousand, maximum-likelihood decoding has been proposed as an alternative to BP [15]. Such a decoder has been used in the standard for 3G wireless broadcast and multicast [16].

5.3.2 Systematic Raptor Codes

Another practical consideration is how to make Raptor codes systematic, meaning the information symbols are also encoding symbols. Consider a Raptor code specified by $(k, \mathcal{C}^\star, P_D(x))$ and an overhead parameter ϵ. We wish to determine a set of *systematic positions* $\{i_1, \dots i_k\}$ such that, if the information symbols are $(x_1, \dots, x_k)$, then the fountain output denoted by z satisfies $z_{i_j} = x_j$ for $j \in \{1, \dots, k\}$. Although it would be ideal for $i_j = j$, the analysis in [12] can only guarantee that the k positions are in the first $k(1+\epsilon)$ encoded symbols.

To achieve this, we consider the matrix interpretation of the code. The pre-code $\mathcal{C}^\star$ can be specified generator matrix $G \in \mathbb{R}^{k \times m}$. Any n encoding symbols of the LT code can also be specified using an adjacency matrix $S_n \in \mathbb{R}^{m \times n}$, where the n columns are drawn independently from $P_D(x)$. Hence, the first n elements of the fountain output is represented by

$$z_1^n = xGS.$$

We now present an algorithm for systematic codes. We use Gaussian elimination on GS for $k' = k(1+\epsilon)$ to determine the rows that form a $k \times k$ submatrix of full rank, denoted R, which correspond to the systematic positions. We then preprocess the information symbols by right-multiplying it by R^{-1} and then using the standard Raptor encoder on the result.

It is clear that this code is systematic and the k information symbols are found in the first k' encoding symbols. The only modification to the Raptor decoder is that it must postprocess the BP output by right-multiplying it by R.

More discussion on cost and error analysis is presented in [12]. Additional details can be found in a patent by Digital Fountain [15].

5.3.3 Matlab Implementation

We build a Raptor code in Matlab using the previous implementation of the LT code along with an irregular LDPC code. For a more detailed discussion, see Appendix A.3.

6 Extensions

This section will provide an overview on work that extends fountain codes to more a broader context. For a more detailed summary, see [17].

6.1 Fountain Codes for Noisy Channels

Given the success of fountain codes (especially Raptor codes) on erasure channels, it is natural to investigate the corresponding performance on other types of noisy channels.

In [18], the performance of LT and Raptor codes are compared on both BSC and AWGNC, and Raptor codes are shown to be more successful. In another closely related work [19], the performance of Raptor codes on arbitrary binary-input memoryless symmetric channels (BIMSCs) is analyzed. We summarize this contribution below.

6.1.1 Raptor Codes on BIMSCs

Since we received noisy bits in a general BIMSC, simply terminating reception at slightly more than k bits is no longer sufficient. Instead, we can calculate the reliability of each bit and use it as a measure of the information we received. The receiver will collect bits until the accumulated reliability is $k(1+\epsilon)$, where ϵ is an appropriate constant, called the *reception overhead*. With the reception completed, BP decoding is used to recover the information bits.

The paper investigates to what extend can the results on erasure channel be carried over to BIMSCs. More specifically, it asks the following main question:

> Is it possible to achieve a reception overhead arbitrarily close to zero, while maintain the reliability and efficiency of the decoding algorithm?

In [19], a partial answer for BIMSCs is provided.

First, the authors define *universal Raptor codes* for a given class of channels as Raptor codes that simultaneously approach capacity for any channel in that class when decoded by the BP algorithm. Then they show a negative result:

Result 6.1. *For channels other than BEC, it is not possible to exhibit "universal" Raptor codes for a given class of communication channels.*

However, the above result only applies to Raptor codes and it may still possible for LDPC codes to be "universal." While the Raptor codes are not universal, the performance loss of an mismatched Raptor codes is not too bad, as shown in the following result:

Result 6.2. *Asymptotically, the overhead of universal Raptor codes designed for the BEC is at most* $\log_2(e)$ *if it is used on any BIMSC with BP decoding.*

This result indicates that, while Raptor codes in general do not achieve capacity on BIMSCs, it can achieve a rate that is within a constant gap to capacity.

The analysis of Raptor codes is closely related to the analysis of LDPC codes. For LDPC codes, a technique called *Gaussian approximation*, which is inspired by the central limit theorem, is extensively used [6, 20, 21, 22], where message densities from a variable node to a check node (as LLR) are approximated as a Gaussian (for regular LDPCs) or a mixture of Gaussians (for irregular LDPCs). For Raptor codes, the assumption is refined to be a *semi-Gaussians approximation*, where it is assumed *only* that the messages from input bits are symmetric Gaussian random variables with the same distribution. Under this assumption, a recursive analysis on the mean of the Gaussian random variables can be carried out and the performance of a Raptor codes with a particular degree distribution can be analyzed.

6.2 Weighted Fountain Codes

By selecting the neighbors of encoding symbols according to a non-uniform distribution, the fountain codes can provide more protection to certain message symbols than some others, hence providing *unequal error protection* (UEP) capability [23].

In a UEP problem, we partition the k information symbols into r sets $s_1, s_2, \cdots, s_r$, each with size $\alpha_1 k, \alpha_2 k, \cdots, \alpha_r k$, where $\sum_i \alpha_i = 1$. Via And-Or tree analysis, the asymptotic recovery probability can be analyzed. For the special case of $r = 2$, where the symbols are partition into MIBs (more important bits) and LIBs (less important bits), [23] shows that weighted fountain code can achieve lower BERs for both MIBs and LIBs for small overhead, comparing to the convectional equal-error protection fountain codes. For large overheads, the BER of the MIBs improves significantly while the performance of LIBs only degrades slightly.

A finite-length analysis for finite-length LT, Raptor, and weighted Raptor codes are presented in [23]. They analyze the probability of error under maximum-likelihood decoding and derives various lower and upper bounds for the BERs. The results show that with weighted Raptor codes, we can achieve very low error rates for MIBs with a small increase in error rates for LIBs.

6.3 Windowed Fountain Codes

An alternative way to extend the idea of LT codes is by assigning the set of message symbols into a sequence of (possibly overlapping) subsets, where each subset is usually consecutive and hence called a *window* [24]. Within each window, the information symbols are usually drawn uniformly to produce an encoding symbol.

Windowed fountain codes have a few good properties:

1. the overhead is constant
2. the probability of completing decoding for such an overhead is essentially one
3. the codes are effective for low number of information symbols, i.e. a few tens.

To achieve these desirable properties, windowed fountain codes incur higher decoding complexity of $O(k^{3/2})$. However, at short block lengths, the increase in complexity may be outweighed by its benefits.

The key component for generating the windowed fountain code is the *random windowed binary matrix* A with window size w, which is of size $k \times (k + m)$ and can be constructed as follows:

1. For each column j, choose a starting position i randomly and uniformly from $\{1, 2, \cdots, k\}$, which gives the starting row of this column. Set $A_{i,j} = 1$.
2. For the w elements $A_{i+1,j}, \ldots, A_{i+w,j}$, generate their (binary) values via some random distributions.
3. Repeat this for each column.

$$\begin{array}{l} 0\\ \vdots \\ 0 \\ 1 \ \leftarrow \text{randomly chosen starting position for current column}\\ \left.\begin{array}{l} *\\ *\\ \vdots\\ * \end{array}\right\} w \text{ randomly generated binary symbols}\\ 0\\ \vdots\\ 0 \end{array}$$

Figure 11: Structure of a windowed column

In practice, one can generate the binary random vector within each window as a fixed weight random vector, or a random vector that is drawn i.i.d. from a Bernoulli distribution. The generated column is called a *windowed column*, and its structure is illustrated in Figure 11.

The analysis in [24] is based on random matrix theory (especially the rank property) and it is proven that when window size w is a constant multiple of $\sqrt{k}$, the random windowed binary matrix has rank k with high probability. Therefore, given large enough window size, one can use Gaussian elimination for decoding. Simulation results show that $w = 2\sqrt{k}$ and a fixed column weight of about $2\log k$ is sufficient to achieve good performance.

6.3.1 Windowed Fountain Codes for UEP

This approach could also be used to design UEP fountain codes [25], where the concept of *Expanding Window Fountain* (EWF) codes is introduced. For an EWF code, a predefined sequence of strictly increasing subsets of the information symbols are chosen to be a sequence of windows, as shown in Figure 12. In the end, the input symbols in the smallest window will have the strongest error protection, because they are also contained in all larger windows and are more likely to be used when producing encoding symbols.

The encoding and decoding for EWF codes are similar to the encoding and decoding of LT codes. Hence, it can be viewed as a generalization of LT codes, and the analysis is based on the AND-OR tree technique [4] for fountain codes.

6.4 Distributed Fountain Codes

Fountain codes has been applied to the problem of implementing a robust, distributed network memory using unreliable sensor nodes in [26].

In the distributed sensor network, assume there are k *data-generating* nodes, which generates *independent* data packets [4] over a certain time interval. In addition, assume there are $n > k$ *storage* nodes that act as storage and relay devices, where each storage node has limited memory and can store only one data packet. Given these, one would like to find a strategy that stores the k data

[4] The independence assumption is appropriate if we assume distributed source coding has been used.

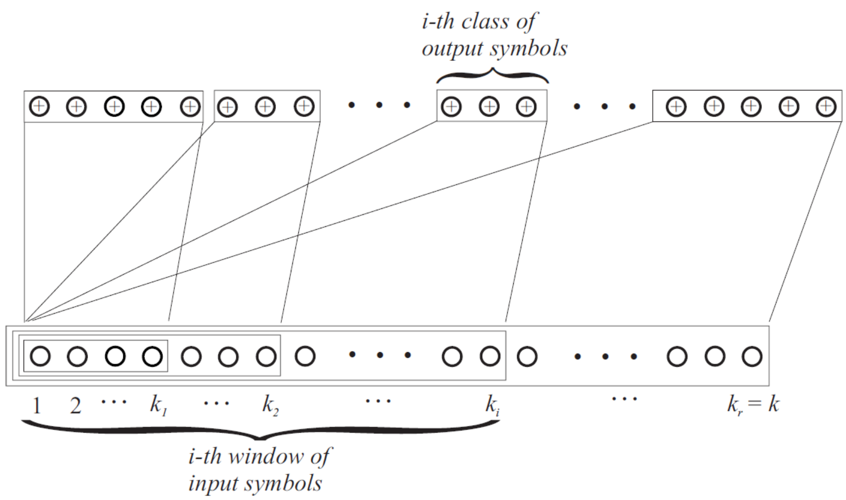


Figure 12: The design of EWF codes (from [25])

packets in the n unreliable storage nodes. More specifically, the paper investigates a strategy for answering the *approximate data collection queries* problem, where a data collector wishes to recover at least $(1-\delta)k$ original data packets after querying *any* $(1+\epsilon)k$ storage nodes.

Assuming linear codes over GF(2) are used, each data node routes its data to d randomly selected storage nodes, each storing the XOR of what it receives. These actions happen before the actual data collection and is called *pre-routing*. Therefore, any linear network strategy can be described as

$$z = xG,$$

where

- z is an $1 \times n$ vector of stored data,
- x is an $1 \times k$ data vector,
- G is a $k \times n$ matrix,

with each of the nonzero entries in G corresponding to the data packets that is sent to a storage node.

In this scenario, standard erasure codes are not directly applicable as the communication cost in sensor network is significant and should be minimized. In addition, the code should allow a distributed construction. These give rise to two desired properties for codes for distributed storage:

1. the code generation matrix should be as sparse as possible to minimize communication, since each nonzero in the generation matrix corresponds to a transmission.
2. the construction should be random and independent, allowing it to be generated in a decentralized fashion.

Dimakis et al. then show that it is not possible to achieve $\delta = 0$ and $\epsilon = 0$ at the same time due to the unreliability of each sensor node. Then applying Lemma 5.1, they show that the LT code used inside the Raptor code can be used for approximate network storage, which allows one to recover $\left(1 - \frac{\epsilon}{4(1+\epsilon)}\right) k$ data nodes by querying $\left(1 + \frac{\epsilon}{2}\right) k + 1$ storage nodes. Since the average degree of the LT code is about $\ln(1/\epsilon)$, the decoding complexity is $O(k \log(1/\epsilon))$, which is linear in k.

The above construction of LT code requires the storage nodes to find data nodes randomly and request for their data packets. The authors then provide an algorithm with only *local* randomized decisions that works on grid topology, and show that the cost of encoding (finding random data nodes) in this case is asymptotically negligible via the trapping set analysis of random walk on finite lattices.

6.5 Fountain Codes for Source Coding

It is known that linear source codes can achieve the entropy rate of memoryless sources [27] and a good channel code is intimately related to a good source code. More precisely, as explained in [28], let s be a length-n source vector and let H be its compression matrix with dimension $n \times k (k < n)$, then the decompresser tries to select the most likely $u = g_H(sH)$ such that $uH = sH$. Analogously, for a linear channel code with parity check matrix H, when receiving y, a receiver tries to select a codeword $y - g'_H(yH)$ (assuming we have an additive channel), where $u' = g'_H(yH)$ the most likely error pattern among that leads to the syndrome yH. Therefore, if a source and the noise of an additive channel has the same statistics, then $g_H(\cdot)$ and $g'_H(\cdot)$ will be identical. In this sense, the source coding problem and the channel coding problem are equivalent.

However, the above results does not directly lead to practical source coding schemes because the scope of this approach is limited to memoryless sources whose statistics are known to the decompresser. To overcome the difficulties, one can use the Burrows-Wheeler (block sorting) transform (BWT) [29]. BWT is an invertible transform that outputs the last column of a square matrix, which is formed by sorting all the cyclic shifts of the original sequences lexicographically. BWT is a very useful tool for universal compression because as the block length grows, the output of BWT is asymptotically piecewise i.i.d. for stationary ergodic tree sources, where the properties of the segments depend on the statistics of the source [30]. Therefore, we can assume the source is piecewise i.i.d. without much loss of generality.

For a piecewise i.i.d. (i.e., memoryless non-stationary) source, it is possible to detect the i.i.d. segments and then use capacity-achieving codes such as LDPC code and a modified BP algorithm for compression [28, 31]. Since fountain codes are also good channel codes, they are also used for source compression [32, 33] and achieve comparable performances. Furthermore, since fountain codes are rateless, they handle heterogeneous sources more naturally and may be more useful in practice.

6.6 Shannon Theory: Fountain Capacity

In the standard Shanon setup, the rate is defined from the perspective of an encoder, which is the *ratio of the information symbols and transmitted symbols* (“pay-per-use”). Under this definition, the rate of a fountain code is simply zero as infinite amount of redundancy are added to the information bits. This motivates the authors of [34] to define a new quantity *fountain capacity*, which is the *ratio of information symbols transmitted to channel symbols received* (“pay-per-view”).

This setup can be motivated by a broadcast channel with multiple receivers, where receivers may have different channel quality. The conventional compound channel approach, though ensures reliability, caters to the worst channel and is wasteful for receivers that have better channels.

In the fountain capacity setup, a receiver will stop "listening" once it has gather enough channel symbols, and receivers with better channels will need to gather less channel symbols.

Following the above motivation, the concept of fountain capacity is more of interest in the multiuser scenario. For example, the results on fountain codes show that for erasure broadcast channels, if the Shannon capacity between the sender and receiver i is C_i, then the fountain capacity region is approximately equal to the Cartesian product $[0, C_1] \times [0, C_2] \times \cdots \times [0, C_n]$, which is in general larger than the conventional broadcast capacity region. However, only the point-to-point case is focused in [34], where they show that the fountain capacity is always upper bounded by the Shannon capacity, and equal to Shannon capacity for stationary memoryless channels. The finding of fountain capacity regions for multiuser systems is left as an open problem.

6.7 Applications

With many good properties, such as capacity-achieving, low encoding/decoding complexity, rateless, and unequal error protection, fountain codes have been applied to a variety of engineering applications, such as hybrid ARQ [35], scalable video streaming [36], and sensor networks.

6.7.1 Usage in Standards

Raptor codes have been adopted in several standards, such as the *3rd Generation Partnership Project* (3GPP), where it is used for use in mobile cellular wireless broadcast and multicast [16], and the DVB-H standards, where it is used for IP datacast to handheld devices. In both cases, systematic Raptor codes are used.

7 Summary

The table below summarizes the characteristics of various codes that are designed for the digital fountain ideal:

	Tornado	**LT**	**Raptor**
Rateless	No	Yes	Yes
Overhead	ϵ	$\epsilon \to 0$	$\epsilon \to 0$
Encoding complexity per symbol	$O(\epsilon \ln(1/\epsilon))$	$O(\ln(k))$	$O(1)$
Decoding complexity per symbol	$O(\epsilon \ln(1/\epsilon))$	$O(\ln(k))$	$O(1)$
Space per symbol	$O(1)$	$O(1)$	$O(1)$, with a larger constant.

Table 1: Summary for fountain codes

A Implementation Details

We now present a more detailed discussion on the MATLAB implementation of Tornado, LT and Raptor codes designed for this report. Our code is available on request.

A.1 Tornado Code

Although Tornado codes were historically interesting for being the first modern fountain codes and the innovation for codes on irregular graphs, they are rather impractical due to the requirement for a cascade of graphs. In fact, they were quickly replaced by irregular LDPC and LT codes, which have much simpler structure and equal or even better performance.

We originally planned on designing a general Tornado code as a pre-code for the Raptor code, but found it extremely cumbersome to design for various choices of n and k. We found that it was much more of an art than science, and quickly abandoned trying to create a very general code. Instead, we focused on a two-stage version in [7][Section V.B] called the Tornado-Z. In the paper, $k = 16,000$ and $n = 32,000$ were given as the system parameters, the left graph was chosen using a truncated Soliton distribution and the right graph was a right-regular LDPC with the exact degree distributions given.

We implemented a simple codebook generator using the specifications and quickly realized the limitations of MATLAB for fountain codes. Fountain codes are known to behave better for very large block lengths, but even for this example memory becomes an issue. More compact representations using MATLAB's built-in functionality led to much longer decoding times and proved to be infeasible as well. This suggests the use of more efficient programming languages to develop better data structures for doing encoding and decoding.

For Tornado and other fountain codes, encoding is extremely simple; it takes less than a second for the Tornado-Z. However, decoding requires a lot of bookkeeping and ends up taking about a minute an iteration, for about 10 iterations at $p = 0.1$. We point out that we are not doing the most efficient decoding possible, using the best data structure for finding nearest neighbors, or finding the optimal ordering of decoding exploiting the graph structure due to time constraints. Addressing these issues is probably an essential factor in why Digital Fountain is able to make practical implementations.

An interesting observation we made is that the Tornado effect is observed. In the initial iterations of the algorithm, the decoder struggles in uncovering information symbols. After a few runs, however, it quickly speeds through the remaining bits and correctly decodes. This is illustrated in Figure 13 for a sample run at $p = 0.1$.

A.2 LT Code

To develop more intuition for LT codes and design of degree distributions, we have implemented LT codes in MATLAB and obtained simulation results. Our implementation encodes k information symbols into n encoding symbols. Since LT codes are rateless, in theory n may be potentially infinite. But for the purpose of implementation we choose a finite value for n. This is a valid assumption because in practice, only slightly more than k symbols are necessary to decode the information bits. The LT code implementation consists of the following three functions:

- **Codebook generation:** This function takes k, the number of information symbols, n the number of output symbols, and δ, the tolerable error probability as inputs. Using these parameters we create the robust Soliton distribution as shown in Definition 4.5. The output is

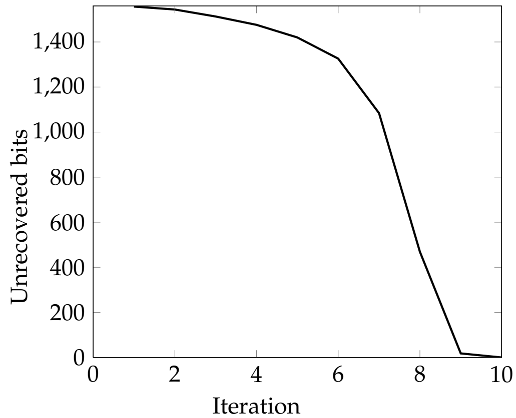


Figure 13: Number of bits still erased versus decoder iteration number for $k = 16,000$, $n = 32,000$ and $p = 0.1$. We observe the Tornado effect in that the algorithm struggles for most of the run and quickly converges to the solution in a very short number of runs.

a $k \times n$ size generator matrix G. The 1's in the i^{th} column of G correspond to the information symbols whose linear combination is transmitted in the i^{th} time slot. All other entries are 0.

- **LT encoder:** This function takes that input data-stream and performs the XOR operation on its symbols according the generator matrix G. The output data-stream is then transmitted on the binary erasure channel.

- **LT decoder:** The input to the decoder is the output of the LT encoder with some symbols randomly erased by the channel. The decoding process is as described in Section 4.1.2. At every iteration, the algorithm finds a check node with degree one and decodes it. All edges connected to this node are removed and the corresponding check node values and degrees are updated. The algorithm stops when all information symbols have been decoded. The decoder declares a failure after a finite number of iterations if the encoding symbols are insufficient to decode the input.

The parameters of an LT code directly affect its performance, such as the overhead of the code. In Figure 14 we plot the change of overhead with respect to both c and δ, and choose two set of (c, δ) values with reasonable expected overhead.

In Figure 15, we plot the number of information symbols decoded versus the number of (unerased) encoded symbols received at the decoder. The parameters chosen are $k = 10000$, $\delta = 0.5$, $c = 0.03$ or $c = 0.3$. It appears that when $c = 0.03$, the relationship between the number of received symbols and number of decoded symbols exhibits a thresholding effect—before the number of received symbols reaches the threshold, most of the symbols are not decoded, and after the threshold is exceeded, all the symbols get decoded shortly. For $c = 0.3$, this effect is less significant. This difference in behavior is due to the degree distributions that different c values result in. A higher value of c implies a larger number of degree one check nodes $R = c \ln(k/\delta)\sqrt{k}$. We observe that for a higher c value, the curve has a large value initially, but the increase in number of information symbols decoded is not as fast as for lower c values. This is because degree one check nodes can be easily decoded without waiting for more encoded symbols. However, higher number of degree one check nodes implies that a smaller number of information symbols are covered in every encoded symbol. Due to this, for higher c, more encoded symbols are required to decoded all the $k = 10000$ information symbols.

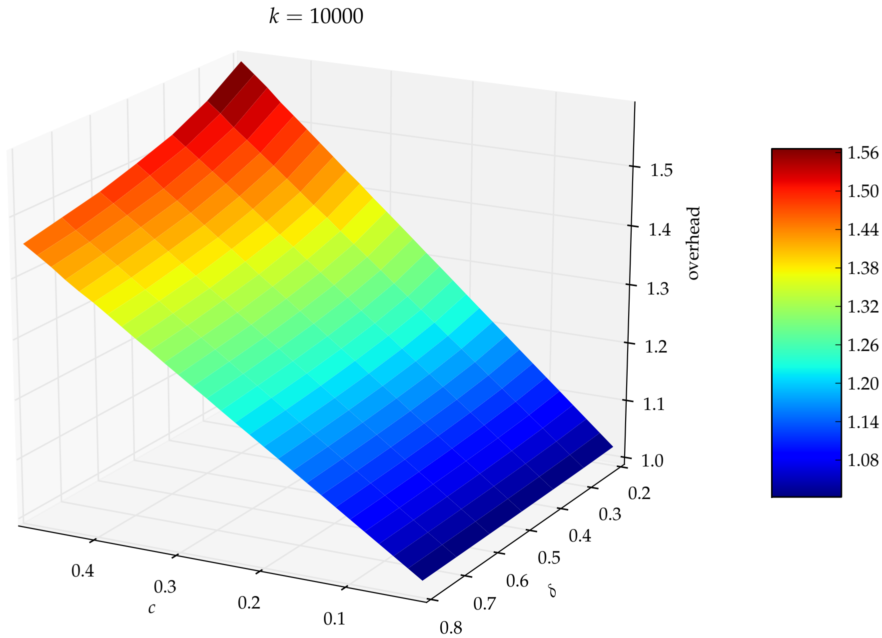
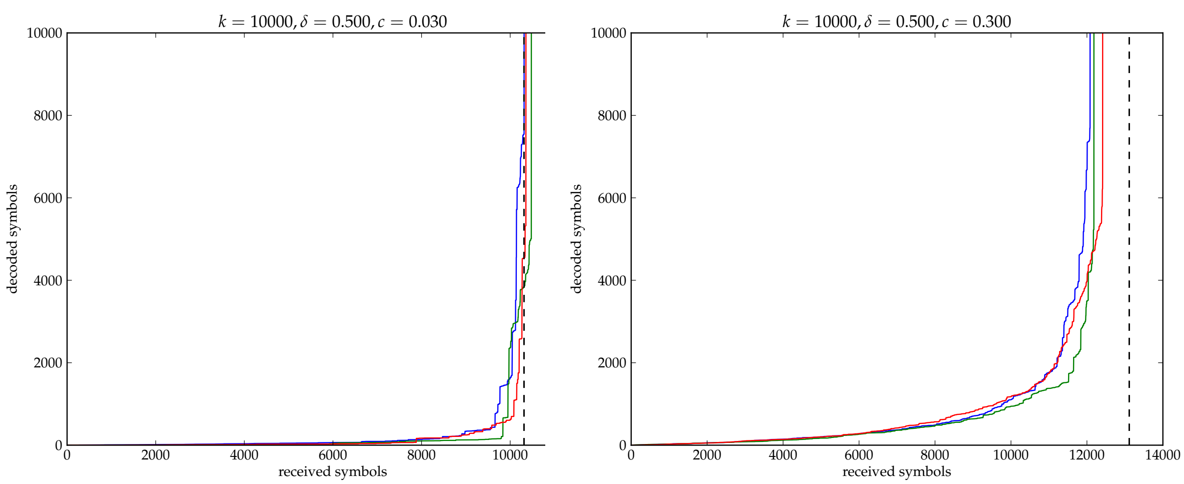


Figure 14: The overhead versus c and δ for an LT code with $k = 10000$.

In Figure 16 we show the number of information symbols decoded as the decoding algorithm iterates when $n = 11000$. It can be seen that the number of decoded symbols start to increase rapidly after several initial iterations. This shows that LT codes also have the "Tornado" characteristic during decoding.



(a) $c = 0.03$

(b) $c = 0.3$

Figure 15: Plot of number of information symbols decoded versus number of encoded symbols processed, illustrating the tornado effect in decoding. The dashed line is the theoretical overhead calculated from the δ and c values. Each solid curve represents a randomly generated LT code under a different random seed.

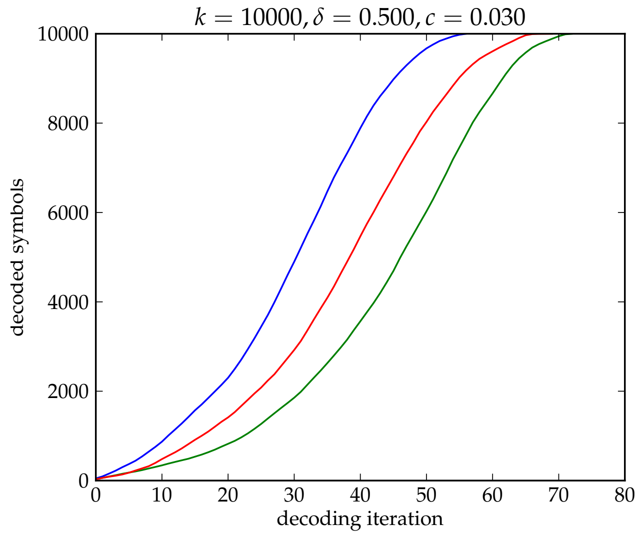


Figure 16: Plot of number of information symbols decoded versus number of iterations in the decoding process. Each curve represents a randomly generated LT code under a different random seed.

A.3 Raptor Code

We have implemented Raptor codes in MATLAB. We have used the same LT codes implementation as in Appendix A.2 and implemented irregular LDPC codes as a precode. The degree distribution of left nodes (variable nodes) in the LDPC codes is fixed as $\lambda(x) = \frac{2}{5}x + \frac{3}{5}x^2$. The degree distribution of right nodes (factor nodes) is varied according to the numbers of left and right nodes: the average degree of right nodes needs to be $d_\rho = d_\lambda n/(n-k) = 2.5n/(n-k)$. The LDPC code implementation consists of the following three functions:

- **Generator-matrix constructor:** This function takes the numbers of information bits and output bits and the degree distributions of left and right nodes. It constructs an $(n-k) \times n$ parity-check matrix according to the given degree distributions. Then the parity-check matrix is transferred to a new parity-check matrix that still satisfies the given degree distributions but has a $k \times n$ regular generator matrix. The outputs are the resulting generator matrix and parity-check matrix.

- **LDPC encoder:** This function takes the input sequence of length k and the $k \times n$ regular generator matrix. The output is the codeword of length n, which is computed by the binary multiplication of the input sequence and the generator matrix.

- **LDPC decoder:** This function takes the received sequence of length n through a binary erasure channel and the $(n-k) \times n$ parity-check matrix. It performs the message-passing algorithm on the bipartite graph defined by the parity-check matrix. The message is the log-likelihood ratio:

$$L(y) = \ln \frac{P_{Y|X}(y|0)}{P_{Y|X}(y|1)} = \begin{cases} \infty, & y = 0 \\ -\infty, & y = 1 \\ 0, & y = ? \end{cases} . \tag{9}$$

  The function stops iteration if no more left nodes are recovered and returns the decoded sequence. It returns the last k bits as the recovered input sequence because the generator matrix is regular, i.e., the rightmost $k \times k$ submatrix of the generator matrix is the identical matrix.

We plot the bit error rate (BER) curve of Raptor codes in Figure 17. The LDPC code with $n = 11000$, $k = 10000$, and $\rho(x) = \frac{27}{55}x^{26} + \frac{28}{55}x^{27}$ is used as precoding and the LT codes with $k = 11000$ and varying $n = k/R$ is used. In Figure 17, we also plot the BER curve of the LT codes for comparison. We observe that the BER of the Raptor codes is lower than that of the LT codes. However, Figure 17 does not show how effective the Raptor codes are in the error floor region. We need to simulate LT codes at least 10000 times to observe the error floor in Figure 8, but we have time to simulate them only several hundreds times. Nevertheless, as shown in Figure 17, we expect that the precoding can recover the symbols that LT codes cannot decode in the error floor region, too.

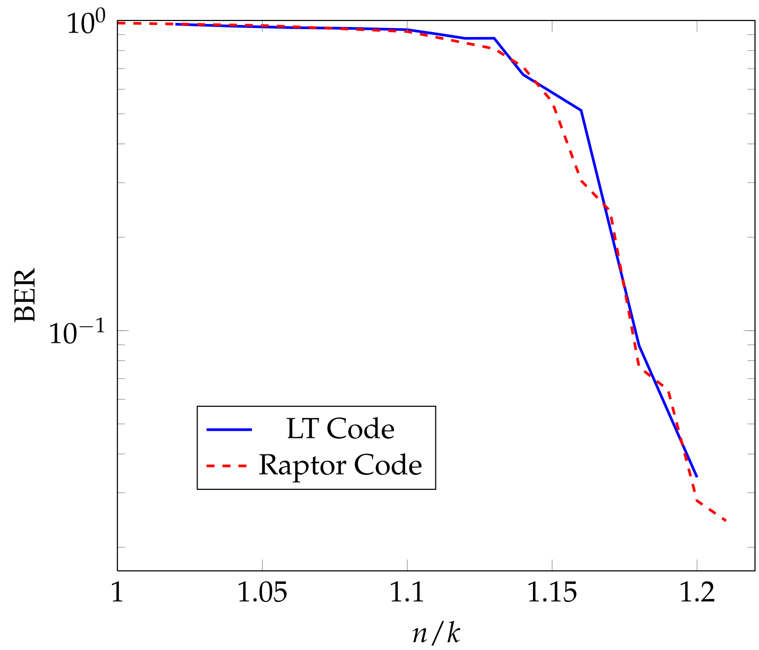


Figure 17: The bit error rate of Raptor codes and LT codes

References

[1] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Prc. ACM Symp. Theory Comp.*, El Paso, TX, 1997, pp. 150–159.

[2] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman, "Efficient erasure correcting codes," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 569–584, 2001.

[3] R. G. Gallager, *Low Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[4] M. Luby, M. Mitzenmacher, and A. Shokrollahi, "Analysis of random processes via and-or tree evaluation," in *Proc. ACM-SIAM Symp. Discrete Algo.*, San Francisco, CA, 1998, pp. 364–373.

[5] M. Luby, M. Mitzenmacher, M. Shokrollahi, and D. Spielman, "Improved low-density parity-check codes using irregular graphs," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 585–598, 2001.

[6] T. J. Richardson, A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 619—637, 2001.

[7] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *Proc. ACM SIGCOMM*, Vancouver, Sep. 1998, pp. 56–67.

[8] J. Byers, M. Luby, and M. Mitzenmacher, "Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads," vol. 1, pp. 275–283, 1999.

[9] M. Luby, "LT codes," in *Proc. IEEE Symp. Found. Comp. Sci.*, Vancouver, Nov. 2002, pp. 271–280.

[10] T. Richardson and R. Urbanke, *Modern Coding Theory*. New York: Cambridge University Press, 2008.

[11] A. Shokrollahi, S. Lassen, and M. Luby, "Multi-stage code generator and decoder for communication systems," U.S. Patent #7,068,729, Issued Jun. 27, 2006.

[12] A. Shokrollahi, "Raptor codes," *IEEE Trans. Inform. Theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[13] P. Maymounkov, "Online codes," NYU, Tech. Rep. TR2002-833, Nov. 2002.

[14] R. Karp, M. Luby, and A. Shokrollahi, "Finite length analysis of lt codes," in *Proc. IEEE Int. Symp. Inform. Theory*, Chicago, IL, Jun. 2004, p. 39.

[15] A. Shokrollahi, S. Lassen, and R. Karp, "Systems and processes for decoding chain reaction codes through inactivation," U.S. Patent #6,856,263, Issued Feb. 15, 2005.

[16] "Technical specification group services and system aspects; multimedia broadcast/multicast services (mbms); protocols and codecs (release 6)," 3rd Generation Partnership Project (3GPP), Tech. Rep. 3GPP TS 26.346 V6.3.0, 2005.

[17] D. Sejdinovi, "Topics in fountain coding," Ph.D. dissertation, University of Bristol, 2009.

[18] R. Palanki and J. Yedidia, "Rateless codes on noisy channels," in *Proc. IEEE Int. Symp. Inform. Th.*, Chicago, IL, Jun. 2004, p. 37.

[19] O. Etesami and A. Shokrollahi, “Raptor codes on binary memoryless symmetric channels,” *IEEE Trans. Inform. Theory*, vol. 52, no. 5, pp. 2033–2051, 2006.

[20] M. Ardakani and F. R. Kschischang, “A more accurate one-dimensional analysis and design of irregular LDPC codes,” *IEEE Trans. Commun.*, vol. 52, no. 12, pp. 2106–2114, 2004.

[21] H. El Gamal and A. R. Hammons, “Analyzing the turbo decoder using the gaussian approximation,” *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 671–686, 2001.

[22] S. Chung, T. Richardson, and R. Urbanke, “Analysis of sum-product decoding of low-density parity-check codes using a gaussian approximation,” *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 657–670, 2001.

[23] N. Rahnavard, B. N. Vellambi, and F. Fekri, “Rateless codes with unequal error protection property,” *IEEE Trans. Inform. Theory*, vol. 53, no. 4, pp. 1521–1532, 2007.

[24] C. Studholme and I. Blake, “Windowed erasure codes,” in *Proc. IEEE Int. Symp. Inform. Th.*, Seattle, WA, Jun. 2006, pp. 509–513.

[25] D. Sejdinovic, D. Vukobratovic, A. Doufexi, V. Senk, and R. Piechocki, “Expanding window fountain codes for unequal error protection,” *IEEE Trans. Commun.*, vol. 57, no. 9, pp. 2510–2516, 2009.

[26] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, “Distributed fountain codes for networked storage,” in *Proc. IEEE Int. Conf. Acoust., Speech, and Signal Process.*, vol. 5, Toulouse, France, May 2006, pp. 2491–2494.

[27] I. Csiszár, “Linear codes for sources and source networks: Error exponents, universal coding,” *IEEE Trans. Inform. Theory*, vol. 28, no. 4, pp. 585–592, 1982.

[28] G. Caire, S. Shamai, and S. Verdú, “A new data compression algorithm for sources with memory based on error correcting codes,” in *Proc. IEEE Inform. Theory Workshop*, Paris, France, Apr. 2003, pp. 291–295.

[29] M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler, “A block-sorting lossless data compression algorithm,” Tech. Rep., 1994.

[30] M. Effros, K. Visweswariah, S. R. Kulkarni, and S. Verdú, “Universal lossless source coding with the Burrows-Wheeler transform,” *IEEE Trans. Inform. Theory*, vol. 48, no. 5, pp. 1061–1081, 2002.

[31] G. Caire, S. Shamai, and S. Verdú, “Noiseless data compression with low-density parity-check codes,” in *Advances in Network Information Theory*, 2004, p. 263.

[32] G. Caire, S. Shamai, A. Shokrollahi, and S. Verdú, “Fountain codes for lossless data compression,” in *Advances in Network Information Theory*, Piscataway, NJ, Dec. 2005, p. 1.

[33] B. N. Ndzana, A. Shokrollahi, and J. Abel, “Burrows-Wheeler text compression with fountain codes,” in *Proc. IEEE Data Compression Conf.*, Snowbird, UT, Mar. 2006, p. 462.

[34] S. Shamai, E. Telatar, and S. Verdú, “Fountain capacity,” *IEEE Trans. Inform. Theory*, vol. 53, no. 11, pp. 4372–4376, 2007.

[35] E. Soijanin, N. Varnica, and P. Whiting, "Punctured vs rateless codes for hybrid ARQ," in *Proc. IEEE Inform. Theory Workshop*, Unta del Este, Uruguay, Mar. 2006, pp. 155–159.

[36] M. C. O. Bogino, P. Cataldi, M. Grangetto, E. Magli, and G. Olmo, "Sliding-window digital fountain codes for streaming of multimedia contents," in *IEEE Int. Symp. Circuits. Syst.*, New Orleans, LA, May 2007, pp. 3467–3470.