Perpetual Codes: Cache-friendly Coding

Petar Maymounkov
petar@csail.mit.edu

January 11, 2006

Abstract

Devising capacity-achieving linear-time encodable and decodable codes for the erasure channel has been an open problem of great practical importance since Elias' seminal paper [6] in 1955. The first resolution of this problem appeared in the form of a somewhat impractical but also ingenious construction [11, 10] using an irregular version of Gallager's low-density parity check [8] codes. In 2002 this construction was significantly improved [9, 12, 13], resulting in the first highly practical rateless erasure codes, dubbed Fountain Codes, which decode messages of size k in $O(k \ln 1/\epsilon)$ symbol operations, and achieve any fixed overhead $\epsilon > 0$.

This paper presents a new family of linear-time rateless erasure codes, called Perpetual Codes, that bear little resemblance to Fountain Codes. Perpetual Codes improve over Fountain Codes by achieving very good cache performance, and obviating the (inherent in Fountain Codes) need for symbol sorting. Even though Perpetual Codes decode in $O(k \operatorname{poly}(1/\epsilon))$ symbol operations, our preliminary experimental results with Perpetual Codes achieve over 300Mbit/sec decoding bandwidth, which suggests that they are competitive with Fountain Codes (whose performance in unpublicized) in practice. We discuss future directions that could increase decoding performance significantly.

We believe that Perpetual Codes (with minor modifications) are both rateless and capacity achieving for a large class of error channels, and in particular the Binary Symmetric Channel, while still being linear-time encodable and decodable. In contrast, it has been proven [7] that Fountain Codes cannot be extended to have the rateless property on error channels.

Contents

1 Introduction

1.1 Generalities

Coding for the erasure channel has recently enjoyed growing popularity due to its applications to very tangible practical problems. A line of work by Luby et al. [11, 9, 13], which resulted in the creation of Raptor Codes, has introduced the current state of the art practical erasure codes and has thus set certain standards regarding what is required of practical codes in view of their applications. These standards have caused the design focus for practical codes to deviate from the focus of classical Shannon Coding Theory. In this paper paper we present a new class of cache-friendly practical erasure codes that are not only competitive with Raptor Codes but also enable applications that are infeasible using Raptor Codes. Before we discuss the merits of our constructions we review the setting and goals of erasure coding.

The erasure coding problem was introduced by Elias [6] in 1955. A sender and a receiver communicate over a one-way lossy communication channel. An input message at the sender has to be delivered in full fidelity to the receiver, while minimizing the number of transmissions. Formally, an input message X consisting of k *input symbols* $X = (x_1, \ldots, x_k) \in \Sigma^k$ over a finite *input alphabet* Σ is given at the sender. The communication channel $\mathcal{C}$ is modeled by a function $\mathcal{C} : \Gamma^n \to \Lambda^n$, where Γ is a finite alphabet of *channel input symbols*, $\Lambda = \Gamma \cup \{e\}$ is an alphabet of *channel output symbols* with e being a special symbol representing "erasure." The channel transmission function is such that for $Y = (y_1, \ldots, y_n) \in \Gamma^n$, $\mathcal{C}(Y) = (y'_1, \ldots, y'_n) \in \Lambda^n$, where $y'_i \in \{y_i, e\}$. An erasure correcting code is a pair of probabilistic functions:

$$\begin{aligned} \mathcal{E} : \Sigma^k \to \Gamma^n, &\qquad \text{the encoder algorithm,} \\ \mathcal{D} : \Lambda^n \to \Sigma^k, &\qquad \text{the decoder algorithm} \end{aligned}$$

For which $\mathcal{D}(\mathcal{C}(\mathcal{E}(X))) = X$ fails to hold with probability $O(e^{-k})$ as long as the fraction of erased symbols is no greater than a predetermined number $1-R \in [0, 1)$, and the transmission function $\mathcal{C}$ is not adversarial, i.e. it is independent of $\mathcal{E}$ and $\mathcal{D}$. The parameter R is called the *channel capacity* (or the *channel rate*). A variation of this definition requires that the the channel transmission function is the unique function that erases each symbol independently with probability $1 - R$.

The *rate of transmission* $T \in (0, 1)$ of an erasure correcting code is defined to be $T = k \log |\Sigma| / n \log |\Gamma|$. The *reception overhead* ϵ of an erasure code is defined to be $\epsilon = Rn \log |\Gamma| / k \log |\Sigma| - 1$. The reception overhead represents the extra amount of information that has to be received faithfully before message recovery is possible.

Elias showed that random linear codes can transmit at rates arbitrarily close to channel capacity as the message size k grows large. Elias' codes demonstrate that transmission at rates approaching the channel capacity can be achieved, however they are not fulfilling from a practical and algorithmic point of view, since the running time of the encoder and decoder entail $O(k^2)$ symbol operations. Conversely, Elias also showed that no code can transmit at rates higher than the channel capacity.

Without loss of generality, and in line with the real-world applications of erasure codes, we will assume (for the rest of this paper) that the input message is binary, i.e. $\Sigma = \{0, 1\}$, and that $\Gamma = \{0, 1\}^l$ for some constant l. Indeed, we are mostly concerned with sending binary files over various packet-based communication networks like the Internet, Satellite, Multicast, etc., where the basic transmission unit is an IP packet. For example, when using 1KB IP packets at the transmission layer we have $l = 8192$. We further adopt the convention that the input message consists of input symbols over Γ as well. This convention allows us to say that the code rate is simply k/n and the reception overhead is $Rn/k - 1$. Note that the channel capacity, the code transmission rate and the reception overhead are related by $T = R/(1 + \epsilon)$. For the rest of this paper, we will primarily be concerned with the reception overhead as it is a simple indicator of how close the transmission rate is to the channel capacity.

1.2 Related Work

Elias' introduction of erasure coding posed the natural algorithmic problem of finding fast, and in particular linear-time, encoding and decoding algorithms for the erasure channel that approach capacity. For practical purposes the erasure coding requirements have been relaxed to require an arbitrarily small but fixed probability of failure $\pi > 0$, rather than the originally required exponentially vanishing probability of failure $O(e^{-k})$.

This problem was resolved by the construction of Tornado Codes [11] and was later significantly improved upon by the currently best (for practical purposes) erasure codes, dubbed Raptor Codes [13], which we use as a fair benchmark for comparison with the codes we introduce in this paper.

For any desired reception overhead $\epsilon > 0$ and probability of failure $\pi > 0$ there is a Raptor Code that encodes and decodes in $O(k \ln 1/\epsilon\pi)$ symbol operations, where a symbol operation is a bitwise XOR of the values of two symbols. Industrial grade implementations of Raptor Codes achieve failure probabilities to the order of 10^{-5},10^{-10}, or 10^{-12} depending on specific application requirements.

Another important aspect of Raptor Codes, pioneered by LT and Online Codes [9, 12], is the property of being *rateless* (also referred to as *universal*). A code is rateless if a single code design can be used on any erasure channel regardless of its channel capacity. To make this more concrete, a rateless code with reception overhead ϵ and probability of failure π, is a pair of functions:

$$\begin{aligned} \mathcal{E}_{\epsilon,\pi} &: \Gamma^k \times \mathbb{N} \to \Gamma, & &\text{the encoder algorithm,} \\ \mathcal{D}_{\epsilon,\pi} &: \Gamma^{(1+\epsilon)k} \to \Gamma^k, & &\text{the decoder algorithm} \end{aligned}$$

Such that for any input message $X = (x_1, \ldots, x_k)$, and $y_i = \mathcal{E}_{\epsilon,\pi}(X, i)$, the equality $\mathcal{D}_{\epsilon,\pi}(y_{i_1}, \ldots, y_{i_{(1+\epsilon)k}}) = X$ fails to hold with probability at most π, whenever the indices $i_1, \ldots, i_{(1+\epsilon)k}$ are all different and have been chosen in a non-adversarial way, i.e. independently of the functions $\mathcal{E}_{\epsilon,\pi}$ and $\mathcal{D}_{\epsilon,\pi}$.

Rateless codes are best interpreted in the context of broadcast transmission, where a sender uses a rateless encoder function to transmit a potentially infinite stream of output symbols. A receiver is able to recover the original message as soon as $(1+\epsilon)k$ faithfully transmitted output symbols have been received. This setting naturally models applications like Satellite broadcast, multicast transmission of bulk data, and other applications described in more detail in [2].

In the rateless setting, the running time of the encoder is measured per output symbol produced. For practical codes, it is required that the time to produce one output symbol is constant. Raptor Codes produce an individual output symbol in $O(\ln 1/\epsilon\pi)$ symbol operations. The encoder is typically also allowed to perform preprocessing of the input message in linear time.

In summary, for a desired overhead ϵ and failure probability π, a practical erasure code is a rateless erasure code which produces an output symbol in $O(\ln 1/\epsilon\pi)$ symbol operations and recovers the input message from $(1+\epsilon)k$ output symbols in $O(k \ln 1/\epsilon\pi)$ symbol operations with probability at least $1-\pi$.

1.3 Our Contributions

While Raptor Codes have proven to be very attractive for various applications, a closer look at their internal design exhibits two problems that pose difficulties in the most common case applications.

Each output symbol that Raptor Codes generate is a sparse linear combination of the input symbols. Decoding Raptor Codes entails solving a large sparse system consisting of the linear equations represented by the output symbols. The output symbols are generated in a carefully crafted manner so that when $(1+\epsilon)k$ output symbols are received at the decoder, the resulting system of equations can be solved entirely via back-substitution.

Because the system of equations at the decoder is random, the solution of this system entails random-access (to the output symbols) at each step of the back-substitution order, thereby incurring essentially worst-case cache performance. This drawback is tolerable, when the entire input message can be fit in memory since memory access is still fairly fast, but it is utterly prohibitive when messages are larger and need to be partially stored on disk. As a result Raptor Codes are practically unable to decode messages that are larger than a few gigabytes.

This problem has been identified in [3] and a non-rigorous work-around has been proposed. In particular, [3] suggests encoding (and decoding) using Raptor Codes over a large sliding window across the input message. This work-around however sacrifices various properties of the original Raptor Codes and is not a satisfactory solution. Furthermore, it still fails to improve cache performance significantly as the sliding window size is required to be fairly large.

Sequential-access decodable codes are interesting not only because they enable coding over very large messages, but also because they can potentially increase the coding speed of smaller in-memory messages. Coding speed for smaller messages is a very sensitive issue when coding is applied to live transmission of data (e.g. audio or video) [4].

A second problem with Raptor Codes stems from the fact that the Raptor decoder can begin decoding only after a large majority of output symbols have been received. This could be particularly inconvenient in certain applications where input messages are sent back-to-back as is the case with live data transmission. In yet other applications, the message sizes could be very large (to the order of 500GB), as is the case with TV programs' dailies distribution. In such instances the receiver has to wait for the transmission to complete, and then also wait for the message to be decoded. Ideally, a majority of the decoding algorithm should be performed during the time of reception, so that as soon as sufficiently many output symbols have been received the receiver will have the decoded input message.

A related problem with Raptor Codes has to do with symbol ordering. For virtually all practical applications, a message is not considered decoded until it is represented in a contiguous fashion in the original order in memory or on disk. Since Raptor Codes are based on decoding a random system of equations, after the system has been solved the correct values of the input symbols are permuted randomly and a proper sorting is due before decoding can be declared complete. Sorting is an expensive and cache-unfriendly step that incurs further time overhead. To make matters worse in the case of Raptor Codes, sorting cannot be performed before all output symbols have been received and the rest of the decoding process has been completed. Therefore the latency of the decoder is further increased.

This paper presents a rateless erasure coding design, called *Perpetual Coding*, that tries to obviate all of the above-mentioned problems with Raptor Codes. Perpetual Coding bears similarities to Forney's code concatenation techniques and convolutional codes. We present two varieties of Perpetual Codes: Smooth and Block, both of which we implement and test experimentally.

Smooth Perpetual Codes exhibit performance competitive with that of Raptor Codes. While the performance of Raptor Codes is not publicized, we have implemented our own version for comparison purposes. Our experiments indicate that the Smooth codes decode at over 300Mbit/sec. Careful analysis of Smooth Perpetual Codes is complicated and evades us at the moment. We thus investigate Block Perpetual Codes as a simplified toy-version of Smooth Perpetual Codes that has managable and relatively clear analysis. In practice, they achieve 33Mbit/sec decoding bandwidth. We show that Block Perpetual Codes produce output symbols in $O(1/\epsilon^2 \log(1/\pi))$ symbol operations, and decode in $O(k/\epsilon^2 \log(1/\pi))$ symbol operations.

Despite the fact that the running time of Block Perpetual Codes and most likely Smooth Perpetual Codes is poly-logarithmic in $1/\epsilon$, their good performance is achieved due to their cache-friendly characteristics. We therefore suggest that in the realm of practical coding cache-friendliness is at least as important as running time dependence on $1/\epsilon$.

As we have already alluded, Perpetual Coding entails a mostly sequential access to memory, and thus it enables coding over very large messages, which are unmanagable with Raptor Codes. The Perpetual design shifts most of the computational cost of the decoder within the time of reception and it performs the symbol sorting procedure implicitly within this time frame.

In the conclusion of this paper we describe future improvements of the Perpetual design that have the potential of increasing performance significantly. We also offer intuitive arguments, suggesting that the techniques used in this paper can be easily extended to more general error channels, possibly resulting in practical capacity-achieving rateless error-correcting codes. Such codes have not yet been constructed, and in particular it has been shown that the Raptor Code design cannot be extended to have the rateless property on error channels [7].

To the best of our no knowledge, no codes have been previously constructed that fall within an order of magnitude of the performance of Raptor Codes. Therefore we belive that our constructions of Perpetual Codes comprise a non-trivial contribution to the field of practical coding.

2 Two Code Designs and Algorithms

We devote this section to describing the two flavors of Perpetual Codes, dubbed Block and Smooth, and the encoding and decoding algorithms that go with them.

In both code designs described below the general setting will be as follows. We use the term *linear combination* of symbols to refer to linear combinations of symbol values over $\mathbb{F}_2$. We elaborate on this more later.

At the sender (encoding step):

1. Begin with an input message at the sender, consisting of k input symbols, denoted $x = (x_0, \ldots, x_{k-1})$
2. Sender uses *outer encoder* to compute n *source symbols*, denoted $y = (y_0, \ldots, y_{n-1})$, from the input symbols for some prescribed $n \geq k$. Where each source symbol is a linear combination of some subset of the x's
3. Sender uses *inner encoder* to repeatedly compute a potentially limitless sequence of output symbols $z_0, z_1, \ldots$ from the source symbols and sends it over the communication channel to the receiver. Each output symbol z_i is independently computed as a linear combination of some subset of the y's, which depends on i and a universally agreed-upon pseudo-random function

At the receiver (decoding step):

1. Receiver collects $m = (1 + \epsilon)k$ faithfully transmitted output symbols. Since output symbols are pseudo-random and independent, without loss of generality we say that the receiver faithfully received the first m output symbols, and denote them as $z = (z_0, \ldots, z_{m-1})$
2. Receiver applies *inner decoder* which using z and the system of linear dependencies among the z's "partially recovers" a large fraction of the y's
3. Using the partially recovered y's, receiver produces a considerably smaller system of equations on the "missing" subset of the y's
4. Using *outer decoder* receiver solves the smaller system of linear dependencies to produce exact solutions to the mssing y's
5. Receiver uses the exact solutions from the previous step to find an exact solution to all of y (and consequently all of x)

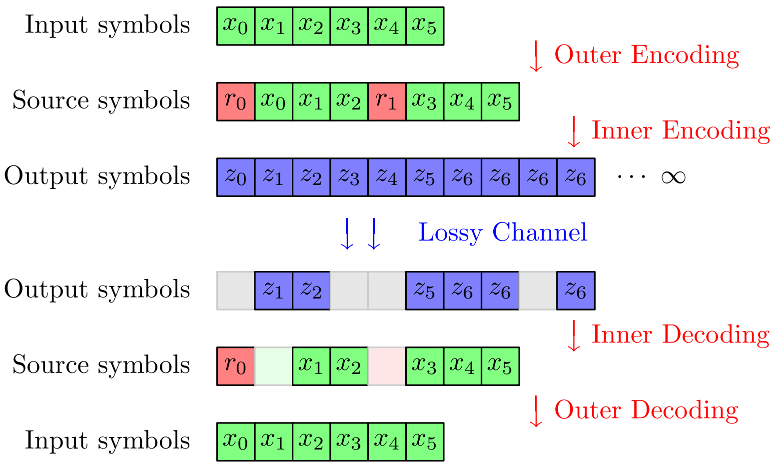

Figure 1: A crude illustration of the encoding and decoding processes.

This process is roughly illustrated in Figure 1.

Perpetual encoding is the process of generating the output symbols from the source symbols. Respectively, Perpetual Decoding is the process that recovers (not explicitly) a large portion of the values of the source symbols from the output symbols.

The source symbols incorporate the information of the input symbols while adding some redundancy. The redundancy is necessary so that when the Perpetual Decoding recovers a large fraction of the information in the source symbols, the added redundancy can help recover all input symbols.

The technique of adding redundancy to the input message before using a rateless code is called precoding. In the case of Block Perpetual Codes any linear-time (not necessarily capacity-achieving) code can be used as a precode. Nevertheless, we describe our construction using a standard random right-regular code, which was also used as a precode in the constructions of Online and Raptor Codes.

In the case of Smooth Perpetual Codes it is not immediately obvious to us that any precode can be used, because as we shall see there is an intricate interplay between the Perpetual Code and the precode during decoding. Thus for the Smooth Perpetual Code we describe the specific precode that we used in our implementation and which produced good experimental results.

It should be understood that all random choices made in the encoding algorithms described below are made using a pseudo-random function, which is known to the sender and the receiver. This is naturally necessary so that the receiver can attach the same meaning to output and source symbol as was done by the sender.

2.1 Smooth Codes

The goal of this section is to describe the Smooth Codes, give some intuition as to why they work, and emphasize some crucial algorithmic optimizations that make Smooth Codes practical.

2.1.1 Some Conventions

In line with our conventions so far, the basic unit of information that we will operate with is a *symbol*, which is an element of $\Gamma = \{0,1\}^l$ for some fixed constant l. We shall implicitly use the natural equivalence Γ and the l-dimensional vector space $\mathbb{F}_2^l$ over the field $\mathbb{F}_2 = \mathbb{Z}/2$. All algorithms, described below, entail two kinds of symbol operations: copying and addition in $\mathbb{F}_2^l$. The latter one is also refered to as addition over $\mathbb{F}_2$. The result of adding two symbols over $\mathbb{F}_2$ is simply the bitwise XOR of the values of its operands. Nevertheless, we find it useful to keep the algebraic generality of these operations in mind, because later it will come handy in our discussion regarding future directions and improvements.

Whenever we use the term "linear combination" we will always refer to a linear combination of symbols over $\mathbb{F}_2$. In our algorithmic exposition below, we will often create a new symbol b as a linear combination of some pre-existing symbols $a_1, \ldots, a_r$. This operation will be denoted using the shorthand notation $b := a_1 \oplus \cdots \oplus a_r$, and we shall say b is *associated* with a_i (for all i). Trivially, $b := a$ shall mean that b is a copy of a's value.

2.1.2 Inner Encoder and Initial Descent

Similarly to the Raptor Code design, Smooth Codes entail two layers: an outer code and a rateless inner code. The input message $x = (x_0, \ldots, x_{k-1})$ is first encoded into a *source message* $y = (y_0, \ldots, y_{n-1})$, for a prescribed $n \geq k$, using the outer code. Subsequently, the source message is used to produce a potentially limitless sequence of output symbols $z_0, z_1, \ldots$ using the inner code.

We shall begin with a description of the inner code, as its properties motivate the design of the outer code. Given a source message y, the inner encoder is responsible for generating a single output symbol. The inner encoder can be called repeatedly to generate an effectively infinite sequence of output symbols. Each such output symbol is generated in a randomized fashion, independently from the others.

Define a *window* at offset i and aperture α to be a set of α source symbols with consecutive indices starting at the i-th source symbol and allowing for wrap-around. Formally:

$$W_i = \{y_{(i+j) \bmod n} \mid 0 \leq j < \alpha\}$$

An instance of a Smooth perpetual code is parameterized, among others, by a *code aperture* α. Given a code aperture and source message size n, a *smooth* collection of windows is defined as:

$$\Omega_S = \{W_i \mid 0 \leq i < n\}$$

This should be contrasted with the *block* collection of windows (used in the Block version of the codes):

$$\Omega_B = \{W_i \mid i = \alpha j, \text{ for } 0 \leq j < n/\alpha\}$$

An output symbol z_i is generated by the Smooth inner encoder as follows:

- ✠ *A window W is selected uniformly at random from Ω_S*
- ✠ *The output symbol z_i is computed to equal a linear combination of a subset of the source symbols in W, such that each source symbol is associated with z_i with probability 1/2*

We should note that all random choices throughout the algorithms described here are actually pseudo-random with respect to some function shared by the encoder and the decoder. This is necessary so that both of them attach the same meaning to the various symbols that are being manipulated (source symbols and output symbols). Also the random choices made in the generation of output symbol z_i are further keyed by the output symbol's index i, so that different output symbols can be properly distringuished.

The main data structure at the inner decoder represents a linear system of equations over $\mathbb{F}_2$: $Ay = v^A$, where A is a 0/1-matrix of dimensions $n \times n$, y is a vector of unknown symbols corresponding to the source symbols, and v^A is a vector of symbols. We use the notation A_i to denote the i-th row of A, and $A_{i,j}$ to denote the (i, j)-th entry of A.

The inner decoder will maintain the invariant that the i-th row of A, for all $0 \leq i < n$, is allowed to have non-zero entries only at columns $\{i \bmod n, \ldots, (i + \alpha - 1) \bmod n\}$, i.e. along a thin strip to the right of the main diagonal. This allows for a very efficient in-memory representation of A as an n-element array of α-bit strings. As we shall expand upon later, α will be a small number. In fact, our experiments show that $\alpha = 32$ is sufficient for practical purposes. Therefore, A is particularly efficiently represented as an array of 32-bit machine

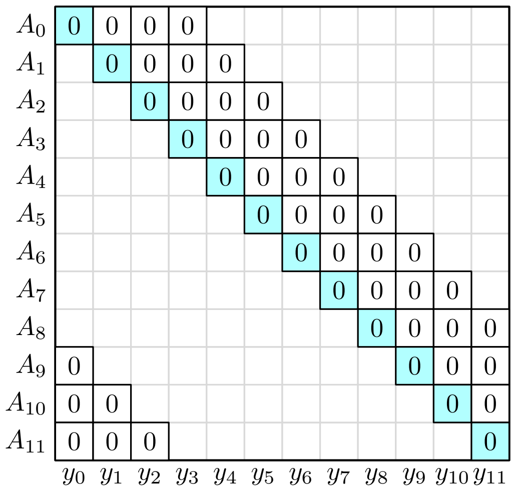

Figure 2: The initial form of the inner decoder matrix A before decoding begins. In this example $n = 12$, $\alpha = 4$ and the blue cells mark the diagonal.

words. Furthermore, all matrix operations will be efficiently implemented using machine-word bitwise operations like AND, OR, NOT, XOR and LEFT/RIGHT SHIFT. The form of A is illustrated in Figure 2.

The y vector is not represented explicitly. At the end of the entire decoding algorithm (inner and outer) the matrix A will be diagonal, and the values of the unknowns y will be stored in v^A. The vector v^A itself is represented as an array of symbols in memory. We shall see later that this array will actually be internally permuted in order to ensure that the symbol values of the input symbols (which will be a subset of the y's) are represented by a contiguous block in memory.

The decoder receives a prescribed number $m = (1 + \epsilon)k \geq k$ of faithfully transmited (i.e. not erased) output symbols, before it is able to recover the input message. The work of the decoder algorithm is divided into two parts: pre-reception work and post-reception work. Each time a faithfully received output symbol arrives, the decoder performs an $O(1)$-cost processing of the newly arrived symbol. This is part of the pre-reception work. After all m output symbols have been received, the decoder performs some more work before it completes. This separation of work is particularly important because it emphasizes an important feature of Perpetual Codes that Fountain Codes do not posses: the majority of the decoding work is pre-reception.

The inner decoder begins by initializing the system $Ay = v^A$ to the trivial system, i.e. $A := 0$ and $v^A := 0$. The goal of the pre-reception work is to leave the system A in a state similar to what the system state would have been if all symbols were received and collected first and then Gaussian forward substitution was performed.

Initial Descent (Step 1)

Each time a new output symbol is received, the inner decoder performs the *initial descend* of the output symbol into the system. Let z^v be the value of the newly received symbol z, and let z^A be the n-dimensional 0/1-vector representing its associates from among the y's. Then the initial descend proceeds as follows:

- ✠ *Define* $g : \mathbb{F}_2^n \rightarrow \mathbb{N}$ *to be such that its arguments are allowed to be only vectors* z^A *that have non-zero entries in some window of aperture* α*. If* w *is the unique such window whose first entry is non-zero, then* $g(z^A)$ *is defined as the index of that entry*
- ✠ *Repeat:*
  - ✠ *Set* $i := g(z^A)$
  - ✠ *If* $A_i = 0$*, set* $A_i := z^A$ *and* $v_i^A = z^v$*, and stop (symbol successfully pivoted into the system)*
  - ✠ *Otherwise, set* $z^A := z^A \oplus A_i$ *and* $z^v := z^v \oplus v_i^A$*. If* $z^A \neq 0$ *repeat, else stop (symbol has been cancelled)*

Observe that the initial descend algorithm maintains the invariant that if a row i of A is non-zero, then the diagonal entry $A_{i,i}$ must equal 1. We say that the source symbol i has been *filled* if $A_{i,i} \neq 0$. Alternatively, we say that there is a *hole* at source symbol i if $A_{i,i} = 0$. The inner code exhibits the asymptotically-sharp probabilistic property that after $(1 + \epsilon)n$ output symbols have been descended into the system, the number of outstanding holes is δn where $\delta = \delta(\epsilon)$ is purely a function of ϵ as n grows large. For practical applications $\delta \sim 0.5\% - 1\%$ and $\epsilon \sim 4\%$. Experimentally, the relationship between ϵ and δ holds true even for very small values of $n \sim 100$ and $\alpha = 32$. The holes are approximately uniformly distributed (across the source symbols), although a closer look shows that they tend to form small clusters of roughly $\alpha/2$ holes spread across varying ranges of roughly $\alpha^2/4$ consecutive source symbols.

The initial descend accounts for 75% of the overall decoding work, primarily because it is the most "cache unfriendly" part. Indeed, each output symbol is descended upon a random location of the diagonal of A and therefore it requires random memory access. For that reason, we found the following optimization useful.

Let the *effective aperture* of an n-dimensional vector b be the aperture of the smallest window that covers the non-zero entries of b. Denote this aperutre by $\text{eff}(b)$. When an output symbol descends down the diagonal of A, the length of the descent "path" tends to be proprtional to α^2 on average. It can be seen that the expected length of the descent path of an output

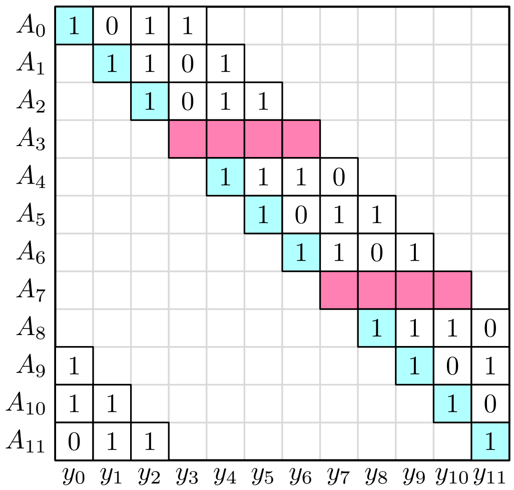

Figure 3: The form of the inner decoder matrix A after the initial descent stage. In this example $n = 12$ and $\alpha = 4$. The red cells mark holes in the system.

symbol is increased every time the output symbol passes through a non-zero row of A_i with larger effective aperture. Observing that, we modify the descend algorithm to pivot the descending symbol as soon as it encounters a previously pivoted symbol of larger effective aperture, and continue descending the previously pivoted symbol. Formally:

- ✠ *Repeat:*
  - ✠ *Set* $i := g(z^A)$
  - ✠ *If* $A_i = 0$*, set* $A_i := z^A$ *and* $v_i^A = z^v$*, and stop (symbol successfully pivoted into the system)*
  - ✠ *Otherwise:*
    - ✠ *If* $\mathit{eff}(z^A) < \mathit{eff}(A_i)$*, then switch the values of* z^A *and* A_i*, and switch the values of* z^v *and* v_i^A
    - ✠ *Set* $z^A := z^A \oplus A_i$ *and* $z^v := z^v \oplus v_i^A$*. If* $z^A \neq 0$ *repeat, else stop (symbol has been cancelled)*

After the initial descent stage of the inner decoder, the matrix A looks like the one shown in Figure 3.

2.1.3 Outer Encoding

The goal of the outer code is to add some redundancy to the input message (thereby producing the source message) so that when the inner decoder reaches a state with δn holes, the

0	0	0	r_0	x_0	x_1	r_1	x_2	x_3	r_2	x_4	x_5
y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}

Figure 4: The form of the source message, when $k = 5$, $n = 10$, $\gamma = 2/5$, and $\alpha = 4$.

redundancy in the source message can be utilized to "fill in the holes" and help diagonalize the matrix A.

The outer code that we designed to match the Smooth inner code is systematic, meaning that the source message (which is the codeword of the outercode) incorporates all k input symbols as is and adds some additinoal redundancy symbols. We employ two kinds of redundancy symbols: (a) $\alpha - 1$ *0-padding* redundancy symbols are placed at the front of the source message, and (b) γk *parity-check* redundancy symbols $r = (r_0, \ldots, r_{\gamma k})$, for some parameter $\gamma > 0$, are uniformly interleaved among the input symbols. Hence, we have the identity $n = (\alpha - 1) + (1 + \gamma)k$. Each parity-check redundancy symbol is computed as a linear combination of some subset of the input symbols. The parameter γ is chosen so as to ensure that the number of redundacies is a constant multiple larger than the number of holes. We use $\gamma = 0.03$.

The parity-check redundancy symbols are uniformly interleaved among the input symbols in order to ensure that both kinds of symbols are equally likely to be in a hole (at the inner decoder). Formally the source message y is composed using an interleaving permutation σ as follows:

$$y_{\sigma(i)} = \begin{cases} 0, & \text{for } 0 \leq i < \alpha - 1 \\ x_{i-(\alpha-1)}, & \text{for } \alpha - 1 \leq i < k + \alpha - 1 \\ r_{i-(k+\alpha-1)}, & \text{for } k + \alpha - 1 \leq i < n \end{cases}$$

Where σ is defined to be (this can be skipped upon first read):

$$\sigma(i) = \begin{cases} i, & \text{for } 0 \leq i < \alpha - 1 \\ (i - \alpha + 1) + \left\lfloor \frac{i-\alpha+1}{\lfloor 1/\gamma \rfloor} \right\rfloor, & \text{for } \alpha - 1 \leq i < k + \alpha - 1 \\ (i - k - \alpha + 1)\lfloor 1/\gamma \rfloor, & \text{for } k + \alpha - 1 \leq i < n \end{cases}$$

The form of the source message is illustrated in Figure 4.

Let us now describe how the parity redundancy symbols r are constructed. The γk redundant symbols are subdivided into two groups $r' = (r'_0, \ldots, r'_{\gamma k - \sqrt{k} - 1})$ and $r'' = (r''_0, \ldots, r''_{\sqrt{k}-1})$.

Each symbol in r' is assigned a sliding scope of $\omega\sqrt{k}$ input symbols (with truncation at the edges of the source message) that they are responsible for, such that r'_i is responsible for:

$$\left\{x_i \mid \max\left(0, i\frac{k}{|r'|} - \omega\sqrt{k}/2\right) \leq i < \min\left(k, i\frac{k}{|r'|} + \omega\sqrt{k}/2\right)\right\}$$

where ω is parameter, which in our experimental setup is $\omega = 10$. Further, r'_i is computed to be the linear combination of $d = \omega$ uniformly and randomly chosen input symbols from its scope. Intuitively, each r'_i will be used to correct for some hole within its scope.

A small number of holes will remain unaccounted for by the r' redundancies. They will be taken care of by the r'' redundancies. Each r''_i is computed as the linear combination of $\sqrt{k}$ randomly and uniformly chosen input symbols.

Note that overall the parity redundancies $r = (r' \; r'')$ can be computed in time linear in k.

2.1.4 Outer Decoding

At the decoder, the redundancy in the source message is represented by a linear system $By = v^B$, where B is a $(\gamma k \times n)$-dimensional 0/1-matrix, y is as before the vector of source symbols treated as unknowns, and v^B is a (γk)-dimensional vector of symbol values. The matrix B differs from A in the sense that it will look a lot like a sparse random matrix, and for this reason its representation is done using standard pointer-based data strcutures for sparse matrices (similarly to the ones used in Raptor Codes e.g.).

Observe that the inner decoder system $Ay = v^A$ and the outer decoder system $By = v^B$ can be viewed as a single system of linear equations, which we call simply the *decoder system*:

$$\begin{bmatrix} A \\ B \end{bmatrix} y = \begin{bmatrix} v^A \\ v^B \end{bmatrix}$$

Outer Decoder Initialization

For all i, the i-th equation $B_i y = v^B_i$ of this system is initialized to reflect the redundancy r_i in the natural way. Specifically, $v^B_i := 0$, and $B_{i,\sigma(j)} := 1$ for $j = k + \alpha - 1 + i$ and for all j for which $x_{(j-\alpha+1)}$ is associated with r_i.

Tail Clearance (Step 2)

Recall that after the end of the initial descent (Step 1), the matrix A may have some non-zero entries in the lower left corner, which we call the *tail*, due to the wrap-around nature of the output symbols. These are generally not a problem, but the decoding algorithm is significantly simpler if it does not have to deal with them. This is why we introduced the

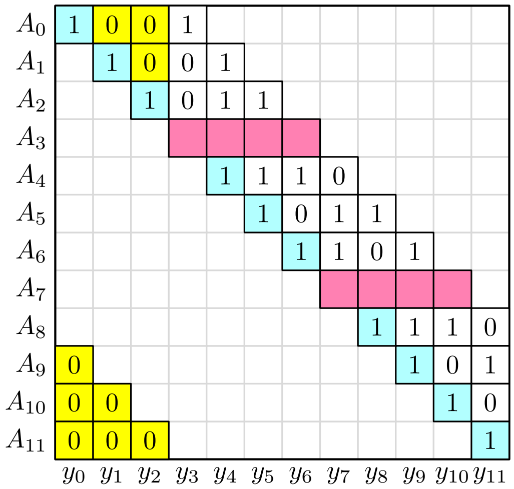

Figure 5: The inner decoder matrix A, after the tail clearance step. The yellow cells represent the cells that were modified.

0-padding redundnat symbols in the source message. The 0-padding symbols correspond exactly to the first $\alpha - 1$ columns of A where the tail resides, and hence they allow us to "delete" the tail without affecting the rest of the system.

Formally, after the initial descent we perform the *tail clearance* step, which looks like this:

- ✠ *Set* $A_{i,i} := 1$ *for* $0 \leq i < \alpha - 1$
- ✠ *Set* $A_{i,j} := 0$ *for* $0 \leq i < \alpha - 1$ *and* $i < j < \alpha - 1$
- ✠ *Set* $A_{i,j} := 0$ *for* $n - (\alpha - 1) \leq i < n$ *and* $0 \leq j < \alpha - 1$

After the tail clearance A has non-zero entries only along a thin strip including and to the right of the main diagonal, See Figure 5.

Outer System Reduction (Step 3b)

The goal of this step is to use the current state of the system A in order to reduce the system B to a much smaller system only on the unknowns that are holes in A.

Conceptually, this can be done in a straightforward manner:

- ✠ *For* $0 \leq i < n$ *in ascending order, if* $A_i \neq 0$ *do:*
  - ✠ *For* $0 \leq j < \gamma k$, *if* $B_{j,i} = 1$, *then:*
    - ✠ $B_j := B_j \oplus A_i$
    - ✠ $v_j^B := v_j^B \oplus v_i^A$

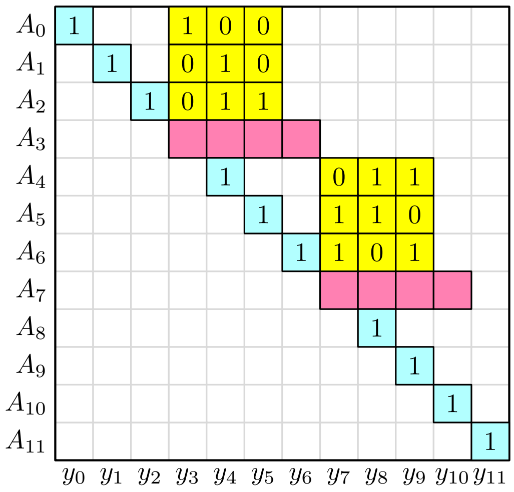

Figure 6: The inner decoder matrix A in disjoined canonical form.

In practice, this routine is unacceptably slow. Observe that when a row A_i is used to cancel a non-zero entry $B_{j,i}$ in B, it typically introduces new non-zero entries to the right of $B_{j,i}$. This effect is called the *trickle*. A non-zero entry in B is observed to trickle for about $O(\alpha^2)$ steps on average before it disappears or falls into a hole.

We have developed an optimized version of the Outer System Reduction step which speeds up this step by orders of magnitude. This is perhaps one of the more important optimizations in the code design. The main observation that enables us to optimize is that many "trickle paths" overlap and undergo roughly the same transformations. We have thus developed a technique that helps us piggy-back the work of all trickles in a single fast sweep transformation of the matrix A, described in the next paragraph.

Inner System Disjoin (Step 3a)

The Inner System Disjoin routine performs a back-substituion-like transformation of the A system, which converts each non-zero row of A to a special *disjoined* canonical form. In the canonical form, the i-th row of A, if non-zero, must have $A_{i,i} = 1$ and all other non-zero entries can appear only in $\{A_{i,h}, \ldots, A_{h+\alpha-2}\}$, where h is the index of the first hole that appears to the right of the i-th column. The disjoined canonical form of A is illustrated in Figure 6.

To transform A into a disjoined form, the following algorithm is employed:

✠ *Set $l := n$. This variable will be used to remember the position of the most recent hole to the right*

✠ *Set $i := n - 1$, repeat while $i \geq 0$:*

- ✠ *If* $A_i \neq 0$, *do:*
  - ✠ *Set* $\beta := \min(\alpha, l - i)$
  - ✠ *Set* $j := 1$, *repeat until* $j < \beta$*:*
    - ✠ *If* $A_{i,i+j} = 1$, *then:*
      - ✠ *Update* $A_i := A_i \oplus A_{i+j}$
      - ✠ *Update* $v_i^A := v_i^A \oplus v_{i+j}^A$
    - ✠ *Increment* $j := j + 1$
- ✠ *Else:*
  - ✠ *Update* $l := i$
- ✠ *Decrement* $i := i - 1$

Note that a variety of algorithms can be used to transform A into disjoined form. Some cleverer and more eleborate ones can also reduce the number of non-zero entries in A, thereby reducing the overall runtime. We choose not to go into such detail for the sake of keeping this exposition relatively clearer.

Observe that the disjoin form of A maintains that each row has no more than α non-zero entries, thereby preserving the nice property that it can be represented by a compact bit-array in memory. More aggressive disjoin forms that violate this property are possible, especially in view of the fact that bookkeeping operations do not comprise much of the running time. We leave this as a future research direction.

The disjoin transformation (Step 3a) is meant to be applied to A before the Outer System Reduction (Step 3b) is performed. It can be seen that the disjoin form of A significantly reduces the number of trickle steps during Step 3b. In particular, trickle steps "jump" from hole to hole, instead of travel smoothly across columns.

Solving the Reduced Outer System (Step 4)

The reduced outer system has γk equations and $\delta(1+\gamma)k \sim \delta k$ unknowns, where typically $\gamma/\delta \sim 5$. The reduced system consists of two parts: the top part induced by r' and the bottom one induced by r''. The former comprises the majority of the system (as $\gamma k - \sqrt{k} \gg \sqrt{k}$) and it is roughly diagonal. More specifically, it has a thin cloud of non-zero entries scattered around the main diagonal (note this is not a square matrix). The latter is essentially random.

We solve the reduced outer system using plain-vanilla Gaussian elimination, using "the textbook elimination order", i.e. forward-substitute from left to right, and then backward substitute from right to left. According to our experiments, this is achieved in observed linear time.

When done, the solutions of the source symbols that were in holes are "pushed back" into the inner decoder system. So, for example, if after solving the reduced system $B_i = e_j$ (where e_j is the j-th basis vector), then we push back by setting $A_j := e_j$ and $v_j^A := v_i^B$.

Diagonalizing the Inner System (Step 5)

At last, we perform a single round of plain back-substitution in A from right to left, in order to diagonalize the system. Now, the values of the input message are located in the corresponding entries of v^A. In a real-world implementation, the actual memory locations of the symbols in v^A should be logically permuted by σ^{-1} so as to ensure that at the end of decoding the input message resides in a contiguous block of memory.

2.1.5 Computational Costs

We distinguish two kinds of computational costs: bookkeeping and symbol operations. Bookkeeping operations consist of the manipulations over the matrices A and B. Symbol operations consist of the bitwise XOR operations over a pair of symbols. Bookkeeping operations comprise a tiny fraction of the decoder runtime, so we generally disregard them. Therefore, we measure the runtime costs by the number of XOR operations. Since the main contribution of the Perpetual design is improved cache performnace, we will also be interested in the number of times a symbol has to be recalled in cache, assuming that the cache scheduler is based on the least-recently used eviction policy.

To give a slight touch of generality, we will use the terms *local* and *remote* memory, instead of cache and memory, because our arguments could be transparently carried over to other layers of the memory hierarchy, like disk vs. memory, e.g. In the analysis of the cache performance, we will make various assumptions about the relative sizes of the local and remote memories. We say that an algorithm has (N, M)-cost, if the algorithm performs N symbol operations and M cache recalls. We will also adopt the notation L for the size of the local memory, and R for the size of the remote memory, where L and R are measured in multiples of the symbol length.

As a rough benchmark for comparison, let us briefly discuss the runtime of Raptor Codes. A practical Raptor decoder performs approximately $8k$ symbol operations do decode a message, such that each operation is performed on a pair of symbols, where one of the symbols was recently used and the other one is essentially random (from the set of symbols in memory). This amounts to $(8k, 8k)$-cost. At the end of decoding a Raptor Code needs to sort the decoded symbols in (k, k)-cost. Thus the Raptor Decoder incurs a total $(9k, 9k)$-cost.

Let us now step through the costs incurred by the Smooth Perpetual Decoder. All numeric costs listed below are given according to their observed values from our experiments where $\alpha = 32, \delta = 0.005, \gamma = 0.03, \omega = 10, d = 10$:

1. *Initialization* $(0, 0)$*-cost:* A clever implementation does not require explicit symbol value initialization

2. *Initial Descend* $(11k, 11k)$*-cost:* Each output symbol is descended at a random location along A's diagonal, and on average we experimentally observed that each descend makes 11 trickle steps before it settles

3. *Tail Clearance* $(0, 0)$*-cost:* This step entails only bookkeeping operations

4. *Disjoin Tranform* $(6k, k)$*-cost:* We have experimentally observed that each filled source symbol (in A) has 6 non-zero entries on average. On the other hand, due to the sequential nature of the disjoin step, each symbol has to be recalled at most once

5. *Outer System Reduction* $(2k, \gamma k)$*-cost:* The diagonal nature of B ensures that each redundancy symbol has to be recalled only once, assuming that $L/R > \sqrt{k}$. This assumption holds for most practical applications both at the cache/memory and memory/disk layers

6. *Outer System Solution* $(0.3k, \gamma k)$*-cost:* Due to the diagonal nature of the reduced outer system, each symbol needs to be recalled only once

7. *Solutions Pushback* $(\delta k, 2\delta k)$*-cost:* There are δk solutions in v_B each of which has to be recalled and copied to its corresponding location in v_A

8. *Inner back-substitution* $(6k, k)$*-cost:* This step is, in some sense, "symmetric" to the prior disjoin transform and seems to incur the same cost

Overall, we get $(25k, 13k)$-cost. In practice, this amounts to 300Mbit/sec (on hardware described below), compared to Raptor Codes, whose practical performance is 400Mbit/sec (using our own implementation) and a corresponding $(9k, 9k)$-cost.

We should mention that our cost notation does not account for lookahead-prone memory, like disks, primarily because we have not run tests on files larger than 1GB at the moment. However, it should be intuitively clear that, considering the large discrepancy between random vs. sequantial disk read access, Perpetual Codes would probably significantly outperform Raptor Codes on messages over a few gigabytes. This is explained by the following arguments:

1. By far the most expensive step is the initial descent where each output symbol descent will require one seek, followed by a few sequential reads, amounting to about k random seeks

2. The disjoin transform amounts to one complete round of sequential read of the entire message length, where the assumption is that a clever implementation reverses the direction of A's diagonal so that walking up the diagonal corresponds to a forward disk read

3. The Outer System Reduction comprises another complete sequential read (note the diagonal nature of the precode)

4. The Pushback will entail a very small number δk of disk seeks

5. And finally the backsubstitution entails another complete sequential read/write

In total, the Perpetual Decoder requires approximately k seeks and 3 sequential sweeps. In contrast a Raptor Code requires $9k$ seeks, which undoubtedly will incur significantly higher costs.

2.1.6 Experimental Results

We have performed various experiments with Smooth Perpetual Codes using various parameter settings. We manually found the following configuration of parameters most useful:

1. Aperture $\alpha = 32$

2. $\delta = 0.05$ and $\gamma = 6\delta = 0.03$

3. $\omega = d = 10$

The values of all parameters are interrelated and our choices will certainly be suboptimal without precise analysis. Nevertheless we observed consistent results for all message sizes.

We fixed the symbol size to 1KB which amounts to $l = 8192$, and varied the message size from 100KB to 1GB. We enforced that the initial descend ends as soon as δk holes remain. The observed reception overhead was no greater than 8%. The encoder bandwidth was consistent at 650Mbit/sec and the decoder bandwidth was consistent at 320Mbit/sec. We performed all experiments on a Intel Xeon 2.8GHz CPU with 2GB of RAM memory, 16KB of L1 cache, and 1MB of L2 cache.

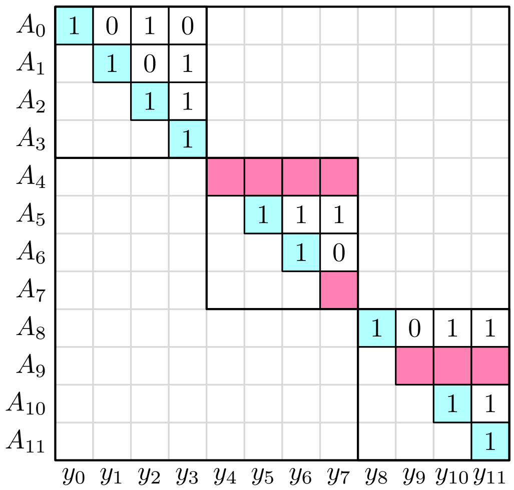

Figure 7:

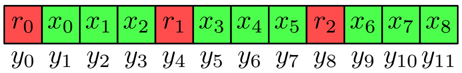

Figure 8:

2.2 Block Codes

Block Perpetual Codes are a simplified toy version of Smooth codes with relatively simple analysis. We describe them briefly mainly because they provide some intuition about the underlying processes that take place in Smooth codes.

The inner encoder here uses the block window collection Ω_B, instead of the smooth one. This ensures that after the initial descent, the matrix A is in a convenient block form, as shown in Figure 7.

The block nature of A removes the need for 0-padding symbols in the source messsage. Thus the source message is simply an interleaving of input and redundance symbols, as shown in Figure 8.

All blocks in A that are full rank after the initial descent are locally back-substituted to produce exact solutions to all participating source symbols. Conversely, in order to simplify analysis, all source symbols in blocks of non-full rank are declared to be holes.

This ensures that at this stage of the decoding process $(1-\delta)n$ source symbols are entirely recovered, and δn are missing. This should be contrasted with the smooth case, where if a source symbol is filled (i.e. not in a hole) it is not explicitly recovered just yet.

The outer precode for block codes can be any fixed-rate erasure code. We use the right-regular random precode, also used in Raptor and Online Codes, whereby each redundant symbol is a linear combination of input symbols, such that each input symbol is associated with $d = 3$ randomly selected redundancy symbols.

After reducing the outer system, the reduced system is solved via back-substitution only. That this is possible is a property of the right-regular random precode. The solutions of the reduced outer system are pushed back into the inner system and this completes the decoding algorithm.

2.2.1 Experimental Results

As will be seen from our analysis, Block Perpetual Codes are not very practical. However, we performed a simple measurement of the running time of a Block Perpetual Code which was at the least informative.

We implemented a Block Perpetual Decoder without a precode and with aperture $\alpha = 1000$. We tested it on an source message of size 1GB, with symbol size 1KB (i.e. $n = 1000000$). Our experiments indicated that no more than 13% overhead was ever needed to recover the entire message, and the decoding speed was stable at 33Mbit/sec. As the message size is scaled down, the decoding speed remains unchanged, while the reception overhead improves (decreases). This alone is a useful code in practice for light-weight applications, like multi-source downloads in peer-to-peer systems, where the reception rate is typically well below 33Mbit/sec. In contrast, we saw that Smooth Perpetual Codes achieve better reception overhead and better running times with apertures as small as $\alpha = 32$.

3 Block Perpetual Codes Analysis

The main result the desribes the behavior of the Block Perpetual Inner code is the following:

Theorem 3.1 (The Block Perpetual Theorem). *For any $\epsilon > 0$ and $\delta > 0$, there is an aperture $\alpha = \alpha(\epsilon, \delta) \sim 2\ln(1/\delta)/\epsilon^2$ such that the Block Perpetual Code with aperture α and source message size n can recover at least $(1-\delta)n$ source symbols from at most $(1+\epsilon)n$ output symbols with probability $O(e^{-n})$ in expected (and sharply concentrated) $n\alpha/2$ symbol operations.*

In the theorem statement, $\alpha = \alpha(\epsilon, \delta) \sim 2\ln(1/\delta)/\epsilon^2$ means that α is a function of ϵ and δ and α approaches the value $2\ln(1/\delta)/\epsilon^2$ as ϵ and δ become small.

To produce a code that recovers the complete input message, the inner code can be combined with any linear-time, not necessarily capacity-achieving, fixed rate erasure correcting code. The composite code will inherit the failure probability of the precode. One such precode, that we use in our algorithm descriptions above, is the right-regular random precode, also used in [12, 13, 14].

3.1 Proof Outline

Definition 3.1. *A block of source symbols is said to be fully recovered, if all α source symbols comprising that block have been recovered by the decoder.*

We are going to prove Theorem (3.1) by showing that the probability that a fixed block is not fully recovered is bounded from above by $2\delta/3$. Therefore in expectation at most $2\delta n/3\alpha$ blocks are not fully recovered, and hence in expectation at most a fraction $2\delta/3$ of the source symbols are not recovered.

Finally, we will prove that the outcome fails to meet the expectation with a vanishing probability by applying a martingale argument. In particular, we will show that the probability that less than $2\delta n/3\alpha$ blocks have been fully recovered is at most $O(1/n)$.

3.2 Proof of the Block Perpetual Theorem

Definition 3.2. *A block is said to be β-filled for some non-negative integer β if at least $\alpha+\beta$ output symbols received at the decoder are assigned to that block.*

For a fixed block B let B_F be the event that B is β-filled, where β is a parameter. Further, let B_R be the event that B is fully recovered. Then we have that:

$$\begin{aligned}\mathbf{Pr}\left[\overline{B_R}\right] &= \mathbf{Pr}\left[\overline{B_R}|B_F\right]\mathbf{Pr}\left[B_F\right] + \mathbf{Pr}\left[\overline{B_R}|\overline{B_F}\right]\mathbf{Pr}\left[\overline{B_F}\right] \\ &\leq \mathbf{Pr}\left[\overline{B_R}|B_F\right] + \mathbf{Pr}\left[\overline{B_F}\right] \qquad (3.1)\end{aligned}$$

We will examine the terms of (3.1) separately, and we will require the stronger condition that:

$$\mathbf{Pr}\left[\overline{B_R}|B_F\right] \leq \delta/3 \qquad (3.2)$$

$$\mathbf{Pr}\left[\overline{B_F}\right] \leq \delta/3 \qquad (3.3)$$

$\mathbf{Pr}\left[\overline{B_R}|B_F\right]$ is easily computed using the following lemma:

Lemma 3.1 (The Full Rank Lemma). *Let A be an $(\alpha+\beta)\times\alpha$ random 0/1-matrix where each entry equals "1" independently with probability 1/2. Then A does not have full rank over $\mathbb{F}_2$ (the integers modulo 2) with probability at most $2^{-\beta}$.*

Proof. A does not have full rank if and only if all rows A_i of A fall within some hyperplane of the α-dimensional vector space $\mathbf{V}$ over $\mathbb{F}_2$. Let $H = [2^\alpha]$ be the index set of hyperplanes of $\mathbf{V}$, and let C_j denote the event that all rows of A fall within the j^{th} hyperplane of $\mathbf{V}$. Then, the probability that A does not have full rank equals $\mathbf{Pr}\left[\,\vee_{j\in H} C_j\right]$.

A hyperplane in $\mathbf{V}$ is an $(\alpha-1)$-dimensional subspace of $\mathbf{V}$, and each such hyperplane spans exactly $2^\alpha/2$ points of $\mathbf{V}$. Therefore $\mathbf{Pr}[C_j] = 2^{-(\alpha+\beta)}$. Using the union bound:

$$\mathbf{Pr}\left[\,\vee_{j\in H} C_j\right] \leq \sum_{j\in H}\mathbf{Pr}[C_j] \leq 2^{-\beta}$$

Using the Inclusion-Exclusion Principle, also known as the Bonferroni Inequalities (see Chapter 8.3 of [1]), one can verify that this bound is sharp when $\beta = o(\alpha)$. $\square$

The lemma immediately implies that:

$$\mathbf{Pr}\left[\overline{B_R}|B_F\right] \leq 2^{-\beta}$$

And combining with (3.2) we get:

$$\beta \geq \log(1/\delta) + \log 3 = \frac{\ln 1/\delta + \ln 3}{\ln 2}$$

Now we turn our attention to the second term of (3.1) an an attempt to fulfill condition (3.3). Let X be the random variable that equals the number of output symbols that are assigned to block B. And let X_i be the indicator random variable that output symbol i is assigned to block B. For $\mu = \mathbf{E}[X]$ we have:

$$\mu = \mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{(1+\epsilon)n} X_i\right] = \sum_{i=1}^{(1+\epsilon)n}\mathbf{E}[X_i] = (1+\epsilon)\alpha$$

We are interested in bounding:

$$\mathbf{Pr}\left[\overline{B_F}\right] = \mathbf{Pr}[X < \alpha+\beta]$$

On an intuitive level, Y is approximately Poisson-distributed (as n grows large) with mean μ. We are trying to bound the likelihood of the event $X < \alpha+\beta$. If we view α as a growing parameter with respect to the fixed β and ϵ, our goal is to grow α just enough so that $\alpha+\beta$ is sufficiently far below the mean $\mu = (1+\epsilon)\alpha$ so that the tail probability $\mathbf{Pr}[Y < \alpha+\beta]$ becomes desirably small.

Formally, we model X as a binomial random variable with $Y = \sum_{i=1}^{(1+\epsilon)n} X_i$ and $\mathbf{Pr}[X_i = 1] = \alpha/n$. To get the desired bound, we will apply the following Chernoff bound:

Theorem 3.2 (Chernoff Bound). *For $X = X_1 + \cdots + X_n$ a sum of independent Bernoulli random variables with $\mu = \mathbf{E}[X]$ and $\omega > 0$:*

$$\mathbf{Pr}[X \le (1-\omega)\mu] \le e^{-\omega^2\mu/2}$$

In order to apply that bound we first have to ensure that $\alpha+\beta \le \mu$, or equivalently:

$$\alpha \ge \frac{\ln 1/\delta + \ln 3}{\epsilon \ln 2} \tag{3.4}$$

(Note that relaxing this requirement makes the analysis excessively complicated and does not win us anything asymptotically.)

For our needs we want $(1-\omega)\mu = \alpha+\beta$, and hence we set:

$$\omega = \frac{\alpha\epsilon - \beta}{(1+\epsilon)\alpha}$$

Then the Chernoff bound then yields:

$$\mathbf{Pr}\left[\overline{B_F}\right] = \mathbf{Pr}[X \le \alpha+\beta] \le \exp\left[-\frac{(\alpha\epsilon-\beta)^2}{2(1+\epsilon)\alpha}\right] \tag{3.5}$$

Combining condition (3.3) and the Chernoff bound (3.5) yields:

$$\alpha \ge \frac{2\ln 1/\delta}{\epsilon^2}\big(1 + o_\epsilon(1)\big) \tag{3.6}$$

Where $o_\epsilon(1)$ vanishes as ϵ approaches 0. The details of the asymptotics of $o_\epsilon(1)$ and the derivation of (3.6) are given in a Lemma A.1 in the Appendix, as they are not vital for our discussion.

At last, we have established that when condition (3.6) holds, $\mathbf{Pr}[\overline{B_R}] \le 2\delta/3$. Now we show that the actual number of fully recovered blocks does not deviate far from its expectation. Let the vector of independent random variables $Y = (Y_1, \ldots, Y_{(1+\epsilon)n})$ be such that Y_i represents the outcome of the i^{th} output symbol's choice of associates. And let $f(Y)$ be the number of not fully recoverable blocks, given a specific Y. Define the martingale $X_0, X_1, \ldots, X_{(1+\epsilon)n}$ by:

$$X_i = \mathbf{E}\left[f(Y)|Y_j \text{ for } 1 \leq j \leq i\right]$$

Lemma B.1 (in the Appendix) shows that this is indeed a martingale. Note that $X_0 = \mathbf{E}[f(Y)] = 2\delta n/3\alpha$ is a constant, and that $X_{(1+\epsilon)n} = f(Y)$. The function f has the Lipschitz property that if Y and Y' differ in one coordinate, then $|f(Y) - f(Y')| \leq 1$. In English, if two sets of output symbols differ in only one symbol, then the number of fully recoverable blocks that each set induces can differ by at most 1. The Lipschitz property implies that $|X_{i+1} - X_i| \leq 1$, which allows us to apply the Azuma's Inequality (see Chapter 7.2 of [1]):

Theorem 3.3 (Azuma's Inequality). *Let $c = X_0, \ldots, X_n$ be a martingale with*

$$|X_{i+1} - X_i| \leq 1$$

for all $0 \leq i < n$. Then:

$$\mathbf{Pr}\left[X_n - c > \lambda\sqrt{n}\right] < e^{-\lambda^2/2}$$

We set $\lambda = \frac{\delta\sqrt{n}}{3\alpha\sqrt{1+\epsilon}}$, $c = 2\delta n/3\alpha$ and $n = (1+\epsilon)n$ to get:

$$\mathbf{Pr}[f(Y) > \delta n/\alpha] < \exp\left(-n\,\frac{\epsilon^4}{1+\epsilon}\,\frac{\delta^2}{\ln^2 1/\delta}\,\frac{1}{72}\right) = O(e^{-n})$$

This completes the proof of the Block Perpetual Theorem.

4 Conclusions

We believe that Perpetual Codes present an interesting approach to practical erasure coding, because:

1. To the best of our knowledge, they are no other codes known that fall within an order of magnitude of the in-memory performance of the current state of the art Raptor Codes
2. They shift most of the decoding work to the pre-reception time window, unlike Raptor Codes where most of the decoding is done post-reception. In particular, the post-reception performance of Perpetual Codes is an order of magnitude better than that of our implementation of Raptor Codes
3. They are more appropriate for messages whose sizes exceed the RAM memory limit

4.1 Future Optimizations

In addition to our preliminary design, we have identified a couple of promising optimizations that could lead to significant improvements in performance. Most prominently, we believe that arithmetic over fields larger than $\mathbb{F}_2$ (in particular, $\mathbb{F}_4$ and $\mathbb{F}_8$) could have a profound impact on performance. A doubling of the field size would allow for halving the aperture size, and therefore doubling the decoding speed. This intuition is contingent on the hope that arithmetic in larger fields would not incur significant increases in symbol operation costs. Similar techniques have been investigated in the context of Raptor Codes [5], however, it is not currently clear to us that they can be leveraged as much in that case.

There is a plethora of implementational optimizations which have identified, but have not investigated in detail yet. We have observed that towards the end of the initial descend stage, there tend to be output symbols that trickle for thousands of steps only to get cancelled out at the end. This observation calls for a natural optimization whereby the decoder first checks whether a symbol will be cancalled out, and only if this is not the case, the actual symbol operations corresponding to the descent are performed. Furthermore, as mentioned earlier, a well-engineered disjoin transform could reduce the number of non-zero entries in A, and therefore the running time, by a few percent.

4.2 Error channels

Assuming Perpetual Coding proves to be a good practical choice, we believe that it would be worth applying similar techniques to error channels, like the Binary Symmetric Channel. Our intuition for that case stems once again from Block Perpetual Codes. If we use the same encoder, according to the main theorem above, each block window will be approximately equally filled with output symbols and we can use maximum likelihood decoding on each block independently. The code layer can than be backed up by a fixed-rate linear-time precode, e.g. an Expander Code [14]. This design produces a rateless code, and it is an open question whether this code will be capacity-achieving like its erasure channel counterpart. In contrast, we should mention that it has been proven that Raptor Codes cannot have the rateless property on symmetric channels [7]. It is also an interesting question to devise a decoder for the Smooth Perpetual Code for error channels.

4.3 Final Words

Our work on Perpetual Codes leaves us with two pressing questions: (a) how to analyse the smooth perpetual code, and (b) how to analyse and possibly improve the smooth perpetual precode.

References

[1] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley and Sons, Inc., 2000.

[2] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of ACM SIGCOMM*, 1998.

[3] Digital Fountain, Inc. U.S. Patent 6,486,803, November 26, 2002, On demand encoding with a window.

[4] Digital Fountain, Inc. U.S. Patent Application 20030226089, February 26, 2003, System and method for reliably communicating the content of a live data stream.

[5] Digital Fountain, Inc. U.S. Patent Application 20050219070, December 1, 2004, Protection of Data from Erasures Using Subsymbol Based Codes.

[6] Peter Elias. Coding for two noisy channels. In *The 3rd London Symposium on Information Theory*, 1955.

[7] Omid Etesami, Mehdi Molkaraie, and Amin Shokrollahi. Raptor codes on symmetric channels. *Preprint*, 2005.

[8] Robert Gallager. Low density parity check codes. *IRE Transaction on Information Theory*, 8:21–28, 1962.

[9] Michael Luby. Lt codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 271–282, 2002.

[10] Michael Luby, Michael Mitzenmacher, and Amin Shokrollahi. Analysis of random processes via and-or tree evaluation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[11] Michael Luby, Michael Mitzenmacher, Amin Shokrollahi, Daniel Spielman, and Volker Stemann. Practical loss-resilient codes. In *29th Annual ACM Symposium on Theory of Computing*, pages 150–159, 1997.

[12] Petar Maymounkov. Online codes. Technical Report TR2002-833, New York University, 2002.

[13] Amin Shokrollahi. Raptor codes. Technical Report DF2003-06-001, Digital Fountain, Inc., Fremont, CA, 2003.

[14] Michael Sipser and Daniel Spielman. Expander codes. In *IEEE Symposium on Foundations of Computer Science*, pages 566–576, 1994.

A Aperture Asymptotics

Lemma A.1. *For $\epsilon > 0$, $\delta > 0$ and $\beta = \ln(3/\delta)/\ln 2$, the inequality:*

$$\exp\left[-\frac{(\alpha\epsilon - \beta)^2}{2(1+\epsilon)\alpha}\right] \leq \delta/3 \tag{A.1}$$

resolves to:

$$\begin{aligned}\alpha \geq \frac{\ln 1/\delta}{\epsilon^2}&\left[1 + \frac{\epsilon}{\ln 1/\delta}\left(\frac{\ln 3}{\ln 2} + \ln 3\right) + \epsilon\left(\frac{1}{\ln 2} + 1\right) + \frac{\ln 3}{\ln 1/\delta}\right] + \\ &+ \frac{\ln 1/\delta}{\epsilon^2}\sqrt{(1+\epsilon)\left(\frac{2\epsilon}{\ln 2} + \epsilon + 1\right)} + \\ &+ \frac{\ln 3}{\epsilon^2}\sqrt{(1+\epsilon)\left(\frac{2\epsilon}{\ln 2} + \epsilon + 1\right)}\end{aligned}$$

Or equivalently:

$$\alpha \geq \frac{2\ln 1/\delta}{\epsilon^2}\big(1 + o_\epsilon(1)\big) \tag{A.2}$$

Proof. After taking the logarithm on both side of inequality (A.1), we have:

$$\frac{(\alpha\epsilon - \beta)^2}{2(1+\epsilon)\alpha} \geq \ln 3/\delta$$

Which unfolds to:

$$\epsilon^2\alpha^2 - 2\big(\epsilon\beta + \ln(3/\delta)(1+\epsilon)\big)\alpha + \beta^2 \geq 0$$

This is a quadratic inequality in α of the form $a\alpha^2 + b\alpha + c \geq 0$, where:

$$\begin{aligned}&0 < a = \epsilon^2\\ &0 > b = -2\big(\epsilon\beta + \ln(3/\delta)(1+\epsilon)\big)\\ &0 < c = \beta^2\end{aligned}$$

To fulfill the quadratic inequality we require that:

$$\alpha \geq -\frac{b}{2a} + \frac{\sqrt{b^2 - 4ac}}{2a}$$

And finally we get the desired result using that $\beta = \ln(3/\delta)/\ln 2$ in calculating:

$$-\frac{b}{2a} = \frac{\ln 1/\delta}{\epsilon^2}\left[1 + \frac{\epsilon}{\ln 1/\delta}\left(\frac{\ln 3}{\ln 2} + \ln 3\right) + \epsilon\left(\frac{1}{\ln 2} + 1\right) + \frac{\ln 3}{\ln 1/\delta}\right]$$

$$\frac{\sqrt{b^2 - 4ac}}{2a} = \frac{\ln 3 + \ln 1/\delta}{\epsilon^2}\sqrt{(1+\epsilon)\left(\frac{2\epsilon}{\ln 2} + \epsilon + 1\right)}$$

The asymptotic inequality (A.2) is self-evident from the above formulas. □

B The Independent Disclosure Martingale

Lemma B.1 (Independent Disclosure Martingale). *Let $Y = (Y_1, \dots, Y_m)$ be a vector of independent random variables, and let $f(Y)$ be any real-valued function. Also let $\varnothing = W_0 \subseteq W_1 \subseteq \cdots \subseteq W_n = \{Y_i\}$ be a gradation of the set of variables Y_i. Define the Independent Disclosure Martingale $X_0, \dots, X_n$ as:*

$$X_i = \mathbf{E}\left[f(Y)|W_i\right]$$

Then $X_0, \dots, X_n$ has the martingale property:

$$\mathbf{E}[X_{i+1}|X_i, \dots, X_0] = X_i$$

Furthermore, let f satisfy the Lipschitz condition, i.e. if Y and Y' differ only in $W_{i+1} \backslash W_i$ for some i, then $|f(Y) - f(Y')| \leq 1$. Then the corresponding martingale satisfies $|X_{i+1} - X_i| \leq 1$.

Note that this is a generalized version of the well-known edge and vertex exposure martingales for random graphs. (Other abstract generalizations of the exposure martingales are discussed in Chapter 7.4 of [1].) Also observe that $X_0 = \mathbf{E}[f(Y)]$ and $X_n = f(Y)$ are constants and are typically included in the martingale for convenience.

Proof. Using the indepence of the Y_i's and linearity of expectation:

$$\begin{aligned}
\mathbf{E}\left[X_{i+1}|X_i\right] &= \mathbf{E}\Big[\mathbf{E}\left[f(Y)|W_{i+1}\right]\big|W_i\Big] \\
&= \sum_{W_{i+1}\backslash W_i} \mathbf{Pr}\left[W_{i+1}\backslash W_i\right] \mathbf{E}\left[f(Y)|W_{i+1}\right] \\
&= \sum_{W_{i+1}\backslash W_i} \mathbf{Pr}\left[W_{i+1}\backslash W_i\right] \sum_{W_n\backslash W_{i+1}} \mathbf{Pr}\left[W_n\backslash W_{i+1}\right] f(Y) \\
&= \sum_{W_n\backslash W_i} \mathbf{Pr}\left[W_n\backslash W_i\right] f(Y) \\
&= \mathbf{E}\left[f(Y)|W_i\right] \\
&= X_i
\end{aligned}$$

□

Note that Y_i's do not have to be real-valued variables, and indeed we use this in our proofs. Consider, for example, a universe set U and a collection of disjoint subsets $\{A_i\}$ of U. Let $R \subset U$ be a random subset of U where each point $r \in U$ is included in R independently with some probability. Then let Y_i be the random variable that represents $R \cap A_i$, i.e. the domain of Y_i is $2^{|A_i|}$. We can then use the Y_i's to construct an Independent Disclosure Martingale that investigates some real-valued property $f(R)$ of R.