Fountain Code Schemes

An Introduction to `fountain_scheme` Library

Shenghao Yang

April 3, 2026

Abstract

This document provides an overview of fundamental fountain code schemes available in the `fountain_scheme` library (released on crates.io). These schemes are designed to be compatible with the encoding and decoding algorithms provided by the `fountain_engine` library. We discuss the design of the code schemes, including the precodes, degree sets, code parameters, and decoding configuration.

Contents

1 Introduction **2**
- 1.1 The CodeScheme Interface . . . 3
- 1.2 Encoding Pipeline . . . 4
- 1.3 Decoding Configuration . . . 4
- 1.4 Code Schemes . . . 5

2 Precodes **6**
- 2.1 LDPC Precodes . . . 7
  - 2.1.1 Formulation of LDPC Precode . . . 7
  - 2.1.2 Raptor10 LDPC . . . 8
  - 2.1.3 RaptorQ LDPC . . . 9
  - 2.1.4 Reversed LDPC Design . . . 9
- 2.2 HDPC Precodes . . . 10

2.2.1 Formulation of HDPC Precode . . . 10
2.2.2 Raptor10 HDPC . . . 11
2.2.3 RaptorQ HDPC . . . 12
2.3 Verification of Precodes . . . 13
2.3.1 LDPC Verification . . . 14
2.3.2 HDPC Verification . . . 14

3 Degree Sets **15**
3.1 Degree Distributions . . . 15
3.1.1 Degree Distribution Design Guidelines . . . 16
3.1.2 Random Sampling of Degree Distribution . . . 17
3.1.3 Deterministic Generation of Degree Values . . . 18
3.2 Sampling Degree Sets . . . 18
3.2.1 Random Approaches . . . 19
3.2.2 Deterministic Approach . . . 20
3.2.3 Degree Sets for Systematic Codes . . . 21
3.3 Degree Set Generators . . . 21

4 Code Parameters **22**
4.1 Raptor10 Code Parameters . . . 22
4.2 RaptorQ Code Parameters . . . 23

5 Decoding Configuration **24**
5.1 Maximum Number of Inactivations . . . 24
5.2 Number of Padding Vectors . . . 24
5.3 Inactivation Strategy . . . 25
5.4 Back Substitution Method . . . 25

A Constant-Weight Reflected Gray Code **26**

1 Introduction

The `fountain_engine` library (crates.io) provides a flexible framework for fountain codes, designed according to [1]. The library's encoder and decoder are implemented for a generic class of fountain codes, which can be specified by implementing the `CodeScheme` trait. An implementation of the `CodeScheme` trait is referred to as a code scheme. The `fountain_engine` library does not implement any specific fountain code scheme.

The `fountain_scheme` library (crates.io) offers components and examples for constructing fountain code schemes. Using these building blocks, both Raptor10 and RaptorQ codes can be implemented. The purpose of this document is to provide an overview of the `fountain_scheme` library and its use in developing various code schemes.

1.1 The CodeScheme Interface

The `CodeScheme` trait is shown in Listing 1. It specifies the code scheme through the following methods:

- `get_params`: generate the coding parameters,
- `code_type`: specify the code type,
- `create_degree_set_fn`: create the degree set generator,
- `create_precode`: create the precode (returns a `PrecodePair`, i.e., a tuple of optional `HDPC` and `LDPC` components),
- `decoding_config`: provide decoder options.

A code scheme has a **code type**, which can be either ordinary or systematic: ordinary codes do not include the message vectors among the coded vectors, whereas systematic codes do. The code type is specified by the `code_type` function, which returns a `CodeType` enum.

A code scheme has the following parameters:

- k: number of message vectors,
- a: number of active message vectors,
- l: number of LDPC vectors,
- h: number of HDPC vectors, and
- $b = k - a$: number of inactive message vectors.

The code parameters are specified by the `get_params` function, which returns a `CodeParams` struct.

Given a value of k, the choice of the other parameters is critical for performance and are affected by the precode and degree set design. The

`fountain_utility` library (crates.io) provides some tools to evaluate the performance of a code scheme, and hence can help with the parameter optimization. But how to optimize the choice of the parameters is beyond the scope of this document.

1.2 Encoding Pipeline

Encoding proceeds in two stages. First, a precode is applied to the k message vectors to produce $k + l + h$ variable vectors. The precode includes l LDPC (Low-Density Parity-Check) and h HDPC (High-Density Parity-Check), and the variable vectors satisfy the parity-check constraints of the precode matrices. The precode of a code scheme is specified by the `create_precode` function, which returns a `PrecodePair` value: a tuple of optional HDPC and LDPC trait objects. In §2, we discuss the precode design, with LDPC and HDPC designs in §2.1 and §2.2, respectively.

Following precoding, LT encoding forms coded vectors as linear combinations of the variable vectors. Each coded vector is the sum of a small subset of variable vectors, specified by its degree set. A degree set function is returned by the `create_degree_set_fn` function, which generates the degree set for a given coded vector ID. Degree set design is discussed in §3. Together, the precode and degree set fully define the encoding process.

1.3 Decoding Configuration

The decoding algorithm is implemented in the `fountain_engine` library [2]. The code scheme configures the decoder by specifying:

- `max_inactive_num`: the maximum number of inactive vectors,
- `num_padding`: the number of padding vectors,
- `pre_inactivation`: the type of pre-inactivation,
- `inac_strategy`: the strategy of choosing the inactive variable vectors during dynamic inactivation, and
- `subs_method`: the method of back substitution during the GE phase.

By default, only pre-inactive vectors are inactivated during the BP phase. If the `max_inactive_num` is larger than the number of pre-inactive vectors, the decoder will support dynamic inactivation.

In `fountain_engine` version 1.1 (the line used by `fountain_scheme` 1.0.1), the configurable fields of `DecodingConfig` are as described above; further engine evolution may extend this surface.

```
pub trait CodeScheme {
    fn get_params(&self) -> CodeParams;
    fn code_type(&self) -> CodeType;
    fn create_degree_set_fn(&self) -> DegreeSetFn;
    fn create_precode(&self) -> PrecodePair;
    fn decoding_config(&self) -> DecodingConfig {
        DecodingConfig {
            max_inactive_num: self.get_params().num_pre_inactive(),
            num_padding: 0,
            pre_inactivation: PreInactivation::YesPre,
            inac_strategy: InactivationStrategy::ByIndex,
            subs_method: SubstitutionMethod::Direct,
        }
    }
}
```

Listing 1: The `CodeScheme` trait (essential methods; defaults abbreviated)

1.4 Code Schemes

In addition to various precodes and degree set generators, the `fountain_scheme` library provides the following code scheme types (each implements `CodeScheme`):

- `RandomLTCode`: ordinary LT without precode (ideal soliton, robust soliton, or custom CDF).
- `LDPCLTCode`: ordinary LT with R10 parameters and `R10LDPC` only ($h = 0$ in parameters); call `as_systematic()` for `CodeType::Systematic`.
- `HDPCLTCode`: ordinary LT with RaptorQ-like parameters, `RQLDPC`, and RQ-style cyclic HDPC (`default_rq_hdpc`), plus Raptor-style degree sets; call `as_systematic()` for systematic mode.

- `BinaryHDPCLTCode`: LT with R10-style `R10LDPC` and `R10HDPC`, same degree-set family; defaults to ordinary encoding—call `as_systematic()` for systematic mode.

Systematic behavior is not separate struct types: use `as_systematic()` on `LDPCLTCode`, `HDPCLTCode`, or `BinaryHDPCLTCode` so that `code_type()` returns `Systematic`.

In the following of this document, we discuss how these code schemes are implemented, focusing on the design of precodes and degree sets. For how to use a code scheme in applications and how to assess its performance, see [2].

2 Precodes

For encoding k message vectors, the $k+l+h$ variable vectors are generated by linear combinations of the message vectors, where l vectors are LDPC vectors, h vectors are HDPC vectors. The variable vectors are assumed to be ordered as follows:

- ID 0 to $a-1$ are for active message variable vectors.
- ID a to $a+l-1$ are for LDPC variable vectors.
- ID $a+l$ to $k+l-1$ are for inactive message variable vectors.
- ID $k+l$ to $k+l+h-1$ are for HDPC variable vectors.

The total number of variable vectors is $k+l+h$, with variable vector IDs from 0 to $k+l+h-1$.

The overall parity-check matrix of the system is given by

$$H = \begin{bmatrix} S_a & I & S_b & S_h \\ D_a & D_s & D_b & I' \end{bmatrix}, \quad (1)$$

where S_a, S_b, and S_h are sparse matrices, and D_a, D_s, and D_b are dense matrices. The number of dense rows in H affects the tradeoff between computation cost and overhead.

The `create_precode` function returns a `PrecodePair`: a pair whose first element is an optional `HDPC` implementation and whose second is an optional `LDPC` implementation. If a component is `None`, that part of the precode is omitted; otherwise it is a boxed trait object implementing the corresponding trait.

2.1 LDPC Precodes

2.1.1 Formulation of LDPC Precode

An LDPC precode is specified by an $l \times a$ active parity-check matrix S_a and an $l \times (b+h)$ inactive parity-check matrix S_i. The matrix S_i has two parts: the first b columns S_b correspond to the inactive message vectors, and the last h columns S_h correspond to the HDPC vectors. The LDPC parity-check matrix S is given by

$$S = \begin{bmatrix} S_a & I_l & S_b & S_h \end{bmatrix}$$

where I_l is the $l \times l$ identity matrix.

When specifying the parity-check matrix in terms of S_a and S_i, we use the absolute row and column indices starting from 0. This convention allows the design of the precode to be independent of the vector ID conventions. Conversion from the absolute row and column indices to the vector IDs is done by the data manager in the `fountain_engine` library.

The `LDPC` trait in the `fountain_engine` library specifies the following functions to define S_a and S_i:

- `active_column(v)`: gives the (row) indices of the non-zero entries of the column v of S_a. Here v is in the range 0 to $a+l-1$, and the output is an array of indices in the range 0 to $l-1$.

- `active_row(c)`: gives the (column) indices of the non-zero entries of the row c of S_a. Here c is in the range 0 to $l-1$, and the output is an array of indices in the range 0 to $a+l-1$.

- `inactive_column(v)`: gives the (row) indices of the non-zero entries of the column v of S_i. Here v is in the range 0 to $b+h-1$, and the output is an array of indices in the range 0 to $b-1$.

- `inactive_row(c)`: gives the (column) indices of the non-zero entries of the row c of S_i. Here c is in the range 0 to $b-1$, and the output is an array of indices in the range 0 to $b+h-1$.

The default implementation of the `LDPC` trait is to return empty vectors for all the functions. This is a trivial but useful LDPC precode called `NoLDPC`. `NoLDPC` should be used when $l = 0$.

The code scheme library provides the following non-trivial implementations of LDPC precodes:

- `R10LDPC`: an LDPC precode follows the design of Raptor10 codes.
- `RQLDPC`: an LDPC precode follows the design of RaptorQ codes.
- `ReversedLDPC`: an LDPC precode with reversed order of the columns of S_a and S_i in RaptorQ design.

We introduce these implementations in the following subsections.

2.1.2 Raptor10 LDPC

We first discuss the design of the LDPC precode for Raptor10 codes in [3], where $a = k$ and S_i is the all-zero matrix. This LDPC precode is implemented as `R10LDPC`.

The $l \times a$ matrix S_a is formed by $\lceil a/l \rceil$ circulant matrices.

- The first $\lfloor a/l \rfloor$ circular matrices are square and have l columns.
- For $i = 0, 1, \dots, \lceil a/l \rceil - 1$, the first column of the ith circulant matrix has ones at positions 0, $(i + 1) \bmod l$, and $(2i + 2) \bmod l$ and zeros elsewhere.
- The other columns are cyclic shifts (down-shift) of the first column.

Each square circulant matrix is any-symmetric, i.e., symmetric w.r.t. the anti-diagonal. The following is an example of S_a with $a = 16$ and $l = 7$.

$$S_a = \left[\begin{array}{ccccccc|ccccccc|cc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]$$

To make sure that 0, $(i+1) \bmod l$, and $(2i+2) \bmod l$ are all distinct, we need $(i + 1) \not\equiv 0 \pmod{l}$ and $(2i + 2) \not\equiv 0 \pmod{l}$. Equivalently, we need $i \not\equiv l - 1 \pmod{l}$, and (when l is even) also $i \not\equiv \frac{l}{2} - 1 \pmod{l}$. Therefore, it is sufficient to require that $\lceil a/l \rceil < l/2$.

In [3], it is required that l is the smallest prime number larger than or equal to $0.01k + X$, where X is the smallest positive integer such that $X(X - 1) \geq 2k$.

2.1.3 RaptorQ LDPC

Now we give a specific design of LDPC precode as in [4]. The design of S_a is the same as the Raptor10 design, but the value of l is determined as a prime number approximately equal to $0.01k + \sqrt{2k}$. For RaptorQ codes, S_i is non-zero. This LDPC precode is implemented as `RQLDPC`.

The $l \times (b+h)$ submatrix S_i has cyclic rows, and each row consists of two consecutive ones. The following is an example of S_i with $l = 7$, $b = 2$ and $h = 4$.

$$S_i = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

S_i also has the property of anti-symmetry. In the example above, the first 6 rows of S_i are anti-symmetric. Usually, l is a multiple of $b + h$, and each square submatrix from $i(b + h)$ to $(i + 1)(b + h) - 1$ is anti-symmetric.

2.1.4 Reversed LDPC Design

It is possible to reverse the order of the columns of S_a and S_i in the previous design, so that each squared block of S_a and S_i is symmetric (w.r.t. the diagonal). The following is an example of S_a with symmetric structure:

$$S_a = \left[\begin{array}{cc|ccccccc|ccccccc} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}\right].$$

The following is an example of S_i with symmetric structure:

$$S_i = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

Note that this design does not essentially change the coding structure, but it can simplify some analyses. In the code scheme library, the reversed-column layout is implemented as `ReversedLDPC`. The `R10LDPC` type reuses that same implementation (Raptor10 is obtained with the reversed-column convention), so the matrix logic is not duplicated under two separate names.

2.2 HDPC Precodes

2.2.1 Formulation of HDPC Precode

The HDPC precode is specified by an $h \times (k+l)$ generator matrix D, so that the HDPC parity-check matrix is

$$H = [D \ I_h]$$

where I_h is the $h \times h$ identity matrix. The matrix D can be written as

$$D = [D_a \ D_s \ D_b]$$

where D_a is an $h \times a$ matrix, D_s is an $h \times l$ matrix, and D_b is an $h \times b$ matrix.

The `HDPC` trait specifies three functions: `mul_binary`, `mul_sparse_sh`, and `mul_data`.

- `mul_data`: implements the multiplication DX for the case that X is a matrix formed by data vectors. This function involves the data manager to perform the operation on vectors.
- `mul_binary`: implements the multiplication DX for the case that X is a binary matrix, but not necessary sparse. This function does not involve the data manager.

- `mul_sparse_sh`: implements the multiplication $D_s S_h$. This function does not involve the data manager.

Note that `mul_sparse_sh` can be implemented using `mul_binary`. But the specialized implementation could be more efficient.

The default implementation of the `HDPC` trait returns empty vectors for all the functions. This is a trivial but useful HDPC precode called `NoHDPC`. `NoHDPC` should be used when $h = 0$. The code scheme library provides the following non-trivial implementations of HDPC precodes:

- `R10HDPC`: an HDPC precode follows the design of Raptor10 codes.
- `GenericRQHDPC`: a generic HDPC precode follows the design of RaptorQ codes.

For standard Raptor codes [3, 4], the matrix D is of the form $\Delta\Gamma$, where Δ is an $h \times (k+l)$ binary sparse matrix, and Γ is a $(k+l) \times (k+l)$ triangular matrix.

2.2.2 Raptor10 HDPC

The matrix D of Raptor10 is formed by the weight $h' \triangleq \lceil h/2 \rceil$ subsequence of the binary reflected Gray code of h bits. Denote by $G_{h,h'}$ the $h \times \binom{h}{h'}$ matrix formed by the binary reflected Gray code of h bits with weight h'. The matrix D is formed by the first $k+l$ columns of $G_{h,h'}$. By this design, we also require that

$$k + l \leq \binom{h}{h'}. \tag{2}$$

Directly using this formula of D involves $h'(k+l)$ vector additions to calculate DX.

According to [5], the matrix D is of the form $\Delta\Gamma$, where Γ is a $(k+l) \times (k+l)$ upper-triangular matrix with all the entries are ones above the diagonal and Δ is an $h \times (k+l)$ binary matrix with all columns has degree 2 except that the first column has h' ones and $h - h'$ zeros. If Δ is known, we can use the following formula to calculate $DX = \Delta\Gamma X$:

- Let $Z = \Gamma X$, where $Z[k+l-1] = X[k+l-1]$.
- For $i = k+l-2, \ldots, 0$: $Z[i] = Z[i+1] + X[i]$.

- Calculate $DX = \Delta Z$.

We only need to store one vector but not all the vectors in Z: during the calculation, $Z[i]$ can be added to the final results based on matrix Δ. The computation cost of this code is $k + l + h$ finite field multiplications and $3k + 3l + h'$ vector additions. So this approach has lower computation cost if $h' \geq 3$ compared to the direct calculation of DX.

Now let us discuss how to enumerate the columns of Δ from the last column to the second column in the reverse order. As D is the first $k + l$ columns of $G_{h,h'}$, for $j > 1$, the j-th column of Δ is $G_{h,h'}[j] \oplus G_{h,h'}[j-1]$. An algorithm is provided in [6] to enumerate the columns of $G_{h,h'}$ and hence the columns of Δ. The last column of Δ is the $(k + l)$-th column of $G_{h,h'}$, which can be reached by the enumerating algorithm. The previous columns of Δ can be obtained by reversing the procedure in [6]. See the detailed discussion in Appendix A.

In RFC5053, S_h is all-zero and hence $D_s S_h$ is all-zero. So the `mul_sparse_sh` function in `Raptor10HDPC` just output an all-zero matrix. But our implementation of `mul_sparse_sh` supports the calculation of $D_s S_h$ when S_h is not all-zero.

The `mul_binary` function is implemented to calculate DE for decoding, where E is a binary matrix constructed from the BP decoding process. This implementation also does not assume that S_h is all-zero.

2.2.3 RaptorQ HDPC

We first discuss the design of the HDPC precode for RaptorQ codes in [4]. The $h \times (k + l)$ matrix $D = \Delta\Gamma$ is defined as follows: Γ is a $(k + l) \times (k + l)$ lower-triangular matrix:

$$\Gamma = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \alpha & 1 & \cdots & 0 \\ \alpha^2 & \alpha & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha^{k+l-1} & \alpha^{k+l-2} & \cdots & 1 \end{bmatrix},$$

where α is a primitive element of GF(256). Δ is an $h \times (k + l)$ matrix with all columns has degree 2 (with ones in two random positions) except that the last column is

$$\begin{bmatrix} 1 & \alpha & \alpha^2 \cdots \alpha^{k+l-1} \end{bmatrix}^\top.$$

As an example,

$$\Delta = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & 0 & \cdots & 1 & \alpha \\ 0 & 0 & 1 & 0 & \cdots & 1 & \alpha^2 \\ 1 & 0 & 0 & 1 & \cdots & 0 & \alpha^3 \\ 0 & 1 & 0 & 0 & \cdots & 0 & \alpha^4 \\ 0 & 0 & 1 & 0 & \cdots & 0 & \alpha^5 \\ 0 & 0 & 0 & 1 & \cdots & 0 & \alpha^6 \end{bmatrix}$$

For a sequence of vectors X, the calculation DX can be done efficiently: Let $\Gamma X = Z$. We first calculate Z and then calculate ΔZ. Calculation of Z uses the following iterative formula:

- $Z[0] = X[0]$
- For $i = 1, \ldots, k-1$: $Z[i] = \alpha Z[i-1] + X[i]$

This approach however is not in-place as we need a vector to store $Z[i]$. We only need to store one but not all the vectors in Z: during the calculation, $Z[i]$ can be added to the final results based on matrix Δ.

The computation cost of this code is $k + l + h$ finite field multiplications and $3k + 3l + h$ finite field additions.

To calculate $D_s S_h = [D_s \; D_b] \begin{bmatrix} S_h \\ 0 \end{bmatrix}$, we see that the $h \times (l + b)$ matrix $[D_s \; D_b] = \Delta' \Gamma'$ where Δ' is the last $l + b$ columns of Δ and Γ' is the $(l + b) \times (l + b)$ matrix of the same formula of Γ.

The code scheme library provides a generic implementation of RaptorQ HDPC, `GenericRQHDPC`, which is parameterized by a function `delta_column_fn`. The function `delta_column_fn` returns the two non-zero row indices for the jth column of Δ. Some examples of `delta_column_fn` are given in Listing 2. In RFC6330, the function `delta_column_fn` is implemented using a specific pseudo-random number generator, and hence can be implemented similar as `delta_column_fn_random`.

2.3 Verification of Precodes

The verification module provides functions to verify the LDPC and HDPC codes, which is available for testing the correctness of the precode implementation.

```
let delta_column_fn_default = Box::new(move |j: usize| vec![j % h, (j + 1) % h]);

let delta_column_fn_random = Box::new(move |j: usize| {
        let mut rng = StdRng::seed_from_u64(j as u64);
        let pos1 = rng.gen_range(0..h);
        let pos2 = rng.gen_range(0..h);
        vec![pos1, pos2]
    });
```

Listing 2: Examples of delta column function

2.3.1 LDPC Verification

To verify an LDPC code, we can cross validate the `active_column` and `active_row` functions, and the `inactive_column` and `inactive_row` functions. The validation module provides a function `cross_validate_ldpc` to verify the LDPC code.

```
use fountain_scheme::validation::cross_validate_ldpc;
cross_validate_ldpc(&ldpc, l);
```

The function `cross_validate_ldpc` takes two arguments: a reference to the LDPC code and the number of LDPC vectors. The LDPC code is an instance of a type implementing the `LDPC` trait, and the number of LDPC vectors is the number of LDPC vectors in the LDPC code. The function panics if the LDPC implementation is inconsistent, with a detailed error message indicating which row/column pair failed validation.

2.3.2 HDPC Verification

To verify an HDPC code, we can cross validate the `mul_binary`, `mul_sparse_sh`, and `mul_data` functions. First, we can verify that `mul_binary` and `mul_sparse_sh` are consistent in terms of calculating $D_s S_h$. Then, we can verify that `mul_binary` and `mul_data` are consistent in terms of calculating DX, where X is a binary matrix.

The validation module provides the following functions to verify an HDPC code:

- `validate_hdpc_mul_sparse`: validate the `mul_binary` and `mul_sparse_sh` functions.
- `validate_hdpc_mul_data`: validate the `mul_binary` and `mul_data` functions, where two cases are considered: X is a (k+l)-row matrix and X is an (a+l)-row matrix.
- `cross_validate_hdpc`: cross validate with the above two functions.

These functions take two arguments: a reference to the HDPC code and the coding parameters. The HDPC code is an instance of a type implementing the `HDPC` trait, and the coding parameters are the coding parameters of the HDPC code. The function panics if the HDPC implementation is inconsistent, with a detailed error message indicating which validation step failed.

3 Degree Sets

A degree set function is of type `DegreeSetFn`, which is a function that takes a coded vector ID i and returns a list of the indices of the variable vectors $\mathcal{A}(i)$. The indices of the variable vectors are in the range of 0 to $a+l+h-1$. We call $|\mathcal{A}(i)|$ the degree of the ith coded vector.

```
type DegreeSetFn = Box<dyn FnMut(usize) -> Vec<usize>>;
```

3.1 Degree Distributions

A degree distribution is a function $\Psi : \mathbb{N} \to [0,1]$ that satisfies $\sum_{d=1}^{k} \Psi(d) = 1$, where k is the maximum degree. The degree distribution is used to sample the degrees of the coded vectors, which further affect the performance of the coding scheme. A degree distribution can be specified by the degree values $(\Psi_1, \ldots, \Psi_k)$.

A degree distribution can be specified by its probability density function (PDF) or cumulative distribution function (CDF). In `fountain_scheme`, instead of floating point numbers, we use integers to represent a probability value. So the total probability is not 1, but an integer m.

- The CDF is a vector $(c_0, c_1, \ldots, c_k)$ of $k+1$ integers, where $c_d \leq c_{d+1}$ for $d = 0, 1, \ldots, k-1$, and $c_k = m$.
- The PDF is a vector $(p_0, p_1, \ldots, p_k)$ of $k+1$ integers, where $p_0 = c_0$ and $p_d = c_d - c_{d-1}$ for $d = 1, \ldots, k$. For a degree distribution, we always have $p_0 = 0$.

In `fountain_scheme`, an array of `u32` values is used to represent the PDF and CDF, we need that $m \leq 2^{32} - 1$.

3.1.1 Degree Distribution Design Guidelines

For different variations of fountain codes, different degree distribution can be applied.

Without precoding and pre-inactivation, the maximum degree should be equal to the number of message vectors k. Example degree distributions include the ideal soliton distribution and the robust soliton distribution. The ideal soliton distribution is defined as follows:

$$\rho_d = \begin{cases} \frac{1}{k} & d = 1, \\ \frac{1}{d(d-1)} & d = 2, \ldots, k. \end{cases}$$

The robust soliton distribution has two parameters, c and δ. Define

$$S = c\sqrt{k} \ln \frac{k}{\delta},$$

and

$$\tau_d = \begin{cases} \frac{S}{kd} & d = 1, 2, \ldots, k/S - 1 \\ \frac{S}{k} \ln(S/\delta) & d = k/S \\ 0 & \text{otherwise.} \end{cases}$$

The robust soliton distribution is obtained by normalizing $\rho_d + \tau_d$. In `fountain_scheme`, we provide the following functions to generate the soliton distributions:

- `ideal_soliton_cdf(m, k)`: generate the CDF of the ideal soliton distribution.
- `robust_soliton_cdf(m, k, c, delta)`: generate the CDF of the robust soliton distribution.

With precoding, the maximum degree can be a constant number that does not change with the number of message vectors. Typically, the maximum degree D is not larger than $k/(l+h)$. In this case, an asymptotically optimal degree distribution is

$$\rho_d^* = \begin{cases} 0 & d = 1, \\ \frac{1}{d(d-1)} & d = 2, \ldots, D-1, \\ \frac{1}{D} & d = D. \end{cases}$$

Though this degree distribution has a 0 probability for degree 1, inactivation decoding can be applied to trigger the BP decoding.

Without inactivation decoding, it is necessary to have a number of degree-1 coded vectors so that the BP decoding can start and continue with a high probability.

In `fountain_scheme`, we provide the following functions to generate the asymptotically optimal degree distribution:

- `asymp_optimal_cdf(m, d1, dmax)`: generate the CDF of the asymptotically optimal degree distribution. Here `d1/m` is the value of degree-1 probability, and `dmax` is the maximum degree.

The degree distributions for Raptor10 and RaptorQ are provided as follows:

- `RAPTOR10_CDF`: the CDF of the Raptor10 degree distribution.
- `RAPTORQ_CDF`: the CDF of the RaptorQ degree distribution.

3.1.2 Random Sampling of Degree Distribution

We first discuss how to sample a degree distribution to get the degree values. The first approach uses a pseudo-random number generator (PRNG).

For an m-bit CDF $(c_0, c_1, \ldots, c_k)$, we can sample a uniformly distributed random number r in $[0, 2^m - 1]$, and then find the first value of $d \geq 1$ such that $r < c_d$. This value d is the sampled degree. More degree values can be obtained by repeating the above procedure. In `fountain_scheme`, we use `sample_degree_from_cdf` to sample a degree:

```
let cdf = ideal_soliton_cdf(m, k);
let degree = sample_degree_from_cdf(&cdf, r);
```

To enable both the encoder and the decoder have the same value of degree for a coded vector, the PRNG can be initialized using the ID of the coded vector.

Both Raptor10 and RaptorQ use this approach to sample the degree distribution, where the CDF is scaled to $[0, 2^{20})$, so that the PRNG only need to generate a 20-bit integer.

3.1.3 Deterministic Generation of Degree Values

We discuss an approach to reduce the calling of PRNG using the quota method.

Suppose that the degree distribution is specified by the PDF $(\Psi_0, \Psi_1, \ldots, \Psi_k)$, where $\Psi_0 = 0$ and $\sum_{d=0}^{k} \Psi_d = m$. We first consider how to sample the degrees of the first n coded vectors. Denote by n_d the number of coded vectors of degree d among the first n coded vectors. We want to find the values of n_d that minimize $\max_d(n\Psi_d - n_d m)$ subject to $\sum_d n_d = n$. We can then use any order to assign the degree values to the coded vectors.

To achieve this goal, write $n\Psi_d = mq_d + r_d$, where q_d is the integer part and r_d is the remainder. Then, the number of coded vectors of degree d is q_d. Initially, all $n_d = q_d$. Now, we have $n\Psi_d - q_d m = r_d$. Then from the descending order of r_d, we add one to q_d to form n_d until $\sum_d n_d = n$.

If we want to further transmit $n' - n$ coded vectors, we can use the same rule to calculate n'_d, and $n'_d - n_d$ is the number of coded vectors of degree d among the additional $n' - n$ coded vectors.

Note the above approach applies to both the degrees of the active vectors and inactive vectors, though the latter is much simpler.

3.2 Sampling Degree Sets

Suppose that we have a sequence of degree values sampled. Now we discuss how to sample the corresponding vectors. The sampling of *active* vectors and *inactive* vectors are similar. To simplify the discussion, we consider the sampling problem among $0, 1, \ldots, k-1$, where n can take different values for different cases. The sampling results discussed here can be converted to the corresponding vector IDs.

3.2.1 Random Approaches

Using a pseudo-random number generator (PRNG), one common method to select d distinct values from the set $v = (0, 1, \ldots, k-1)$ is as follows. We iteratively sample an integer for each selection: on the ith round, sample an integer from $0, 1, \ldots, k-1-i$. Swap the selected element with the element at position $k-1-i$ in v. By repeating this process d times, the last d elements of v are guaranteed to be a uniformly random subset of size d with no duplicates. This approach is also known as the Fisher-Yates shuffle.

In `fountain_scheme`, the following function can be used to generate a degree set based on a sequence of sampled values:

- `sample_degree_set_with_values(k, sample)`: produces a degree set by using the input `sample`, a vector of d integers where each `sample[i]` is chosen uniformly from $0, 1, \ldots, k-1-i$. This method ensures a random selection without duplicates, following the Fisher-Yates algorithm.

To reduce the complexity when d is large, we can sample the d values with equal distance:

- generates two integers a and b, where a is uniformly distributed in $0, 1, \ldots, k-1$ and b is uniformly distributed in $1, 2, \ldots, n-1$.
- use $a + i \cdot b \bmod n$ for $i = 0, 1, \ldots, d-1$.

This is the sampling approach used by the Raptor10 and RaptorQ codes. It is possible that this equal-distance sampling may have duplications is k is a multiple of b. In the RFCs of Raptor10 and RaptorQ, a prime number k is used to avoid this issue. If the original k is not a prime number, we can use the smallest prime number $k' \geq k$.

In `fountain_scheme`, we provide the following function to sample the degree set with equal distance:

- `sample_degree_set_equal_distance(k', k, d, a, b)`: sample the degree set with equal distance.

Using the PRNG in the RFCs to generate d, a, b, this function can be used to sample the degree set of Raptor10 and RaptorQ codes.

3.2.2 Deterministic Approach

To avoid the PRNG in the random approach, we can use a deterministic approach to sample the degree set. Our goal is to sample the values in $[0, k-1]$ almost evenly.

Suppose we have n degree values $(d_1, d_2, \ldots, d_n)$ sampled as in §3.1.3. We want to sample n degree sets $(s_1, s_2, \ldots, s_n)$ such that s_i is a subset of $[0, k-1]$ and $|s_i| = d_i$. We provide a lightweight cyclic-stride sampling method which has time complexity $O(\sum_i d_i)$. Assume $k \geq 2$ and $d_i \leq k$ for all i (so no index repeats within a single s_i; see below).

Maintain integers p, a, and b. Initialize $p = 0$, $a = 0$, and $b = 1$ (so $\gcd(b, k) = 1$). For $i = 1, 2, \ldots, n$:

- Let $d = d_i$.
- Generate

$$s_i = \{(a + (p + t)b) \bmod k \mid t = 0, 1, \ldots, d-1\}.$$

- Update $p \leftarrow p + d$ (an unreduced offset; p may exceed $k - 1$).
- While $p \geq k$:
  - update $a \leftarrow a + 1 \bmod k$;
  - update $b \leftarrow b + 1 \bmod k$ until $\gcd(b, k) = 1$;
  - update $p \leftarrow p - k$.

For fixed (a, b) with $\gcd(b, k) = 1$, the map $t \mapsto (a + tb) \bmod k$ is a bijection on $t \in \{0, 1, \ldots, k-1\}$, hence a permutation of $[0, k-1]$. At step i, with current b satisfying $\gcd(b, k) = 1$ and with $d_i \leq k$, the values $(a + (p + t)b) \bmod k$ for $t = 0, \ldots, d_i - 1$ are pairwise distinct, so s_i has no duplicate indices. The (a, b) refresh after each completed lap of length k breaks strict periodicity compared with fixed (a, b) and a single moving pointer alone.

In `fountain_scheme`, we provide a struct `CyclicStrideDegreeSets` to sample the degree set with cyclic stride.

- `new(k)`: create a new `CyclicStrideDegreeSets` with the given parameters.

- `sample_set(d)`: sample a degree set with the given degree.
- `sample_sets(degrees)`: sample multiple degree sets with the given degrees.

3.2.3 Degree Sets for Systematic Codes

For systematic codes, the initial stage requires solving for all $k+l+h$ variable vectors by utilizing the $l+h$ precode (LDPC/HDPC) constraints, together with the degree sets associated with the first k coded vectors. The previously described degree set sampling methods do not inherently guarantee that this system will always be solvable.

To increase the chance that the systematic encoding produces a system that can be solved for all variable vectors, `fountain_scheme` provides a specialized triangular degree set construction [7]. Specifically, for the first a coded vectors, each degree set corresponding to the ith coded vector always contains the index i itself, as well as possibly other indices less than i. This guarantees that each message vector directly participates in its own equation and fosters triangular structure in the system, greatly increasing the likelihood that the combined message and precode constraints form a full-rank, solvable system.

It is important to clarify that this approach does not absolutely guarantee solvability in every case—certain parameter choices may still lead to an unsolvable system, in which case the systematic encoder will fail and return an error. For Raptor10 and RaptorQ codes, each value of k is specifically assigned a systematic index, ensuring that the generated degree sets provide a system of equations for the $k+l+h$ variable vectors, supported by the $l+h$ precode constraints.

3.3 Degree Set Generators

The `fountain_scheme` library provides example degree-set generators in two styles: one is parameterized by a generic PRNG; the other is fully deterministic and does not use a PRNG—for instance:

The `RaptorDegreeSetGenerator` struct implements the random approach for Raptor10 and RaptorQ codes. It is parameterized by a generic PRNG defined in the `PseudoRandom` trait. This generator can be created for ordinary

codes and systematic codes using the `new` and `new_systematic` functions, respectively.

Without using PRNG, the `CyclicStrideDegreeSetGenerator` struct implements the cyclic-stride construction.

4 Code Parameters

A code has the following parameters:

- k: number of message vectors,
- a: number of active message vectors,
- l: number of LDPC vectors,
- h: number of HDPC vectors, and

In `fountain_engine`, `CodeParams::new(k, a, l, h)` builds a parameter set with explicit active count a, then derives $b = k - a$ and $i = b + h$. For non-pre-inactivation settings, `CodeParams::new_without_pre_inact(k, l, h)` is used instead; it fixes $a = k$, $b = 0$, and $i = 0$. For the basic LT code, we have $k = a$ and $l = h = 0$. For Raptor codes, we have $l + h > 0$. In the following, we discuss the design of the code parameters for Raptor codes.

4.1 Raptor10 Code Parameters

As Raptor10 code does not use pre-inactivation, all the message vectors are active, i.e., $a = k$.

Following the discussion of LDPC precode of Raptor10 code in §2.1.2, using $l(l-1) \geq 2k$ can ensure the LDPC precode has column weight 3. In [3], it is required that l is the smallest prime number larger than or equal to $0.01k + x$, where x is the smallest positive integer such that $x(x-1) \geq 2k$.

Following the discussion of HDPC precode of Raptor10 code in §2.2.2, (2) is required for k, l, and h. In [3], h is the smallest integer satisfying (2).

In `fountain_scheme`, we provide the following function to generate the Raptor10 code parameters:

- `generate_r10_parameters(k)`: generate the Raptor10 code parameters.

Given a value of k, the function builds a `CodeParams` via `CodeParams::new_without_pre_inact(k, l, h)`, so $a = k$, $b = 0$, and $i = 0$ as required when pre-inactivation is not used.

4.2 RaptorQ Code Parameters

RaptorQ codes utilize pre-inactivation, introducing inactive message vectors. This means in general $a \leq k$, where a is the number of active message vectors and k is the total number of message vectors. According to [4], the parameters a, l (LDPC), and h (HDPC) are provided for each supported k in the standard's parameter table. As observed in [5], l is typically chosen as a prime near $0.01k + \sqrt{2k}$.

In `fountain_scheme`, we provide a function that generates RaptorQ-inspired code parameters based on the core rules from [4], but without strictly following the parameter table:

- `generate_rq_like_parameters(k)`: generate RaptorQ-like code parameters for any integer k.

The implementation returns `CodeParams::new(k, a, l, h)` with the computed (a, l, h), so $i = b + h$ where $b = k - a$, matching pre-inactivation in the parameter struct.

Unlike the RFC, this function accepts arbitrary k values, not just those listed in the official table. When called, it computes a `CodeParams` struct using the following procedures:

- The HDPC parameter h is set as follows:
  - $h = 10$ for $k = 60$
  - $h = 16$ for $k = 56403$
  - For other k, h is interpolated linearly on a logarithmic scale:

$$h = 10 + 6 \cdot \frac{\log k - \log 60}{\log 56403 - \log 60}$$

- The active variable count a is chosen so that $a + l$ is prime, with:
  - For $k < 60$, a is nearly k;
  - For $k \geq 60$, a is approximately $k - 1.57\sqrt{k}$.

5 Decoding Configuration

During decoding, `fountain_engine` [2] recovers the variable vectors from the received coded vectors by a two-stage process: a belief-propagation (BP) phase that exploits the sparse parity structure, followed when needed by a Gaussian-elimination (GE) phase on a reduced system involving inactive variables. Inactive variables are those treated explicitly in the GE step—initially this set can include the inactive message indices, the HDPC variables under pre-inactivation, and any variables chosen for dynamic inactivation when BP alone does not converge.

The maximum size of that inactive set, how padding is accounted for, whether HDPC rows are pre-inactivated, how dynamic candidates are ordered, and how GE back-substitution is performed are all exposed as fields of `DecodingConfig`.

5.1 Maximum Number of Inactivations

The field `max_inactive_num` caps how many variable vectors may be **inactive** at once: the pre-inactive part (inactive message indices together with HDPC variables when pre-inactivation is enabled) plus any variables moved to inactive status by **dynamic inactivation** while BP is still running. The decoder may continue inactivate more variables only while the inactive count stays at or below this limit.

The computation cost of the GE phase is related to the number of inactive variables, and hence `max_inactive_num` also affects the computation cost of the GE phase. Typically, if no dynamic inactivation is used, the `max_inactive_num` is set to the number of pre-inactive variables, i.e., $b + h$ for RaptorQ and 0 for Raptor10. If full dynamic inactivation is used, the `max_inactive_num` is set to the number of variable vectors, i.e., $k + l + h$.

5.2 Number of Padding Vectors

RFC6330 provides RaptorQ code parameters for a discrete set of k values (up to 56403), presented in a table. If the target k is not listed, the RFC prescribes selecting parameters from the smallest table entry $k' \geq k$. In this case, encoding treats the additional $k' - k$ message symbols as all-zero padding vectors, which are not transmitted.

When encoding, `fountain_engine` does not need to explicitly handle these padding vectors. On the decoding side, however, these padding positions can act as LDPC constraints for recovering inactive variables. The field `num_padding` specifies how many such padding vectors are available for the decoder to use. Ideally, to recover all k' message symbols, k' coded vectors would be required.

At present, the `num_padding` field is effectively ignored in `fountain_engine` v1.1, which behaves as if `num_padding` is set to 0. Nevertheless, it will decode all required message symbols (producing zeros in the padded positions), using at least k coded vectors.

5.3 Inactivation Strategy

Inactivation strategy is used to choose the inactive variable vectors during the BP phase. The `fountain_engine` library v1.1 only supports the `ByIndex` strategy that uses the first active variable vectors as the inactive variable vectors.

5.4 Back Substitution Method

After solving the inactive variables, the back substitution method is used to substitute the inactive variables into the involved decoded vectors in the GE phase. The `fountain_engine` library v1.1 only supports the `Direct` method that solves the inactive variables directly using the GE-reduced matrix.

A Constant-Weight Reflected Gray Code

Here we discuss the algorithm for enumerating the columns of a binary constant-weight reflected Gray code. Our algorithm here slightly improves the Bitner-Ehrlich-Reingold (BER) algorithm [6] by allowing enumerating the columns in the reverse order from a specific column number. The idea is simple:

- Using the existing BER algorithm enumerate forwardly to the specified column number.
- Then enumerate backwardly from the specified column number to the first column.

To enumerate backwardly, we modify the iterative formula of the BER algorithm.

Denote by $G_{n,k}$ the $n \times \binom{n}{k}$ matrix formed by the binary constant-weight reflected Gray code. For two special cases, we have the formulas: $G_{i,0}$ be a column vector with i zeros, and $G_{i,i}$ be a column vector with i ones. The general iterative formula is:

$$G_{i,k} = \begin{bmatrix} G_{i-1,k} & G^R_{i-1,k-1} \\ 0 & 1 \end{bmatrix}, \tag{3}$$

where G^R is the reverse of matrix G in terms of columns.

This iterative formula can be visualized as a tree structure, where the root is $G_{n,k}$ and the leaves are either $G_{i,0}$ or $G_{i,i}$. Figure 1 illustrates this with $G_{5,2}$ as an example. At each level, every vertex in the tree represents a matrix $G_{i,j}$ or its reverse $G^R_{i,j}$, called a regular node and a reverse node, respectively. We use the term "vertex" loosely, as the same matrix $G_{i,j}$ may appear multiple times in different parts of the tree. However, this notational flexibility does not cause confusion in the implementation. Each edge is labeled with a corresponding bit. In this tree, each leaf node represents a column of the binary constant-weight reflected Gray code, determined by the sequence of bits along the edges from the root to that leaf.

Now we discuss the algorithm for enumerating the columns of $G_{n,k}$. We first discuss the forward enumeration algorithm, and then the backward enumeration algorithm. Initially, the first column $g = [1, 1, \ldots, 1, 0, 0, \ldots, 0]^\top$, where the first k entries are ones and the remaining $n - k$ entries are zeros.

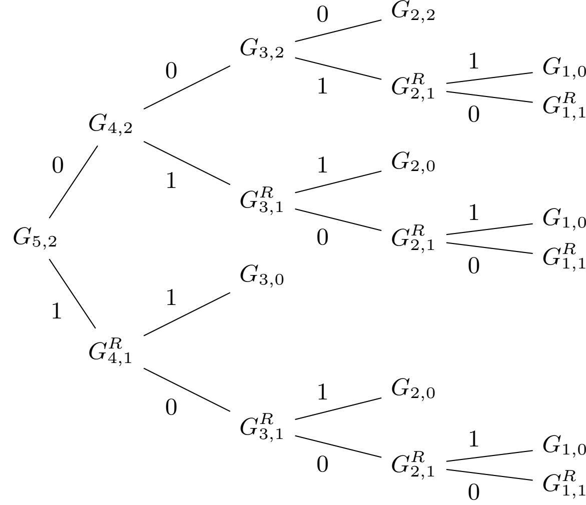

Figure 1: The tree structure of $G_{5,2}$.

A vector τ is initialized to $\tau[i] = i + 1$ for $i = 1, \ldots, n - 1$. The meaning of $\tau[i]$ for $i > 0$ is the next level to visit in the tree after the ith level. $\tau[0]$ is the current level to visit. We have another parameter t to store the weight number of the vertex to visit. Initially, $\tau[0] = k$ and $t = k$. See the `init` function in Listing 3 for the detailed initialization.

```
1  def init(n, k):
2      g = [1]*n + [0]*(n-k)    # first column of G_{n,k}
3      # forward enumeration
4      tau[i] = i+1, for i = 1,2,...,n-1
5      t = k
6      tau[0] = k
7      # backward enumeration
8      tau_r[i] = i+1, for i = 1,2,...,n-1
9      t_r = k
10     tau_r[0] = n
```

Listing 3: Initialization of the column enumeration algorithm for $G_{n,k}$.

Before going to the next column, we first keep the current level $i = \tau[0]$ fixed. For forward enumeration, we will visit the vertex $G_{i,t}$ and update the

path to use its sibling R node, and hence visit the next column. Forward enumeration only visits regular nodes. Even though there may be multiple $G_{i,t}$ in the tree, their updates are the same. Therefore, we do not need to distinguish them. The first vertex to visit is $G_{k,k}$. There are three things to do for moving forward:

- First, update $\tau[0]$ to the next level to visit, and $\tau[i]$ to its initial value.
- Second, update g. Two positions of g are flipped: Depending on the value of $g[i]$ and t, another position is determined, and flipped. Then, flip $g[i]$.
- Update t to the weight number t' of the sibling node $G^R_{i,t'}$ of $G_{i,t}$, which depends on the value of $g[i]$.
- Last, update the next vertex to visit, which is a descendant of $G^R_{i,t'}$, which affects t and τ.

See the `next_column` function in Listing 4 for the detailed implementation.

The forward enumeration algorithm has been explained in detail in [6]. But for our application, we need to enumerate the columns in the reverse order from a specific column number. In the following, we discuss how to enhance the algorithm to enumerate the columns in the reverse order.

For backward enumeration, we need new variables t_r and τ_r to store the weight number and the next level to visit in the tree after the ith level, respectively. Initially, $\tau_r[0] = n$ and $t_r = k$. See the `init` function in Listing 3 for the detailed initialization. The forward enumeration also affects t_r and τ_r. As forward enumeration moves from a visited regular node to its sibling reverse node, the backward enumeration reverses the operations. Therefore, the next vertex to visit in backward enumeration is the sibling of the current vertex in forward enumeration. See the `previous_column` function in Listing 5 for the detailed implementation.

The backward enumeration algorithm is almost the same as the forward enumeration, except that only the reversed nodes are visited. See Listing 5 for the detailed implementation. Compared to the forward enumeration, the only difference is in Line 12.

```
def next_column():
    i = tau[0]
    # update tau[0] and tau[i]
    tau[0] = tau[i]
    tau[i] = i+1
    # update g and t
    if g[i] == 1:
        if t > 0: g[t-1] = not g[t-1]
        else: g[i-1] = not g[i-1]
        t = t + 1
    else:
        if t > 1: g[t-2] = not g[t-2]
        else: g[i-1] = not g[i-1]
        t = t - 1
    g[i] = not g[i]
    # update backward enumeration
    t_r = t
    if i < tau_r[0]:
        tau_r[i] = tau_r[0]
    else if i < tau_r[tau_r[0]]:
        tau_r[i] = tau_r[tau_r[0]]
    tau_r[0] = i
    # update the next vertex to visit
    if t == i || t == 0:
        t = t + 1
    else:
        t = t - g[i-1]
        tau[i-1] = tau[0]
        if t == 0: tau[0] = i-1
        else: tau[0] = t
```

Listing 4: Forward enumeration of the columns of $G_{n,k}$.

```
def previous_column():
    i = tau_r[0]
    # update tau_r[0] and tau_r[i]
    tau_r[0] = tau_r[i]
    tau_r[i] = i+1
    # update g and t_r
    if g[i] == 1:
        if t_r > 0: g[t_r-1] = not g[t_r-1]
        else: g[i-1] = not g[i-1]
        t_r = t_r + 1
    else:
        if t_r > 1: g[t_r-2] = not g[t_r-2]
        else: g[i-1] = not g[i-1]
        t_r = t_r - 1
    g[i] = not g[i]
    # update forward enumeration
    t = t_r
    if i < tau[0]:
        tau[i] = tau[0]
    else if i < tau[i-1]:
        tau[i] = tau[i-1]
    tau[0] = i
    # update the next vertex to visit
    if t_r == i || t_r == 0:
        t_r = t_r + 1
    else:
        t_r = t_r - g[i-1]
        tau_r[i-1] = tau_r[0]
        if t_r == 0: tau_r[0] = i-1
        else: tau_r[0] = t_r
```

Listing 5: Backward enumeration of the columns of $G_{n,k}$.

References

[1] S. Yang, "Fountain codes: Designs and algorithms," 2025, under development.

[2] ——, "Fountain engine: A rust library for fountain codes," 2025. [Online]. Available: https://shhyang.github.io/fountain_docs/docs/doc-engine.pdf

[3] A. Shokrollahi, T. Stockhammer, M. Luby, and M. Watson, "Raptor Forward Error Correction Scheme for Object Delivery," RFC 5053, Oct. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc5053

[4] L. Minder, A. Shokrollahi, M. Watson, M. Luby, and T. Stockhammer, "RaptorQ Forward Error Correction Scheme for Object Delivery," RFC 6330, Aug. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6330

[5] A. Shokrollahi and M. Luby, *Raptor Codes*, ser. Foundations and Trends in Communications and Information Theory. now, 2011, vol. 6.

[6] J. R. Bitner, G. Ehrlich, and E. M. Reingold, "Efficient generation of the binary reflected gray code and its applications," *Communications of the ACM*, vol. 19, no. 9, pp. 517–521, 1976.

[7] L. Mao, S. Yang, X. Huang, and Y. Dong, "Design and analysis of systematic batched network codes," *Entropy*, vol. 25, no. 7, p. 1055, 2023.