Fountain Engine: A User Guide

Shenghao Yang

March 6, 2026

About this Document

This document introduces the Fountain Engine, a generic framework for fountain codes. The engine enables flexible implementation of various fountain code designs through a code scheme interface, and it is independent of any specific data operation implementation. The encoder and decoder provided by the engine generate the operations that need to be performed on data using an abstract data operation format. Users of the engine have complete freedom and control over how to carry out these operations. This document describes how to use the engine, including details on the code scheme interface, the data operation format, and the data operator interface. A companion Fountain Utility library is also introduced for code scheme validation and performance analysis.

This document applies to fountain$_{\text{engine}}$ and fountain$_{\text{utility}}$ as published on crates.io at version 1.0.x.

Contents

1 Introduction

Fountain codes are a class of erasure codes capable of generating an unlimited number of coded vectors from a given set of message vectors. Both LT codes and Raptor codes are subclasses of fountain codes [1, 2]. Raptor 10 [3] and RaptorQ [4] are two standardized Raptor codes by the IETF that demonstrate various techniques used in fountain code design.

Fountain codes are particularly well-suited for network communications, where they provide reliability without requiring feedback mechanisms, flexible erasure correction capability, and low computational complexity. They have been adopted in several practical applications: 3GPP and DVB-H utilize Raptor 10 for mobile wireless broadcasting and multicasting; the ANDL protocol employs a Reliable Large Datagram Protocol (RLDP) [5] based on RaptorQ [4]; and the Librecast Project [6] implements IPv6 multicast using RaptorQ. Raptor codes have been implemented in numerous open source projects across multiple languages, including C++ [7], Java [8], and Rust [9, 10].

This document introduces *fountain engine*, henceforth referred to as *the engine*, a generic implementation framework for fountain code encoding and decoding. The engine is different from the existing implementations of fountain code mentioned above due to its generic nature, which is elaborated in two aspects.

The first generic aspect of the engine is that it does not implement any specific fountain code scheme. Instead, it provides a framework that can encode and decode any fountain code specified through the code scheme interface. In contrast to the existing implementations of fountain codes that support only one specific fountain code design, the engine is much more flexible and can be used to implement various fountain code designs. This design benefits from the common encoding and decoding algorithm of fountain codes.

The second generic aspect of the engine is its independence from any specific data operation implementation. Fountain code encoding and decoding involve various data operations, including data padding, blocking, and vector operations such as scalar-vector multiplication and vector addition. The engine does not perform these data operations, but only specifies the operations that need to be performed on data vectors, assuming that the original data has already been properly padded and separated into equal-length data vectors. The encoding and decoding algorithms are vector operations performed on the data vectors, such as "add vector x to vector y" or "multiply vector z by a scalar" based on vector IDs "x, y, z". Users of the engine can freely use their own data operator to perform the operations.

The design of the engine is described in [11]. In this document, we introduce the use of the engine implemented in Rust.

2 Basic Usage

The `fountain engine` is implemented as a Rust library. To use the library, you need to add it as a dependency to your Rust project.

```toml
[dependencies]
fountain_engine = "1.0"
```

Remember to replace 1.0 with the version number of the engine you want to use.

In this section, we introduce a basic use case of the engine, which does not need a data operator. This use case can help to verify the correctness of a code scheme and evaluate the computation and memory cost by analyzing the operations generated by the engine using the `fountain_utility` library to be introduced in Section 6.

2.1 Encoding

The encoding process is implemented by the `Encoder` struct. To create an encoder, call the `new` function, which takes a coding scheme as argument.

```rust
use fountain_engine::Encoder;
let encoder = Encoder::new(code_scheme);
```

The `code_scheme` is an implementation of the `CodeScheme` trait, which defines the interface for a code scheme. We will discuss the code scheme in Section 3.

After the encoder is created, the encoding process can be started by calling the `encode_coded_vector` function. The function can be called multiple times for different coded vector IDs. The function returns `Option<usize>`: if the `coded_id` is valid, `Some(data_id)` is returned, where `data_id` is the ID of the data vector for the coded vector; otherwise `None` is returned. The valid range of coded vector IDs will be discussed in Section 4.

```rust
if let Some(data_id) = encoder.encode_coded_vector(coded_id) {
    // Use the data_id
}
```

After performing encoding, the operations generated by the encoder can be obtained by calling the `get_operations` function of the `manager` field. The operations can be executed by a data operator to generate the coded vectors. See Section 5 for the details about the data operator.

```rust
let operations = encoder.manager.get_operations();
```

2.2 Decoding

The decoding process is implemented by the `Decoder` struct. To create a decoder, call the `new` function, which takes a coding scheme as argument. The `code_scheme` should be the same as the one used to create the corresponding encoder.

```rust
use fountain_engine::Decoder;
let decoder = Decoder::new(code_scheme);
```

After the decoder is created, the decoding process can be started by calling either `add_coded_id` or `add_coded_vector` function, both of which return the decoding status.

The `add_coded_id` function is used for delayed execution, where the coded vector data is not provided immediately. The coded vector data will be provided later through the operations generated by the decoder.

```
let status = decoder.add_coded_id(coded_id);
```

The `add_coded_vector` function is used for on-the-fly execution, where the coded vector data is provided immediately.

```
let status = decoder.add_coded_vector(coded_id, vector);
```

The `coded_id` is the ID of the coded vector to be decoded, and it should be the same as the one used to encode the coded vector. The `status` is a `DecodeStatus` enum, which can be `Decoded` or `NotDecoded`. If `NotDecoded`, more coded vectors can be added to the decoder.

The current decoding status can also be checked without adding a new coded vector by calling the `decode_status` function:

```
let status = decoder.decode_status();
```

The operations generated by the decoder can be obtained by calling the `get_operations` function of the `manager` field. The operations can be executed by a data operator to generate the decoded message vectors.

```
let operations = decoder.manager.get_operations();
```

3 Coding Scheme

A fountain code scheme consists of the following components: code parameters, a code type, a degree set generator, a precode generator, and decoding configuration. These components should be implemented following the

```rust
use fountain_engine::traits::{CodeScheme, HDPC, LDPC};
use fountain_engine::types::{CodeParams, CodeType};

struct MyCodeScheme {
    k: usize,
}

impl MyCodeScheme {
    fn new(k: usize) -> Self {
        Self { k }
    }
}

impl CodeScheme for MyCodeScheme {
    fn get_params(&self) -> CodeParams {
        CodeParams::new(self.k, self.k, 0, 0)
    }
    fn code_type(&self) -> CodeType {
        CodeType::Ordinary
    }
    fn create_degree_set_fn(&self) ->
    Box<dyn FnMut(usize) -> (Vec<usize>, Vec<usize>)> {
        let k = self.k;
        Box::new(move |coded_id| ((0..k).collect::<Vec<usize>>(), vec![]))
    }
    fn create_precode(&self) ->
    (Option<Box<dyn HDPC>>, Option<Box<dyn LDPC>>) {
        (None, None)
    }
}
```

Listing 1: Example of a code scheme implementation

`CodeScheme` trait. Listing 1 shows an example of a code scheme implementation. Using this code scheme implementation, a code scheme for k message vectors can be created by calling the `new` function.

The code scheme example in Listing 1 is not an effective one. Some non-trivial code schemes are provided in the `fountain_scheme` library [12].

In the remainder of this section, we first describe the general model of the engine used for encoding and decoding, and then discuss the components of a code scheme.

3.1 Fountain Code Model

The engine performs the fountain code encoding and decoding by processing vectors over a vector space over GF(256), each of which is represented by a sequence of 8-bit integers. The vector length does not necessarily need to be known by the engine, but is assumed to be the same for all the vectors.

Suppose that the encoder operates on k message vectors, where k is a positive integer. The encoding has two steps: precoding and LT encoding. The precoding generates $k + l + h$ variable vectors by linear combinations of the message vectors, where l vectors are LDPC vectors, h vectors are HDPC vectors, and the remaining k vectors are called base vectors. The base vectors are the message vectors for the systematic encoding, but not for the ordinary encoding.

The variable vectors are then separated into two groups: active vectors and inactive vectors:

- The first a base vectors, and all the LDPC vectors are active vectors.
- The remaining $b = k - a$ base vectors and all the HDPC vectors are inactive vectors.

The LT encoding generates coded vectors by linear combinations of the variable vectors. Each coded vector is the sum of only a small subset of the variable vectors, which is called the degree set of the coded vector. The degree set of a coded vector is required for both encoding and decoding.

3.2 Code Scheme Interface

The code scheme interface is defined by the `CodeScheme` trait. The trait defines the following functions:

- `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,
- `max_inactive_num`: specify the maximum number of inactive vectors, which has the default implementation as $h + b$.

3.2.1 Coding Parameters

A code scheme can generate the coding parameters by the `get_params` function, which returns a `CodeParams` struct. The `CodeParams` struct stores 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.

When implementing the `get_params` function, the return value is created by the `CodeParams::new` function.

```rust
let params = CodeParams::new(k, a, l, h);
```

3.2.2 Code Type

The `code_type` function returns a `CodeType` enum.

```rust
pub enum CodeType {
    Ordinary,
    Systematic,
}
```

`Ordinary` and `Systematic` are used for the ordinary fountain code and the systematic fountain code, respectively. The ordinary fountain code does not include the message vectors in the coded vectors, while the systematic fountain code includes the message vectors in the coded vectors.

3.2.3 Degree Set Generator

The `create_degree_set_fn` function returns a function `degree_set` of type `DegreeSetFn`.

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

The function `degree_set` generates the degree set for a given coded vector ID. The function returns two lists: the indices of the active variable vectors with distinct values from 0 to $a+l-1$ and the indices of the inactive variable vectors with distinct values from 0 to $b+h-1$.

3.2.4 Precode Generator

The precode generator is a function that generates the precode, which consists of an LDPC code and an HDPC code. The `create_precode` function returns two options. Both the LDPC and HDPC codes can be `None`, which means no LDPC or HDPC code is used. If not `None`, the LDPC code is an instance of the `LDPC` trait, and the HDPC code is an instance of the `HDPC` trait.

3.3 Precode Traits

3.3.1 LDPC Precode

The LDPC code is defined by the parity-check matrix

$$\begin{bmatrix} S_a & I & S_i \end{bmatrix}$$

where I is the $l \times l$ identity matrix, S_a is the $l \times a$ active parity-check matrix, and S_i is the $l \times (b+h)$ inactive parity-check matrix. All the variable vectors must satisfy the parity-check constraints. While S_a and S_i do not have to be sparse, they must be binary matrices.

The `LDPC` trait specifies four functions to define the parity-check matrix.

- The `active_column` function returns the indices of the non-zero entries of a column of S_a.
- The `active_row` function returns the indices of the non-zero entries of a row of S_a.
- The `inactive_column` function returns the indices of the non-zero entries of a column of S_i.
- The `inactive_row` function returns the indices of the non-zero entries of a row of S_i.

```
fn active_column(&self, var_col: usize) -> Vec<usize>;
fn active_row(&self, check_row: usize) -> Vec<usize>;
fn inactive_column(&self, var_col: usize) -> Vec<usize>;
fn inactive_row(&self, check_row: usize) -> Vec<usize>;
```

3.3.2 HDPC Precode

The HDPC code is specified by the parity-check matrix

$$\begin{bmatrix} D & I \end{bmatrix}$$

where D is the $h \times (k+l)$ generator matrix, and I is the $h \times h$ identity matrix. All the variable vectors must satisfy the parity-check constraints. Here D is a dense matrix, and may not be binary. 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.

To implement the HDPC precode, the `HDPC` trait specifies three functions: `mul_binary`, `mul_sparse_sh`, and `mul_data`.

The `mul_binary` function implements the multiplication DX for the case that X is a binary matrix. The `mul_binary` function takes the following arguments:

- `params`: the coding parameters,
- `n`: the number of columns of X, which is the number of inactive variables.

- `v`: a function that returns the a row of X specified by the input argument.

The output is a $h \times n$ matrix DX.

```
fn mul_sparse(&self,
    params: &CodeParams,
    n: usize,
    s: &dyn Fn(usize) -> Vec<usize>) -> Vec<Vec<u8>>;
```

The `mul_sparse_sh` function implements the multiplication $D_s S_h$, where S_h is the last h columns of S_i. The `mul_sparse_sh` function takes the following arguments:

- `params`: the coding parameters,
- `s`: a function that returns the indices of the non-zero entries of a row of S_h.

The output is a $h \times h$ matrix $D_s S_h$.

```
fn mul_sparse_sh(&self,
    params: &CodeParams,
    s: &dyn Fn(usize) -> Vec<usize>) -> Vec<Vec<u8>>;
```

The `mul_data` function implements the multiplication DX for the case that X is a matrix formed by data vectors. As this function is applied to the data vectors, it uses the data manager and the data vector IDs to perform the operations. The `mul_data` function takes the following arguments:

- `manager`: the data manager, which will be introduced in Section 4.
- `params`: the coding parameters.
- `x_ids`: the indices of the vectors forming the matrix X. When the length of `x_ids` is less than $k + l$, the extra rows are assumed to be all-zero vectors.
- `y_ids`: the indices of the vectors forming the matrix $Y = DX$.

```
fn mul_data(&self,
    manager: &mut DataManager,
    params: &CodeParams,
    x_ids: &[usize],
    y_ids: &[usize]);
```

3.4 Decoding Configuration

The coding scheme described above applies to both encoding and decoding.

For decoding, one additional configuration is about inactivation. Note that the order of solving variable vectors during the BP phase and the solving of the inactive variables in the GE phase do not affect the final decoding result.

One parameter is to control the maximum number of inactive variables, including both the pre-inactive and dynamically inactivated variables. This parameter affects the computation cost of the GE phase: The GE phase adopts Gaussian elimination to solve the inactive variables, which has a higher computation cost than the BP phase.

It is also possible to adjust the strategy of choosing the inactive variable vectors during the BP phase. The current implementation chooses the inactive variable vectors in the order of the variable vectors. This strategy is efficient for systematic coding, especially when the number of erased variable vectors is small. Optimizing the strategy can further reduce the total number of inactive variables and hence reduce the computation cost of the GE phase.

4 Data Manager

The fountain engine specifies **what** tasks the encoder and decoder must accomplish (for example, that a coded vector should be the linear combination of certain message vectors), but it does not dictate **how** these tasks are executed. To model these actions abstractly, the fountain engine defines a set of operations, each of which includes a name and a list of data vector IDs it involves.

The encoder and decoder use the data manager to manage the data vector IDs and the associated operations, which can be accessed by the `manager` field of the encoder/decoder. In practice, users only need to interact with the data manager when implementing the HDPC precode. However, understanding how data vector IDs are allocated, as well as how the data manager records operations, is important for using the engine correctly and efficiently.

4.1 Types of Vectors

The engine uses data vector IDs to refer to the data vectors. The data vector IDs range is shared by message vectors, variable vectors generated by precoding, and coded vectors generated by the LT encoding.

4.1.1 Message Vectors

Message vectors are used as the encoding input and decoding output, and are indexed from 0 to $k-1$, where k is the number of message vectors. The following rules of the data vector IDs for message vectors are used:

- Before encoding, the message vectors have data vector IDs from 0 to $k-1$.
- After decoding, the decoded message vectors have data vector IDs from 0 to $k-1$.

Due to this rule, data vector IDs for message vectors are not explicitly shown in the encoder and decoder interfaces.

4.1.2 Variable Vectors

Variable vectors are generated by precoding, and are indexed from 0 to $k+l+h-1$, where $k+l+h$ is the total number of variable vectors. The first $a+l$ variable vectors are the active variable vectors, and the remaining $b+h$ variable vectors are the inactive variable vectors.

The data manager maintains a consecutive range of data vector IDs for the $k+l+h$ variable vectors. By knowing the initial data vector ID for the variable vector 0, all the other data vector IDs for the variable vectors can be determined.

The data vector IDs for the variable vectors are assigned according to the code type:

- For ordinary code `CodeType::Ordinary`, the data vector IDs for the variable vectors are from 0 to $k + l + h - 1$.
- For systematic code `CodeType::Systematic`, the data vector IDs for the variable vectors are from k to $2k + l + h - 1$.

4.1.3 Coded Vectors

Coded vectors include the message vectors for systematic encoding, LDPC and HDPC constraints and LT encoded vectors. We adopt the following convention of coded vector IDs:

- IDs from 0 to $k - 1$ are for message vectors, which are the first k coded vectors for systematic encoding. Ordinary encoding does not use these coded vector IDs.
- IDs from k to $k + l - 1$ are for LDPC constraints.
- IDs from $k + l$ to $k + l + h - 1$ are for HDPC constraints.
- IDs from $k + l + h$ are for LT encoded coded vectors.

The maximum coded vector ID should be known from the design.

LDPC and HDPC constraints are always the all-zero vectors and hence do not need to be transmitted. The other coded vector IDs should be accompanied with the coded IDs for transmission.

4.2 Usage of Coded Vector IDs

4.2.1 Encoding

Now we can explain the input and output of the `encode_coded_vector` function of `Encoder` struct with more details:

- if `coded_id < k` and the code type is `CodeType::Systematic`, the encoder returns `Some(coded_id)`, which is the data vector ID for the message vector with ID `coded_id`.
- if `coded_id < k` and the code type is `CodeType::Ordinary`, the encoder returns `None`, as message vectors are not used as coded vectors in ordinary encoding.

- if `k <= coded_id < k+l+h`, the encoder returns `None`, as LDPC and HDPC constraints are not encoded as separate coded vectors (they are always all-zero vectors).
- if `coded_id >= k+l+h`, the encoder returns `Some(data_id)`, where `data_id` is the data vector ID for the LT encoded coded vector with ID `coded_id`.

4.2.2 Decoding

For systematic decoding, the input coded vector IDs can be either less than `k` or greater than or equal to `k+l+h`. For ordinary decoding, the input coded vector IDs should be greater than or equal to `k+l+h`. The coded IDs from `k` to `k+l+h-1` are used internally by the decoder for LDPC and HDPC constraints, and should not be used when calling the `add_coded_vector` function and the `add_coded_id` function of `Decoder` struct.

4.3 Management of Data IDs

The data manager maintains many functions to manage the data vector IDs. These functions are most used internally by the engine, and are not needed by the users.

For encoding a coded vector with ID `coded_id`, in the `encode_coded_vector` function of `Encoder` struct, the `coded_data_id` function of the data manager is used to get the data vector ID for the coded vector. In addition to assigning a data vector ID for the coded vector, the `coded_data_id` function also performs a data vector operation to ensure that the data vector is an all-zero vector.

For decoding a coded vector with ID `coded_id`, in the `add_coded_id` function of `Decoder` struct, the `insert_coded_id` function of the data manager is used to get the data vector ID for the coded vector. The `insert_coded_id` function does not perform any data vector operation.

One data ID function that may be used by the users is the `temp_data_id` function. This function is used by the engine to get a temporary data vector ID for a data vector generated by the engine, but not used by the users directly. The `temp_data_id` function may be used in the implementation of the HDPC trait `mul_data` function.

4.4 Data Operation Interfaces

The encoder and decoder perform the data vector operations through the interface provided by the data manager. The data manager does not implement these operations, but saves them. The saved operations can be obtained by the `get_operations` function of the data manager. We discuss these operations provided by the data manager in the following two categories:

- Data vector operations, and
- Informative operations.

4.4.1 Data Vector Operations

The following data vector operations are supported by the data manager:

- `ensure_zero(list_id)`: ensure the vectors indexed in `list_id` are all zero vectors.
- `add_to_vector(list_id,id)`: add the vectors indexed in `list_id` to the vector `id`.
- `broadcast_add(id,list_id)`: add a vector `id` to all the vectors in `list_id`.
- `multiply_alpha(id)`: multiply the vector `id` by the primitive element α.
- `multiply_scalar(scalar,id)`: multiply the vector `id` by an 8-bit scalar, represent an element in GF(256).
- `divide_scalar(scalar,id)`: divide the vector `id` by an 8-bit scalar, represent an element in GF(256).
- `multiply_add(src_id,scalar,target_id)`: multiply the vector `src_id` by an 8-bit scalar, represent an element in GF(256), and add the result to the vector `target_id`.
- `move_to(src_id,target_id)`: move the data vector with `src_id` to the vector with `target_id`. After moving, the original data vector becomes an all-zero vector.

- `copy_to(src_id,target_id)`: copy the vector with `src_id` to the vector with `target_id`.
- `remove(id)`: remove the vector with `id`.

Users of the engine do not need to directly use these operations. For code scheme implementation, the `mul_data` function in the `HDPC` trait uses the data operation interfaces.

4.4.2 Informative Operation

The data manager also provides one informative operation:

- `InfoCodedVector(coded_id,data_id)`: the data vector with `data_id` corresponds to the coded vector with `coded_id`.

For encoding, the `InfoCodedVector` operation is used to tell the data operator that a coded vector has been generated. For decoding, the `InfoCodedVector` operation is used to tell the data operator that a coded vector should be added.

The data manager provides two functions that generate this operation:

- `add_coded_vector(coded_id,data_id)`: used for decoding to add a coded vector.
- `encode_coded_vector(coded_id,data_id)`: used for encoding to indicate a coded vector has been generated.

Both functions create the same `InfoCodedVector` operation.

4.4.3 Extract Operations

For an encoder or a decoder, the operations generated can be extracted via the data manager in the `manager` field, which provides the following functions for managing the operations:

- `get_operations`: return a reference to the saved operations by the data manager.
- `clear_operations`: clear all the saved operations.

- `move_new_operations`: return a vector of the new operations generated since the last extraction, but do not change the operations saved in the data manager.

```rust
impl DataManager {
    pub fn get_operations(&self) -> &[Operation];
    pub fn clear_operations(&mut self);
    pub fn move_new_operations(&mut self) -> Vec<Operation>;
}
```

5 Data Operators

Various data operators can be defined for the engine. We will discuss the implementation of a data operator according to several typical use cases. Before discussing the implementation of a data operator, we first introduce the `Operation` enum, which is required by all data operators.

5.1 Operation Enum

The `Operation` enum is defined as follows:

```rust
pub enum Operation {
    EnsureZero { list_id: Vec<usize> },
    MultiplyAlpha { id: usize },
    MultiplyScalar { scalar: u8, id: usize },
    AddToVector { list_id: Vec<usize>, target_id: usize },
    BroadcastAdd { src_id: usize, target_ids: Vec<usize> },
    MulAdd { src_id: usize, scalar: u8, target_id: usize },
    MoveTo { src_id: usize, target_id: usize },
    CopyTo { src_id: usize, target_id: usize },
    Remove { id: usize },
    InfoCodedVector { coded_id: usize, data_id: usize },
}
```

All these operations correspond to the data vector operations provided by the data manager. Note that though the data manager provides the divide_scalar operation, it is not included in the Operation enum. This is because the division by a scalar is converted to other operations internally by the data manager.

5.2 Delayed Data Operation

Following the basic use case in Section 2, we can implement a data operator to execute the operations generated by the engine. For this purpose, the data operator does not need to follow any specific interface, but only needs to execute the operations generated by the engine.

For delayed execution, the data operator can implement any interface to execute the operations. If implementing the DataOperator trait (see Section 5), the execute function takes a single &Operation, so operations need to be executed one at a time. For example, a data operator might have:

- insert_vector(vector: &[u8], data_id: usize): insert the data vector vector into the data operator with data ID data_id.
- execute(op: &Operation): execute a single operation op.

Note that when executing multiple operations, you need to iterate over them and call execute for each one.

5.2.1 Encoding Operation

Before executing any operations, the data_operator should prepare the message vectors for encoding, which are the first k data vectors. The preparation of the message vectors is not included in the operations extracted from the encoder, but is assumed by the encoder. Initialization of the message vectors can be executed even after calling the encoder, but should be before calling the execute function of the data operator.

```rust
for i in 0..k {
    data_operator.insert_vector(&message_vectors[i], i);
}
```

After creating the encoder by the new function, the operations of precoding are already generated and can be executed by the data operator.

```rust
let encoder = Encoder::new(code_scheme);
let operations = encoder.manager.move_new_operations();
for op in operations {
    data_operator.execute(&op);
}
```

To perform encoding, the `encode_coded_vector` function is called for each desired coded vector ID. The function `move_new_operations` of the data manager can be called to retrieve the new operations generated since the last extraction, which can be passed to the data operator for execution.

You have flexibility in how you interleave encoding and execution. For example, you may process operations one vector at a time:

```rust
for coded_id in coded_ids {
    encoder.encode_coded_vector(coded_id);
    let operations = encoder.manager.move_new_operations();
    for op in operations {
        data_operator.execute(&op);
    }
}
```

Alternatively, you may encode all desired vectors first, and then execute all pending operations together:

```rust
let encoder = Encoder::new(code_scheme);
for coded_id in coded_ids {
    encoder.encode_coded_vector(coded_id);
}
for i in 0..k {
    data_operator.insert_vector(&message_vectors[i], i);
}
let operations = encoder.manager.get_operations();
// optionally clear the operations after extraction
encoder.manager.clear_operations();
for op in operations {
    data_operator.execute(op);
}
```

The informational operation `InfoCodedVector` tells the data operator that a coded vector has been generated.

5.2.2 Decoding Operation

After creating the decoder by the `new` function, the operations of initializing some all-zero vectors are generated and can be executed by the data operator.

```
let decoder = Decoder::new(code_scheme);
let operations = decoder.manager.move_new_operations();
for op in operations {
    data_operator.execute(&op);
}
```

To perform decoding, a sequence of coded vectors are added to the decoder by the `add_coded_id` function. The operations of adding the coded vectors can be retrieved by the `move_new_operations` function of the data manager and executed by the data operator.

```
for coded_id in coded_ids {
    let status = decoder.add_coded_id(coded_id);
    let operations = decoder.manager.move_new_operations();
    for op in operations {
        data_operator.execute(&op);
    }
    if status == DecodeStatus::Decoded {
        break;
    }
}
```

Alternatively, all the operations can be retrieved by the `manager.get_operations` function of the `manager` field and executed by the data operator after the decoding is complete.

```
let decoder = Decoder::new(code_scheme);
for coded_id in coded_ids {
    let status = decoder.add_coded_id(coded_id);
    if status == DecodeStatus::Decoded {
        break;
    }
}
let operations = decoder.manager.get_operations();
// optionally clear the operations after extraction
decoder.manager.clear_operations();
for op in operations {
    data_operator.execute(op);
}
```

It is not necessary to explicitly call the `insert_vector` function to add coded vectors to the data operator, as the `operations` retrieved by the `get_operations` function of the `manager` field already contain the informative operations `InfoCodedVector` to tell the data operator when a coded vector should be added.

5.2.3 Optimization of Data Operation Execution

Though the operations are generated in order, they are not necessary to be executed in the same order. Each operation has one or multiple data ID affected, and some data IDs used (no matter affected or not). We have the following rules:

- Two operations A and B must be executed in the same order as they are generated if a data ID affected by one operation is referred to by another operation.
- A consecutive sequence of operations can be executed in parallel (or any order) if any data ID affected by an operation is not used by all other operations in the sequence.

In delayed data operation execution, the operations retrieved can be optimized before execution. How to optimize the execution depends on the specific data operator implementation, and is not discussed in this guide.

5.3 On-the-fly Data Operation Retrieval

The engine also provides an interface for on-the-fly retrieval and execution of the operations. For this purpose, the data operator needs to implement the DataOperator trait, and create encoder and decoder with such a data operator.

```
pub trait DataOperator {
    fn insert_vector(&mut self, vector: &[u8], data_id: usize);
    fn get_vector(&self, data_id: usize) -> &[u8];
    fn execute(&mut self, op: &Operation);
}
```

The default implementation of get_vector returns an empty slice. Data operators used only for capturing or logging operations (e.g. DisplayDataOperator) can rely on this default; operators that store and retrieve vectors must implement get_vector to return the stored data.

5.3.1 Encoder and Decoder with Data Operator

The encoder and decoder can be created with a data operator by calling the new_with_operator function.

```
let encoder = Encoder::new_with_operator(
        code_scheme,
        Box::new(data_operator));
let decoder = Decoder::new_with_operator(
        code_scheme,
        Box::new(data_operator));
```

In Rust, the above operation moves the ownership of the data operator into the encoder and decoder. The data operator can be retrieved by the move_operator function of the data manager for other data vector operations, and then set back to the data manager for on-the-fly data operation execution.

Note that move_operator will panic if no operator has been set. It is safe to call move_operator after creating an encoder or decoder with new_with_operator.

```
let data_operator = encoder.manager.move_operator();
```

```rust
encoder.manager.set_operator(data_operator);
let data_operator = decoder.manager.move_operator();
decoder.manager.set_operator(data_operator);
```

5.3.2 Display Data Operator

Here we illustrate a simple example of data operator that prints the data operations on the console. This data operator only needs to implement the `execute` function of the `DataOperator` trait, and leaves the other functions with the default implementation.

```rust
struct DisplayDataOperator;
impl DataOperator for DisplayDataOperator {
    fn execute(&mut self, op: &Operation) {
        println!("operation: {:#?}", op);
    }
}
```

In the above example, the `execute` function prints the operation on the console. The other two functions of the `DataOperator` trait use the default implementation. In other words, this is a trivial data operator that exposes the operations. The exposed operations can be captured and executed by other programs.

After creating the encoder and decoder with the data operator, call the `encode_coded_vector` and `add_coded_id` functions to generate the operations, and the operations will be printed on the console.

```rust
// Encoding
for coded_id in coded_ids {
    encoder.encode_coded_vector(coded_id);
}
// Decoding
for coded_id in coded_ids {
    decoder.add_coded_id(coded_id);
}
```

See Section 6.1.1 for a more flexible IO data operator provided in the utility library.

5.4 On-the-fly Data Operation Execution

Finally, we illustrate how to execute the operations on the fly. This approach is useful when testing the code engine but is not recommended for production use as the user loses the flexibility of executing the operations in the desired order. It is recommended to use the on-the-fly data operation retrieval instead to achieve the same purpose.

A data operator for on-the-fly data operation execution needs to implement the `insert_vector` and `get_vector` functions of the `DataOperator` trait nontrivially. We use the `VecDataOperater` struct as an example, which is provided in the utility library.

Before creating the encoder, initialize the data operator with the message vectors. Then create the encoder with the data operator.

```
let data_vector_length = 4; // length of each data vector in bytes
let mut data_operator = VecDataOperater::new(data_vector_length);
for i in 0..k {
    data_operator.insert_vector(&message_vectors[i], i);
}
let encoder = Encoder::new_with_operator(
        code_scheme,
        Box::new(data_operator));
```

Then call the `encode_coded_vector` function to generate encoded vectors, which can be retrieved by the `get_data_vector` function of the `manager` field of the encoder.

To decode the encoded vectors, call the `add_coded_vector` function to add the coded vectors to the decoder.

```
for coded_id in coded_ids {
    if let Some(data_id) = encoder.encode_coded_vector(coded_id) {
        let data_vector = encoder.manager.get_data_vector(data_id);
        decoder.add_coded_vector(coded_id, data_vector);
    }
}
```

6 Fountain Utility

Accompanying the code engine, we provide the `fountain_utility` library for fountain code performance evaluation and testing. The utility library includes the following modules:

- two data operators: `IoDataOperator` and `VecDataOperater`, and
- a testing module `code_testing` for testing fountain code schemes.
- a statistics module `testing_statistics` for collecting and analyzing statistics from multiple test runs of the same code scheme.

To use the utility library, you need to add it as a dependency to your Rust project.

```
[dependencies]
fountain_utility = "1.0.0"  # change the version number
```

6.1 Data Operators

A couple simple data operators are provided in the `fountain_utility` library for code scheme validation and performance evaluation.

- `IoDataOperator` is a data operator that writes the operations to an I/O stream.
- `VecDataOperater` is a data operator that stores the data vectors in a vector.

6.1.1 IoDataOperator

The `IoDataOperator` is a data operator that writes the operations to an I/O stream. It can be customized to write the operations to a file or a console for further processing. To use the `IoDataOperator`, add the following code to your Rust program:

```
use fountain_utility::IoDataOperator;
```

The following code shows how to use the `IoDataOperator` to write the operations to the console, as `DisplayDataOperator`.

```rust
use std::io::stdout;

let mut io_operator = IoDataOperator::new(stdout());
let encoder = Encoder::new_with_operator(code_scheme,
        Box::new(io_operator));
```

The following code shows how to use the `IoDataOperator` to write the operations to a file:

```rust
use std::fs::File;
let file = File::create("operations.log")?;
let mut file_logger = IoDataOperator::new(file);
let encoder = Encoder::new_with_operator(code_scheme,
        Box::new(file_logger));
```

The following code shows how to use the `IoDataOperator` to write the operations to a memory buffer:

```rust
use std::io::Cursor;
let mut buffer = Vec::new();
let mut test_logger = IoDataOperator::new(Cursor::new(&mut buffer));
let encoder = Encoder::new_with_operator(code_scheme,
    Box::new(test_logger));
```

6.1.2 VecDataOperater

The `VecDataOperater` is a data operator that stores the data vectors in a vector, and performs the data vector operations on the vectors. It is used for testing the correctness of the code engine and a code scheme for data vector operations. See Section 5.4 for an example of using the `VecDataOperater`.

6.2 Performance Metrics

The basic performance metrics for a code scheme include

1. the encoding and decoding computational costs,
2. the encoding and decoding storage costs, and
3. the decoding success probability or overhead.

The overall computational cost includes the execution time of the fountain code engine and the data operation execution time. The former can be directed measured when executing the engine. Here we mainly care about the data operation computational costs. Instead of directly measuring the data operation execution time during execution, we count the operations transferred to the data manager, which provides an implementation-independent way to measure the computational costs.

6.2.1 `PerformanceMetrics` Struct

The performance metrics can be counted by the `PerformanceMetrics` struct in the `fountain_utility` library. The structure provides the following fields:

- `multiply_alpha`: the number of scalar-vector multiplication operations, where the scalar is the primitive element α.
- `multiply_scalar`: the number of scalar-vector multiplication operations, where the scalar can be any 8-bit scalar.
- `vector_add`: the number of vector addition operations.
- `mul_add`: the number of operations that multiply a vector by an 8-bit scalar, and add the result to another vector.
- `max_storage`: the maximum number of data vectors stored in the data operator.
- `num_coded_vectors`: the number of coded vectors generated or added to the decoder.

These fields can be counted by the `from_operations` function of the `PerformanceMetrics` struct.

```
let metrics = PerformanceMetrics::from_operations(&ops);
```

In the following, we discuss how the performance of a code scheme is evaluated by the code engine.

6.2.2 Computational Costs

We classify the computational costs into the following categories: `multiply_alpha`, `multiply_scalar`, `vector_add`, and `mul_add`. The following operations in the `Operation` enum involve the computational costs:

- `MultiplyAlpha(id)`: one `multiply_alpha` operation.
- `MultiplyScalar(scalar,id)`: one `multiply_scalar` operation.
- `AddToVector(list_id,target_id)`: size of `list_id` number of `vector_add` operations.
- `BroadcastAdd(src_id,target_ids)`: size of `target_ids` number of `vector_add` operations.
- `MulAdd(src_id,scalar,target_id)`: one `mul_add` operation.

Here we count scalar division the same as scalar-vector multiplication, as the former can be implemented by first inverting the scalar and then calling scalar-vector multiplication. Actually, to reduce the complexity of implementing the data operations, scalar division should not be used. Only the code engine performs scalar division.

Scalar-vector multiplication with a general scalar and α, the primitive element, are counted separately as the latter is a special case of the former that can be implemented simpler. Some operations of HDPC are designed specifically using scalar-vector multiplication with α.

One `MulAdd` operation can be counted as one `multiply_scalar` operation and one `vector_add` operation, but that is not the optimal implementation. This approach needs two passes of the vector. A better one is to implement `MulAdd` as a single operation so that one pass is enough.

6.2.3 Decoding Overhead

The minimum number of LT coded vectors required for decoding is k. The decoding overhead o is the number of LT coded vectors used for decoding subtracting k.

The decoding overhead can be derived based on the informative operation `InfoCodedVector` generated by the decoder. The total number of `InfoCodedVector` operations is counted as the `num_coded_vectors` field of the `PerformanceMetrics` struct, from which subtracting k gives the decoding overhead.

6.2.4 Storage Costs

The following operations in the `Operation` enum affect the storage:

- `EnsureZero(list_id)`: initialize and set the list of vectors to all-zero, and generate `list_id` number of new data vectors.
- `MoveTo(src_id,target_id)`: move the vector from the source data vector ID to the target data vector ID, and do not generate any new data vectors.
- `CopyTo(src_id,target_id)`: copy the vector from the source data vector ID to the target data vector ID. It generates one new data vector.
- `Remove(id)`: remove the vector with the data vector ID, and do not generate any new data vectors. It removes one data vector.

The maximum number of data vectors stored in the data operator is counted as the `max_storage` field of the `PerformanceMetrics` struct.

We discuss the storage costs for different types of encoding and decoding, and show that the storage costs can be derived from the coding parameters and the decoding overhead. This discussion can help us to verify the storage cost calculation of a code scheme.

For ordinary encoding, the k message vectors must be stored in the data operator, and they are used to generate the extra $l + h$ variable vectors. Note that during the HDPC calculation, one temporary data vector is used, which is removed after the HDPC calculation. Therefore, after precoding, the storage costs are $k+l+h$ vectors. The storage cost for each coded vector generated is 1 vector.

For systematic encoding, the k message vectors must be stored in the data operator, and they are duplicated for the $k+l+h$ variable vectors. Therefore, after precoding, the storage costs are $2k+l+h$ vectors. However, the duplication operations `CopyTo` of the message vectors can be executed as `MoveTo` in a data operator if the message vectors are not used anymore. For this implementation, the storage costs are $k+l+h$ vectors.

For ordinary decoding, l LDPC coded vectors are added to the decoder initially, and then LT encoded coded vectors are added one by one. The minimum number of LT coded vectors required for decoding is k. If decoding is successful in the BP phase, the storage cost is $k+l+o$ vectors, where o is the decoding overhead. If decoding goes into the GE phase, h HDPC coded vectors are added to the decoder, and then the storage cost is $k+l+h+o$ vectors.

For systematic decoding, if an LT coded vector has coded ID less than k, it is the original message vector, and the coded vector is duplicated. Therefore, the storage cost is $2k+l+h+o$ vectors.

6.3 Code Testing

6.3.1 Testing Templates

The utility library provides a testing module `code_testing` for testing fountain code schemes. The module provides two testing templates and a helper function:

- `test_code_scheme`: tests a code scheme without actual data vector operations.
- `test_code_scheme_with_data_vectors`: tests a code scheme with actual data vector operations, using the `VecDataOperater` data operator.
- `test_code_scheme_multiple`: runs multiple tests and collects the results.

```
pub fn test_code_scheme<C>(
    code_scheme: &C,
    k: usize,
    num_coded_vectors: usize
    ) -> TestResult
where
    C: CodeScheme + Clone;

pub fn test_code_scheme_with_data_vectors<C>(
    code_scheme: &C,
    k: usize,
    data_vector_length: usize,
    num_coded_vectors: usize
    ) -> TestResult
where
    C: CodeScheme + Clone;

pub fn test_code_scheme_multiple<C>(
    num_runs: usize,
    code_scheme: &C,
    k: usize,
    num_coded_vectors: usize
    ) -> Vec<TestResult>
where
    C: CodeScheme + Clone;
```

Both test templates return a `TestResult` struct that contains the performance metrics and the test result. The `TestResult` struct provides the following fields:

- `k`: the number of source symbols.
- `num_mismatches`: the number of mismatches between decoded vectors and message vectors.
- `precoding_metrics`: the performance metrics of the precoding operations.
- `encoding_metrics`: the performance metrics of the encoding operations.
- `decoding_metrics`: the performance metrics of the decoding operations.

- `precoding_time_ms`: the time taken for precoding in milliseconds.
- `encoding_time_ms`: the time taken for encoding in milliseconds.
- `decoding_time_ms`: the time taken for decoding in milliseconds.

Test results can be saved to and loaded from files using JSON Lines format:

```rust
// Save results to a file
save_test_results(&results, "results.jsonl")?;

// Load results from a file
let loaded_results = load_test_results("results.jsonl")?;
```

6.3.2 Testing Statistics

A coding scheme needs to be tested multiple times to get a reliable result. As packet loss is simulated by randomly shuffling the coded vectors, the test result is not the same. The following code shows how to test a code scheme multiple times and collect the testing results.

```rust
let results = test_code_scheme_multiple(
    num_runs,
    &code_scheme,
    k,
    num_coded_vectors);
```

Alternatively, you can manually collect results:

```rust
let results: Vec<TestResult> = (0..num_runs).map(|_i| {
        test_code_scheme(&code_scheme, k, num_coded_vectors)
    }).collect();
```

All the testing results have the same value of `k` and `num_coded_vectors`. The `TestStatistics` struct provides static methods to get the statistics from the test results:

- `success_rate(results)`: returns `(num_runs, successful_runs, success_rate)` tuple.
- `overhead_stats(k, results)`: returns `Statistics` with mean, min, and max of the decoding overhead.
- `storage_stats(results)`: returns `Statistics` with mean, min, and max of the storage costs.
- `avg_computation_costs(k,results)`: returns a tuple of `(precoding_avg_comp, encoding_avg_comp, decoding_avg_comp)` with average computational costs.
- `avg_time_costs(results)`: returns `AverageTime` with average time taken for precoding, encoding, and decoding when executing the code engine.

References

[1] M. Luby, "LT codes," in *Foundations of Computer Science (FOCS), 2002. Proceedings. The 43rd Annual IEEE Symposium on*, Nov. 2002, pp. 271–280.

[2] A. Shokrollahi, "Raptor codes," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2551–2567, June 2006.

[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] TON Foundation, "Reliable large datagram protocol (rldp)," TON Documentation, 2024, tON blockchain network protocol documentation. [Online]. Available: https://docs.ton.org/v3/documentation/network/protocols/rldp

[6] Librecast Team, "Librecast project," Open Source Project, 2024, iPv6 multicast implementation based on RaptorQ. [Online]. Available: https://codeberg.org/librecast

[7] L. Fulchir, "libraptorq: C++11 implementation of rfc6330," Open Source Project, 2016, header-only C++11 library implementing RFC6330 RaptorQ with C/C++98 API support. Features deterministic builds, LZ4 compression for cached precomputations, and multi-language bindings. [Online]. Available: https://github.com/LucaFulchir/libRaptorQ

[8] OpenRQ Team, "Openrq: Java implementation of raptorq," Open Source Project, 2024, java implementation of RaptorQ forward error correction. [Online]. Available: https://openrq-team.github.io/openrq/

[9] Y. Poirier, "A rust library for implementing forward error correction (fec) using raptor codes," Crates.io Package, 2025, rust implementation of RFC5053 Raptor 10 forward error correction. [Online]. Available: https://crates.io/crates/raptor-code

[10] C. Berner, "Raptorq: Rust implementation," Crates.io Package, 2024, rust implementation of RFC6330 RaptorQ forward error correction. [Online]. Available: https://crates.io/crates/raptorq

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

[12] ——, "Basic fountain code schemes: An introduction to fountain_scheme library," 2025, under development.