FRI protocol

Karthik Inbasekar[a]

E-mail: karthik@ingonyama.com

Abstract: FRI API definitions

1 Summary of current requirements for ICICLE

In this section, we summarize some of the low level stuff that needs to be in place for the FRI API to work. FRI protocol is relevant for the following small fields $\mathbb{F}_p$.

- Baby Bear (32 bits) - implemented in icicle
- m31 (32 bits) - implemented in icicle
- Koala Bear (32 bits) (Only used in Plonky3)
- Polar Bear (40 bits) and Teddy Bear (32 bits) [1]
- Goldilocks (64 bits)

We also need to support the k'th extension fields $\mathbb{F}_{p^k}$ with k limbs. The most relevant extension fields are

- Baby Bear: $\mathbb{F}_{p^4}$ Quartic extension - commonly used, $\mathbb{F}_{p^5}$ Quintic extension - not commonly used.
- M31: cubicm31Plonky3, cm31, qm31. $Cm31$ and $qm31$ are the common use cases.

Other extension fields for Koala bear, Polar bear, Goldilocks are only relevant if we decide to support the base fields. Comparison targets are

- RISC0 FRI, Plonky3 FRI for baby bear field and $cm31$.
- STWO FRI and Plonky3 FRI for m31 circle group. (this imposes circle constraints, conditons at NTT etc)

Before we go deep into algorithms etc. The following operations are missing in ICICLE and are critical for FRI definition. We denote a vector of size n containing field elements as $\mathbb{F}_p^n$, a vector of size n containing k limb extension fields as $\mathbb{F}_{p^k}^n$.

- Field API

  - base field , extension field arithmetic, where $a \in \mathbb{F}_p$, and $c, r \in \mathbb{F}_{p^k}$

$$\begin{aligned} r &= a \pm c \\ r &= a \cdot c \end{aligned} \tag{1.1}$$

  - extension field - extension field arithmetic where $c_1, c_2, r \in \mathbb{F}_p^k$

$$\begin{aligned} r &= c_1 \pm c_2 \\ r &= c_1 \cdot c_2 \end{aligned} \tag{1.2}$$

- Vecops
  - base field , extension field arithmetic, where $\vec{a} \in \mathbb{F}_p^n$, $e \in \mathbb{F}_p$ and $\vec{c}, \vec{r} \in \mathbb{F}_{p^k}^n$

$$\begin{aligned} \vec{r} &= \vec{a} \pm \vec{c} \\ \vec{r} &= \vec{a} \cdot e \end{aligned} \tag{1.3}$$

  - extension field - extension field arithmetic, where $\vec{c}_1, \vec{c}_2, \vec{r} \in \mathbb{F}_{p^k}^n$

$$\begin{aligned} \vec{r} &= \vec{c}_1 \pm \vec{c}_2 \\ \vec{r} &= \vec{c}_1 \cdot \vec{c}_2 \end{aligned} \tag{1.4}$$

- Hashes requirements in Fiat Shamir: Bitwise hashes (Keccak, Blake2s/3s), finite field hashes (poseidon2) to be potentially used in Fiat shamir. see §3.5
  - The first use case is in generating randomness from merkle roots with a hashing structure

```
entry1 = [commit_label[u8]||entry0|| roots[0].LE32()]
```

  - When the hash output is in bytes, we should safely encode the result to base/extension fields to be used in the protocol. Currently this seems easy to do with all bitwise hashes, since they have a natural sponge mode.
  - So far the use case for Poseeidon/Poseidon2 is questionable due to efficiency. Since we do support $3 \to 1$ poseidon/poseidon2, this is still doable, subject to safe byte to field/extension field conversions.
- Merkle Tree: Keccak, Blake2s/3s,Poseidon2,Poseidon used in Merkle trees of different arities. Vectors that need to be Merkle committed logarithmically decrease in size by one of the constant folding factors : $2, 4, 8, 16$ inside the FRI protocol. (see §3.2)
  - Merkle trees can have leaf elements of type base field $\mathbb{F}_p$ or extension field $\mathbb{F}_p^k$. Take this into account while determining leaf element sizes.
  - The variations of Merkle tree that we support are applicable. Including, pruning, and a efficient storage optimization by storing fri layers in bit reversed form §4.1

  – If we support batched trees, then we can also use batched FRI efficiently. We can still do this by wrapping around existing implementations. §4.3, as the FRI protocol (algorithm) itself is unaffected by this.

- Uniformly random sampler for query phase.

2 FRI background *minimal* :)

Reed Solomon Codewords: Consider a univariate polynomial of degree d, defined by $P(x) = \sum_{i=0}^{d} a_i \cdot x^i$, where $a_i \in \mathbb{F}_p$ or $\mathbb{F}_{p^k}$. A RS code $RS(P(x), \mathcal{L}, n)$ is a mapping into domain $\mathcal{L}$ of size $n = 2^l$ (for $l \in \mathbb{Z}$) given by $\{P(1), P(\alpha), P(\alpha^2) \dots, P(\alpha^{n-1})\}$ such that $d < n$. [1] For FRI, we are only interested in the case where

$$d = 2^k - 1 \tag{2.1}$$

we will assume this in the rest of the note. Furthermore in all our cases $\mathcal{L} \in \mathbb{F}^*$ (a coset of the multiplicative subgroup of $\mathbb{F}$), this is known as a smooth RS code. What this means is that $\alpha = \omega$ is a nth root of unity. [2] The rate of a RS code is defined as

$$\rho = \frac{d+1}{n} = \frac{1}{2^{l-k}} \ , \ l \geq k \in \mathbb{Z} \tag{2.2}$$

The rate is the relative size of the codeword w.r.t to the coefficients, typical values are $\rho = 1/2, 1/4, 1/8, 1/16$. Sometimes $\mathcal{R} = \rho^{-1}$ is called a blow up factor, the size of our target starting domain $|D| = \mathcal{R} \cdot (d+1)$.

For our purposes, A RS code word in short is a set of evaluations of a low degree polynomial in a root of unity domain $\mathcal{L}$. For FRI the folding algorithm uses the property that a random linear combination of a RS code is a RS code in the same space.

$$RS(P_1(x), \mathcal{L}, n) + \gamma \cdot RS(P_2(x), \mathcal{L}, n) \equiv RS(P_3(x), \mathcal{L}, n) \tag{2.3}$$

where $x \in \mathcal{L}$ and $\gamma \in \mathbb{F}_p$ or $\mathbb{F}_{p^k}$. This is easy to understand since that we can write $P_{eval} = NTT(P_{coeff})$ and NTT is a linear operation that allows scalar multiplication.

2.1 FRI quick walkthrough

The goal of FRI is to check if a given codeword of length $n = 2^k$ corresponds to a low degree polynomial of degree $d < n$.

The prover claim: I know a low degree polynomial of $F(x)$ degree d, I will give you oracle access to a code word $RS(F(x), \mathcal{L}, n)$.

The prover consists of two phases, a commit phase, and a query phase. In the commit phase, the prover recursively reduces the claim that he knows a low degree polynomial $F(x)$ of degree d corresponding to a codeword $RS(F(x), \mathcal{L}, n)$ to a claim that he knows low degree polynomial $F'(x)$ of degree $\frac{d+1}{2} - 1 = \frac{d-1}{2}$ corresponding to a codeword $RS(F'(x), \mathcal{L}^2, n/2)$

[1]The code is error correcting upto $\frac{d-n}{2}$ symbols if d is even, upto $\frac{d-n-1}{2}$ symbols if d is odd.

[2]This can be extended to MLE, but we do not consider it in this iteration.

using a folding algorithm. This process continues untill the desired final length of the codeword is reached.

The example described below follows the FRI paper [2] and a close implementation/blog is [3].

2.1.1 Folding in two

Below, we explain a one round folding process for a folding factor of two. [2]. Given a polynomial

$$P_0(X) = [P_0(1), P_0(\omega), \ldots, P_0(\omega^{n-1})] \tag{2.4}$$

in a domain

$$\mathcal{L} = \{1, \omega, \omega^2, \ldots \omega^{n-1}\} \tag{2.5}$$

The prover first commits(Merkle/MMCS) the polynomial,

$$root_0 \leftarrow Commit(P_0(x)) \tag{2.6}$$

and generates a random parameter $\beta \leftarrow hash(root_0)$. Where β is necessarily in an extension field $\mathbb{F}_{p^k}$. We then have the FRI folding function

$$P_1(\beta, X^2) = P_{0,even}(X^2) + \beta \cdot P_{0,odd}(X^2) \tag{2.7}$$

where

$$\begin{aligned} P_{0,even}(X^2) &= \frac{P_0(X) + P_0(-X)}{2} \\ P_{0,odd}(X^2) &= \frac{P_0(X) - P_0(-X)}{2X} \end{aligned} \tag{2.8}$$

Note that the folded poly P_1 is defined in the Domain $\mathcal{L}^2$

$$\mathcal{L}^2 = \{1, \omega^2, \omega^4, \ldots \omega^{2n-2}\} \tag{2.9}$$

P_1 can be computed either in coefficient or evaluation form (most practical FRI evaluations have a folding in evaluation form). It is much easier to express as

$$\begin{aligned} P_{0,even}(\omega^{2i})\Big|_{i=0}^{n/2-1} &= \frac{P_0(\omega^i) + P_0(-\omega^i)}{2}\Big|_{i=0}^{n/2-1} \\ P_{0,odd}(\omega^{2i})\Big|_{i=0}^{n/2-1} &= \frac{P_0(\omega^i) - P_0(-\omega^i)}{2\omega^i}\Big|_{i=0}^{n/2-1} \end{aligned} \tag{2.10}$$

and

$$P_1(\beta, \omega^{2i})\Big|_{i=0}^{n/2-1} = P_{0,even}(\omega^{2i})\Big|_{i=0}^{n/2-1} + \beta \cdot P_{0,odd}(\omega^{2i})\Big|_{i=0}^{n/2-1} \tag{2.11}$$

The prover then continues, the recursion untill the length reduces to 1 (or if user provided desired stopping length). For each folding round the offset in the original domain increases $\mathcal{L}, \mathcal{L}^2, \mathcal{L}^4, \ldots$. The prover needs to store, the commit data (merkle tree), and the codeword for each layer to be used in the query phase. For this example, the recursion stops when $P.len() = 1$.

In general, the folding factor and stopping degree can be varied by the user (as a protocol configuration). For example, in RISC0, the stopping degree is 255 (length 256), and folding factor is 16.

2.1.2 Query phase: for folding in two

In the query phase the prover samples a uniformly random set of indices, (constrained by Fiat Shamir). The number of queries is set by the user/application. The prover needs to provide the polynomial evaluations at the relevant indices for EACH layer of recursion, and their authentication paths.

In this example, in order to prove the folding relation (2.11), we need three evaluations. The prover samples indices as many as specified by number of queries by the user. The query provides the necessary data to check the authenticity of the evaluations via the merkle paths and verify (2.11). The sampling process uses a uniform sampler based on the previous data added to the transcript.

For each query in the list of query indices, the prover computes the necessary indices in each folded layer of FRI. For example, if the query index is $i \in \{0, n/2\}$. The indices in each FRI layer is computed as

$$\begin{aligned} index &= index \mod layer.len() \qquad (2.12) \\ indexsym &= index + layer.len()/2 \mod layer.len() \qquad (2.13) \end{aligned}$$

For $P_0(x)$ length is n and $P_1(x)$ length is $n/2$. Prover provides

$$\begin{aligned} &[P_0(\omega^i), P_0(\omega^{i+n/2})] \ , \ [MT.path(P_0(\omega^i)), MT.path(P_0(\omega^{i+n/2}))] \\ &[P_1(\omega^{2(i)}), P_1(\omega^{2(i+n/4)})] \ , \ [MT.path(P_1(\omega^{2(i)})), MT.path(P_1(\omega^{2(i+n/4)}))] \end{aligned} \qquad (2.14)$$

The proof consists of

$$\begin{aligned} &[root_0, root_1] \\ &[P_0(\omega^i), P_0(\omega^{i+n/2})], [P_1(\omega^{2(i)}), P_1(\omega^{2(i+n/4)})] \\ &[MT.path(P_0(\omega^i)), MT.path(P_0(\omega^{i+n/2}))] \\ &[MT.path(P_1(\omega^{2(i)})), MT.path(P_1(\omega^{2(i+n/4)}))] \end{aligned} \qquad (2.15)$$

- In the full protocol (1), for each FRI layer, the prover needs to provide the leaves and the commit path for each FRI layer. With the layer indices
- The number of FRI layers is given by $\log_2(n) - \log_2(d_f + 1)$, where $n = 2^N$ is the initial codeword length, and d_f is the degree of the final polynomial.

2.1.3 Verifier

The verifier can check easily the merkle authentication paths of each leaf.

$$\begin{aligned} &[P_0(\omega^i), P_0(-\omega^i)] \ , \ [MT.path(P_0(\omega^i)), MT.path(P_0(-\omega^i))] \\ &[P_1(\omega^{2i}), P_1(-\omega^{2i})] \ , \ [MT.path(P_0(\omega^{2i})), MT.path(P_0(-\omega^{2i}))] \end{aligned} \qquad (2.16)$$

However the folding relation is checked using the collinearity relation as follows, Since $\omega^{n/2} = -1$. Now, the verifier needs to construct from the prover values

$$\gamma = hash(root_0) \qquad (2.17)$$

and constructs the single field element

$$\tilde{P}_1 = \frac{P_0(\omega^i) + P_0(-\omega^i)}{2} + \gamma \cdot \frac{P_0(\omega^i) - P_0(-\omega^i)}{2\omega^i} \tag{2.18}$$

The verifier checks if

$$\tilde{P}_1 \overset{?}{=} P_1(\omega^{2i}) \tag{2.19}$$

Note that this is still only one point where (2.11) is checked, so this needs to be repeated at different points. Usually this number is specified by the number of queries in protocol configuration. Nevertheless, we still cannot check all points, so there is still a small chance that the prover is cheating.

Technically the **FRI verifier** makes a few queries to the committed codeword, and checks that the prover knows "a" low degree polynomial $P(x)$ such that $\delta(F(x), P(x)) \leq \theta$, where $\theta \geq \frac{|\mathcal{L}|-|\mathcal{L}'|}{|\mathcal{L}|}$. The error θ is the ratio of the number of points $|\mathcal{L}| - |\mathcal{L}'|$ where F, P do not agree w.r.t to the total number of points $|\mathcal{L}|$. What this means is that, verifier determines that $F(x) = P(x)\ \forall x \in \mathcal{L}' \subseteq \mathcal{L}$. The intuition behind the Low degree testing is that, a dishonest prover who did not know the claimed low degree polynomial manifests itself through the error term θ.

- This error can be reduced by increasing the number of queries, which also increases the proof size.
- Some amount of security can be obtained by adding a proof of work component after the commit phase.

The proof of work component supposes that the security in bits w.r.t domain size vs field size is

$$\frac{\text{domain size}}{\text{Field size}} \tag{2.20}$$

For $n = 2^{27}$ and baby bear bit size 2^{31} using a quartic extension field in Fiat Shamir gives $2^{27}/2^{31*4}$ approximately 97 bits of security.

Proof of work: The protocol configuration can ask the prover to find a 64 bit nonce value such that, when added to the last hash chain of Fiat Shamir at the end of commit phase results in a certain number of zero bits (specified in config) in the resulting hash value. [4]. This arbitrary process adds "pow bits" amount of security.

3 FRI Config: Generic

3.1 Protocol config:

These parameters define FRI security in many ways, it is user responsibility to configure them as per their requriements. Another important issue that factors into this is also the choice of field and extension field.

security formula for user guide

```
pub struct fri_config {
    blow_up_factor: usize, //[1,2,4,8,16,user_specified] Default: 1 (assume
↪   input is a codeword)
    folding_factor: usize// values: [2,4,8,16], Default: 2,
    pow_bits: usize, //optional: determines number of leading zeros at POW
↪   stage if exists, Default: 0
    num_queries: usize, //user must specify this as per the security
↪   requirement of his implementation,
    //num_queries<= 2*(d_f+1)
    stopping_degree: usize, //Default: 0 , stopping_vector_size =
↪   stopping_degree+1
    commitment_scheme: Merkle //Add more options later as we get support
}
```

The number of queries s is configurable via the relation (but user must choose this number), the guideline is to follow the equation

$$s = \left\lceil \frac{\lambda}{\log_2 \beta} \right\rceil \tag{3.1}$$

where β is the blowup factor and λ is the desired security level λ (for the query phase). Eg: For baby bear extension field, the query phase is 100 bits secure with a blowup factor 4 for number of queries = 50.

3.2 Merkle config:

Since FRI folding algorithm will decrease codeword size and padding is not allowed, merkle must be able to support different arities, It is guaranteed that the codewords are always length of power of 2.

```
pub struct Merkle {
    hasher: Hasher, //options: Blake2s/3s, SHA256, Keccak, Poseidon2, Poseidon
↪   (optional)
    hasher_config: Domain_Tag_optional,
    compression: Hasher,//options: Blake2s/3s, SHA256, Keccak, Poseidon2,
↪   Poseidon (optional)
    compression_config: Domain_Tag_optional,
    //hasher = compression in most cases
    arity: usize, //binary,ternary quaternary etc (need to decide if to leave
↪   to user: can impact proof size and performance)
    padding_policy: Not_allowed,
    }
```

3.3 MMCS config

Once we have it in ICICLE

3.4 Proof structure and prover data structure

```
    pub struct FriProof <F> {
        commit_phase_commits: Vec<Merkle_root>,
        query: Vec<F>, //for each query the number of leafs are as long as
↪   folding factor
        query_path: Vec<Merklepaths>, // same as above
        final_poly: Vec<F>, //in case of canonical FRI it is a constant value
        pow_nonce: u64,
    }
```

Prover needs to store layer data and layer merkle tree for query phase.

```
    pub struct  Frilayer {
        layer_data: Vec<Vec<F>>,
        merkle_tree: Vec<MerkleTree>,
    }
```

3.5 Fiat Shamir config:

The hasher must be capable of taking inputs in $\mathbb{F}_p$ as well as $\mathbb{F}_{p^k}$, it must also be capable of outputting transcript based $\mathbb{F}_p$ as well as $\mathbb{F}_{p^k}$ values. The only committed data from which randomness is generated is the merkle commits. So it needs a label.

```
pub struct Transcript {
    hasher: Hasher, //options: Blake2s/3s, SHA256, Keccak, Poseidon2, Poseidon
↪   (optional)
    domain_separator: [u8] //Default: b"ICICLE FRI" user provided ONCE per
↪   protocol instantiation
    label_for_commit_phase_commits: [u8] //Default: b"FRI_layer_commits"
    label_for_round_challenge: [u8],
    label_for_nonce: "nonce" //default
    encode_policy: LE/BE // max length u32::max_value()
    seed_rng: rng //Optional
    public: Public [u8] //public and prevstate of the transcript
    }
```

Following our work in sumcheck we will always use this structure for any round message[3]

```
encoded_round_msg = [label||length(prover_msg_in_round)||round_number||
↪   prover_msg_in_round]
hash(public||prev_challenge||round_challenge_label||encoded_round_msg)
```

[3]Since msg length is constant for FRI roots, it is ok to skip the meta data length(provermsginround) as it offers no added security. It is also ok to skip "roundnumber", since swapping order of roots, will cause either merkle verify to fail, or inconsistency in collinearity check

Prover sequence for transcript in commit to query phase looks like (assume LE32 encoding), refer to algorithm (1).

```
//define transcript config
//initiate transcript
//entries_i are prover messages encoded in le32,
//alpha_i are challenges in extension field.
DS =[domain_seperator||log_2(initial_domain_size).LE32()]
entry_0 =[DS||public.LE32()]
//round 0
//compute merkle commit get root_0
entry_1 = [commit_label[u8]|| len(root_0)|| 0|| root_0.LE32()]
alpha_0 = hash(entry_0||rng||round_challenge_label[u8]||entry_1).to_ext_field()
//round 1
//compute merkle commit get root_1
entry_2 = [commit_label[u8]|| len(root_1)|| 1|| root_1.LE32()]
alpha_1 =
↪  hash(entry_0||alpha_0||round_challenge_label[u8]||entry_2).to_ext_field()
...
//compute merkle commit get root_n
entry_{n} = [commit_label[u8]|| len(root_{n-1})|| n-1|| root_{n-1}.LE32()]
alpha_{n-1} = hash(entry0||alpha_{n-2}||round_challenge_label[u8]||entry_n).to ⌋
↪  _ext_field()
// end of commit phase
-------------------------
Case 1: Proof of work ON
The hash that produce the required pow_bits behavior should compute following
↪  the prescription
//nonce = pow_func();
//temp = Hash(entry_0||alpha_{n-1}||"nonce"||nonce)
// after finding nonce use the following prescription to generate seed
entry_{nonce} = ["nonce".LE32()||nonce.LE32()]
seed = hash(entry_0||entry_{nonce})
-------------------------
Case 2: Proof of work OFF
seed = hash(entry_{0}||alpha_{n-1})
-------------------------
//for query phase
query_indices:Vec<u32> =  UniformSample(seed, range, friconfig.num_queries)
//given same seed, uniform sample must provide same query list!
```

Verifier sequence commit to query phase is as follows

```
//define transcript config
// initiate transcript with same definitions as prover
rng = transcript.seed_rng;
// log domain size must be equal to log_2n
inital_domain_size_log2 = friproof.len()+ log_2(final_poly.len())
DS =[domain_seperator||initial_domain_size.LE32()]
entry_0 =[DS||public.LE32()]
roots = read friproof.commit_phase_commits;
nonce = friproof.pow_nonce

//compute transcript data
entry_1 = [commit_label[u8]||len(root_0)|| 0|| roots[0].LE32()]
entry_2 = [commit_label[u8]||len(root_1)|| 1|| roots[1].LE32()]
...
entry_{n} = [commit_label[u8]||len(root_{n-1})|| n-1 || roots[n-1].LE32()]

//compute hashes
alpha_0 = hash(entry_0||rng||round_challenge_label[u8]||entry_1).to_ext_field()
alpha_1 =
↪  hash(entry_0||alpha_0||round_challenge_label[u8]||entry_2).to_ext_field()
..
alpha_{n-1} = hash(entry_0||alpha_{n-2}||round_challenge_label[u8]||entry_n).t⌋
↪  o_ext_field()
-------------------------
Case 1: Proof of work ON
//check nonce
entry_{nonce} = ["nonce".LE32()||nonce.LE32()]
bits = num_leading_zeros(hash(entry_0||entry_{nonce}));
assert_eq!(bits,friconfig.pow_bits);
seed = hash(entry_0||entry_{nonce})
-------------------------
Case 2: Proof of work OFF
seed = hash(entry_0||alpha_{n-1})
-------------------------
//for query phase
query_indices:Vec<u32> =  UniformSample(seed, range, friconfig.num_queries)
//given same seed, uniform sample must provide same query list!
```

3.6 Generic Pseudocode

In this section, we describe a generic FRI algorithm (1), that has folding factor 2, Proof of Work and arbitrary blow up factor.

Algorithm 1 Canonical FRI: with folding factor 2, POW, and any blowupfactor

```
Define: friconfig, Merkle, Transcript $T$
Init: $T$, frilayer, friproof
Input data: Univariate Poly: $P(x)$: $d = 2^k - 1$ or Codeword:len $n = 2^N$, [Public]
if poly.state = coeff then
    $n = friconfig.blowupfactor \times (d+1)$ , $p_{eval} \leftarrow NTT(P(x), n)$
else
    $p_{eval} \leftarrow \{p_0, p_1, \ldots, p_{2^N-1}\}$, $n = p_{eval}.len()$
end if
$D_{inv} \leftarrow g \cdot \{1, \omega^{-1}, \omega^{-2}, \ldots \omega^{-2^N+1}\}$ , $T \leftarrow [\mathsf{public}, n]$    ▷ Coset gen $g = 1$ or gen of $\mathbb{F}_p$

Phase 1: Commit and fold Phase : compute intensive
CommitFold2(friconfig,Merkleconfig,$T$,$p_{eval}$, n, $D_{inv}$,frilayer,friproof)
▷ Currently for folding factor two, replace commitment and folding algorithm here for folding factor> 2
Proof of work: optional    ▷ This can be any type of POW
while temp.leadingzeros ! = friconfig.powbits do
    let $nonce = rand ::< u64 > ()$
    let $temp = Hash(T, nonce)$    ▷ Same hash as in FiatShamir
end while
T.append(nonce) , friproof.pownonce.append(nonce)

Phase 2: Query phase : data retrieval
let seed = $hash(T)$, num queries = $friconfig.queries$
max = $n$, min = $friproof.finalpoly.len()$,
let queryindices = UniformSample(seed, range(min,max), num queries)
    ▷ Hash of transcript is seed
for query in queryindices do
    for layer in frilayer.layerdata do    ▷ Last layer is sent in plain text
        layersize = layer.len(), layertree = frilayer.merkletree[layer]
        let index = query % layersize
        let indexsym = (query+layersize/2) % layersize
        $leaf = layer[index]$, $leafsym = layer[indexsym]$
        friproof.query.append([$leaf$,$leafsym$])
        friproof.querypath.append([$layertree.path(leaf), layertree.path(leafsym)$])
    end for
end for
return friproof
```

3.7 some sample FRI configs:

The algorithm (1) with RISC0 parameters, and no proof of work can be compared with RISC0FRI. Note that Risc0 uses baby bear field.

Algorithm 2 **CommitFold2**(friconfig,Merkleconfig,T,p_{eval}, n, D_{inv},frilayer,friproof)

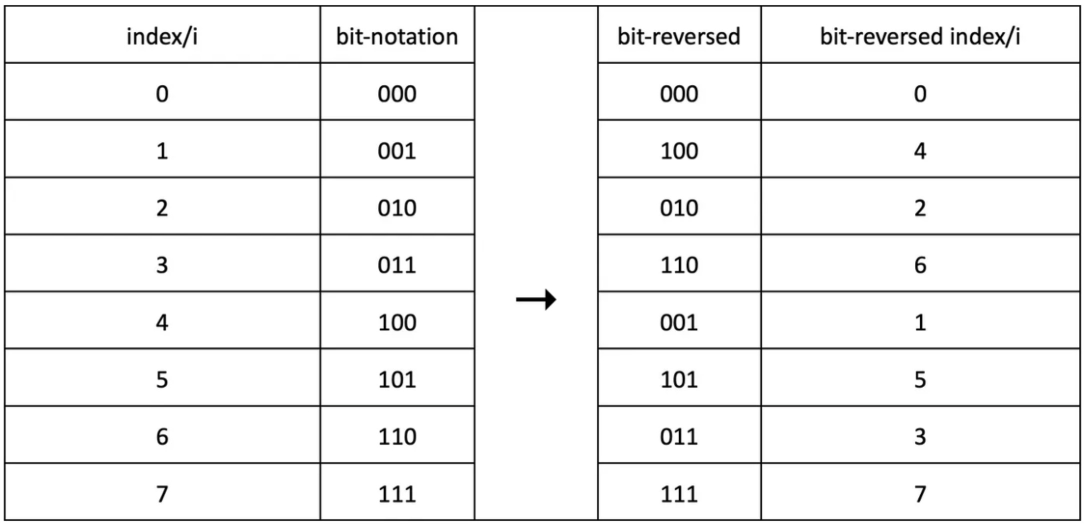
```
 1: for $r = 0, 1, 2, ..., \log_2(n) - \log_2(d_f + 1) - 1$ do          ▷ $d_f$: friconfig.stoppingdegree
 2:     $l \leftarrow p_{eval}.len()$
 3:     if $l$ =friconfig.stoppingdegree+1 then          ▷ Stopping degree = 0 in canonical
 4:         friproof.finalpoly.append($p_{eval}$)   ▷ Final poly: not committed, sent in plain text
 5:         Break
 6:     end if
 7:     frilayer.layerdata.append($p_{eval}$)          ▷ Prover needs to store this for query
 8:     frilayer.merkletree.append(Merkletree($p_{eval}$))
 9:     $root_r \leftarrow merkleroot(p_{eval})$
10:     friproof.commitphasecommits.append($root_r$)          ▷ update proof
11:     $T.append(root_r)$, $\alpha_i \leftarrow hash(T)$          ▷ Fiat Shamir $\alpha_i \in \mathbb{F}_{p^k}$ extension field
12:     $p_{even} \leftarrow (p_{eval}[0 : l/2] + p_{eval}[l/2 : l])/2$          ▷ Take top half and bottom half
13:     $p_{odd} \leftarrow ((p_{eval}[0 : l/2] - p_{eval}[l/2 : l])/2) \circ D[0 : l/2, offset = r]$
14:     $p_{eval} \leftarrow p_{even} + \alpha_i \cdot p_{odd}$          ▷ update folded vector
15: end for
```

```
pub struct fri_config_risc0 {
    blow_up_factor: 4,
    folding_factor: 16
    pow_bits: None, //risc0 does not use this at all
    num_queries: 50
    stopping_degree: 255, //stopping_vector_size = 256
    commitment_scheme: Merkle //Add more options later as we get support
}
```

This only gets 97 bits of security, as discussed earlier in (2.20). This config should be measured for raw performance vs our code. Besides, they have metal and GPU implementations from supranational.

In the case of Plonky3FRI (not circle) the parameters are tunable.

```
pub struct fri_config_p3 {
    blow_up_factor: usize, //[1,2,4,8,16,user_specified] Default: 1
↪   (assume input is a codeword)
    folding_factor: 2//  only arity 2 folding supported in Plonky3
    pow_bits: usize, //optional: determines number of leading zeros at POW
↪   stage if exists, Default: 0
    num_queries: usize //user must specify this as per the security
↪   requirement of his implementation
    stopping_degree: 0, //Default: 0 , stopping_vector_size =
↪   stopping_degree+1
    commitment_scheme: MMCS,
}
```



index/i	bit-notation		bit-reversed	bit-reversed index/i
0	000		000	0
1	001		100	4
2	010		010	2
3	011		110	6
4	100	→	001	1
5	101		101	5
6	110		011	3
7	111		111	7

Figure 1. Bit reversed storage for merkle trees

Note that in Plonky3 fold applies to every row in a matrix, and is closely related to MMCS scheme.

4 Generalizations

MMCS is an obvious inclusion that must be considered, for a note see Tomer's work. Currently Plonky3 implements binary MMCS.

4.1 Merkle optimization

Before computing the merkle tree, one can bit reverse the vectors as shown in fig 1 [5]. This causes the symmetric indices to be one next to the other, leading to shorter proof lengths. The security of this can be questioned, but it may be used as a configurable option.

check if this is ok!

4.2 Arbitrary fold

A given polynomial $P(x) = \sum_{i=0}^{d} a_i x^i$ can be split into parts [6]

$$P(x) = P_0(x^f) + x \cdot P_1(x^f) + \ldots + x^{f-1} P_{f-1}(x^f) \tag{4.1}$$

The folded polynomial with a random value β is constructed as

$$P_{fold}(x^f) = P_0(x^f) + \beta \cdot P_1(x^f) + \beta^2 \cdot P_2(x^f) + \ldots \beta^{f-1} P_{f-1}(x^f) \tag{4.2}$$

As expected, the domain goes from $\mathcal{L} \to \mathcal{L}^f$ in the split polynomials. For $f = 2$, we get the even and odd split (2.11). For $f = 4$

$$P(x) = P_0(x^4) + x \cdot P_1(x^4) + x^2 \cdot P_2(x^4) + x^3 \cdot P_3(x^4) \tag{4.3}$$

$$P_j(x^4) = \sum_{i=0}^{(d+1)/4-1} a_{4i+j} x^{4i} \tag{4.4}$$

where $j = 0, 1, 2, 3$. we can write in coefficient form, using the fact that $\omega^{n/2} = -1$ and $\omega^{n/4} = i$

$$\begin{aligned}
P_0(x^4) &= \left(\frac{P(x)+P(-x)}{4}\right) + \left(\frac{P(ix)+P(-ix)}{4}\right)\\
P_1(x^4) &= \left(\frac{P(x)-P(-x)}{4x}\right) + \left(\frac{P(ix)-P(-ix)}{4ix}\right)\\
P_2(x^4) &= -\left(\frac{P(x)+P(-x)}{4x^2}\right) + \left(\frac{P(ix)+P(-ix)}{4x^2}\right)\\
P_3(x^4) &= -\left(\frac{P(x)-P(-x)}{4x^3}\right) + \left(\frac{P(ix)-P(-ix)}{4ix^3}\right)
\end{aligned} \tag{4.5}$$

In evaluation form, the vector operation appears to be in interleaved form (perhaps there is a more efficient way to do this). ToDO

$$\begin{aligned}
P_0(\omega^{4i})\Big|_{i=0}^{n/4-1} &= \left(\frac{P(\omega^i)+P(-\omega^i)}{4}\right)\Big|_{i=0}^{n/4-1} + \left(\frac{P(\omega^{i+n/4})+P(-\omega^{i+n/4})}{4}\right)\Big|_{i=0}^{n/4-1}\\
P_1(\omega^{4i})\Big|_{i=0}^{n/4-1} &= \left(\frac{P(\omega^i)-P(-\omega^i)}{4\omega^i}\right)\Big|_{i=0}^{n/4-1} + \left(\frac{P(\omega^{i+n/4})-P(-\omega^{i+n/4})}{4\omega^{i+n/4}}\right)\Big|_{i=0}^{n/4-1}\\
P_2(\omega^{4i})\Big|_{i=0}^{n/4-1} &= -\left(\frac{P(\omega^i)+P(-\omega^i)}{4\omega^{2i}}\right)\Big|_{i=0}^{n/4-1} + \left(\frac{P(\omega^{i+n/4})+P(-\omega^{i+n/4})}{4\omega^{2i}}\right)\Big|_{i=0}^{n/4-1}\\
P_3(\omega^{4i})\Big|_{i=0}^{n/4-1} &= -\left(\frac{P(\omega^i)-P(-\omega^i)}{4\omega^{3i}}\right)\Big|_{i=0}^{n/4-1} + \left(\frac{P(\omega^{i+n/4})+P(-\omega^{i+n/4})}{4\omega^{3i+n/4}}\right)\Big|_{i=0}^{n/4-1}
\end{aligned} \tag{4.6}$$

Similar formulas can be written down for $f = 8, 16$, but this will lead to lot of interleaving access the above way ToDO

See RISC0cuda4fold or RISC0Cpufold for reference implementations for folding factor 16.

4.3 Batched FRI

This is referred in both Risc0 and plonky3 white papers and the definition of batching is very simple (see [6]). We are given a series of polynomials

$$f_0(x), f_1(x), \ldots f_m(x) \tag{4.7}$$

for low degree testing.

- The prover Batch commits using batch merkle or batch mmcs all the polynomials.
- Samples a random value $\beta = hash(T)$ and computes

$$F(x) = \sum_{i=0}^{m} \beta^i f_i(x) \tag{4.8}$$

- Prover does regular FRI with $F(x)$.
- The query indices in query phase are used to check the consistency of (4.8)

Batched FRI is very practical and is often used in large scale proving systems in production like RISC0, for small scale problems this is kind of useless.

4.4 DEEP-FRI

future work

The method used in DEEP FRI is quotienting, and out of domain sampling, which is fundamentally different from FRI. (it is not used in RISC0). The claim is that it does not offer improved soundness compared to FRI [7].But all of this is debatable and sitting on unproven conjectures.

The DEEP-FRI used in plonky3 [8] is not what is discussed in [9]. In Plonky3 they do quotienting and then prove that the result of the quotient is a low degree polynomial with regular FRI. It can be considered as a wrap around the FRI defined in (1).

4.4.1 related: FRI-PCS

Future Work

The quotienting is important to understand how to do FRI-PCS.

Acknowledgments

We stand on the shoulders of giants.

References

[1] T. Solberg, "New bears at the bear market - introducing polar bear and teddy bear fields."

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast reed-solomon interactive oracle proofs of proximity."

[3] A. Szepieniec, "Anatomy of a stark, part 3: Fri."

[4] StarkWare, "ethstark documentation." Cryptology ePrint Archive, Paper 2021/582, 2021. https://eprint.iacr.org/2021/582.

[5] Kroma, "Brain fried - diving into fri protocol."

[6] U. Haböck, "A summary on the FRI low degree test." Cryptology ePrint Archive, Paper 2022/1216, 2022.

[7] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, "Proximity gaps for reed-solomon codes." Cryptology ePrint Archive, Paper 2020/654, 2020.

[8] H. Masip-Ardevol, M. Guzmán-Albiol, J. Baylina-Melé, and J. L. Muñoz-Tapia, "eSTARK: Extending STARKs with arguments." Cryptology ePrint Archive, Paper 2023/474, 2023.

[9] E. Ben-Sasson, L. Goldberg, S. Kopparty, and S. Saraf, "DEEP-FRI: Sampling outside the box improves soundness." Cryptology ePrint Archive, Paper 2019/336, 2019.