STARK Math

Introduction

This document serves as an overview of STARK math, providing deep insights whilst not going too much into formal verification of axioms. The idea is to provide an implementation specification for developers who seek to code STARKs from scratch, or for mathematicians who are looking for a summary of the methods involved in constructing basic STARKs.

Explanation by Example

Most of the information in this section stems from the `Stark 101` series by Starkware. The content has been modified to convey the core principles in a more straight forward manner, eliminating the need to read their Python code line-by-line. Some details related to domain choices have been omitted whilst other sections of the source material have been extended.

> STARK 101
> STARK 101 is a hands-on tutorial on how to write a STARK prover from scratch (in Python).
> https://starkware.co/stark-101/

We start with an example of a `trace table`:

column	x	y

0	10	20
1	30	40
2	70	80

The `trace table` represents the trace of an arbitrary computation. This could be the output of a domain specific program written in a `DSL` like `Circom`, or the trace of a `Virtual Machine` as is the case with most `STARK` proving systems (`Risc0`, `SP1`, `ZKEVMs`).

We interpolate our `trace polynomials` for `x` and `y` using either `FFT` or `lagrange interpolation`:

`g(x)` = Interpolate(`(0,10)`, `(1,30)`, `(2,70)`)

`h(x)` = Interpolate(`(0,20)`, `(1,40)`, `(2,80)`)

Next we define a `constraint polynomial` that evaluates to `0` for our `domain`:

`C(x)` = `h(x)` - `g(x)` - `10`

This constraint polynomial may not be inherently useful, however it conveys the idea of how we can enforce `constraints`. Multiple `constraint polynomials (c1, c2, ..., cn)` can be combined into a single `constraint polynomial`:

`C(x)` = `c1(x)` + `c2(x)` + `...` + `cn(x)`

Commit to the Trace Polynomials

We commit to all `trace polynomials` by evaluating them over an `extended domain` and inserting the evaluations into a `merkle tree`.

The `extended domain` is constructed from powers of `g`. `g` is a generator of a `finite field`:

$$D = g^0, g^1, g^2..g^N$$

Where `N` is the size of the multiplicative subgroup and should be a power of `2`. The advantages of choosing a power of `2` will be explained later in this document.

For each `trace polynomial` we commit to a single `merkle tree` the evaluations over the `extended domain`.

Since we utilize `FFT` under the hood of `FRI` for efficient polynomial math it is best to choose an `extended domain` that is a power of 2, as well as a `root of unity`.

For the size of the extended domain `N` to support a `root of unity`, it must satisfy:

$$1 = g^N$$

where `g` is the generator of the extended domain.

When the size of the input is a power of two,

$$N = 2^n$$

it allows FFT to evenly split the problem in half at each recursive step. This leads to a natural and efficient division, reducing the time complexity of the algorithm's complexity to:

$$O(N \log(N))$$

`N` is not a power of two, splitting evenly becomes more complex, leading to less efficient computation.

Commit to the Constraint Polynomial

The process of committing to the `constraint polynomial` is more complex than that of committing to the `trace polynomials` since it involves `FRI` :

Anatomy of a STARK, Part 3: FRI

STARK tutorial with supporting source code in python.

https://aszepieniec.github.io/stark-anatomy/fri.html

For the first round of our `FRI` folding we evaluate the `constraint polynomial` over the `extended domain` and once again commit the results to a `merkle tree`. For each additional folding step, until we reach a `low degree polynomial` , we fold the degree of the polynomial and the `extended domain` in half.

We evaluate the `folded polynomial` over the `folded domain` in each step and commit the results to `merkle trees`. Ultimately we are left with a `merkle tree` for each folding step.

Probabilistic Verifier Queries

Now that we have constructed our `merkle trees` for the evaluations of the `trace polynomials` and the `folded constraint polynomial` , the verifier can issue random challenges (interactive proving). `Fiat-shamir transformation` can be utilized to make this protocol non-interactive, but for the scope of this section we will assume an interactive setup.

For each query the prover must reveal not only the evaluations of the `trace polynomial` over the originally extended domain, but also the corresponding values from the `FRI merkle trees` . If the values are persistent for each step in `FRI` and therefore satisfy the rules of our `constraint polynomial`, the query is considered valid.

The verifier is not required to fold the polynomials but instead employs a consistency check for each step in the queried evaluations. The formula for the consistency check looks like this:

$$f'(x) = \frac{f(x) + f(g \cdot x)}{2}$$

Where `g` is a generator of the `finite field` over which the `extended domain` is defined. `f'(x)` is the next folded polynomial and `f(gx)` is the preceding polynomial.

A generator `g` can be brute-forced and must satisfy:

$$g^{(q-1)/p_i} \not\equiv 1 \pmod{q}$$

Where `q` is the size of a field and `pi` are all possible prime factors of `q-1`. In commonly used fields `q` is chosen such that factoring the prime factors of `q-1` is feasible.

The amount of queries that are necessary for the verifier to trust the prover can be calculated according to this probability formula:

$(1 - \frac{1}{m})^k$, where `k` is the amount of queries and `m` the amount of columns in the `trace`.

> 💡 The more columns we have in the `trace` the more queries we need to perform in order to be able to trust the prover with negligible fraud tolerance.

FFT and IFFT Introduction

We utilize FFT to convert a polynomial from its `coefficient representation` to its `value representation`.

IFFT on the other hand is the inverse process of converting the polynomial back to the `coefficient representation`.

The size of the `value representation` depends on the size of the `extended domain` we choose to evaluate our polynomial over.

By using the formula for each point in our `value representation` , we can get the set of points required to interpolate the next folded polynomial using `IFFT` :

$$f'(x) = \frac{f(x) + f(g \cdot x)}{2}$$

As explained earlier the verifier will only need to check the consistency of the folding steps in combination with verifying the merkle queries and won't need to re-compute the entire folding process.

Extend Verification with DEEP

Deep is used to add an extra layer of security to the verification phase. By randomly querying the `constraint polynomial` at a random point `z` that lies outside the extended evaluation domain, it can be verified that the folding and commitment outputs correctly represent the relationship between the `trace` and `constraints`.