PCS multiproofs using random evaluation

Jun 18, 2021

Multiproof scheme for polynomial commitments

This post describes a multiproof scheme for (additively homomorphic) polynomial commitment schemes. It is very efficient to verify (the dominant operation for the verifier being one multiexponentiation), as well as efficient to compute as long as all the polynomials are fully available (it is not suitable in the case where the aggregations has to be done from only proofs without access to the data).

It is thus very powerful in the setting of verkle tries for the purpos of implementing weak statelessness.

Note that this post was written with the KZG commitment scheme in mind. It does however work for any “additively homomorphic” scheme, where it is possible to add two commiments together to get the commitment to the sum of the two polynomials. This means it can also be applied to Inner Product Arguments (the core argument behind bulletproofs) and is actually a very powerful aggregation scheme for this use case.

Verkle multiproofs using random evaluation

Problem: In a verkle tree of width d, we want to provide all the intermediate KZG (“Kate”) proofs as efficiently as possible.

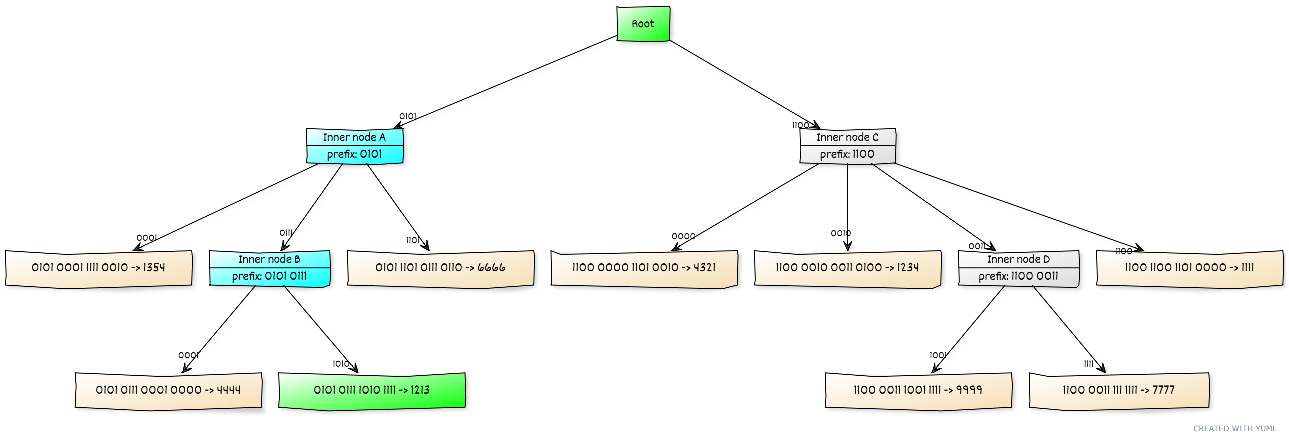
We need to provide all intermediate commitments, there is no way around that in Verkle trees, but we only need a single KZG proof in the optimal case. There are efficient multiverification techniques for KZG if all proofs are given to the verifier, but we want to do with just a small constant number of proofs.

For the notation used here, please check my post on KZG commitments.

Please see here for an introduction to verkle tries.

Connection to verkle tries

Quick recap: looking at this verkle trie:



In order to prove the leaf value `0101 0111 1010 1111 -> 1213` we have to give the commitment to `Node A` and `Node B` (both marked in cyan), as well as the following KZG proofs:

- Proof that the root (hash of key and value) of the node `0101 0111 1010 1111 -> 1213` is the evaluation of the commitment of `Inner node B` at the index `1010`
- Proof that the root of `Inner node B` (hash of the KZG commitment) is the evaluation of the commitment of `Inner node A` at the index `0111`
- Proof that the root of `Inner node A` (hash of the KZG commitment) is the evaluation of the `Root` commitment at the index `0101`

Each of these commitments, let's call them C_0 (`Inner node B`), C_1 (`Inner node A`) and C_2 (`Root`) is to a polynomial function $f_i(X)$, What we are really saying by the claim that the commitment C_i evaluates to some y_i at index z_i is that *the function committed to* by C_i, i.e. $f_i(X)$, evaluates to y_i at z_i, i.e. $f_i(z_i) = y_i$, So what we need to prove is

- $f_0(\omega^{0b1010}) = H(0101\ 0111\ 1010\ 1111, 1213)$ (hash of key and value), where $C_0 = [f_0(s)]_1$, i.e. C_0 is the commitment to $f_0(X)$
- $f_1(\omega^{0b0111}) = H(C_0)$, where $C_1 = [f_1(s)]_1$
- $f_2(\omega^{0b0101}) = H(C_1)$, where $C_2 = [f_2(s)]_1$

Note that we replaced the index with $z_i = \omega^{\text{the index}}$, where ω is a d-th root of unity which makes many operations more efficient in practice (we will explain below why). H stands for a collision-resistant hash function, for example `sha256`.

If we have a node deeper inside the trie (more inner nodes on the path), there will be more proofs to provide. Also, if we do a multiproof, where we provide the proof for multiple key/value pairs at the same time, the list of proofs will be even longer. Overall, we can end up with hundreds or thousands of evaluations of the form $f_i(z_i) = y_i$ to prove, where we have the commitments $C_i = [f_i(s)]_1$ (these are part of the verkle proof as well).

Relation to prove

The central part of a verkle multiproof (a verkle proof that proves many leaves at the same time) is to prove the following relation:

Given m KZG commitments $C_0 = [f_0(s)]_1, \ldots, C_{m-1} = [f_{m-1}(s)]_1$, prove evaluations

$$f_0(z_0) = y_0$$
$$\vdots$$
$$f_{m-1}(z_{m-1}) = y_{m-1}$$

where $z_i \in \{\omega^0, \dots, \omega^{d-1}\}$, and ω is a d-th root of unity.

Proof

1. Let $r \leftarrow H(C_0, \dots, C_{m-1}, y_0, \dots, y_{m-1}, z_0, \dots, z_{m-1})$ (H is a hash function) The prover computes the polynomial

$$g(X) = r^0 \frac{f_0(X) - y_0}{X - z_0} + r^1 \frac{f_1(X) - y_1}{X - z_1} + \ldots + r^{m-1} \frac{f_{m-1}(X) - y_{m-1}}{X - z_{m-1}}$$

If we can prove that $g(X)$ is actually a polynomial (and not a rational function), then it means that all the quotients are exact divisions, and thus the proof is complete. This is because it is a random linear combination of the quotients: if we just added the quotients, it could be that two of them just "cancel out" their remainders to give a polynomial. But because r is chosen after all the inputs are fixed (see Fiat-Shamir heuristic), it is computationally impossible for the prover to find inputs such that two of the remainders cancel.

Everything else revolves around proving that $g(X)$ is a polynomial (and not a rational function) with minimal effort for the prover and verifier.

Note that any function that we can commit to via a KZG commitment is a polynomial. So the prover computes and sends the commitment $D = [g(s)]_1$, Now we only need to convince the verifier that D is, indeed, a commitment to the function $g(X)$, This is what the following steps are about.

2. We will prove the correctness of D by (1) evaluating it at a completely random point t and (2) helping the verifier check that the evaluation is indeed $g(t)$, Let $t \leftarrow H(r, D)$, We will evaluate $g(t)$ and help the verifier evaluate the equation

$$g(t) = \sum_{i=0}^{m-1} r^i \frac{f_i(t) - y_i}{t - z_i}$$

with the help of the prover. Note that we can split this up into two sums

$$g(t) = \underbrace{\sum_{i=0}^{m-1} r^i \frac{f_i(t)}{t - z_i}}_{g_1(t)} - \underbrace{\sum_{i=0}^{m-1} r^i \frac{y_i}{t - z_i}}_{g_2(t)}$$

The second sum term $g_2(t)$ is completely known to the verifier and can be computed using a small number of field operations. The first term can be computed by giving an opening to the commitment

$$E = \sum_{i=0}^{m-1} \frac{r^i}{t - z_i} C_i$$

at t, Note that the commitment E itself can be computed by the verifier using a multiexponentiation (this will be the main part of the verifier work), because they have all the necessary inputs.

The prover computes

$$h(X) = \sum_{i=0}^{m-1} r^i \frac{f_i(X)}{t - z_i}$$

which satisfies $E = [h(s)]_1$,

3. Let $f_D(X)$ denote the polynomial committed to by D – if the prover is honest, then this will be $g(X)$, however this is what the verifier wants to check. Due to the binding property there can be at most one polynomial that the prover can open D at, which makes this valid.

   What remans to be checked for the verifier to conclude the proof is that

   $$f_D(t) = h(t) - g_2(t)$$

   or, reordering this:

   $$g_2(t) = h(t) - f_D(t)$$

   The verifier can compute the left hand side $y = g_2(t)$ without the help of the prover. What remains is for the prover to give an opening to the commitment $E - D$ at t to prove that it is equal to y. The KZG proof $\pi = [(h(s) - g(s) - y)/(s - t)]_1$ verifies that this is the case.

   The proof consists of D and π.

Verification

The verifier starts by computing r and t,

As we have seen above, the verifier can compute the commitment E (using one multiexponentiation) and the field element $g_2(t)$

Then the verifier computes

$$y = g_2(t)$$

The verifier checks the Kate opening proof

$$e(E - D - [y]_1, [1]_2) = e(\pi, [s - t]_2)$$

This means that the verifier now knows that the commitment D, opened at a completely random point (that the prover didn't know when they committed to it), has exactly the value of $g(t)$ which the verifier computed with the help of a prover. According to the Schwartz-Zippel lemma this is extremely unlikely (read: impossible in practice, like finding a hash collision), unless D is actually a commitment to $g(X)$; thus, $g(X)$ must be a polynomial and the proof is complete.

Optimization: Do everything in evaluation form

This section is to explain various optimizations which make all of the above easy to compute, it is not essential for understanding the correctness of the proof (but it is for making an efficient implementation). The great advantage of the above version of KZG multiproofs, compared to many others, is that very large parts of the prover and verifier work only need to be done in the field. In addition, all these operations can be done on the evaluation form of the polynomial (or in maths terms: in the “Lagrange basis”). What that means and how it is used is explained below.

Evaluation form

Usually, we see a polynomial as a sequence of coefficients $c_0, c_1, \ldots$ defining a polynomial function $f(X) = \sum_i c_i X^i$, Here we define another way to look at polynomials: the so-called “evaluation form”.

Given d points $(\omega^0, y_0), \ldots, (\omega^{d-1}, y_{d-1})$, there is always a unique polynomial f of degree $< d$ that assumes all these points, i.e. $f(\omega^i) = y_i$ for all $0 \leq i < d$, Conversely, given a polynomial, we can easily compute the evaluations at the d roots of unity. We thus have a one-to-one correspondence of

$$\{\text{all polynomials of degree } < d\} \leftrightarrow \{\text{vectors of length } d\text{, seen as evaluations of a polynomial at } \omega$$

This can be seen as a “change of basis”: On the left, the basis is “the coefficients of the polynomial”, whereas on the right, it’s “the evaluations of the polynomial on the ω^i”.

Often, the evaluation form is more natural: For example, when we want to use KZG as a vector commitment, we will commit to a vector $(y_0, \ldots, y_{d-1})$ by committing to a function that is defined by $f(\omega^i) = y_i$, But there are more advantages to the evaluation form: Some operations, such as multiplying two polynomials or dividing them (if the division is exact) are much more efficient in evaluation form.

In fact, all the operations in the KZG multiproof above can be done very efficiently in evaluation form, and in practice we never even compute the polynomials in coefficient form when we do this!

Lagrange polynomials

Let’s define the Lagrange polynomials on the domain $x_0, \ldots, x_{d-1}$.

$$\ell_i(X) = \prod_{j \neq i} \frac{X - x_j}{x_i - x_j}$$

For any $x \in x_0, \ldots, x_{d-1}$,

$$\ell_i(x) = \begin{cases} 1 & \text{if } x = x_i \\ 0 & \text{otherwise} \end{cases}$$

so the Lagrange polynomials can be seen as the “unit vectors” for polynomials in evaluation form. Using these, we can explicitly translate from the evaluation form to the coefficient form: say we’re given $(y_0, \ldots, y_{d-1})$ as a polynomial in evaluation form, then the polynomial is

$$f(X) = \sum_{i=0}^{d-1} y_i \ell_i(X)$$

Polynomials in evaluation form (given by the y_i) are sometimes called “Polynomials in Lagrange basis” because of this.

For KZG commitments, we can use another trick: Recall that the G_1 setup for KZG to commit to polynomials of degree $< k$ consists of $[s^i]_1, 0 \leq i < d$, From these, we can compute $[\ell_i(s)]_1, 0 \leq i < d$, Then we can simply compute a polynomial commitment like this:

$$[f(s)]_1 = \sum_{i=0}^{d-1} y_i [\ell_i(s)]_1$$

There is no need to compute the polynomial in coefficient form to compute its KZG commitment.

FFT to change between evaluation and coefficient form

The Discrete Fourier Transform $u = \mathrm{DFT}(v)$ of a vector v is defined by

$$u_i = \sum_{j=0}^{d-1} v_j \omega^{ij}$$

Note that if we define the polynomial $f(X) = \sum_{j=0}^{k-1} v_j X^j$, then $u_i = f(\omega^i)$, i.e. the DFT computes the values of $f(X)$ on the domain $\omega^i, 0 \leq i < d$, This is why in practice, we will the roots of unity as our domain whenever it is available because then we can use the DFT to compute the evaluation form from the coefficient form.

The inverse, the Inverse Discrete Fourier Transform $v_i = \mathrm{DFT}^{-1}(u_i)$, is given by

$$v_i = \frac{1}{d} \sum_{j=0}^{d-1} u_j \omega^{-ij}$$

Similar to how the DFT computes the evaluations of a polynomial in coefficient form, the inverse DFT computes the coefficients of a polynomial from its evaluations.

To summarize:

$$\text{coefficient form} \underset{\mathrm{DFT}^{-1}}{\overset{\mathrm{DFT}}{\rightleftarrows}} \text{evaluation form}$$

The "Fast Fourier Transform" is a fast algorithm, that can compute the DFT or inverse DFT in only $\frac{d}{2} \log d$ multiplications. A direct implementation of the sum above would take d^2 multiplications. This speedup is huge and makes the FFT such a powerful tool.

In strict usage, DFT is the generic name for the operation, whereas FFT is an algorithm to implement it (similar to sorting being an operation, whereas quicksort is one possible algorithm to implement that operation). However, colloquially, people very often just speak of FFT even when they mean the operation as well as the algorithm.

Multiplying and dividing polynomials

Let's say we have two polynomials $f(X)$ and $g(X)$ such that the sum of the degrees is less than k, Then the product $h(X) = f(X) \cdot g(X)$ is a polynomial of degree less than k. If we have the evaluations $f_i = f(x_i)$ and $g_i = g(x_i)$, then we can easily compute the evaluations of the product:

$$h_i = h(x_i) = f(x_i) g(x_i) = f_i g_i$$

This only needs d multiplications, whereas multiplying in coefficient form needs $O(d^2)$ multiplications. So multiplying two polynomials is much easier in evaluation form.

Now lets assume that $g(X)$ divides $f(X)$ exactly, i.e. there is a polynomial $q(X)$ such that $f(X) = g(X) \cdot q(X)$, Then we can find this quotient $q(X)$ in evaluation form

$$q_i = q(x_i) = f(x_i)/g(x_i) = f_i/g_i$$

using only d divisions. Again, using long division, this would be a much more difficult task taking $O(d^2)$ operations in coefficient form.

We can use this trick to compute openings for Kate commitments in evaluation form, where we need to compute a polynomial quotient. So we can use this trick to compute the proof π above.

Dividing when one of the points is zero

There is only one problem: What if one of the g_i is zero, i.e. $g(X)$ is zero somewhere on our evaluation domain? Note by definition this can only happen if $f(X)$ is also zero at the same point, otherwise $g(X)$ cannot divide $f(X)$, But if both are zero, then we are left with $q_i = 0/0$ and can't directly compute it in evaluation form. But do we have to go back to coefficient form and use long division? It turns out that there's a trick to avoid this, at least in the case that we care about: Often, $g(X) = X - x_m$ is a linear factor. We want to compute

$$q(X) = \frac{f(X)}{g(X)} = \frac{f(X)}{X - x_m} = \sum_{i=0}^{d-1} f_i \frac{\ell_i(X)}{X - x_m}$$

Now, we introduce the polynomial $A(X) = \prod_{i=0}^{d-1}(X - x_i)$. The roots of the polynomial are all the points of the domain. So we can also write $A(X)$ in another form as

The formal derivative of A is given by

$$A'(X) = \sum_{j=0}^{d-1} \prod_{i \neq j} (X - x_i)$$

This polynomial is extremely useful because we can write the Lagrange polynomials as

$$\ell_i(X) = \frac{1}{A'(x_i)} \frac{A(X)}{X - x_i}$$

so

$$\frac{\ell_i(X)}{X - x_m} = \frac{1}{A'(x_i)} \frac{A(X)}{(X - x_i)(X - x_m)} = \frac{A'(x_m)}{A'(x_i)} \frac{\ell_m(X)}{X - x_i}$$

Now, let's go back to the equation for $q(X)$, The one problem we have if we want to get this in evaluation form is the point $q(x_m)$ where we encounter a division by zero; all the other points are easy to compute. But now we can replace

$$q(X) = \sum_{i=0}^{d-1} f_i \frac{\ell_i(X)}{X - x_m} = \sum_{i=0}^{d-1} f_i \frac{A'(x_m)}{A'(x_i)} \frac{\ell_m(X)}{X - x_i}$$

Because $\ell_m(x_m) = 1$, this lets us compute

$$q_m = q(x_m) = \sum_{\substack{i=0 \\ i \neq m}}^{d-1} f_i \frac{A'(x_m)}{A'(x_i)} \frac{1}{x_m - x_i}$$

The reason why we can exclude $i = m$ in this summation is that $f_m = 0$ (otherwise the division would not be possible and have a remainder).

For all $j \neq m$, we can compute directly

$$q_j = q(x_j) = \sum_{i=0}^{d-1} f_i \frac{\ell_i(x_j)}{x_j - x_m} = \frac{f_j}{x_j - x_m}$$

This allows us to efficiently allow all q_j in evaluation form for all j, including $j = m$, This trick is necessary to compute $q(X)$ in evaluation form.

In order to make this efficient, it's best to precompute the $A'(x_i)$ as computing them takes $O(d^2)$ time, but only needs to be performed once.

Special case roots of unity

In the case where we are using the roots of unity as our domain, we can use some tricks so that we don't need the precomputation of $A'(x_i)$. The key observation is that $A(X)$ can be rewritten in a simpler form:

$$A(X) = \prod_{i=0}^{d-1}(X - \omega^i) = X^d - 1$$

Because of this the formal derivative becomes much simpler:

$$A'(X) = dx^{d-1} = \sum_{j=0}^{d-1} \prod_{i \neq j} (X - \omega^i)$$

And we can now easily derive $A'(x_i)$:

$$A'(\omega^i) = d(\omega^i)^{d-1} = d\omega^{-i}$$

Evaluating a polynomial in evaluation form on a point outside the domain

Now there is one thing that we can do with a polynomial in coefficient form, that does not appear to be easily feasible in evaluation form: We can evaluate it at any point. Yes, in evaluation form, we do have the values at the x_i, so we can evaluate $f(x_i)$ by just taking an item from the vector; but surely, to evaluate $f(X)$ at a point z *outside* the domain, we have to first convert to coefficient form?

It turns out, it is not necessary. To the rescue comes the so-called barycentric formula. Here is how to derive it using Lagrange interpolation:

$$f(z) = \sum_{i=0}^{d-1} f_i \ell_i(z) = \sum_{i=0}^{d-1} f_i \frac{1}{A'(x_i)} \frac{A(z)}{z - x_i} = A(z) \sum_{i=0}^{d-1} \frac{f_i}{A'(x_i)} \frac{1}{z - x_i}$$

The last part can be computed in just $O(d)$ steps (assuming the precomputation of the $A'(x_i)$), which makes this formula very useful, for example for computing $g(t)$ and $h(t)$ without changing into coefficient form.

This formula can be simplified in the case where the domain is the roots of unity:

$$f(z) = \frac{z^d - 1}{d} \sum_{i=0}^{d-1} f_i \frac{\omega^i}{z - \omega^i}$$

Dankrad Feist

Dankrad Feist
mail .at. dankradfeist .dot. de

dankrad
dankrad

Researcher at Ethereum Foundation