Overview of Plonky2

Plonky2 is a proof system developed by Polygon Zero, based on "Plonkish" arithmetization and FRI polynomial commitments.

The primary design goal of Plonky2 was to allow very efficient recursive proofs, and it's still interesting in that aspect (the next-generation Plonky3 toolkit does not support recursion, or even Plonk circuits, at the time of writing this).

In this set of notes I try to describe the internal workings of Plonky2 in detail (as the original authors provided essentially no documentation at all…)

Links to the topical notes

- Layout.md - Layout of all the columns
- Gates.md - The different "custom gates" present in the Plonky2 code base
- Selectors.md - Gate selectors and constants
- GateConstraints.md - Gate constraint equations
- Wiring.md - The permutation argument
- Poseidon.md - Poseidon hash function
- FRI.md - FRI commitment scheme
- Challenges.md - Fiat-Shamir challenges
- Protocol.md - Overview of the protocol
- Lookups.md - Lookup gates and the lookup argument
- Recursion.md - Recursive proofs

Some basic design choices

Plonky2 uses a Plonkish arithmetization with wide rows and FRI polynomial commitment scheme, over a small (64-bit) field.

Features

- Plonkish arithmetization:
  - the witness is organized in a $2^n \times M$ matrix (called "advice wires");
  - the circuit is described by "gates" and wiring constraints
  - with optional lookup tables
- wide rows (by default $M = 135$)
- gates are single-row, and at most 1 gate in a row (no rotations a la Halo2)
- custom gates (any number of equations per gate)
- relatively high-degree gates (by default, up to 8)
- optimized for recursive proofs

Having such a large number of columns is not a problem in practice, because using FRI whole rows can be committed (and opened) at together. With KZG this would be rather expensive.

Field choice

Plonky2 uses the Goldilocks field $\mathbb{F}_p$ with $p = 2^{64} - 2^{32} + 1$, and a degree two extension $\mathbb{F}_{p^2} = \widetilde{\mathbb{F}} := \mathbb{F}_p[X]/(X^2 - 7)$. This is essentially the smallest irreducible polynomial over $\mathbb{F}_p$ to use.

In theory, the code supports higher degree field extensions too, though I don't think they are actually used; the crate implements the degree 4 and 5 extension $\mathbb{F}_p[X]/(X^4 - 7)$ and $\mathbb{F}_p[X]/(X^5 - 3)$.

Recently Telos announced the integration of other fields in their fork.

Hash choice

Plonky2 can use either Keccak or the Poseidon hash (with custom constants) with `t=12` (that is, the internal state is 12 field elements, approximately 750 bits wide).

For recursive proofs obviously Poseidon is used. To make this fast, a 135 column wide Poseidon gate is used; see Poseidon.md for more details.

The hash function is used for several purposes:

- the most important is the FRI commitment (both for computing linear hashes of rows and then the Merkle tree on the top);
- but also used for Fiat-Shamir heuristic;
- and handling of public inputs.

Because the public inputs are always hashed into 4 field elements (approx. 256 bits), in practice all circuits contain a Poseidon gate, and thus are 135 columns wide.

In theory it's possible to add further hash function choices, eg. Monolith (faster proofs) or Poseidon2-BN254 (more efficient EVM-compatible wrapper).

Custom Gates

Plonky2 has about a dozen of custom gates included by default (but the user can in theory add more). Most of these are intended for implementing the recursive verifier circuit.

A custom gate is essentially several (low-degree) polynomial constraints over the witness cells of a single row, plus recipes to calculate some of these (for example in the Poseidon gate, all cells apart from the 12 inputs are calculated). The latter is encoded in a so called "Generator" using the `SimpleGenerator` trait.

On the user side, it seems that custom gates are abstracted away behind "gadgets".

Unfortunately the actual gate equations never appear explicitly in the code, only routines to calculate them (several ones for different contexts…), which 1) makes it hard to read and debug; and 2) makes the code very non-modular.

However, there is also a good reason for this: The actual equations, if described as (multivariate) polynomials, can be very big and thus inefficient to calculate, especially in the case of the Poseidon gate. This is because of the lack of sharing between intermediate variables. Instead, you need to described *an efficient algorithm* to compute these polynomials (think for example Horner evaluation vs. naive polynomial evaluation).

Note that while in theory a row could contain several non-overlapping gates, the way Plonky2 organizes its gate equations would make this unsound (it would also complicate the system even more). See the details at the protocol description.

List of gates

The default gate set is:

- `arithmetic_base`
- `arithmetic_extension`
- `base_sum`
- `constant`
- `coset_interpolation`
- `exponentiation`
- `lookup`
- `lookup_table`
- `multiplication_extension`
- `noop`
- `poseidon`
- `poseidon_mds`
- `public_input`
- `random_access`
- `reducing`
- `reducing_extension`

Arithmetic gates

These evaluate the constraint $w = c_0xy + c_1z$, either in the base field or in the quadratic extension field, possibly in many copies, but with shared $c_0, c_1 \in \mathbb{F}$ (these seem to be always in the base field?)

Note: in ArithmeticExtensionGate, since the constraints are normally already calculated in the extension field, we in fact compute in a doubly-extended field $\widetilde{\mathbb{F}}[Y]/(Y^2 - 7)$! Similarly elsewhere when we reason about $\widetilde{\mathbb{F}}$.

Base sum gate

This evaluates the constraint $x = \sum_{i=0}^{k} a_i B^i$ (where B is the radix or base). It can be used for example for simple range checks.

The the coefficient ranges $0 \le a_i < B$ are checked very naively as

$$\forall\, i. \quad \prod_{k=0}^{B-1}(a_i - k) = 0$$

which is a degree B equation, so this is only useful for very small B-s (typically $B = 2$).

The corresponding witness row looks like $[x, a_0, a_1, \ldots a_{B-1}, 0, \ldots, 0]$.

Constant gate

A very simple gate enforcing the $x_i = c_i$. Here the c_i are the same type of constants as in the arithmetic circuit. The reason for the existence of this is presumably that the constant columns *are not routed columns*, otherwise you could use the permutation (wiring) argument to enforce such constants.

I'm not convinced this is the right design choice, but probably depends on the circumstances.

Coset interpolation gate

This gate interpolates a set of values $y_i \in \widetilde{\mathbb{F}}$ on a coset ηH of a small subgroup H (typically of size 2^4), and evaluates the resulting polynomial at an arbitrary location $\zeta \in \widetilde{\mathbb{F}}$.

The formula is the barycentric form of Lagrange polynomials, but slightly modified to be chunked and to be iterative.

This is used for recursion.

Exponentiation

This computes $y = x^k$ using the standard fast exponentiation algorithm, where k is number fitting into some number of bits (depending on the row width).

I believe it first decomposes k into digits, and then does the normal thing. Though for some reason it claims to be a degree 4 gate, and I think it should be degree 3… TODO: sanity check this

Lookups

There are two kind of lookup gates, one containing $(\mathsf{inp}, \mathsf{out})$ pairs, and the other containing $(\mathsf{inp}, \mathsf{out}, \mathsf{mult})$ triples.

Neither imposes any constraint, as lookups are different from usual gates, as their behaviour is hardcoded in the protocol.

The 2 gates (`LookupGate` for the one without multiplicities and `LookupTableGate` for the one with) encode the lookup usage and the table(s) themselves, respectively. Plonky2 uses a logarithmic derivative based lookup argument.

See Lookups.md for more details.

Multiplication extension gate

This is the same as `ArithmeticExtensionGate`, except that it misses the addition. So the constraint is $z = c_0xy \in \widetilde{\mathbb{F}}$. In exchange, you can pack 13 of these into 80 columns instead of 10.

Noop gate

This doesn't enforce any constraint. It's used as a placeholder so each row corresponds to exactly a single gate; and also lookup tables as implemented require an empty row.

Poseidon gate

This computes the Poseidon hash (with Plonky2's custom constants and MDS matrix).

The poseidon gate packs all the (inputs of the) nonlinear sbox-es into a 135 wide row, this results in the standard configuration being 135 advice columns.

Poseidon hash is used for several purposes: hashing the public inputs into 4 field elements; the recursion verification of FRI; and generating challenges in the verifier.

See Poseidon.md for more details.

Poseidon MDS gate

This simply computes the multiplication by the 12x12 MDS matrix (so all linear constraints), but with the input vector consisting of field extension elements. It is used in the recursive proof circuit.

Public input gate

This simply packs the hash of the public inputs (4 field elements) into the first four cells of a row.

The actual public inputs are presumably at arbitrary locations in the routed advice wires; then Poseidon gates are automatically added and wired to compute the hash of those. The final hash result is "wired" to be equal to these values.

Finally these four values are constrained (with 4 linear equations) to be the actual hash values, which I guess is how the verifier checks these (that is, the hash values are hardcoded in the equations).

Random access gate

This gate allows dynamically indexing (relatively short) vector of size 2^n.

The idea is to decompose the index into bits $\mathsf{idx} = \sum b_i 2^i$; then if you can write down "selectors" like for example

$$S_{11} := S_{\mathtt{0b1011}} := b_0(1-b_1)b_2b_3 = \begin{cases} 1, & \text{if} \quad \mathsf{idx} = \mathtt{0b1011} = 11 \\ 0, & \text{if} \quad \mathsf{idx} \neq 11 \end{cases}$$

Then $A_{\mathsf{idx}} = \sum_{i=0}^{2^n-1} S_i \cdot A_i$.

Or at least that's how I would do it :)

The degree of the gate is $n+1$, so they probably inline the above selector definitions.

In the actual implementation, they repeat this as many times as it fits in the routed wires, followed by (at most) 2 cells used the same way as in the constant gate, finally followed by the bit decompositions

So the cells look like (for `n=4`): `copies ++ consts ++ bits` with

```
copies = [ i,a[i],a0,...,a15; j,b[j],b0,...b15; ... ]
consts = [ c0, c1 ]
bits   = [ i0,i1,2,i3; j0,j1,j2,j3; ...; 0... ]
```

Reducing gates

These compute $y = \sum_{i=0}^{k} \alpha^i \cdot c_i$ with the coefficients c_i in either the base field or the extension field, however with $\alpha \in \widetilde{\mathbb{F}}$ always in the extension field.

It assumes that everything fits into a single row.

Selectors and constants

The circuit descriptions consists not only of the permutation, but also the gate selectors and gate constants.

(There are also "lookup selectors", for those see Lookups)

Typically we have 2 or 3 selector columns (to switch between the different gates) and 2 constant columns (for the constants of the arithmetic and other gates).

For example in the Fibonacci example circuit uses 5 gates, which are:

- `NoopGate`
- `ConstantGate { num_consts: 2 }`
- `PublicInputGate`
- `ArithmeticGate { num_ops: 20 }`
- `PoseidonGate(...)<WIDTH=12>`

(in fact the `NoopGate` is used only when changing from the 100th Fibonacci number to say the 200th one, because apparently the 100 example just fills the trace exactly...)

This can be seen for example by serializing the `CommonCircuitData` struct.

The Poseidon gate is present because Plonky2 handles public input by always hashing it into 4 field element (encoded in the public input gate).

These gates are numbered `0..4`. Looking at the selector polynomials in the Fibonacci example, in the first one we see a lot of `3` (arithmetic gates, encoding the Fibonacci computation); a `UNUSEDGATE = 2^32-1`, which is a placeholder for gates not using the given selector polynomial, a 2 (public input), an 1 (constants), and some 0-s (no-op). In the other selector columns, it's all UNUSED except one 4 (Poseidon hash gate), exactly where the other one is unused.

We can see that a full selector column is only used for Poseidon, while the first column is used for everything else.

This makes sense, as the selector strategy of Plonky2 is the following: Enumarates the gates `0..#ngates-1`; it then groups as many gates into a single selector as many is possible with the equation degree limit. If you have `K` gates gate_i in a single selector column, then the corresponding selector polynomials have degree `K`:

$$S_k(x) = \prod_{i \neq k} \frac{U(x) - \mathsf{gate}_i}{\mathsf{gate}_k - \mathsf{gate}_i} \qquad S_k(\omega^i) = \begin{cases} 1, & \text{if } S(\omega^i) = \mathsf{gate}_k \\ ?, & \text{if } S(\omega^i) = 2^{32} - 1 \\ 0, & \text{otherwise} \end{cases}$$

where $U(x)$ is the (degree $N-1$) polynomial encoding the given selector column. Observe that the normalization is actually not necessary (as the gate equations are all normalized to zero) and Plonky2 doesn't do it. The actual polynomials used by Plonky2 are instead

$$\mathcal{S}_k(x) := \big((2^{32} - 1) - U(x)\big) \cdot \prod_{i \neq k} \big(\mathsf{gate}_i - U(x)\big)$$

The factor cancelling unused gates is only added if there are more than 1 selector columns, but because the Poseidon gate is always included (to handle the public input), and has degree 7, this is always the case.

The Poseidon gate has degree 7, while the degree limit is 8, so we can only use a degree 1 or 2 selector there (note that the degree limit is for the *quotient* polynomial, which is always "one less". Though it seems to me that the code has ± 1 error here, and as a result is overly cautious...).

The arithmetic gate has degree 3 (because $\deg(c_0 xy) = 3$: the constant coefficients also count!); the noop has degree 0 and both the constant and public input gates have degree 1. As $4 + \max(3, 0, 1, 1) = 7 \leq 9$ this still fits.

The constant columns contain the c_0, c_1 constants for the arithmetic gates (they are all 1 here); also the values for the constant gates. For the remaining gates (Poseidon, public input and noop) they are simply set to zero.

Gate constraints

Each type of gate (see Gates.md for the different gates supported by Plonky2) is defined by a set of polynomial equations (up to degree 8, though via a usual ± 1 error Plonky2 may restrict this to degree 7) whose variables are cells in a single row in the witness.

TODO: continue

Wiring (or permutation argument)

The wiring constraints enfore equalities between concrete cells in the (routed) witness columns (by default the first 80 columns out of the 135 in total).

This is done essentially the same way as in the original Plonk paper, except generalized for more columns (80 instead of 3), and reducing the number of required running product columns because Plonky2 needs high-degree constraints anyway.

Sub-protocol

The verifier samples $\beta_i, \gamma_i \in \mathbb{F}_p$ challenges for the permutations. There are $r =$ `num_challenges` number of these, which is set so that $(\deg /|\mathbb{F}|)^r$ is small enough. Typically $r \in \{2, 3\}$, in this case $r = 2$.

If "lookup gates" are present, the corresponding Δ_j challenges ($4r$ in total, but $2r$ reuses β_i, γ_i) are also sampled at this point.

The permutation σ polynomials are encoded on as many disjoint cosets $k_i H$ as many routed columns are. The $k_i \in \mathbb{F}$ is chosen simply as g^i with g being a multiplicative generator of $\mathbb{F}^\times$ (however, these k_i-s are listed in the `CircuitCommonData`, with the very helpful name `"k_is"`...) They use the following generators:

$$
\begin{aligned}
g &= \texttt{0xc65c18b67785d900} = 14293326489335486720 \\
h &= \texttt{0x64fdd1a46201e246} = 7277203076849721926 = g^{(p-1)/2^{32}} \\
\omega &= h^{(2^{32}/2^n)} \qquad \text{where} \qquad H = \langle \omega \rangle
\end{aligned}
$$

(remark: the smallest generator would be $g = 7$).

So the Plonk permutation σ is a permutation of $[N] \times [M]$ where there are $N = 2^n$ rows and M routed columns (in the default configuration we have $M = 80$. The cells indices are then mapped into $\mathbb{F}$ by

$$
\begin{aligned}
\phi : [N] \times [M] &\longrightarrow \mathbb{F}^\times \\
(i \;\; , \;\; j) &\longmapsto k_j \cdot \omega^i
\end{aligned}
$$

where ω is the generator of the subgroup $H \subset \mathbb{F}^\times$ of size n.

We can then form the $2M$ polynomials encoding the permutation (these are part of the fixed circuit description - though in practice we don't store S_{id} as that's easy to compute):

$$
\begin{aligned}
S_{\mathsf{id}}^{(j)}(\omega^i) &:= \phi(\;\;(i,j)) \\
S_{\sigma}^{(j)}(\omega^i) &:= \phi(\sigma(i,j))
\end{aligned}
$$

Next, we would normally compute the partial products

$$
A_k := \prod_{i=0}^{k-1} \prod_{j=0}^{M-1} \frac{W_{i,j} + \beta \cdot \phi((i,j)) + \gamma}{W_{i,j} + \beta \cdot \phi(\sigma(i,j)) + \gamma}
$$

for $k \leq 0 < M$, exactly as in the Plonk protocol. However, a problem with this is that corresponding equations would have degree M, which is too big (recall that in our case $M = 80$).

The actual layout

So what Plonky2 does, is to just enumerate all the $N \times M$ terms

$$
T_{i,j} := \frac{W_{i,j} + \beta \cdot \phi((i,j)) + \gamma}{W_{i,j} + \beta \cdot \phi(\sigma(i,j)) + \gamma}
$$

into chunks determined by the maximal degree we allow (`max_quotient_degree_factor = 8`). Note: This can be confusing when looking at the Fibonacci example, as there we also have $N = 2^3 = 8$, but this is just a coincidence!

So we should get NM/maxdeg partial products. In our case $M/\mathsf{maxdeg} = 80/8 = 10$, so we can organize this into an $N \times 10$ matrix (in row-major order), resulting 10 "partial product columns".

Essentially the 80 routed columns' permutation argument is compressed into 10 partial product columns, and we will have equations ensuring that these are constructed correctly, and that the full product is 1 (which in turn proves the wire constraints).

Note: the Plonky2 source code uses some absolute horrible names here, and then does shiftings, reorderings, basically moving the last column to the first one, and calls this "z" while the rest "partial products", and then reorders even these between the challenge rounds; but I think this is just a +1 error and programmers not understanding what they are doing... Note that the first column needs to also opened "on the next row", while the rest only on "this row", but really, that's not a valid reason to do all this shit.

Here is an ASCII graphics explaining what happens in the source code (indices denote the corresponding partial product):

```
                                                        zs | partial_products
+------------------+      +------------------+      +----+---------------+
|  1  2 ...   9 10 |      |  1  2 ...   9  0 |      |  0 |  1  2 ...   9 |
| 11 12 ...  19 20 |      | 11 12 ...  19 10 |      | 10 | 11 12 ...  19 |
| 21 22 ...  29 30 |  ->  | 21 22 ...  29 20 |  ->  | 20 | 21 22 ...  29 |
| 31 32 ...  39 40 |      | 31 32 ...  39 30 |      | 30 | 31 32 ...  39 |
|                  |      |              40 |      | 40 |               |
```

First, the partial products are generated as in the first table. The final one (in the bottom-right corner) should be equal to 1. Then, the last column is shifted down, with constant 1 coming (denoted by 0 index) appearing in the top-right corner. Then, the last column is moved in the front. Finally (no picture for the lack of space), the first columns are separated from the rest and bundled together, so for r challeges you will get r first columns (called `"zs"`), and then $9r$ of the remaning columns (called `"partial products"`). Seriously, WTF.

In any case, then we commit to these (reordered!) $10 \times r$ columns (together, as before). If there are lookup gates, the corresponding polynomials are also included here. The commitments are added to the Fiat-Shamir transcript too.

Constraints

There are two types constraints (repeated for all challenge rounds): that the starting (and also ending, because of the cyclic nature of the multiplicative subgroup) value should equal to 1:

$$\mathcal{L}_0(x) \cdot \big[A_0(x) - 1\big]$$

where $\mathcal{L}$ denotes the Lagrange polynomials and $A_0(x)$ denotes the first column; and 10 transition constraints (one for each column), ensuring that the partial products are formed correctly:

$$A_{i+1}(x) = A_i(x) \cdot \prod_{j=0}^{7} \frac{W_{8i+j} + \beta \cdot k_{8i+j} x + \gamma}{W_{8i+j} + \beta \cdot \Sigma_{8i+j}(x) + \gamma}$$

where A_i denotes the i-th partial product column – with the convention $A_{10}(x) = A_0(\omega x)$ to simplify the notation –, and Σ_j denotes the "sigma" columns encoding the permutation.

Multiplying with the denominator we get a degree 9 constraint, but that's fine because the quotient polynomial will have only degree 8.

Poseidon hash

Plonky2 primarily uses the Poseidon hash function (while Keccak is also supported in the code, as we are interested in recursive proofs, that's not really relevant here).

Poseidon itself is based on the sponge construction: it's defined by a concrete, fixed permutation $\pi : \mathbb{F}^t \to \mathbb{F}^t$ (in our case $t = 12$).

Using the permutation

For linear hashing, Plonky2 uses the permutation in standard sponge fashion (with `rate=8` and `capacity=4`), except in "overwrite mode". Plonky2 does not use any padding, but normally all the inputs should have constant length. The only exception seems to be the Fiat-Shamir domain separator, which is unused by default.

Here overwrite mode means that instead of adding in the new input to the existing state, it overwrites it. It's claimed that this is still secure (see this paper from the Keccak authors).

Hashing in linear mode is used when committing the matrices (constants, witness, permutation argument, quotient polynomial): First the rows of the (LDE extensions of the) matrices are hashed, and then a Merkle tree of the size corresponding to the number of rows is built on the top of these hashes. This means you have to open full rows, but normally you want to do that anyway.

For Merkle tree compression function, the standard $f(x, y) := \pi(x, y, 0)_0$ construction is used.

To generate challenges, Plonky2 uses the permutation in an "overwrite duplex" mode. Duplex means that input and output is interleaved (this makes a lot of sense for IOPs). This is implemented in `iop/challenger.rs`.

Poseidon gate

A Poseidon permutation is implemented as a "custom gate", in a single row of size 135 (each of the 135 cells containing a field element). These essentially contain the inputs of the sboxes, and you have a separate equation for each such cell (except the inputs, so `135 - 12 = 123` equations). The cells are:

- 12 cells for the permutation input

- 12 cells for the permutation output
- 5 cells for "swapping" the inputs of the Merkle compression function (1 indicating a swap bit, and 4 deltas). This is to make Merkle proofs easier and/or more efficient.
- 3×12 cells for the inputs of the initial full layer sbox-es (the very first ones are linear in the input so don't require separate cells)
- 22 cells for the inputs of the inner, partial layer sbox-es
- 4×12 cells for the inputs of the final full layer sbox-es

Adding these together you get 135, which is also the number of columns in the standard recursion configuration. An advantage of this "wide" setting, is that a whole row is a leaf of the Merkle tree, so the Merkle tree is shallower.

Some remarks:

- When using "standard Plonk gates" or other less wide custom gates, you can pack many of those (in a SIMD-like fashion) in a single row. This allows the large number of columns not being (too) wasteful
- The newer Plonky3 system in fact packs several permutations in a single row (in the examples, 8 permutations). Plonky3 is significantly faster, this may be one of the reasons (??)
- Deriving the equations for each cell (each sbox) can be done by relatively simple computer algebra. However, this is not done anywhere in the Plonky2 codebase. Instead, an algorithm evaluating these constraints is written manually as native code. Indeed, if we would naively write down the equations as polynomials, the size of the resulting polynomials would blow up - you actually need sharing of intermediate results to be able to evaluate them effectively.
- Plonky2 uses their own, non-standard round constants (generated by chacha20), and their own MDS matrix too.

See `gates/poseidon.rs` for the details.

Poseidon MDS gate

There is also a second Poseidon-related gate. Apparently this is used in recursive proof circuits. It appears to do simply a multiplication by the MDS matrix (but in the extension field?)

TODO: figure out why and how is this used

FRI protocol

Plonky2 uses a "wide" FRI commitment (committing to whole rows), and then a batched opening proofs for all the 4 commitments (namely: constants, witness, running product and quotient polynomial).

Initial Merkle commitment

To commit to a matrix of size $2^n \times M$, the columns, interpreted as values of polynomials on a multiplicative subgroup, are "low-degree extended", that is, evaluated (via an IFFT-FFT pair) on a (coset of a) larger multiplicative subgroup of size $2^{n+\mathsf{rate}^{-1}}$. In the standard configuration we have $\mathsf{rate} = 1/8$, so we get 8x larger columns, that is, size 2^{n+3}. The coset Plonky2 uses is the one shifted by the multiplicative generator of the field

$$g := \texttt{0xc65c18b67785d900} = 14293326489335486720 \in \mathbb{F}$$

Note: There may be some reordering of the LDE values (bit reversal etc) which I'm unsure about at this point.

When configured for zero-knowledge, each *row* (Merkle leaf) is "blinded" by the addition of `SALT_SIZE = 4` extra random *columns* (huh?).

Finally, each row is hashed (well, if the number of columns is at most 4, they are left as they are, but this should never happen in practice), and a Merkle tree is built on the top of these leaf hashes.

So we get a Merkle tree whose leaves correspond to full rows (2^{n+3} leaves).

Note that in Plonky2 we have in total 4 such matrices, resulting in 4 Merkle caps:

- constants (including selectors and sigmas)
- the witness (135 columns)
- the wiring (and lookup) protocol's running products (resp. sums)
- the quotient polynomial

Merkle caps Instead of using a single Merkle root to commit, we can use any fixed layer (so the commitment will be 2^k hashes instead of just 1). As the paths become shorter, this is a tradeoff between commitment size and proof size / verification cost. In case we have a lot of Merkle openings for a given tree (like in FRI low-degree proofs here), clearly this makes sense. In the default configuration Plonky2 uses Merkle caps of size $2^4 = 16$

FRI configuration

An FRI proof is configured by the following parameters:

```
struct FriConfig {
  rate_bits:  usize,        // rate = 2^{-rate_bits}.
  cap_height: usize,        // Height of Merkle tree caps.
  proof_of_work_bits: u32,  // Number of bits used for grinding.
  num_query_rounds: usize,  // Number of query rounds to perform.
  reduction_strategy: FriReductionStrategy,
}
```

Here the "reduction strategy" defines how to select the layers. For example it can always do 8->1 reduction (instead of the naive 2->1), or optimize and have different layers; also where to stop: If you already reduced to say a degree 3 polynomial, it's much more efficient to just send the 8 coefficients than doing 3 more folding steps.

The "default" `standard_recursion_config` uses rate = 1/8 (rate_bits = 3), markle cap height = 4, proof of work (grinding) = 16 bits, query rounds = 28, reduction startegy of arity 2^4 (16->1 folding) and final polynomial having degree (at most) 2^5. For example for a recursive proof fitting into 2^{12} rows, we have the degree sequence $2^{12} \to 2^8 \to 2^4$, with the final polynomial having degree $2^4 = 16 \leq 2^5$

For recursion you don't want fancy reduction strategies, it's better to have something uniform.

Grinding is used to improve security. This means that the verifier sends a challenge $x \in \mathbb{F}$, and a prover needs to answer with a witness $w \in \mathbb{F}$ such that $H(x\|w)$ starts with as many zero bits as specified.

The conjectured security level is apparently `rate_bits * num_query_rounds + proof_of_work_bits`, in the above case $3 \times 28 + 16 = 100$. Plonky2 targets 100 bits of security in general. Remark: this is measured in number of hash invocations.

FRI proof

An FRI proof consists of:

```
struct FriProof<F: RichField + Extendable<D>, H: Hasher<F>, const D: usize> {
  commit_phase_merkle_caps: Vec<MerkleCap<F, H>>,  // A Merkle cap for each reduced polynomi
  query_round_proofs: Vec<FriQueryRound<F, H, D>>, // Query rounds proofs
  final_poly: PolynomialCoeffs<F::Extension>,      // The final polynomial in coefficient fo
  pow_witness: F,                                  // Witness showing that the prover did Po
}
```

The type parameters are: `F` is the base field, `D` is the extension degree (usually `D=2`), and `H` is the hash function.

During both the proof and verification process, the verifier challenges are calculated. These are:

```
struct FriChallenges<F: RichField + Extendable<D>, const D: usize> {
  fri_alpha: F::Extension,         // Scaling factor to combine polynomials.
  fri_betas: Vec<F::Extension>,    // Betas used in the FRI commit phase reductions.
  fri_pow_response: F,             // proof-of-work challenge
  fri_query_indices: Vec<usize>,   // Indices at which the oracle is queried in FRI.
}
```

Here powers of $\alpha \in \widetilde{\mathbb{F}}$ is used to combine several polynomials into a single one, and $\beta_i \in \widetilde{\mathbb{F}}$ are the coefficients used in the FRI "folding steps". Query indices (size is `num_query_rounds`) is presumably the indices in the first layer LDE where the Merkle oracle is queried.

See Challenges.md for how these Fiat-Shamir challenges are generated.

Remark: **`batch_fri`**: There is also `batch_fri` subdirectory in the repo, which is not clear to me what actually does, as it doesn't seems to be used anywhere…

The FRI protocol

The FRI protocol proves that a committed Reed-Solomon codeword, which is a priori just a vector, is in fact "close" to a codeword (with high probability).

This is done in two phases: The commit phase, and the query phase. Note that this "commit" is not the same as the above commitment to the witness etc!

In Plonky2, we want to execute this protocol on many polynomials (remember that each column is a separate polynomial)! That would be very expensive, so instead they (well, not exactly them) are combined by the prover with the random challenge α, and the protocol is executed on this combined polynomial.

Combined polynomial So we want to prove that $F_i(x_i) = y_i$ where $F_i(X)$ are a set of (column) polynomials. In our case $\{F_i\}$ consists of two *batches*, and $x_i \in \{\zeta, \omega\zeta\}$ are constants on the batches. The first batch of the all the column polynomials, the second only those which needs to be evaluated at $\omega\zeta$ too (`"zs"` and the lookup polynomials). We can then form the combined quotient polynomial:

$$P(X) := \sum_i \alpha^i \frac{F_i(X) - y_i}{X - x_i} = \sum_b \frac{\alpha^{k_b}}{X - x_b} \sum_{j=0}^{m_b} \alpha^j \left[F_{b,j}(X) - y_{b,j}\right]$$

In practice this is done per batch (see the double sum), because the division is more efficient that way. This polynomial $P(X)$ is what we execute the FRI protocol on, proving a degree bound.

Remark: In the actual protocol, F_i will be the columns and y_i will be the openings. Combining the batches, we end up with 2 terms:

$$P(X) = P_0(X) + \alpha^M \cdot P_1(X) = \frac{G_0(X) - Y_0}{X - \zeta} + \alpha^M \cdot \frac{G_1(X) - Y_1}{X - \omega\zeta}$$

The pair (Y_0, Y_1) are called "precomputed reduced openings" in the code (calculated from the opening set, involving *two rows*), and X will be substituted with $X \mapsto \eta^{\mathsf{query\_index}}$ (calculated from the "initial tree proofs", involving *one row*). Here η is the generator of the LDE subgroup, so $\omega = \eta^{1/\rho}$.

Commit phase Recall that we have a RS codeword of size $2^{n+(1/\rho)}$ (encoding the combined polynomial $P(X)$ above), which the prover committed to.

The prover then repeatedly "folds" these vectors using the challenges β_i, until it gets something with low enough degree, then sends the coefficients of the corresponding polynomial in clear.

As example, consider a starting polynomial of degree $2^{13}-1$. With $\rho = 1/8$ this gives a codeword of size 2^{16}. This is committed to (but the see the note below!). Then a challenge β_0 is generated, and we fold this (with an arity of 2^4), getting a codeword of size 2^{12}, representing a polynomial of degree 2^9-1. We commit to this too. Then generate another challenge β_1, and fold again with that. Now we get a codeword of size 2^8, however, this is represented by a polynomial of at most degree 31, so we just send the 32 coefficients of that instead of a commitment.

Note: as an optimization, when creating these Merkle trees, we always put *cosets* of size 2^{arity} on the leaves, as we will have to open them all together anyway. Furthermore, we use *Merkle caps*, so the proof lengths are shorter by the corresponding amount (4 by default, because we have 16 mini-roots in a cap). So the Merkle proofs are for a LDE size 2^k have length $k - \mathsf{arity\_bits} - \mathsf{cap\_bits}$, typically $k-8$.

step	Degree	LDE size	Tree depth	prf. len	fold with	send & absorb
0	$2^{13}-1$	2^{16}	12	8	β_0	Merkle cap
1	$2^{9}-1$	2^{12}	8	4	β_1	Merkle cap
2	$2^{5}-1$	2^{8}	–	–	–	2^5 coeffs

Grinding At this point (why here?) the grinding challenge is executed.

Query phase This is repeated `num_query_rounds = 28` times (in the default configuration).

A query round consists of two pieces:

- the initial tree proof
- and the folding steps

The initial tree proof consist of a single row of the 4 LDE matrices (the index of this row is determined by the query index challenges), and a Merkle proof against the 4 commited Merkle caps.

The steps consist of pairs of evaluations on cosets (of size 2^{arity}) and corresponding Merkle proofs against the commit phase Merkle caps.

From the "initial tree proof" values $\{F_j(\eta^k)\}$ and the openings $\{y_j, z_j\}$, we can evaluate the combined polynomial at $\eta^k := \eta^{\mathsf{query\_idx}}$:

$$P(\eta^k) = \frac{1}{\eta^k - \zeta} \sum_{j=0}^{M_1-1} \alpha^j \left[F_j(\eta^k) - y_j\right] + \frac{1}{\eta^k - \omega\zeta} \sum_{j=M_1}^{M_2-1} \alpha^j \left[F_j(\eta^k) - z_j\right]$$

Then in each folding step, a whole coset is opened in the "upper layer", one element of which was known from the previous step (or in the very first step, can be computed from the "initial tree proofs" and the openings themselves) which is checked to match. Then the folded element of the next layer is computed by a small 2^{arity} sized FFT, and this is repeated until the final step.

FRI verifier cost

We can try and estimate the cost of the FRI verifier. Presumably the cost will be dominated by the hashing, so let's try to count the hash permutation calls. Let the circuit size be $N = 2^n$ rows.

- public input: $\lceil \#\mathsf{PI}/8 \rceil$ (we hash with a rate 8 sponge)
- challenges: approximately 95-120. The primary variations seem to be number (and size) of commit phase Merkle caps, and the size of the final polynomial
- then for each query round (typically 28 of them), approx 40-100 per round:
  - check the opened LDE row against the 4 matrix commitments:
    - hash a row (typical sizes: 85, 135, 20, 16; resulting in 11, 17, 3 and 2 calls, respectively)
    - check a Merkle proof (size `n+3-4 = n-1`)
    - in total `33 + 4(n-1)` calls
  - check the folding steps
    - for each step, hash the coset (16 $\widetilde{\mathbb{F}}$ elements, that's 4 permutations)
    - then recompute the Merkle root: the first one is `n+3-8`, the next is `n+3-12` etc

For example in the case of a recursive proof of size $N = 2^{12}$, we have 114 permutation calls for the challenges, and then $28 \times (77 + 11 + 7)$, resulting in total

$$114 + 28 \times (77 + 11 + 7) = 2774$$

Poseidon permutation calls (with `t=12`), which matches the actual code.

We can further break down this to sponge vs compression calls. Let's concentrate on the FRI proof only, as that dominates:

- sponge is $(33 + 4 + 4 \ldots)$ per round
- compression is $4(\text{n - 1}) + (\text{n-5}) + (\text{n-9}) + \ldots$ per round

In case of $n = 12$, we have 41 (sponge) vs. 54 (compression). As compression is more than half of the cost, it would make sense to optimize that to `t=8` (say with the Jive construction); on the other hand, use a larger arity (say `t=16`) for the sponge.

It seems also worthwhile to use even wider Merkle caps than the default 2^4.

Soundness

Soundness is bit tricky because of the small field. Plonky2 uses a mixture of sampling from a field extension and repeated challenges to achieve a claimed ~100 bit security:

Sub-protocol	Soundness boost
Permutation argument	parallel repeatition
Combining constraints	parallel repeatition
Lookup argument	parallel repeatition
Polynomial equality test	extension field
FRI protocol	extension field + grinding

See this 2023 paper for more precise soundness arguments along these lines.

Fiat-Shamir Challenges

The verifier challenges are genered via Fiat-Shamir heuristics.

This uses the hash permutation in a duplex construction, alternatively absorbing the transcript and squeezing challenge elements. This is implemented in `iop/challenger.rs`.

All the challenges in the proof are summarized in the following data structure

```
struct ProofChallenges<F: RichField + Extendable<D>, const D: usize> {
  plonk_betas:    Vec<F>,         // Random values used in Plonk's permutation argument.
  plonk_gammas:   Vec<F>,         // Random values used in Plonk's permutation argument.
  plonk_alphas:   Vec<F>,         // Random values used to combine PLONK constraints.
  plonk_deltas:   Vec<F>,         // Lookup challenges (4 x num_challenges many).
  plonk_zeta:     F::Extension,   // Point at which the PLONK polynomials are opened.
  fri_challenges: FriChallenges<F, D>,
}
```

And the FRI-specific challenges are:

```
struct FriChallenges<F: RichField + Extendable<D>, const D: usize> {
  fri_alpha: F::Extension,         // Scaling factor to combine polynomials.
  fri_betas: Vec<F::Extension>,    // Betas used in the FRI commit phase reductions.
  fri_pow_response:  F,            // proof-of-work challenge response
```

```
  fri_query_indices: Vec<usize>,   // Indices at which the oracle is queried in FRI.
}
```

Duplex construction

TODO

Transcript

Usually the communication (in an IOP) between the prover and the verifier is called "the transcript", and the Fiat-Shamir challenger should absorb all messages of the prover.

The duplex state is initialized by absorbing the "circuit digest".

This is the hash of the following data:

- the Merkle cap of the constant columns (including the selectors and permutation sigmas)
- the *hash* of the optional domain separator data (which is by default an empty vector)
- the size (number of rows) of the circuit

Thus the challenge generation starts by absorbing:

- the circuit digest
- the hash of the public inputs
- the Merkle cap of the witness matrix commitment

Then the $\beta \in \mathbb{F}^r$ and $\gamma \in \mathbb{F}^r$ challenges are generated, where `r = num_challenges`.

If lookups are present, next the lookup challenges are generated. This is a bit ugly. We need $4 \times r$ such challenges, but as an optimization, the β, γ are reused. So $2 \times r$ more δ challenges are generated, then these are concatenated into $(\beta \| \gamma \| \delta) \in \mathbb{F}^{4r}$, and finally this vector is chunked into r pieces of 4-vectors…

Next, the Merkle cap of the partial product columns is absorbed; and after that, the $\alpha \in \mathbb{F}^r$ combining challenges are generated.

Then, the Merkle cap of the quotient polynomials is absorbed, and the $\zeta \in \widetilde{\mathbb{F}}$ evaluation point is generated.

Finally, the FRI challenges are generated.

FRI challenges

First, we absorb all the opening (a full row, involving all the 4 committed matrix; and some parts of the "next row").

Then the $\alpha \in \widetilde{\mathbb{F}}$ combining challenge is generated (NOTE: this is different from the above α-s!)

Next, the `commit_phase_merkle_caps` are absorbed, and after each one, a $\beta_i \in \widetilde{\mathbb{F}}$ is generated (again, different β-s from above!).

Then we absorb the coefficients of the final (low-degree) folded FRI polynomial. This is at most $2^5 = 32$ coefficients in the default configuration.

Next, the proof-of-work "grinding" is handled. This is done a bit strange way: *first* we absorb the candidate prover witness, *then* we generate the response, and check the leading zeros of that. I guess you can get away with 1 less hashing in the verifier this way…

Finally, we generate the FRI query indices. These are indices of rows in the LDE matrix, that is, $0 \leq q_i < 2^{n+\texttt{rate\_bits}}$.

For this, we generate `num_query_rounds` field elements, and take them modulo this size.

The Plonky2 protocol (IOP)

I try to collect together here the phases of the Plonky2 protocol. *Some* of this is very briefly described in the only existing piece of official documentation, the "whitepaper".

Circuit building

First the static parameters of the circuit are determined:

- global parameters
- set of custom gates
- number of rows (always a power of two)
- gate selectors & gate constants
- wiring permutation
- optional lookup selectors

The constant columns are committed (see also Layout.md for the details of these), and the circuit digest (hash) is computed. The latter seems to be used primarily to seed the Fiat-Shamir challenge generator.

Witness generation

The $2^n \times M$ witness matrix is generated based on the circuit.

This then again committed via FRI (see FRI.md for the details).

Notes: - in practice, we want to pre-commit to the constant columns, as these describe the circuit itself. - full rows of the (LDE extended) matrices are hashed using the sponge construction, and an Merkle tree is built on these rows, resulting in Merkle root commitment. - in practice however instead a single root, a wider "Merkle cap" is used (this is an optimization)

At this point, the circuit digest, the hash of the public inputs, and these Merkle caps are added to the Fiat-Shamir transcript (which was probably empty before, though Plonky2 has support for initializing with a "proof domain separator").

Gate constraints

For each gate type, a set of gate equations ("filtered" using the gate selector polynomials) are evaluated at $\zeta \in \widetilde{\mathbb{F}}$, and these are combined "vertically" by simple addition; so we get as many values as the maximum number of equations in a single gate.

This unusual combination (simple summation) is safe to do because the selector polynomials ensure that on each element of the multiplicative subgroup, at most 1 gate constraint do not vanish. And what we prove at the end is that (with very high probability) the resulting (vertically combined) constraints vanish on the whole multiplicative subgroup (but we do this by evaluating outside this subgroup).

See GateConstraints.md for the details.

Permutation argument

The verifier samples $\beta_i, \gamma_i \in \mathbb{F}_p$ challenges for the permutations. There are $r =$ `num_challenges` number of these, which is set so that $(\deg / |\mathbb{F}|)^r$ is small enough. See (see Challenges.md for the details).

If there are lookups, the lookup challenges are also sampled here, $4 \times r$ ones. However, as an inelegant optimization, Plonky2 reuses the already computed $2r$ challenges β_i, γ_i for this, so there are only $(4r - 2r) = 2r$ new ones generated. These are called δ_j.

As many "sigma columns" as there are routed advice columns (by default 80) were computed (when buildig the circuit), encoded as though the different routed columns correspond to different cosets.

The running product vectors are computed; these are compressed by adding as many terms in a single step as the degree limit allows (by default 8), so from $8k$ routed columns we get k running product columns. But this is again repeated r times.

See Wiring.md for the details how it's done

These are also committed to. So we will have (at the end) the following 4 commitments:

- constants and sigmas (only depends on the circuit)
- witness (or wires)
- running products and lookups
- quotient polynomial chunks (see below)

Lookups

When lookups are present, this is done next to the permutation argument. See Lookups.md for the details.

Combined constraints

Verifier samples $\alpha_i \in (\mathbb{F}_p)^r$ challenges to combine the constraints.

All constraints are combined via powers of α, namely (in this order):

- the grand products starts/ends with 1
- partial products are built correctly - see Wiring.md
- lookups - see Lookups.md
- all the gate constraints - see GateConstraints.md

Quotient polynomials

Prover computes the combined quotient polynomial(s), by which we mean 1 quotient polynomial per challenge rounds, so in total r many. These are then partitioned into degree N chunks:

$$Q(x) = \sum_{k=0}^{\mathsf{maxdeg}-1} x^N \cdot Q^{(k)}(x)$$

and these are committed to in the usual way (so this should be at most $r \times \mathsf{maxdeg}$ columns).

Evaluation

Verifier samples $\zeta \in \widetilde{\mathbb{F}}$ evaluation challenge (in the 128 bit extension field!).

We open all commitments, that is (see Layout.md for more details):

- selectors & circuit constants & sigmas (in a typical case, $2+2+80$ columns)
- witness wires (135 columns)
- partial products ($r \times 10$ columns)
- optionally, lookup RE and partial sums ($r \times (1+6)$ columns, if presents)
- quotient chunks ($r \times \mathsf{maxdeg}$ columns)

at this challenge ζ, and the case of the first column "zs" of the partial products (and also the lookup ones), also at $\omega \cdot \zeta$.

Corresponding (batched) FRI evaluation proofs are also produced for all these.

That's basically all the prover does.

Verifier

The verifier essentially does three things (?):

- checks the FRI opening proofs
- compute combined constraints (one for each challenge round $1 \dots r$) at ζ
- finally check the quotient equation $\mathcal{Q}(\zeta)Z_H(\zeta) = \mathcal{P}(\zeta)$

Note: the quotient polynomial is "chunked", so the verifier needs to reconstruct it as

$$\mathcal{Q}(\zeta) = \sum_{k=0}^{\mathsf{maxdeg}-1} \zeta^k \cdot Q_k(\zeta)$$

TODO: details

Lookup tables

Plonky2 has added support lookup tables only relatively recently, and thus it is not documented at all (not even mentioned in the "whitepaper", and no example code either).

Logarithmic derivatives

Plonky2 doesn't use the classical `plookup` protocol, but the more recent approach based on the paper Multivariate lookups based on logarithmic derivatives, with the implementation following the Tip5 paper.

In logarithmic derivative based lookups, the core equation is

$$\mathsf{LHS}(X) := \sum_{j=0}^{K-1} \frac{1}{X - w_j} = \sum_{i=0}^{N-1} \frac{m_i}{X - t_i} := \mathsf{RHS}(X)$$

where w_j is the witness, t_i is the table, and we want to prove that $\{w_j\} \subset \{t_i\}$, by providing the multiplicities m_i of w_i appearing in $\{t_i\}$. This then becomes a scalar equation, after the subtitution $X \mapsto \alpha$, where $\alpha \in \mathbb{F}$ is a random number (chosen by the verifier).

Note that the above equation is simply the logarithmic derivative (wrt. X) of the more familiar permutation argument:

$$\prod_j (X - w_j) = \prod_i (X - t_i)^{m_i}$$

The reason for the logarithmic derivative is to move the multipliers from the exponent to a simple multiplication.

Binary lookups

Plonky2 *only* allows lookups of the form `input->output`, encoding equations like $y = f(x)$ where the function f is given by the table. In this case we have $t_i := x_i + a \cdot y_i$ (for some verifier-chosen $a \in \mathbb{F}$) and similarly for the witness.

Gates

Plonky2 uses two types of gates for lookups:

- `LookupGate` instances encode the actual lookups
- while `LookupTableGate` instances encode the tables themselves

Each `LookupGate` can contain up to $\lfloor 80/2 \rfloor = 40$ lookups, encoded as $(\mathsf{inp}, \mathsf{out})$ pairs; and each `LookupTableGate` can contain up to $\lfloor 80/3 \rfloor = 26$ table entries, encoded as $(\mathsf{inp}, \mathsf{out}, \mathsf{mult})$ triples.

This (questionable) design decision has both advantages and disadvantages:

- tables can be encoded in less rows than their size, allowing them to be used in small circuits
- up to 40 lookups can be done in a single row; though they are repeated at their actual usage locations, and need to be connected by "wiring"
- on the other hand, encoding the tables in the witness means that the verifier has to do a relatively expensive (linear in the table size!) verification of the tables
- and the whole things is rather over-complicated

The idea behind the protocol

Four challenges (repeated `num_challenges` times) are used in the lookup argument: $a, b \in \mathbb{F}$ are used to combine the input and output, in the above argument and the "table consistency argument", respectively; $\alpha \in \mathbb{F}$ is the above random element; and $\delta \in \mathbb{F}$ is the point to evaluate the lookup polynomial on. NOTE: in the actual implementation, two out of the four are reused from the from permutation argument (called β, γ there).

A sum like the above can be computed using a running sum, similar to the running product used in the permutation argument:

$$U_k := \sum_{i=0}^{k-1} \frac{\mathsf{mult}_i}{\alpha - (\mathsf{inp}_i + a \cdot \mathsf{out}_i)}$$

Then the correctness can be ensured by a simple equation:

$$(U_{k+1} - U_k) \cdot \big(\alpha - (\mathsf{inp}_i + a \cdot \mathsf{out}_i)\big) = \mathsf{mult}_i$$

with the boundary constraints $U_0 = U_{N+K} = 0$, if we merge the LHS and the RHS of the original into a single sum of size $N + K$.

Similarly to the permutation argument, Plonky2 batches several such "running updates" together, since it's already have high-degree constraints:

$$U_{k+d} - U_k = \sum_{i=0}^{d-1} \frac{\mathsf{mult}_{k+i}}{\alpha - \mathsf{inp}_{k+i} - a \cdot \mathsf{out}_{k+i}} = \frac{\sum_{i=k}^{k+d-1} \mathsf{mult}_i \cdot \prod_{j \neq i} (\alpha - \mathsf{inp}_i - a \cdot \mathsf{out}_i)}{\prod_{i=k}^{k+d-1} (\alpha - \mathsf{inp}_i - a \cdot \mathsf{out}_i)}$$

Also similarly to the permutation argument, they don't start from "zero", and reorder the columns so the final sum is at the beginning.

Consistency check

Since the lookup table itself is *part of the witness*, instead of being some pre-committed polynomials, the verifier needs to check the authenticity of that too. For this purpose, what Plonky2 does is to create new combinations (called "combos" in the codebase):

$$\mathsf{combo}'_i := \mathsf{inp}_i + b \cdot \mathsf{out}_i$$

then out of these, build a running sum of *partial Horner evaluations* of the polynomial whose coefficients are combo'_i:

$$\mathtt{RE}_n := \sum_{i=0}^{26n-1} \delta^{26n-1-i} \left(\mathsf{inp}_i + b \cdot \mathsf{out}_i\right)$$

(recall that in the standard configuration, there are $26 = \lfloor 80/3 \rfloor$ entries per row). This running sum is encoded in a single column (the first column of the lookup columns).

Then the verifier recomputes all this (which have cost linear in the size of the table!), and checks the initial constraint ($\mathtt{RE}_0 = 0$), the final constrint $\mathtt{RE}_N$ is the expected sum, and the transition constraints:

$$\mathtt{RE}_{n+1} = \mathtt{RE}_n + \sum_{j=0}^{25} \delta^{25-j} \cdot \mathsf{combo}'_{26n+j}$$

(note that this could be also written in Horner form).

Naming conventions

As the code is based (very literally) on the Tip5 paper, it also inherits some of the really badly chosen names for there.

- `RE`, short for "running evaluation", refers to the table consistency check; basically the evaluation of a polynomial whose coefficients are the lookup table "combos" (in reverse order…)
- `LDC`, short for "logarithmic derivative ???", refers to the sum of fractions (with coefficients 1) on the left hand side of the fundamental equation, encoding the usage
- `Sum`, short for "summation" (really?!), refers the sum of fractions (with coefficients mult_i) on the right hand side of the fundamental equation
- `SLDC` means `Sum - LDC` (we only compute a single running sum)
- `combo` means the linear combination $\mathsf{inp}_k + a \cdot \mathsf{out}_k$.

In formulas (it's a bit more complicated if there are more than 1 lookup tables, but this is the idea):

$$\begin{aligned}
\mathtt{RE}_{\text{final}} &= \sum_{k=0}^{K-1} \delta^{K-1-k} \left[\mathsf{inp}_k + b \cdot \mathsf{out}_k\right] \\
\mathtt{LDC}_{\text{final}} &= \sum_{j} \frac{1}{\alpha - (\mathsf{inp}_j + a \cdot \mathsf{out}_j)} \\
\mathtt{Sum}_{\text{final}} &= \sum_{k=0}^{K-1} \frac{\mathsf{mult}_k}{\alpha - (\mathsf{inp}_k + a \cdot \mathsf{out}_k)} \\
\mathtt{SLDC}_k &= \mathtt{Sum}_k - \mathtt{LDC}_k
\end{aligned}$$

Layout

Lookup tables are encoded in several `LookupTable` gates, which are just below each other, ordered from the left to right and **from bottom to top**. At the very bottom (so "before" the lookup table) there is an empty `NoopGate`. At the top of the lookup table gates are the actual lookup gates, but these are ordered **from top to bottom**. So the witness will look like this:

```
        |      ....      |
 ---    +----------------+
  v     |                |
  v     |     LU #1      |    <- lookups in the first  lookup table
  v     |                |
 ---    +----------------+
  ^     |                |
  ^     |                |
```

```
 ^      |    LUT #1     |     <- table entries (with  multiplicities)
 ^      |               |        of the first lookup  table
---     +---------------+
        |     NOOP      |     <- empty row (all zeros)
---     +---------------+
 v      |               |
 v      |     LU #2     |     <- lookups in the second lookup table
---     +---------------+
 ^      |               |
 ^      |    LUT #2     |     <- table entries in the second table
 ^      |               |
---     +---------------+
        |     NOOP      |     <- empty row
        +---------------+
        |     ....      |
```

Both type of gates are padded if necessary to get full rows. The `Lookup` gates are padded with the first entry in the corresponding table, while the `LookupTable` gates are padded with zeros (which is actually a soundness bug, as now you can prove that `(0,0)` is an element of any table whose length is not divisible by 26).

Lookup selectors

Lookups come with their own selectors. There are always `4 + #luts` of them, nested between the gate selector columns and the constant columns.

These encode:

- 0: indicator function for the `LookupTable` gates rows
- 1: indicator function for the `Lookup` gate rows
- 2: indicator for the empty rows, or `{last_lut_row + 1}` (a singleton set for each LUT)
- 3: indicator for `{first_lu_row}`
- 4,5…: indicators for `{first_lut_row}` (each a singleton set)

Note: the lookup table gates are in bottom-to-top order, so the meaning of "first" and "last" in the code are sometimes not switched up… I tried to use the logical meaning above.

In the code these are called, respectively:

- 0: `TransSre` is for Sum and RE transition constraints.
- 1: `TransLdc` is for LDC transition constraints.
- 2: `InitSre` is for the initial constraint of Sum and Re.
- 3: `LastLdc` is for the final (S)LDC constraint.
- 4: `StartEnd` indicates where lookup end selectors begin.

These are used in the lookup equations.

This ASCII art graphics should be useful:

```
     witness           0 1 2 3 4 5
+---------------+  -  +-+-+-+-+-+-+
|               |     | |#| |#|   |
|     LU #1     |     | |#| | |   |
|               |     | |#| | |   |
+---------------+  -  +-+-+-+-+-+-+
|               |     |#|     |#| |
|               |     |#|     | | |
|    LUT #1     |     |#|     | | |
|               |     |#|     | | |
+---------------+  -  +-+-+-+-+-+-+
|     NOOP      |     |   |#|     |
+---------------+  -  +-+-+-+-+-+-+
|               |     | |#| |#|   |
|     LU #2     |     | |#| | |   |
+---------------+  -  +-+-+-+-+-+-+
|               |     |#|       |#|
|    LUT #2     |     |#|       | |
|               |     |#|       | |
|               |     |#|       | |
+---------------+  -  +-+-+-+-+-+-+
|     NOOP      |     |   |#|     |
+---------------+  -  +-+-+-+-+-+-+
     witness           0 1 2 3 4 5
```

If there are several lookup tables, the last column is repeated for each of them; hence we have `4+#luts` lookup selector columns in total.

The constraints

The number of constraints is `4 + #luts + 2 * #sldc`, where `#luts` denotes the number of different lookup tables, and `#sldc` denotes the number of SLDC columns (these contain the running sums). The latter is determined as

$$\#\texttt{sldc} = \frac{\#\texttt{lu\_slots}}{\texttt{lookup\_deg}} = \left\lceil \frac{\lfloor 80/2 \rfloor}{\texttt{maxdeg} - 1} \right\rceil = \lceil 40/7 \rceil = 6$$

Let's denote the SLDC columns with A_i (for $0 \leq i < 6$), the RE column with R, the witness columns with W_j, and the selectors columns with S_k. Similarly to the permutation argument, the running sum is encoded in row-major order with degree 7 "chunks", but here it's read *from the bottom to the top* order…

Then these constraints are:

- the final SLDC sum is zero: $S_{\mathsf{LastLDC}}(x) \cdot A_5(x) = 0$ (this corresponds to actual core equation above)

- the initial `Sum` is zero: $S_{\mathsf{InitSre}}(x) \cdot A_0(x) = 0$
- the inital `RE` is zero: $S_{\mathsf{InitSre}}(x) \cdot R(x) = 0$
- final `RE` is the expected value, separately for each LUT: $S_{4+i}(x) \cdot R(x) = \mathsf{expected}_i$ where i runs over the set of lookup tables, and the expected value is calculated simply as a polynomial evaluation as described above
- `RE` row transition constraint:

$$S_{\mathsf{TransSre}}(x) \left[R(x) - \delta^{26} R(\omega x) - \sum_{j=0}^{25} \delta^{25-j} \big(W_{3j}(x) + b W_{3j+1}(x) \big) \right] = 0$$

- row transition constraints for `LDC` and `SUM`, separately:

$$0 = S_{\mathsf{TransLDC}}(x) \left\{ A_k(x) - A_{k-1}(x) - \frac{\sum_{i=7k}^{7k+6} \cdot \prod_{j \neq i} (\alpha - W_{2i}(x) - a W_{2i+1}(x))}{\prod_{i=7k}^{7k+6} (\alpha - W_{2i}(x) - a W_{2i+1}(x))} \right\}$$

where for convience, let's have $A_{-1}(x) := A_5(\omega x)$

and similarly for the `SUM` constraint (but with mulitplicities included there). This is very similar to the how the partial products work, just way over-complicated…

In the source code, all this can be found in the `check_lookup_constraints()` function in `vanishing_poly.rs`.

Remark: we have `lookup_deg = 7` instead of 8 because the selector adds an extra plus 1 to the degree. Since 40 is not divisible by 7, the last column will have a smaller product. Similarly, in the `SUM` case they use a smaller degree instead, namely: $\texttt{lut}_{\texttt{d}}\texttt{egree} = \lceil 26/7 \rceil = 4$; also with a truncated last column.

Recursive proofs

TODO