0.1 Number Theoretical Trasnform: Universal Polynomial Accelerator

In the previous sections, when considering Groth16, we have seen the idea that polynomials are powerful encoders of data. To encode a set of N values $a_0, \dots, a_{N-1} \in \mathbb{F}$, we interpolate a polynomial $p(x)$ that at specific points $x_0, \dots, x_{N-1} \in \mathbb{F}$ evaluates to these values. The only condition we impose on these points is that they are distinct (unless, the interpolation would not be properly defined). This way, generally, we have the following interpolation problem:

$$p(x_j) = a_j, \quad j = 0, \dots, N-1$$

Particularly, in Groth16 our choice of points was $x_j = j$ for $j = 0, \dots, N-1$, which, for the large enough finite field $\mathbb{F}$, does not cause any issues. However, the complexity of interpolation in this case is not optimal. Let us see why.

Recall that the interpolation formula (see **??** for details) is given by:

$$p(x) = \sum_{i=0}^{N-1} a_i \ell_i(x), \quad \ell_i(x) = \prod_{j=0, j\neq i}^{N-1} \frac{x - x_j}{x_i - x_j}.$$

The naive evaluation of this formula requires $O(N^2)$ operations: we need to compute each ℓ_i, costing $O(N)$ operations, and then sum them up with, again, $O(N)$ operations.

With the specific choice of points $\{x_j\}_{0 \le j < N}$, we can do much better: in fact, we can reduce the complexity to $O(N \log N)$ operations or even $O(N)$. This is done by utilizing several techniques, including the **Barycentric Interpolation** and the N**th roots of unity**.

0.1.1 Barycentric Interpolation

The idea of $O(N \log N)$ is to exploit the barycentric formula for polynomial interpolation. Let us derive the formula and see how it helps.

First, introduce the quantity $\gamma(x) = \prod_{j=0}^{N-1}(x - x_j)$. Now note that the Lagrange basis polynomials $\ell_j(x)$ can be rewritten as:

$$\ell_j(x) = \gamma(x) \cdot \frac{w_j}{x - x_j}, \quad w_j = \frac{1}{\prod_{k=0, k\neq j}^{N-1}(x_j - x_k)}, \quad j \in [N]$$

> **Remark.** This step might seem unobvious, so let us be careful here. Let us see why expression $\gamma(x) \cdot \frac{w_j}{x - x_j}$ indeed gives the desired basis polynomial:
>
> $$\gamma(x) \cdot \frac{w_j}{x - x_j} = \left(\prod_{k=0}^{N-1}(x - x_k)\right) \cdot \frac{1}{\left(\prod_{k=0, k\neq j}^{N-1}(x_j - x_k)\right)(x - x_j)}$$

Note that we can cancel out $(x - x_j)$ from both numerator and denominator and get

$$\left(\prod_{k=0,k\neq j}^{N-1}(x - x_j)\right) \cdot \frac{1}{\prod_{k=0,k\neq j}^{N-1}(x_j - x_k)} = \prod_{k=0,k\neq j}^{N-1}\frac{x - x_j}{x_j - x_k} = \ell_j(x),$$

which is exactly what we wanted to show.

Good, so why do we even need such an expression for ℓ_j? Let us substitute it back into the interpolation formula:

$$p(x) = \sum_{i=0}^{N-1} a_i \ell_i(x) = \sum_{i=0}^{N-1} a_i \gamma(x) \cdot \frac{w_i}{x - x_i} = \gamma(x) \sum_{i=0}^{N-1} \frac{w_i}{x - x_i} a_i$$

In regards to this formula, we give the following definition.

Proposition 0.1. The **barycentric interpolation formula** for the interpolation problem $p(x_j) = a_j, j \in [N]$, given by $p(x) = \gamma(x) \sum_{i\in[N]} \frac{w_i}{x-x_i} a_i$ with $\gamma(x) = \prod_{i\in[N]}(x - x_i)$, requires $O(N)$ operations to compute and $O(N^2)$ operations to pre-compute.

Proof. Coefficients $\{w_j\}_{j\in[N]}$ are independent of x, and so are the values $\{a_j\}_{j\in[N]}$. To compute $\{w_j\}_{j\in[N]}$, one needs $O(N)$ operations for each w_j, and thus $O(N^2)$ operations in total. To compute the polynomial $p(x)$, one needs $O(N)$ operations to compute $\gamma(x)$ and $O(N)$ operations to compute the sum, knowing $\{w_j a_j\}_{j\in[N]}$. □

Of course, in reality, storing N values $\{w_j\}_{j\in[N]}$ requires $O(N)$ memory, which is not optimal. Moreover, these points on their own are useless and typically are not used in any other parts of the protocol. This is where the N**th roots of unity** come into play.

0.1.2 Multiplicative Cyclic Subgroup

Again, assume we have the prime field $\mathbb{F}_p$. Let ω be a **primitive** N**-th root of unity**, i.e., $\omega^N = 1$ and $\omega^j \neq 1$ for $j < N$. The set $\Omega = \{\omega^j\}_{0\leq j<N}$ is called the N**-th root of unity subgroup** of $\mathbb{F}_p$ of order N. One might ask the following question: why such primitive root even exists? Consider the following lemma, briefly mentioned in **??**.

Lemma 0.2. For $\mathbb{F}_p$ there exists a primitive N-th root of unity if and only if $N \mid (p - 1)$.

What is so special about the set Ω? The magic of Ω is that it allows to

compute certain polynomial operations (such as interpolation or multiplication) using the **Discrete Fourier Transform** (DFT) or, equivalently, the **Number Theoretic Transform** (NTT) algorithm. Consider the first central lemma.

Lemma 0.3. The vanishing polynomial of the set Ω is given by $z_\Omega(X) = X^N - 1$.

Proof Idea. If ω is the Nth primitive root, then for any $h \in \Omega$ we have $h^N = 1$ and therefore all elements of Ω are the roots of $X^N - 1$. There are precisely N such roots, so $X^N - 1$ can be decomposed as a product of linear factors $c \cdot \prod_{j=0}^{N-1}(X - \omega^j)$. It is easy to see that $c = 1$ by comparing the leading coefficient.

Now, let us come back to the barycentric formula. We have seen that the interpolation polynomial $p(x)$ can be written as $p(x) = \gamma(x) \sum_{i=0}^{N-1} \frac{w_i}{x - x_i} a_i$. The key idea of the FFT is to set $x_j = \omega^j$. What does it give us? We give the following proposition.

Proposition 0.4. Suppose the interpolation domain is chosen so that $a_i = \omega^i$. Then, following the notation of Proposition 0.1, certain expressions simplify to the following:

- $\gamma(x) = x^N - 1$.
- $w_i = \omega^i / N$.

Proof Idea. For the first claim, notice that by definition $\gamma(x) = \prod_{i=0}^{N-1}(x - \omega^i)$, which is exactly the vanishing polynomial of Ω. Thus, $\gamma(x) = z_\Omega(x) = x^N - 1$ from Lemma 0.3.

As for the second claim, recall that $w_i = 1/\prod_{j \neq i}^{N-1}(\omega^i - \omega^j)$. Intuitively, the real analysis shows that $w_i = 1/\gamma'(x_i)$ and since $\gamma(x) = x^N - 1$, we have exactly $w_i = 1/N x^{N-1}\Big|_{x=\omega^i} = \omega^i / N$. The same result can be obtained by direct computation. □

That being said, the barycentric formula is now given by:

$$p(x) = \frac{x^N - 1}{N} \sum_{j \in [N]} \frac{\omega^j}{x - \omega^j} a_j$$

This formula is a much more convenient form for the computation of the interpolation polynomial! First, the evaluation of the sum requires $O(N)$ operations and it depends only on the values $\{\omega^j\}_{j \in [N]}$, which are typically pre-computed and used in many other parts of the protocol. Second, the evaluation of the vanishing polynomial $x^N - 1$ requires $O(\log N)$ operations using the fast exponentiation

algorithm, compared to naive $O(N)$ operations.

0.1.3 Fast Polynomial Multiplication

Forward NTT. Using the Nth roots of unity Ω, we can also compute the polynomial multiplication in $O(N \log N)$ operations. The idea is to use the **Number Theoretic Transform** (NTT) algorithm, which is a generalization of the Fast Fourier Transform (FFT) to the finite fields. But first, let us define what NTT is.

Definition 0.5 (NTT)**.** Suppose the polynomial is given by $p(x) = \sum_{j=0}^{N-1} p_j x^j \in \mathbb{F}[x]$. In its essence, the polynomial is defined as a vector of coefficients $\boldsymbol{p} = (p_0, \ldots, p_{N-1})$. The **Number Theoretic Transform** (NTT) of polynomial $p(x)$ is the vector of evaluations at the Nth roots of unity Ω: $\mathsf{NTT}(\boldsymbol{p}) = (p(\omega^0), p(\omega^1), \ldots, p(\omega^{N-1}))$.

Remark. Typically, the jth component of the NTT vector is denoted as $\hat{p}_j = \mathsf{NTT}(\boldsymbol{p})_j$.

If computed naively, the NTT requires $O(N^2)$ operations, since each evaluation of $p(x)$ requires $O(N)$ operations. However, due to the specifics of the selected domain Ω, the NTT can be computed in $O(N \log N)$ operations. Let us emphasize this in the following lemma.

Lemma 0.6. The **Number Theoretic Transform** (NTT) of a polynomial $p(x)$ can be computed in $O(N \log N)$ operations using the Nth roots of unity. This is possible only if the prime field $\mathbb{F}_p$ allows to find the primitive 2^k root of unity for $k \in [m]$ with large enough m. Equivalently, $2^k \mid (p-1)$ for $k \in [m]$.

To show this lemma is true, let us develop the concrete algorithm. Notice that our task consists in computing:

$$p(\omega^i) = \sum_{j \in [N]} p_j (\omega^i)^j = \sum_{j \in [N]} p_j \omega^{ij} \text{ for each } i \in [N]$$

Suppose the considered polynomial is such that $N = 2^r$ (we can always pad the polynomial if that is not the case). Now, let us proceed with the polynomial

as follows:

$$p(\omega^i) = \sum_{j=0}^{2^r-1} p_j \omega^{ij} = \sum_{j=0}^{2^{r-1}-1} p_{2j} \omega^{2ij} + \sum_{j=0}^{2^{r-1}-1} p_{2j+1} \omega^{i(2j+1)} =$$
$$= \sum_{j=0}^{2^{r-1}-1} p_{2j} (\omega^{2i})^j + \omega^i \sum_{j=0}^{2^{r-1}-1} p_{2j+1} (\omega^{2i})^j$$

This already looks interesting enough. Notice that we can introduce two new polynomials: $p_E(x) = \sum_{j=0}^{2^{r-1}-1} p_{2j} x^j$ and $p_O(x) = \sum_{j=0}^{2^{r-1}-1} p_{2j+1} x^j$, which are polynomials, containing even and odd coefficients of p, respectively. In that case,

$$p(\omega^i) = p_E(\omega^{2i}) + \omega^i p_O(\omega^{2i})$$

This is quite an interesting observation which already screams divide-and-conquer! However, currently it might still be unclear how to use it: we still have to evaluate N expressions of form $p_E(\omega^{2i}) + \omega^i p_O(\omega^{2i})$ where both polynomials p_E and p_O contain roughly $N/2$ coefficients, totalling in $O(N^2)$ operations again. To counter this, we claim the following: we need only half the domain of Ω to compute both p_E and p_O. To see why, consider the expression $p(\omega^{i+N/2})$:

$$p(\omega^i) = p_E(\omega^{2(i+N/2)}) + \omega^{i+N/2} p_O(\omega^{2(i+N/2)}) = p_E(\omega^{2i}) + \omega^i \omega^{N/2} p_O(\omega^{2i})$$

In other words, having computed $p_E(\omega^{2i})$ and $p_O(\omega^{2i})$, we know not only $p(\omega^i)$, but also $p(\omega^{i+N/2})$ for free! This way, to compute the NTT for N-degree polynomial, we need to evaluate two $\frac{N}{2}$-degree polynomials at $\frac{N}{2}$ points. This way, on each step: (a) the evaluation domain shrinks in half, (b) the complexity of computing polynomials shrinks in half, (c) we get two new polynomials. This way, on each step, we reduce the problem complexity in half!

The reason why the prime field should support multiplicative cyclic subgroups of order 2^k for sufficiently many k is that not always if ω is the Nth primitive root, then ω^2 is the $\frac{N}{2}$th primitive root. If that is not the case, all the aforementioned magic breaks.

We summarize everything so far in the Algorithm 1.

NTT Domain. OK, so what next? Suppose we want to multiply two polynomials $p(x), q(x) \in \mathbb{F}[X]$ of degree $N = 2^r$ and we have successfully evaluated their NTTs. Say, we got $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$. What can we do next? Here is another trick.

Proposition 0.7. Suppose $m(x) = p(x)q(x)$ is the product of p and q. Then,

$$\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$$

Algorithm 1: Number Theoretic Transform (NTT)

Input : Polynomial $p(x) = \sum_{j=0}^{N-1} p_j x^j$
Output : Vector of evaluations $\mathsf{NTT}(\boldsymbol{p}, \omega)$ at $\Omega = \{\omega\}_{j \in [N]}$

if $N = 1$ **then**
  Return : (p_0)
end
$H \leftarrow N/2$ /* Compute the domain half-size */
$\boldsymbol{p}_E \leftarrow (p_0, p_2, \ldots, p_{N-2})$ /* Find even-indexed coefficients */
$\boldsymbol{p}_O \leftarrow (p_1, p_3, \ldots, p_{N-1})$ /* Find odd-indexed coefficients */
$\boldsymbol{y}_E \leftarrow \mathsf{NTT}(\boldsymbol{p}_E, \omega^2)$ /* Compute NTT for even polynomial via $\frac{N}{2}$th primitive root ω^2 */
$\boldsymbol{y}_O \leftarrow \mathsf{NTT}(\boldsymbol{p}_O, \omega^2)$ /* Compute NTT for odd polynomial via $\frac{N}{2}$th primitive root ω^2 */
Return : $(y_0, \ldots, y_{N-1})$ with $y_j = y_{E,\, j \bmod H} + \omega^j y_{O,\, j \bmod H}$

Speaking more formally, $\mathsf{NTT} : (\mathbb{F}^{(\leq N)}[X], \times) \to (\mathbb{F}^N, \odot)$ is a homomorphism between a set of polynomials of degree up to N and their NTT domain. With certain appropriate technicalities, NTT can be extended to the isomorphism.

Intuition. Although this fact might come out of random, we give an intuitive explanation why this holds. One of NTT interpretations is well-known to you *interpolation*. Indeed, polynomials p and q satisfy the following interpolation problem:

$$p(\omega^j) = \mathsf{NTT}(p)_j = \hat{p}_j, \quad q(\omega^j) = \mathsf{NTT}(q)_j = \hat{q}_j.$$

In turn, $\hat{m}_j$ is nothing but the evaluation of m at ω^j. But, if m is the product of p and q and values of p and q at ω^j are $\hat{p}_j$ and $\hat{q}_j$, this immediately implies that $m(\omega^j)$ is nothing but $\hat{p}_j \hat{q}_j$. Meaning, $\hat{m}_j = \hat{p}_j \hat{q}_j$. This, in turn, implies $\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$.

Wow! What this essentially means is that multiplication in NTT domain is very cheap: it requires only $O(N)$ field multiplication operations! Now, the algorithm of multiplication of p and q becomes a little bit more clear at this point:

1. Compute NTTs $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$ of p and q.
2. Compute NTT $\hat{\boldsymbol{m}} = \hat{\boldsymbol{p}} \odot \hat{\boldsymbol{q}}$ of their product $m = pq$.
3. Restore $m(x)$ from NTT $\hat{\boldsymbol{m}}$ — this problem is called Inverse NTT (INTT).

Inverse NTT. So, the only problem left is restoring the polynomial from its NTT form. This problem is equivalent to solving the interpolation problem:

$$m(\omega^j) = \hat{m}_j, \quad j \in [N]$$

Suprisingly, the **Inverse NTT** is given by $p_j = \frac{1}{N}\sum_{j=0}^{N-1} \hat{p}_j \omega^{-ij}$, which can be computed by running the forward NTT, but with generator ω^{-1}. Division by N is done over $\mathbb{F}_p$, which is surely trivial. We summarize the whole discussion in Figure 0.1.

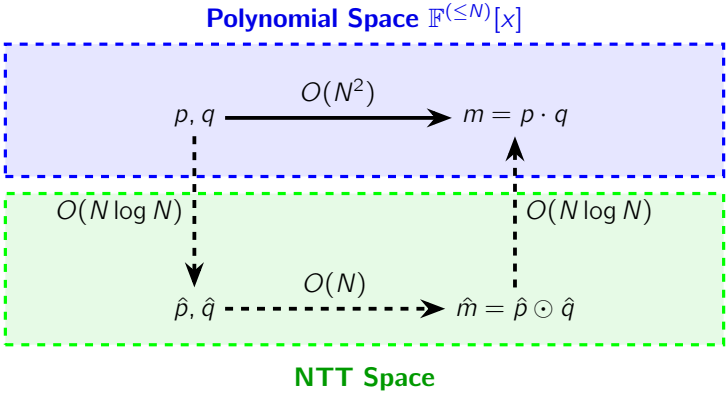

Figure 0.1: Illustration of the NTT Algorithm

0.2 Plonk Arithmetization

Consider we have a certain relation $\mathcal{R}$, which we would like to write down into a processing-prone format over the field $\mathbb{F}$. Plonk arithmetizes this relation into a set of *8 polynomials*, which are then used to verify the witness knowledge. Let us start with the concrete example.

Example. To begin with, observe this fairly simple relation $\mathcal{R}_{\text{example}}$: suppose we have a public input $x \in \mathbb{F}$ and public output $y \in \mathbb{F}$, and we want to prove the knowledge of $e \in \mathbb{F}$ such that $e \times x + x - 1 = y$. Formally, we have the following relation:

$$\mathcal{R}_{\text{example}} = \left\{ \begin{matrix} \textbf{Public Statement:} & x, y \in \mathbb{F} \\ \textbf{Witness}: & e \in \mathbb{F} \end{matrix} \;\middle|\; e \times x + x - 1 = y \right\}$$

Remark. Note that of course, from x and y, it is fairly simple to find e: simply take $\frac{1-x+y}{x}$. However, the Plonk arithmetization is not limited to this simple example, and can be applied to more complex relations, such as hash function pre-image knowledge or any NP statement.

0.2.1 Execution Trace

Standard Plonk is defined as a system with two types of gates: **addition** and **multiplication**. We would explain how to build custom gates later. So, let us consider our program in terms of gates with left, right operands and output.

Example. We need **three gates** to encode our program:

1. **Gate #1**: left e, right x, output $u = e \times x$
2. **Gate #2**: left u, right x, output $v = u + x$
3. **Gate #3**: left v, right x, output $w = v + (-1)$

You might have glanced the intuitive formation of what is called *execution trace table* — a matrix T with columns L, R and O (it is common to denote those as A, B, C to distinguish from another matrix we will discuss later). Moreover, we will mark columns **A**, **B** and **C** in bold to indicate that they are vectors from $\mathbb{F}^N$, where here and hereafter, unless stated otherwise, N is the number of gates in the program.

Example. We might visualize the execution trace table T for the example program as follows:

A	**B**	**C**
2	3	6
6	3	9
9	✗	8

Notice how the last row has no value in **B** column (marked by ✗) — this is reasoned by the fact it is not a variable, but rather a constant, meaning it doesn't depend on execution. Also note that the number of gates in this particular circuit is $N = 3$.

Remark. As you might notice, in contrast to classic R1CS (which we used for Groth16), the standard Plonk arithmetization as is only allows two input values to be processed at a time. This way, if Groth16 requires only one constraint for verifying $x_1(x_2 + x_3 + x_4) = x_5$, Plonk would need three constraints to verify the same statement. Custom gates partially solve this problem as we will see later, but it is important to keep in mind.

0.2.2 Encode the program

It is essential to distinguish the definition of the program and its specific evaluation for the sake of simplicity and efficiency — once having established

encoding for the program, you might apply it for any reasonable inputs. Therefore, let us at first focus on what defines whether execution trace table will be considered valid for our circuit, because having a table by itself does not tell much, since it can be populated with any values.

For that reason, we would define two matrices — $Q \in \mathbb{F}^{N\times 5}$ and $V \in \mathbb{Z}_{\geq 0}^{N\times 3}$ where $N \in \mathbb{N}$, again, is the number of gates in the program:

- Q is the **Gate Matrix**, which encodes the values of the gates and stores all the intermediate values computed.
- V is the **Wiring Matrix**, which encodes the wiring of the gates, i.e., how the output of one gate is carried as input to another.

Definition 0.8. The **gate matrix** $Q \in \mathbb{F}^{N\times 5}$ has one row per each gate with columns $\mathbf{Q}_L$, $\mathbf{Q}_R$, $\mathbf{Q}_O$, $\mathbf{Q}_M$, $\mathbf{Q}_C$ from $\mathbb{F}^N$. If columns $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbb{F}^N$ of the execution trace table form valid evaluation of the circuit, then the following holds:

$$A_i(Q_L)_i + B_i(Q_R)_i + A_iB_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i = 0,\ \forall i \in [N]$$

Using Hadamard product notation, this can be concisely rewritten as:

$$\mathbf{A} \odot \mathbf{Q}_L + \mathbf{B} \odot \mathbf{Q}_R + \mathbf{A} \odot \mathbf{B} \odot \mathbf{Q}_M + \mathbf{C} \odot \mathbf{Q}_O + \mathbf{Q}_C = \mathbf{0}$$

Example. For our program, we would have a following Q table:

$\mathbf{Q}_L$	$\mathbf{Q}_R$	$\mathbf{Q}_M$	$\mathbf{Q}_O$	$\mathbf{Q}_C$
0	0	1	−1	0
1	1	0	−1	0
1	0	0	−1	−1

You can verify that our claim holds for aforementioned trace matrix:

$$2 \times 0 + 3 \times 0 + 2 \times 3 \times 1 + 6 \times (-1) + 0 = 0$$
$$6 \times 1 + 3 \times 1 + 6 \times 3 \times 0 + 9 \times (-1) + 0 = 0$$
$$9 \times 1 + 0 \times 0 + 9 \times 0 \times 0 + 8 \times (-1) + (-1) = 0$$

Recall that columns of trace matrix T are $\mathbf{A} = \begin{bmatrix}2\\6\\9\end{bmatrix}$, $\mathbf{B} = \begin{bmatrix}3\\3\\\times\end{bmatrix}$, $\mathbf{C} = \begin{bmatrix}6\\9\\8\end{bmatrix}$.

Now, we do have a way of encoding gates separately, yet in order to guarantee

how result of one gate is carried in as input of the other (*wirings*), we need another matrix — V.

Definition 0.9. The **wiring matrix** $V \in \mathbb{Z}_{\geq 0}^{N \times 3}$ consists of indices of all inputs and intermediate values, so that if T is a valid trace,

$$\forall (i,j)\ \forall (k,\ell) :\ V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$$

Put more simply, if two values are equal in V, then the corresponding values (corresponding to these indices) in T must be equal as well.

Example. For our program, V can be defined as follows:

L	R	O
0	1	2
2	1	3
3	✗	4

Here 0 is an index of e, 1 is an index of x, 2 — of intermediate value u, 3 — of v and finally 4 — of output w.

0.2.3 Custom Gates

In order to reach beyond classical operations such as addition and multiplication, one may consider composing a custom gate. The main streamliner of this functionality is a matrix Q, using 5 basic columns of which, you already may build custom logic.

Example. Our entire program may be encoded as one custom gate.

$Q =$

Q_L	Q_R	Q_M	Q_O	Q_C
0	1	1	−1	−1

$V =$

L	R	O
0	1	2

$T =$

A	B	C
2	3	8

$$2 \times 0 + 3 \times 1 + 2 \times 3 \times 1 + 8 \times (-1) + (-1) = 0$$

As you can see, custom gates is a good way to reduce the number of constraints needed for the same program.

Remark. Real-world PlonK applications commonly have additional columns in the Q matrix, enabling an even broader set of custom functionality.

0.2.4 Public Inputs

With the current design, we can prove that the computations were done correctly, but we have no restrictions on the values of inputs. For example, when the prover wants to convince the verifier that he knows e for $x = 3$ and $y = 7$, the verifier does not even check whether x is 3 (not to mention whether the result of execution $y = 7$ is correct) in the trace table T. One way of doing this is by incorporating them in three previously defined matrices Q, V, T.

Proposition 0.10. One way to solve this is to use the **equality gates**. Introduce two gadgets:

- **Constant Equality Gate:** Suppose we want to check whether the certain variable equals to the constant value $\alpha \in \mathbb{F}$ at gate with index i. For ith gate, set $(Q_L)_i = -1$, $(Q_C)_i = \alpha$ and other columns to 0. Then, add a row to V with $L = i$, $R =$ ✗ and $O =$ ✗. Then, to satisfy the condition, the ith left input **must** be equal to α.
- **Nodes Equality Gate:** Suppose we want to check whether the ith and jth gates have equal outputs in the kth gate. Set $(Q_L)_k = 1$, $(Q_R)_k = -1$ with other columns to 0. Add a row to V with $L = i$, $R = j$ and $O =$ ✗. Then, to satisfy the condition, the ith and jth outputs **must** be equal.

Example. Suppose the prover wants to prove that he knows e for the public statement $(x, y) = (3, 8)$. We can encode this as follows:

$Q =$

$\mathbf{Q}_L$	$\mathbf{Q}_R$	$\mathbf{Q}_M$	$\mathbf{Q}_O$	$\mathbf{Q}_C$
-1	0	0	0	3
-1	0	0	0	8
1	1	1	-1	1
1	-1	0	0	0

$V =$

L	R	O
0	✗	✗
1	✗	✗
2	0	3
1	3	✗

$T =$

A	B	C
3	✗	✗
8	✗	✗
2	3	8
8	8	✗

As can be seen, besides the original program gate, inscribed in the third row, we have three additional gates:

- The first two gates "allocate" two nodes with indices 0 and 1 to the values 3 and 8 respectively. This is done through the *constant equality gates*.
- The last gate checks whether the result of the third gate is equal to the index 1, corresponding to the allocated value 8. This is done through the *nodes equality gate*.

The primary problem with this approach, is that now we have lost agnosticism in Q and V of concrete evaluations. In other words, our circuit is now "hardcoded" to the specific values of public inputs. In order to resolve this, we would define a separate one-column matrix named $\mathbf{\Pi} \in \mathbb{F}^N$, in which we would encode the public inputs.

Example. With only Q modified, we now have:

$\mathbf{\Pi}$
3
8
0
0

$$Q = \begin{array}{|c|c|c|c|c|} \hline \mathbf{Q}_L & \mathbf{Q}_R & \mathbf{Q}_M & \mathbf{Q}_O & \mathbf{Q}_C \\ \hline -1 & 0 & 0 & 0 & 0 \\ \hline -1 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & -1 & 1 \\ \hline 1 & -1 & 0 & 0 & 0 \\ \hline \end{array}$$

Proposition 0.11 (Wrap-up)**.** The matrix T with columns $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbb{F}^N$ encodes correct execution of the program, if the following two conditions hold:

1. $\forall i \in [N] : A_i(Q_L)_i + B_i(Q_R)_i + A_i B_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i + \Pi_i = 0$
2. $\forall (i,j)\, \forall (k,\ell) : V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$

0.2.5 Matrices to Polynomials

Gates Satisfiability. Now we can traduce the sets of constraints on matrices to just a few equations on polynomials, as we have already done for Groth16. Let ω be a primitive N-th root of unity[1] and let $\Omega = \{\omega^j\}_{0 \le j < N}$. In the Section 0.1, we have already discussed the reasoning for choosing such a set Ω. In particular, denote by $\{L_j\}_{0 \le j < N}$ the Lagrange basis polynomials[2] for the set Ω, which for the selected domain Ω are given by the following convenient form:

$$L_j(x) = \frac{\gamma_j(x^N - 1)}{x - \omega^j}, \quad \text{where } \gamma_j \text{ is some normalizing constant.}$$

[1] Suppose such ω exists, then $\omega^N = 1$ and $\omega^j \neq 1$ for $0 \le j < N$.

[2] Recall that by definition the Lagrange basis $\{L_j\}_{0 \le j < N}$ for set $\mathcal{X} = (x_0, \ldots, x_{N-1})$ is given by $L_j(x_i) = \delta_{ij}$.

Let $a, b, c, q_L, q_R, q_M, q_O, q_C, \pi \in \mathbb{F}^{(\leq N)}[X]$ be polynomials of degree at most N that interpolate corresponding columns from matrices at the domain Ω. In other words, we have $\forall j \in [N] : a(\omega^j) = A_j$ and the same holds for other polynomials.

Notice that if our trace matrix is correct, then the first condition of Proposition 0.11 can be reduced to the following polynomial equation:

$$a(\omega^j)q_L(\omega^j) + b(\omega^j)q_R(\omega^j) + a(\omega^j)b(\omega^j)q_M(\omega^j) + c(\omega^j)q_O(\omega^j) + q_C(\omega^j) + \pi(\omega^j) = 0, \ \forall j \in [N]$$

Notice that this essentially means that the left polynomial $aq_L + bq_R + abq_M + cq_O + q_C + \pi$ has roots at ω^j for all $j \in [N]$. This is equivalent to stating that the vanishing polynomial of Ω divides the left hand side. Due to the Lemma 0.3, this vanishing polynomial is given by $z_\Omega(x) = x^N - 1$. Let us wrap this up in the following proposition.

Proposition 0.12. Now we can reduce down our first condition of Proposition 0.11 to checking valid execution trace into the following claim over polynomials:

$$\exists t \in \mathbb{F}^{(\leq 3N)}[x] : \ aq_L + bq_R + abq_M + cq_O + q_C + \pi = z_\Omega t,$$

where $z_\Omega(x)$ is the vanishing polynomial $x^N - 1$.

Wiring Satisfability. The next step is to shrink the second condition imposed by the V matrix. This may be achieved by introducing the concept of permutation.

Remark. Permutation of the set S is commonly denoted as $\sigma : S \to S$. This function is bijective, meaning that for every $s \in S$ there exists a unique $s' \in S$ such that $\sigma(s) = s'$.

Example. A permutation is a rearrangement of the set, which is in our case:

$$\mathcal{I} = \{(i, j) : \text{such that } 0 \leq i < N, \text{ and } 0 \leq j < 3\}$$

Naturally, the matrix V induces a permutation σ of this set where $\sigma((i, j))$ equals to the pair of indices of the next occurrence of the value at position (i, j). So, for our example:

$V =$

L	R	O
0	✗	✗
1	✗	✗
2	0	3
1	3	✗

We have the following permutation:

$$\sigma(\textbf{(0,0)}) = \textbf{(2,1)},\ \sigma((0,1)) = (0,3),\ \sigma((0,2)) = (0,2)$$

$$\sigma((0,3)) = (0,1),\ \sigma((2,1)) = (0,0),\ \sigma((3,1)) = (2,2)$$

For demonstration purposes, we marked in **green** the index of the first and second occurance of the value 0. For proper σ definition (as it has to be bijective), the application of σ to the last occurance outputs the first one.

Permutation Check. This is probably the most tedious part of PlonK. We split the following derivation into two parts:

- **Set Equality using Polynomials.** We will show how to check whether two sets of field elements are equal using polynomials.
- **Permutation Check using Polynomials.** We will show how to check whether a given function is a permutation using polynomials in several forms.

Additionally, to avoid confusion, we denote the polynomial variable by capital letters (X, Y).

Set equality. Having defined permutation, we can now reduce the second condition of Proposition 0.11 of valid execution trace matrix into the following check:

$$\forall (i,j) \in \mathcal{I} : T_{i,j} = T_{\sigma(i,j)}$$

You may have noticed how this can be reformulated as equality of two sets A and B:

$$A := \{((i,j), T_{i,j}) : (i,j) \in \mathcal{I}\}$$
$$B := \{(\sigma((i,j)), T_{i,j}) : (i,j) \in \mathcal{I}\}$$

We can reduce this check down to polynomial equations! Here is how: suppose for simplicity we have two sets with two elements $A = \{a_0, a_1\}$ and $B = \{b_0, b_1\}$. Introduce two sets of polynomials $A' = \{a_0 + X, a_1 + X\}$ and $B' = \{b_0 + X, b_1 + X\}$.

When do we have the set equality $A' = B'$? Well, $(a_0 + X)(a_1 + X) = (b_0 + X)(b_1 + X)$ works fine. This is true because of linear polynomial unique factorization property, working as prime factors. Now, we can utilize Schwartz-Zippel lemma to replace the latter formula with $(a_0+\gamma)(a_1+\gamma) = (b_0+\gamma)(b_1+\gamma)$ for some random $\gamma \xleftarrow{R} \mathbb{F}$ with overwhelming probability, being at least $1 - 2/|\mathbb{F}|$. If we wish to generalize this for arbitrary sets $A = \{a_0, \ldots, a_{k-1}\}$ and $B = \{b_0, \ldots, b_{k-1}\}$, apply the following equivalent check:

$$\prod_{i=0}^{k-1}(a_i+\gamma)=\prod_{i=0}^{k-1}(b_i+\gamma)$$

Let Ω be a domain of the form $\{1,\omega,\ldots,\omega^{k-1}\}$ for some k-th root of unity ω. Let f and g be polynomials that we interpolate at Ω as follows:

$$f(\omega^j)=a_j+\gamma,\quad g(\omega^j)=b_j+\gamma,\quad j\in[k]$$

Then, $\prod_{i=0}^{k-1}(a_i+\gamma)=\prod_{i=0}^{k-1}(b_i+\gamma)$ holds if and only if there is a polynomial $Z\in\mathbb{F}[X]$ such that for all $h\in\Omega$ we have $Z(\omega^0)=1$ and $Z(h)f(h)=g(h)Z(\omega h)$.

Now that we can encode equality of sets of field elements, let's expand this to sets of tuples of field elements. Let $A=\{(a_0,a_1),(a_2,a_3)\}$ and $B=\{(b_0,b_1),(b_2,b_3)\}$. Then, similarly, if

$$A'=\{a_0+a_1Y+X,a_2+a_3Y+X\},\quad B'=\{b_0+b_1Y+X,b_2+b_3Y+X\},$$

then $A=B$ if and only if $A'=B'$. As before, we can leverage Schwartz-Zippel lemma to reduce this down into sampling two random β and $\gamma\xleftarrow{R}\mathbb{F}$ and checking equality of:

$$(a_0+\beta a_1+\gamma)(a_2+\beta a_3+\gamma)=(b_0+\beta b_1+\gamma)(b_2+\beta b_3+\gamma)$$

Permutation Check. Now, to go back to the second condition of Proposition 0.11 which we are trying to formulate in the polynomial domain, it becomes clear that if we somehow encoded inner indices tuple (i,j) into a one field element, we could use the above fact. Recall that $i\in[N]$ and $j\in\{0,1,2\}$. Thus, take the $3N$-th primitive root of unity η and define the bijective map $((i,j),v)\mapsto(\eta^{3i+j},T_{i,j})$. Thus, consider the modified sets:

$$A=\{(\eta^{3i+j},T_{i,j}):(i,j)\in\mathcal{I}\}$$
$$B=\{(\eta^{3k+\ell},T_{i,j}):(i,j)\in\mathcal{I},\sigma((i,j))=(k,\ell)\}$$

Sample two random field elements β and $\gamma\xleftarrow{R}\mathbb{R}$.
Let $\mathcal{D}=\{1,\eta,\eta^2,\ldots,\eta^{3N-1}\}$. Then, interpolate two polynomials f and g over the defined set $\mathcal{D}$ as follows:

$$f(\eta^{3i+j})=T_{i,j}+\eta^{3i+j}\beta+\gamma,\quad (i,j)\in\mathcal{I}$$
$$g(\eta^{3k+\ell})=T_{i,j}+\eta^{3k+\ell}\beta+\gamma,\quad (i,j)\in\mathcal{I},\quad \sigma((i,j))=(k,\ell)$$

Similarly to our previous discussion, there should be a polynomial $Z\in\mathbb{F}[X]$ such that $\forall d\in\mathcal{D}$, we have $Z(\eta^0)=1$ and $Z(d)f(d)=g(d)Z(\eta d)$. This

would imply the set equality $A = B$ with overwhelming probability according to Schwartz-Zippel lemma.

Shorter Form. Now, using the $3N$-th root of unity is a bit of overkill, so let us try compressing it down to $\Omega = \{\omega^j\}_{0 \le j < N}$ where ω is the Nth root of unity. We will define three polynomials S_{σ_1}, S_{σ_2}, $S_{\sigma_3} \in \mathbb{F}[X]$, which are interpolated as follows:

$$
\begin{aligned}
S_{\sigma_1}(\omega^i) &= \eta^{3k+\ell}, \quad (i,0) \in \mathcal{I}, \quad \sigma((i,0)) = (k,\ell) \\
S_{\sigma_2}(\omega^i) &= \eta^{3k+\ell}, \quad (i,1) \in \mathcal{I}, \quad \sigma((i,1)) = (k,\ell) \\
S_{\sigma_3}(\omega^i) &= \eta^{3k+\ell}, \quad (i,2) \in \mathcal{I}, \quad \sigma((i,2)) = (k,\ell)
\end{aligned}
$$

Let k_1 and k_2 be two field elements such that $\omega^i \neq \omega^j k_1 \neq \omega^\ell k_2$ for all possible triplets i, j, ℓ. Recall that β and γ are random field elements. Let f and g be the polynomials that interpolate, respectively, the following values at Ω:

$$
\begin{aligned}
f(\omega^i) &= \left(T_{i,0} + \omega^i\beta + \gamma\right)\left(T_{i,1} + \omega^i k_1\beta + \gamma\right)\left(T_{i,2} + \omega^i k_2\beta + \gamma\right), \quad i \in [N] \\
g(\omega^i) &= \left(T_{i,0} + S_{\sigma_1}(\omega^i)\beta + \gamma\right)\left(T_{i,0} + S_{\sigma_2}(\omega^i)\beta + \gamma\right)\left(T_{i,0} + S_{\sigma_3}(\omega^i)\beta + \gamma\right).
\end{aligned}
$$

That being said, there is a polynomial $Z \in \mathbb{F}[X]$ such that $\forall d \in \mathcal{D}$ we have $Z(\omega^0) = 1$ and $Z(d)f(d) = g(d)Z(\omega d)$, implying $A = B$ with overwhelming probability. That being said, we now can encode our program using 8 polynomials mentioned at the very beginning:

$$q_L, q_R, q_M, q_O, q_C, S_{\sigma_1}, S_{\sigma_2}, S_{\sigma_3}$$

These are typically called **common preprocessed input**.

0.2.6 Summary

Having a program for relation $\mathcal{R}$, we saw how it can be represented as a sequence of gates with left, right operands and output. The circuit may be encoded using two matrices Q — for capturing gates, and V — for encoding value carries (*wirings*). Upon execution, we get trace execution matrix T and Π for public inputs.

Definition 0.13. Let $T \in \mathbb{F}^{N \times 3}$ be a trace matrix with columns $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C} \in \mathbb{F}^N$ and let $\mathbf{\Pi} \in \mathbb{F}^N$ be a public input vector. They correspond to a valid execution instance with public input given by $\mathbf{\Pi}$ if and only if:

1. $\forall i \in [N] : A_i(Q_L)_i + B_i(Q_R)_i + A_i B_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i + \Pi_i = 0$
2. $\forall (i,j), \forall (k,\ell) : V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$
3. $\forall i > n : \Pi_i = 0$, where n is the number of public inputs.

Then, we encode these conditions in terms of polynomials.

Definition 0.14. Let $z_\Omega(x) = x^N - 1$ be a vanishing polynomial. Let $T \in \mathbb{F}^{N\times 3}$ be a trace matrix with columns $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^N$ and $\mathbf{\Pi} \in \mathbb{F}^N$ be a vector of public signals. They correspond to a valid execution instance with public input given by $\mathbf{\Pi}$ if and only if:

1. $\exists t_1 \in \mathbb{F}[x] : aq_L + bq_R + abq_M + cq_O + q_C + \pi = z_\Omega t_1$
2. $\exists t_2, t_3, z \in \mathbb{F}[x] : zf - gz' = z_\Omega t_2$ and $(z-1)L_1 = z_\Omega t_3$, where $z'(x) = z(\omega x)$.

Remark. We can reduce every needed check down to one equation, if we introduce randomness. Let α be a random field element, then:

$$\begin{aligned} z_\Omega t &= aq_L + bq_R + abq_M + cq_O + q_C + \pi \\ &= \alpha(gz' - fz) \\ &= \alpha^2(z-1)L_1 \end{aligned}$$

The transition between second and third line is very unobvious and requires a bit of algebraic manipulation. Don't worry if you don't see it immediately.

0.3 Plonk Prover and Verifier

In this part, we will observe how the Plonk turns its arithmetization system into the non-interactive proof. We consider how the prover and verifier specifically interact in the protocol.

As in the previous section, let N be the size of our program (number of gates). Let ω be a primitive N-th root of unity. Let $\Omega = \{\omega^j\}_{0\le j<N}$ with the vanishing polynomial $z_\Omega(x) := x^N - 1$.

Assume we have conducted arithmetization of this program, obtaining eight polynomials. Now we will demonstrate commitment-scheme agnostic non-interactive (Fiat-Shamir) algorithms for both prover and verifier.

Remark. Although the further discussion would involve statements like "the prover picks the random $\alpha, \beta, \ldots$", it is important to note that these values are not actually chosen by the prover, but rather computed deterministically from the transcript using Fiat-Shamir heuristic. In case you are not familiar with Fiat-Shamir heuristic, we recommend you to revisit the **??**.

0.3.1 Gadgets

Before diving into the protocol, let us introduce the gadgets that will be used.

Commitment Scheme. We use the polynomial commitment scheme to commit to the polynomials. Similarly to Groth16, one might safely think of KZG commitment scheme to commit to the polynomials: for $p(x) \in \mathbb{F}^{(\le N)}[x]$,

the commitment is $\mathsf{com}(p) = g^{p(\tau)}$, which is evaluated using the powers of the generator g. The opening procedure at point $\zeta \in \mathbb{F}$ of the commitment is simply finding the encryption $g^{Q(\tau)}$ of the quotient polynomial $Q(x) := (p(x)-y)/(x-\zeta)$, where y is the value of the polynomial at ζ. Finally, verification simply involves applying pairing. Note that other commitment schemes might be used as well.

Blindings. In the protocol, to achieve the HZVK property (honest verifier zero-knowledge), the prover must somehow conceal the polynomials he works with. Specifically, consider the polynomial $b(x)$ that is computed as the interpolated values from the trace table (namely, $b(\omega^j) = T_{j,1}$). However, the prover, by providing the opening $b(\zeta)$, should not reveal any information on polynomial $b(x)$ itself. To achieve this, the prover extends $b(x) \in \mathbb{F}^{(\leq N)}[x]$ to the higher-degree polynomial $\widetilde{b}(x)$ with the condition that $\widetilde{b}(\omega^j) = b(\omega^j) = T_{j,1}$. However, for higher degrees, there are multitude of choices for $\widetilde{b}(x)$, so we need to standardize this choice.

Assume we extend $b(x)$ to $\widetilde{b}(x) \in \mathbb{F}^{(\leq N')}[x]$ of higher degree $N' > N$. For convenience, assume $N' = N + \delta$ with $\delta \in \mathbb{N}$. How can we construct such $\widetilde{b}(x)$? We can do this by sampling random values $\gamma_0, \gamma_1, \ldots, \gamma_\delta \xleftarrow{R} \mathbb{F}$ and letting:

$$\widetilde{b}(x) = b(x) + z_\Omega(x) \sum_{j=0}^{\delta} \gamma_j x^j$$

Why does this work? Let us substitute any $\omega^i \in \Omega$:

$$\widetilde{b}(\omega^i) = b(\omega^i) + z_\Omega(\omega^i) \sum_{j=0}^{\delta} \gamma_j \omega^{ij} = b(\omega^i).$$

Here, we use the fact that $z_\Omega(\omega^i) = 0$ for all $i \in [N]$ since z_Ω is, by definition, the vanishing polynomial of $\Omega \ni \omega^i$.

0.3.2 Proving

Let us finally proceed with the prover protocol.

Round 1. Add to the transcript commitments of 8 arithmetizational polynomials:

$$(\mathsf{com}(S_{\sigma_1}), \mathsf{com}(S_{\sigma_2}), \mathsf{com}(S_{\sigma_3}), \mathsf{com}(q_L), \mathsf{com}(q_R), \mathsf{com}(q_M), \mathsf{com}(q_O), \mathsf{com}(q_C))$$

Interpolate polynomials $a'(x), b'(x), c'(x)$ over corresponding columns of T matrix at the domain Ω. Sample random $b_1, b_2, b_3, b_4, b_5, b_6 \xleftarrow{R} \mathbb{F}$. Let the blinded

polynomials be:

$$
\begin{aligned}
a &:= (b_1 x + b_2) z_\Omega(x) + a'(x) \\
b &:= (b_3 x + b_4) z_\Omega(x) + b'(x) \\
c &:= (b_5 x + b_6) z_\Omega(x) + c'(x)
\end{aligned}
$$

Add to the transcript commitments $(\mathsf{com}(a), \mathsf{com}(b), \mathsf{com}(c))$ of computed above polynomials.

Round 2. Sample random scalars $\beta, \gamma \xleftarrow{R} \mathbb{F}$ from the transcript. Let $z_0 = 1$ and define recursively the following sequence $\{z_k\}_{0 \le k < N}$:

$$
z_{k+1} = \frac{(a_k + \beta\omega^k + \gamma)(b_k + \beta\omega^k k_1 + \gamma)(c_k + \beta\omega^k k_2 + \gamma)}{(a_k + \beta S_{\sigma_1}(\omega^k) + \gamma)(b_k + \beta S_{\sigma_2}(\omega^k) + \gamma)(c_k + \beta S_{\sigma_3}(\omega^k) + \gamma)} \cdot z_k.
$$

Interpolate polynomial $z'(x)$ over evaluations $(z_0, \ldots, z_{N-1})$ at the domain Ω. Sample random $b_7, b_8, b_9 \xleftarrow{R} \mathbb{F}$. Let $z(x) = (b_7 x^2 + b_8 x + b_9) z_\Omega(x) + z'(x)$. Add $\mathsf{com}(z)$ to the transcript.

Round 3. Sample $\alpha \xleftarrow{R} \mathbb{F}$ from the transcript. Let $\pi(X)$ be the interpolation polynomial of the input matrix Π at the domain Ω. Define three new polynomials:

$$
\begin{aligned}
p_1(x) &= a q_L + b q_R + ab q_M + c q_O + q_C + \pi \\
p_2(x) &= (a + \beta X + \gamma)(b + \beta k_1 X + \gamma)(c + \beta k_2 X + \gamma) z - \\
&\quad - (a + \beta S_{\sigma_1} + \gamma)(b + \beta S_{\sigma_2} + \gamma)(c + \beta S_{\sigma_3} + \gamma) z(\omega x) \\
p_3(x) &= (z(x) - 1) L_1(x)
\end{aligned}
$$

Define the composite polynomial $p = p_1 + \alpha p_2 + \alpha^2 p_3$. Compute t such that $p = t z_\Omega$. Write $t = t_{\text{low}} + x^{N+2} t_{\text{mid}} + x^{2(N+2)} t_{\text{high}}$, with $t_{\text{low}}, t_{\text{mid}}$, and $t_{\text{high}} \in \mathbb{F}^{\le (N+1)}[x]$ polynomials of degree at most $N + 1$. Sample random $b_{10}, b_{11} \xleftarrow{R} \mathbb{F}$ and define:

$$
\begin{aligned}
t_{\text{low}} &= t'_{\text{low}} + b_{10} x^{N+2} \\
t_{\text{mid}} &= t'_{\text{mid}} - b_{10} + b_{11} x^{N+2} \\
t_{\text{high}} &= t'_{\text{high}} - b_{11}
\end{aligned}
$$

Add to the transcript commitments $(\mathsf{com}(t_{\text{low}}), \mathsf{com}(t_{\text{mid}}), \mathsf{com}(t_{\text{high}}))$.

Round 4. Sample random $\zeta \xleftarrow{R} \mathbb{F}$ from the transcript. Compute:

$$
\bar{a} = a(\zeta), \bar{b} = b(\zeta), \bar{c} = c(\zeta), \bar{S}_{\sigma_1} = S_{\sigma_1}(\zeta), \bar{S}_{\sigma_2} = S_{\sigma_2}(\zeta), \bar{S}_{\sigma_3} = S_{\sigma_3}(\zeta), \bar{z}_\omega = z(\zeta\omega)
$$

and add them to the transcript.

Round 5. Sample random $v \xleftarrow{R} \mathbb{F}$ from the transcript. Let:

$$\begin{aligned}
\hat{p}_{\text{nc},1}(x) &= \bar{a}q_L + \bar{b}q_R + \bar{a}\bar{b}q_M + \bar{c}q_O + q_C \\
\hat{p}_{\text{nc},2}(x) &= (\bar{a} + \beta\zeta + \gamma)(\bar{b} + \beta k_1\zeta + \gamma)(\bar{c} + \beta k_2\zeta + \gamma)z - \\
&\quad - (\bar{a} + \beta\bar{S}_{\sigma_1} + \gamma)(\bar{b} + \beta\bar{S}_{\sigma_2} + \gamma)\beta\bar{z}_\omega S_{\sigma_3} \\
\hat{p}_{\text{nc},3}(x) &= L_1(\zeta)z(x)
\end{aligned}$$

Define:

$$\begin{aligned}
p_{\text{nc}} &= p_{\text{nc},1} + \alpha p_{\text{nc},2} + \alpha^2 p_{\text{nc},3} \\
t_{\text{partial}} &= t_{\text{low}} + \zeta^{N+2} t_{\text{mid}} + \zeta^{2(N+2)} t_{\text{high}}
\end{aligned}$$

The subscript *nc* stands for "non-constant," as it is the part of the linearization of p with nonconstant factors. The subscript *partial* indicates that it is a partial evaluation of t at ζ. Partial means that only some power of x is replaced by the powers of ζ. So in particular $t_{\text{partial}}(\zeta) = t(\zeta)$. Let π_B be the opening (**B**atch) proof at ζ of the polynomial f_B defined as:

$$f_B = t_{\text{partial}} + v p_{\text{nc}} + v^2 a + v^3 b + v^4 c + v^5 S_{\sigma_1} + v^6 S_{\sigma_2}$$

Let π_S be the (**S**ingle) opening proof at $\zeta\omega$ of the polynomial z. Finally, compute $\bar{p}_{\text{nc}} := p_{\text{nc}}(\zeta)$ and $\bar{t} = t(\zeta)$.

Proof. All in all, our proof π looks as follows:

$$\pi = \begin{pmatrix} \text{com}(a), \text{com}(b), \text{com}(c), \text{com}(z), \text{com}(t_{\text{low}}), \text{com}(t_{\text{mid}}), \text{com}(t_{\text{high}}), \\ \bar{a}, \bar{b}, \bar{c}, \bar{S}_{\sigma_1}, \bar{S}_{\sigma_2}, \bar{z}_\omega, \pi_B, \pi_S, \bar{p}_{\text{nc}}, \bar{t} \end{pmatrix}$$

0.3.3 Verification

Transcript Initialization. Similarly to prover, the verifier adds the following commitments to the transcript:

$$(\text{com}(S_{\sigma_1}), \text{com}(S_{\sigma_2}), \text{com}(S_{\sigma_3}), \text{com}(q_L), \text{com}(q_R), \text{com}(q_M), \text{com}(q_O), \text{com}(q_C))$$

Extraction of values and commitments. Firstly, the verifier needs to compute all the challenges. For that, he follows the following steps:

- Add com(a), com(b), com(c) to the transcript.
- Sample two challenges β, γ.
- Add com(z) to the transcript.
- Sample the challenge α.
- Add com(t_{low}), com(t_{mid}), com(t_{high}) to the transcript.
- Sample the challenge ζ.

- Add $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{S}_{\sigma_1}$, $\bar{S}_{\sigma_2}$, $\bar{z}_\omega$ to the transcript.
- Sample the challenge v.

Compute $\pi(\zeta)$. Besides, the verifier needs to compute a few values of all these data. First, he computes the Π matrix with the public inputs and outputs. He needs to compute $\pi(\zeta)$, where $\pi(x)$ is the interpolation of Π over the domain Ω. But he does not need to compute π. He can instead compute $\pi(\zeta)$ as follows:

$$\pi(\zeta) = \sum_{i=0}^{n} L_i(\zeta)\Pi_i,$$

where n is the number of public inputs.

Compute claimed values $p(\zeta)$ and $t(\zeta)$. Next, the verifier computes:

$$\bar{p}_C = p_1(\zeta) + \alpha z_\omega \left(\bar{c} + \gamma\right)\left(\bar{a} + \beta\bar{S}_{\sigma_1} + \gamma\right)\left(\bar{b} + \beta\bar{S}_{\sigma_2} + \gamma\right) - \alpha^2 L_1(\zeta)$$

This is the constant part of the linearization of p. So, adding it to what the prover claims to be $\bar{p}_{\text{nc}}$, he obtains $p(\zeta) = \bar{p}_C + \bar{p}_{\text{nc}}$.

Compute $\mathsf{com}(t_{\text{partial}})$ and $\mathsf{com}(p_{\text{nc}})$. He computes these of the commitments in the proof as follows:

$$\mathsf{com}(t_{\text{partial}}) = \mathsf{com}(t_{\text{low}}) + \zeta^{N+2}\mathsf{com}(t_{\text{mid}}) + \zeta^{2(N+2)}\mathsf{com}(t_{\text{high}})$$

For the second one, compute those:

$$\begin{aligned}
\mathsf{com}(p_{\text{nc},1}) &= \bar{a}\cdot\mathsf{com}(q_L) + \bar{b}\cdot\mathsf{com}(q_R) + \bar{a}\bar{b}\cdot\mathsf{com}(q_M) + \bar{c}\cdot\mathsf{com}(q_O) + \mathsf{com}(q_C)\\
\mathsf{com}(p_{\text{nc},2}) &= (\bar{a} + \beta\zeta + \gamma)(\bar{b} + \beta k_1\zeta + \gamma)(\bar{c} + \beta k_2\zeta + \gamma)\cdot\mathsf{com}(z) -\\
&\quad - (\bar{a} + \beta\bar{S}_{\sigma_1} + \gamma)(\bar{b} + \beta\bar{S}_{\sigma_2} + \gamma)\beta z_\omega\cdot\mathsf{com}(S_{\sigma_3})\\
\mathsf{com}(p_{\text{nc},3}) &= L_1(\zeta)\cdot\mathsf{com}(z)
\end{aligned}$$

Then $\mathsf{com}(p_{\text{nc}}) = \mathsf{com}(p_{\text{nc},1}) + \mathsf{com}(p_{\text{nc},2}) + \mathsf{com}(p_{\text{nc},3})$.

Compute claimed value $f_B(\zeta)$ and $\mathsf{com}(f_B)$. The verifier computes the following two final quantities:

$$\begin{aligned}
f_B(\zeta) &= \bar{t} + v\bar{p}_{\text{nc}} + v^2\bar{a} + v^3\bar{b} + v^4\bar{c} + v^5\bar{S}_{\sigma_1} + v^6\bar{S}_{\sigma_2}\\
\mathsf{com}(f_B) &= \mathsf{com}(t_{\text{partial}}) + v\cdot\mathsf{com}(p_{nc}) + v^2\cdot\mathsf{com}(a) +\\
&\quad + v^3\cdot\mathsf{com}(b) + v^4\cdot\mathsf{com}(c) + v^5\cdot\mathsf{com}(S_{\sigma_1}) + v^6\cdot\mathsf{com}(S_{\sigma_2})
\end{aligned}$$

Proof check. Now the verifier has all the necessary values to proceed with the checks.

- Check that $p(\zeta)$ equals $(\zeta^N - 1)t(\zeta)$.
- Verify the opening of f_B at ζ. Run the check: $\mathsf{Verify}(\mathsf{com}(f_B), \pi_B, \zeta, f_B(\zeta))$.
- Verify the opening of z at $\zeta\omega$. Check the validity of the proof π_S using the commitment $\mathsf{com}(z)$ and the value $\bar{z}_\omega$: $\mathsf{Verify}(\mathsf{com}(z), \pi_S, \zeta\omega, \bar{z}_\omega)$.

If all checks pass, output accept. Otherwise, output reject.