8 Introduction to SNARKs. Arithmetic Circuits. Quadratic Arithmetic Programs

8.1 What is zk-SNARK?

8.1.1 Informal Overview

Finally, we've reached the most interesting part of the course, where we will consider various zk-SNARK constructions we are using on the daily basis. Again, recall that we have the presence of two parties:

- **Prover** $\mathcal{P}$ — the party who knows the data that can resolve the given problem.
- **Verifier** $\mathcal{V}$ — the party that wants to verify the given proof.

Here, the prover wants to convince the verifier that they know the data that resolves the problem (typically, some complex computation) without revealing the data (witness) itself. In the previous lecture, we defined the first practical primitive: zk-NARK — a *zero-knowledge non-interactive argument of knowledge*, and gave the first widely used example: non-interactive Schnorr protocol (which is a special case of a Σ-protocol with the Fiat-Shamir transformation applied). Now, we add one more component which completely changes the game and significantly extends the number of applications: **succinctness**.

> **Definition 8.1. zk-SNARK** — Zero-Knowledge **Succinct** Non-interactive ARgument of Knowledge.

Again, since this is a central question considered, we need to recall what do terms like "argument of knowledge", "succinct", "non-interactive", and "zero-knowledge" mean in this context:

- **Argument of Knowledge** — a proof that the prover knows the data (witness) that resolves a certain problem, and this knowledge can be "extracted".
- **Succinctness** — the proof size and verification time is relatively small relative to the computation size and sometimes even does not depend on the size of the data or statement. This will be explained with examples later.
- **Non-interactiveness** — to produce the proof, the prover does not need any interaction with the verifier.
- **Zero-Knowledge** — the verifier learns nothing about the data used to produce the proof, despite knowing that this data resolves the given problem and that the prover possesses it.

In essence, zk-SNARKs allow one party to prove to another that they know a value without revealing any information about the value itself, and do so with a proof that is both very small and quick to verify. This makes zk-SNARKs a powerful tool for maintaining privacy and efficiency in various cryptographic applications.

This is pretty wide defined and maybe not so obvious if you do not have any background. Let us take a look at the example.

Example. Imagine you are the part of a treasure hunt, and you've found a hidden treasure chest. You want to prove to the treasure hunt organizer that you know where the chest is hidden without revealing its location. Here's how zk-SNARKs can be used in this context:

The problem: you have found a hidden treasure chest (the secret data), and you want to prove to the organizer (the verifier) that you know its location without actually revealing where it is.

How zk-SNARKs Help:
- **Argument of Knowledge**: You create a proof that demonstrates you know the exact location of the treasure chest. This proof convinces the organizer that you have this knowledge.
- **Succinctness**: The proof you provide is very small and concise. It doesn't matter how large the treasure map is or how many steps it took you to find the chest, the proof remains compact and easy to check.
- **Non-interactiveness**: You don't need to have a back-and-forth conversation with the organizer to create this proof. You prepare it once. The organizer can verify it without needing to ask you any questions.
- **Zero-Knowledge**: The proof doesn't reveal any information about the actual location of the treasure chest. The organizer knows you found it, but they don't learn anything about where it is hidden.

Here you can think of zk-SNARK as a golden coin from the chest where the pirates' sign is engraved, so the organizer can be sure you've found the treasure.

But the problems that we want to solve are in a slightly different format. We can't bring a coin to the verifier. Our goal is to prove that we've executed a specific program on a set of data that resolves a specific challenge or gives us a particular result.

8.1.2 Formal Definition

In this section, we will provide a more formal definition of zk-SNARKs. In case you do not want to dive into the technical details, you can skip this part and move to the next sections where we will consider the arithmetic circuits and the Quadratic Arithmetic Programs.

Previously, we considered NARKs that did not require any setup procedure. However, zk-SNARKs are more complex and require a setup phase. This setup phase is used to generate the proving and verification keys (which we call prover parameters pp and verifier parameters vp, respectively), which are then used to create and verify proofs. That being said, let us introduce the **preprocessing NARK**.

Definition 8.2. A **preprocessing non-interactive argument of knowledge** (**preprocessing NARK**) $\Pi_{\text{preNARK}} = (\text{Setup}, \text{Prove}, \text{Verify})$ consists of three algorithms:
- $\text{Setup}(1^\lambda) \to (\text{pp}, \text{vp})$ — the setup algorithm that takes the security parameter λ and outputs the public parameters: proving and verification keys.
- $\text{Prove}(\text{pp}, x, w) \to \pi$ — the proving algorithm that takes the prover parameters pp, statement x, and witness w, and outputs a proof π.
- $\text{Verify}(\text{vp}, x, \pi) \to \{\text{accept}, \text{reject}\}$ — the verification algorithm that takes the verification key, statement x, and proof π, and outputs a bit indicating whether the proof is valid.

Recall, that from NARK (and now preprocessing NARK, respectively) over relation $\mathcal{R}$ we require the following properties:
- **Completeness** — if the prover is honest and the statement is true, the verifier will always accept the proof:

$$\forall (x, w) \in \mathcal{R} : \Pr[\text{Verify}(\text{vp}, x, \text{Prove}(\text{pp}, x, w)) = \text{accept}] = 1$$

- **Knowledge Soundness** — the prover cannot (statistically) generate a false proof π that convinces the verifier.
- **Zero-knowledge** — the verifier "learns nothing" about the witness w from $(\mathcal{R}, \text{pp}, \text{vp}, x, \pi)$.

While we have formally defined all the terms here, including statistical soundness, we have not defined what **knowledge soundness** is. We give a brief informal definition below.

Definition 8.3 (Knowledge Soundness). Π_{preNARK} is (adaptively) **knowledge sound** for a relation $\mathcal{R}$ if for every PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, split into two algorithms, such that:

$$\Pr\left[\text{Verify}(\text{vp}, x, \pi) = \text{accept} \;\middle|\; \begin{array}{c} (\text{pp}, \text{vp}) \leftarrow \text{Setup}(\cdot) \\ x \leftarrow \mathcal{A}_0(\cdot) \\ \pi \leftarrow \mathcal{A}_1(\text{pp}, x) \end{array}\right] > \alpha,$$

where $\alpha = \alpha(\lambda) \neq \text{negl}(\lambda)$ is a non-negligible probability, there exists a PPT extractor $\mathcal{E}^\mathcal{A}$ such that

$$\Pr\left[(x, w) \in \mathcal{R} \;\middle|\; x \leftarrow \mathcal{A}_0(\cdot), \; w \leftarrow \mathcal{E}^\mathcal{A}(x)\right] > \alpha - \epsilon,$$

where $\epsilon = \epsilon(\lambda)$ is a negligible function.

Remark. Informally, the aforementioned definition means that if the prover can generate a false proof with a non-negligible probability, then there exists an extractor that can extract the witness with a probability that is almost as high (and thus is also non-negligible).

Finally, to make zk-NARKs more universal and applicable to a wider range of problems, we introduce the **zk-SNARK** by adding the **succinctness** property.

Definition 8.4. A **zk-SNARK** (Succint NARK) is a preprocessing NARK, where the proof's length $|\pi|$ and verification time $T_\mathcal{V}$ are short: the verification time is sublinear in the size of the computation C (denoted by $|C|$), while the proof size is sublinear in the witness size $|w|$:

$$|\pi| = \mathsf{sublinear}(|w|), \ T_\mathcal{V} = O_\lambda(|x|, \mathsf{sublinear}(|C|)).$$

Remark. Sublinearity means that the function $f : \mathbb{N} \to \mathbb{R}$ grows slower than linearly. For example, functions $f(n) = \log n$ or $f(n) = \sqrt{n}$ are sublinear, while $f(n) = 3n + 2$ is linear. Generally, if $f(n)/(c \cdot n) \xrightarrow[n\to\infty]{} 0$ for any $c \in \mathbb{R} \setminus \{0\}$, then $f(n)$ is sublinear.

Example. Consider the protocol where the proof size is $|\pi| = O(\sqrt{|w|})$ and $T_\mathcal{V} = O(\sqrt[3]{|C|})$. Such protocol is a zk-SNARK, as the proof size is sublinear in the witness size and the verification time is sublinear in the size of the computation.

   Although having a proof size and verification time lower than linear is nice, that is still not sufficient to make zk-SNARKs practical in the wild. For that reason, typically, in practice, we require a stricter definition of the succinctness property, where the proof size and verification time are constant or logarithmic in the size of the computation. This is the case for most zk-SNARKs used in practice.

Definition 8.5. A **zk-SNARK** is **strongly succinct** if the proof size and verification time are constant or logarithmic in the size of the computation:

$$|\pi| = O_\lambda(\log|C|), \ T_\mathcal{V} = O_\lambda(|x|, \log|C|).$$

Example. Consider three major proving systems used in practice with $N = |C|$ being the complexity of a computation:
- **Groth16** with $|\pi| = O_\lambda(1)$, $T_\mathcal{V} = O_\lambda(1)$ is definitely a strongly succinct zk-SNARK since both the proof size and verification time are constant.
- **STARK**s with $|\pi| = O_\lambda(\mathsf{polylog}(N))$ and $T_\mathcal{V} = O_\lambda(\mathsf{polylog}(N))$ are also strongly succinct zk-SNARKs since both the proof size and verification time are logarithmic in the size of the computation.
- **Bulletproofs** with $|\pi| = O_\lambda(\log N)$ and $T_\mathcal{V} = O_\lambda(N)$ is not a strongly succinct zk-SNARK since the verification time is linear in the size of the computation.

8.2 Arithmetic Circuits

8.2.1 What is Arithmetic Circuit?

   The cryptographic tools we have learned in the previous lectures operate with numbers or certain primitives above them (like finite field extensions or elliptic curves), so the first question is: how do we convert a program into a mathematical language? Additionally, we need to do this in a way that can be further (a) made succinct, (b) allows us to prove something about it, and (c) be as universal as possible (to be able to prove quite general statements unlike Σ-protocols considered in the previous lecture).

   The **Arithmetic Circuits** can help us with these problems. Similar to **Boolean Circuits**, they

consist of **gates** and **wires**: gates represent operations acting all elements, connected by wires (see figure below for details). Yet, instead of operations AND, OR, NOT and such, in arithmetic circuits only multiplication/addition/subtraction operations are allowed. Additionally, arithmetic circuits manipulate over elements from some finite field $\mathbb{F}$ (see right figure below).
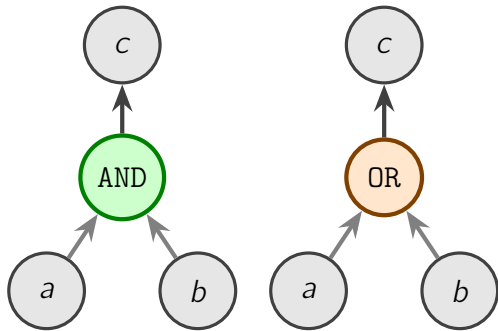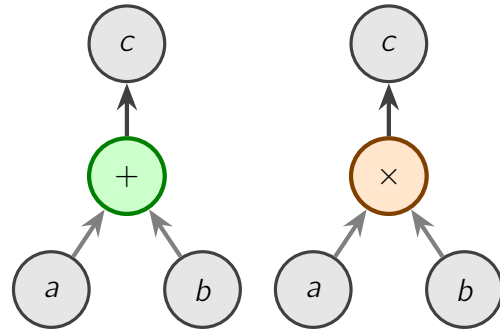

Figure 17: Boolean AND and OR Gates

Figure 18: Addition and Multiplication Gates

Let us come back to boolean circuits for a moment and consider the AND gate. The *AND Gate Truth Table 1* shows us the results we receive if particular values are supplied to the gate. The main point here is that with this table, we can verify the validity of logical statements. Boolean circuits receive an input vector of $\{0, 1\}$ and resolve to true (1) or false (0); basically, they determine if the input values satisfy the statement.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: AND Gate Truth Table

However, more notably, we can combine these gates to create more complex circuits that can resolve more complex problems. For example, we might construct a circuit depicted in Figure 19, calculating $(a \text{ AND } b) \text{ OR } c$.
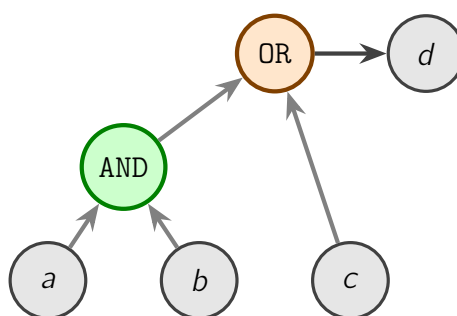

Figure 19: Example of a circuit evaluating $d = (a \text{ AND } b) \text{ OR } c$.

Although we can already represent very complex computations using boolean circuits[13], they are not the most convenient way to represent arithmetic operations.

---

[13]...such as SHA-256 hash function computation, one might take a look here: http://stevengoldfeder.com/projects/circuits/sha2circuit.html

That being said, we can do the same with **arithmetic circuits** to verify computations over some finite field $\mathbb{F}$ without excessive verbosity due to a binary arithmetic, where we had to perceive all intermediate values as binary $\{0, 1\}$.

8.2.2 More advanced examples

Let us take a look at some examples of programs and how can we translate them to the arithmetic circuits.

Example 1: Multiplication. Consider a very simple program, where we are to simply multiply two field elements $a, b \in \mathbb{F}$:

```
def multiply(a: F, b: F) -> F:
    return a * b
```

Since we are doing all the arithmetic in a finite field $\mathbb{F}$, we denote it by F in the code. This can be represented as a circuit with only one (multiplication) gate:

$$r = a \times b$$

The witness vector (essentially, our solution vector) is $\mathbf{w} = (r, a, b)$, for example: $(6, 2, 3)$. We assume that the a and b are input values.

We can think of the "=" in the gate as an assertion, meaning that if $a \times b$ does not equal r, the assertion fails, and the input values do not resolve the circuit.

Good, but this one is quite trivial. Let's consider a more complex example.

Example 2: Multivariate Polynomial. Now, suppose we want to implement the evaluation of the polynomial $Q(x_1, x_2) = x_1^3 + x_2^2 \in \mathbb{F}[X_1, X_2]$ using arithmetic circuits. The corresponding program is as follows:

```
def evaluate(x1: F, x2: F) -> F:
    return x1**3 + x2**2
```

Looks easy, right? But the circuit is now much less trivial. Consider Figure 21. Notice that to calculate x_1^3 we cannot use the single gate: we need to multiply x_1 by itself two times. For that reason, we need three multiplication and one addition gate to represent $Q(x_1, x_2)$ calculation.
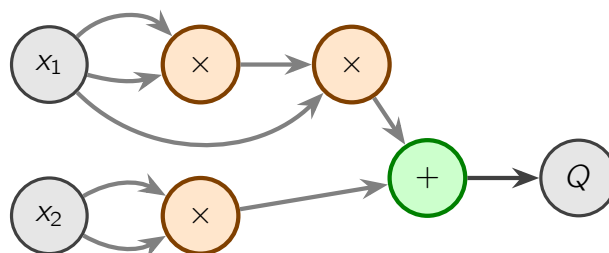

Figure 20: Example of a circuit evaluating $x_1^3 + x_2^2$.

Example 3. `if` **statements.** Well, it is quite clear how to represent any polynomial-like expressions. But how can we translate `if` statements? Consider the program below:

```
def if_statement_example(a: bool, b: F, c: F) -> F:
    return b * c if a else b + c
```

We can express this logic in mathematical terms as follows: "If a is true, compute $b \times c$; otherwise, compute $b + c$." However, only numerical expressions are allowed, so how can we proceed? Assuming that `true` is represented by 1 and `false` by 0, we can transform this logic as follows:

$$r = a \times (b \times c) + (1 - a) \times (b + c)$$

Now, what is the witness vector in this case? One might assume that $\mathbf{w} = (r, a, b, c)$ would suffice. Then, examples of valid witnesses include $(6, 1, 2, 3)$, $(5, 0, 2, 3)$.

But, we need to verify all the intermediate steps! This can be achieved by transforming the above equation using the simplest terms (the gates), ensuring the correctness of each step in the program.

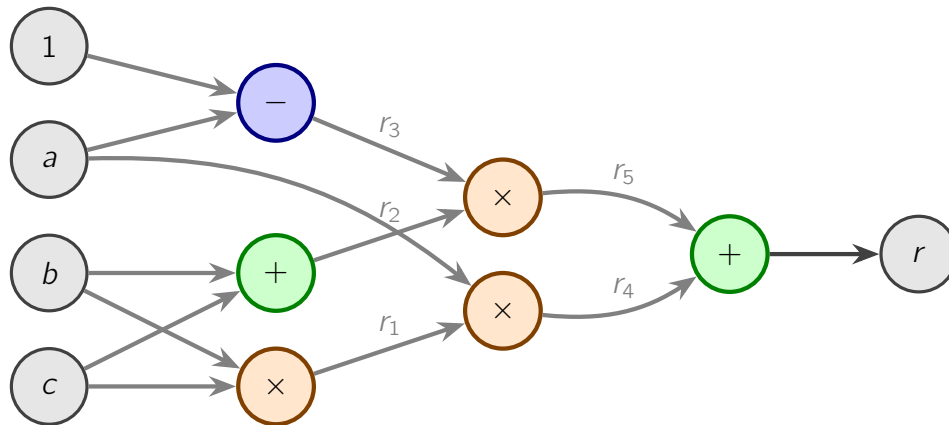Below, we show to visualize the arithmetic circuit for the `if` statement example.

Figure 21: Example of a circuit evaluating the `if` statement logic.

Corresponding equations for the circuit are:

$$r_1 = b \times c \qquad r_2 = b + c$$
$$r_3 = 1 - a \qquad r_4 = a \times r_1$$
$$r_5 = r_3 \times r_2 \qquad r = r_4 + r_5$$

With the witness vector: $\mathbf{w} = (r, r_1, r_2, r_3, r_4, r_5, a, b, c)$. One example of a valid witness is $(6, 6, 5, 0, 6, 0, 1, 2, 3)$.

8.2.3 Circuit Satisfability Problem

Now, let us generalize what we have constructed so far. First, we begin with the arithmetic circuit.

Definition 8.6. Arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ with n inputs over a finite field $\mathbb{F}$ is a directed acyclic graph where internal nodes are labeled via $+$, $-$, and $\times$, and inputs are labeled $1, x_1, x_2, \ldots, x_n$. By $|C|$ we denote the number of gates in the circuit.

Example. For example, previously considered multivariate polynomial $C(x_1, x_2) = x_1^3 + x_2^2$ can be represented as an arithmetic circuit with three multiplication and one addition gates, as shown in Figure 21. It is defined over inputs $\mathbf{x} = (x_1, x_2)$ with $n = 2$ and $|C| = 4$.

Now, suppose that the circuit is defined over n inputs. We can always split this input into two parts: the first ℓ inputs are the *public inputs*, being our statement $\mathbf{x} \in \mathbb{F}^\ell$, and the remaining $n - \ell$ inputs are the *private inputs*, being our secret witness $\mathbf{w} \in \mathbb{F}^{n-\ell}$. The public inputs are known to everyone, while the private inputs are known only to the prover. The goal of the prover is to show that the circuit is satisfiable, i.e., that for the given $\mathbf{x}$, he *knows* a witness $\mathbf{w}$ that resolves the circuit. Resolving in this context means that the output of the circuit is zero.

> **Definition 8.7.** The **Circuit Satisfiability Problem** is defined as follows: given an arithmetic circuit C and a public input $\mathbf{x} \in \mathbb{F}^\ell$, determine if there exists a private input $\mathbf{w} \in \mathbb{F}^{n-\ell}$ such that $C(\mathbf{x}, \mathbf{w}) = 0$. More formally, the problem is determined by relation $\mathcal{R}_C$ and corresponding language $\mathcal{L}_C$ as follows:
>
> $$\mathcal{R}_C = \{(\mathbf{x}, \mathbf{w}) \in \mathbb{F}^\ell \times \mathbb{F}^{n-\ell} : C(\mathbf{x}, \mathbf{w}) = 0\}, \ \mathcal{L}_C = \{\mathbf{x} \in \mathbb{F}^\ell : \exists \mathbf{w} \in \mathbb{F}^{n-\ell}, \ C(\mathbf{x}, \mathbf{w}) = 0\}$$

Let us consider some concrete example of the Circuit Satisfiability Problem.

> **Example.** Suppose our problem (as a prover) is to prove the verifier that we know the point on the circle of "radius $\sqrt{\rho}$"[a], but over the finite field $\mathbb{F}$. More formally, suppose we want to claim that for the given ρ, we have $x_1, x_2 \in \mathbb{F}$ such that:
>
> $$x_1^2 + x_2^2 = \rho$$
>
> For that reason, define the circuit $C(\rho, x_1, x_2) := x_1^2 + x_2^2 - \rho$. It is constructed as shown in the Figure below.
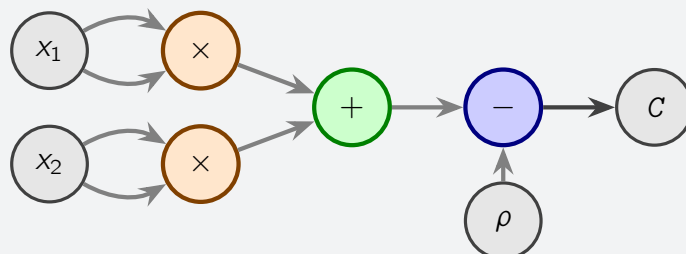>
>
>
> **Illustration:** *Arithmetic circuit for the equation $x_1^2 + x_2^2 - \rho$.*
>
> Now, our statement vector is $\mathbf{x} = \rho \in \mathbb{F}$ (so $\ell = 1$) and the witness vector is $\mathbf{w} = (x_1, x_2) \in \mathbb{F}^2$ (so $n - \ell = 2$). The prover wants to prove that he knows the witness $\mathbf{w}$ such that $C(\mathbf{x}, \mathbf{w}) = 0$. For example, for $\rho = 5$, the prover might have the witness $\mathbf{w} = (2, 1)$ that he wants to show to the verifier[b].
>
> ---
>
> [a]Note that in the finite field the circle equation does not have the geometrical form we are used to (similarly to Elliptic Curve equation, for instance)
> [b]Here, $\mathbb{F} = \mathbb{F}_p$ for some prime $p > 5$

Now, as with any other previously considered proving systems, suppose we are not concerned about the zero-knowledge property and simply want to prove the evaluation integrity of the circuit. Can the prover simply send the witness $\mathbf{w}$ to the verifier? Prover can send the witness, but this will not be a SNARK (and, surely, not a zk-SNARK either).

Proposition 8.8 (Trivial SNARK is not a SNARK)**.** The protocol in which $\mathcal{P}$ sends the witness **w** to $\mathcal{V}$ is not a SNARK for the Circuit Satisfiability Problem. Indeed, in this case, the proof size is $|\pi| = |w|$ (since $\pi = w$) and the verification time is $T_{\mathcal{V}} = O(|\mathcal{C}|)$ (since $\mathcal{C}$ must be evaluated fully). We do not have succinctness (not even mentioning the strong succinctness) in this case.

Proposition above motivates us to look for more advanced techniques to prove the satisfiability of the arithmetic circuits. In the next section, we introduce the Rank-1 Constraint System, which is a more flexible and general way to describe the arithmetic circuits, allowing to further encode the constraints in a more succinct way.

8.3 Rank-1 Constraint System

Almost any program written in high-level programming language can be translated (compiled) into arithmetic circuits, that are really powerfull tool. But for the ZK proof we need slightly different format of it — **Rank-1 Constraint System**, where the simpliest term is **constraint**. This offers a more flexible and general way to describe these parts. However, we need a bit of Linear Algebra to be comfortable with this concept.

8.3.1 Linear Algebra Basics

Although we will not dive deep into the Linear Algebra, we need to understand some basic concepts to be able to work with the Rank-1 Constraint System.

Similarly to group theory working with *groups*, the linear algebra also has a special designated primitive — **vector space**. If previously we were working with the (finite) field $\mathbb{F}$, now we will work with the vector space V over this field. In many practical applications, vector space is formed by **vectors** consisting of a finite fixed collection of elements from the field $\mathbb{F}$. For example, the vector space might be simply $\mathbb{F}^n$: the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ of elements from $\mathbb{F}$. Yet, let us give a bit more general definition.

Definition 8.9. A **vector space** V over the field $\mathbb{F}$ is an abelian group for addition $+$ together with a scalar multiplication operation $\cdot$ from $\mathbb{F} \times V$ to V, sending $(\lambda, x) \mapsto \lambda x$ and such that for any $\mathbf{v}, \mathbf{u} \in V$ and $\lambda, \mu \in \mathbb{F}$ we have:

- $\lambda(\mathbf{u} + \mathbf{v}) = \lambda\mathbf{u} + \lambda\mathbf{v}$
- $(\lambda + \mu)\mathbf{v} = \lambda\mathbf{v} + \mu\mathbf{v}$
- $(\lambda\mu)\mathbf{v} = \lambda(\mu\mathbf{v})$
- $1\mathbf{v} = \mathbf{v}$

Any element $\mathbf{v} \in V$ is called a **vector**, and any element $\lambda \in \mathbb{F}$ is called a **scalar**. We also mark vector elements in boldface.

Example. For example, $V = \mathbb{F}^n$ with operations defined as:

$$\lambda \cdot (x_1, x_2, \ldots, x_n) = (\lambda x_1, \lambda x_2, \ldots, \lambda x_n)$$
$$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$$

is a vector space. Similarly, the following three sets V_1, V_2, V_3 with operations defined above are also valid vector spaces:

$$V_1 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_1 = 0\}$$
$$V_2 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_3 = 2\}$$
$$V_3 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_1 + x_2 + \cdots + x_n = 1\}$$

Now, besides vectors, frequently we are working with the **matrices**. The matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. For example, the matrix A with m rows and n columns, consisting of elements from the finite field $\mathbb{F}$ is denoted as $A \in \mathbb{F}^{m \times n}$. Additionally, we use notation $A = \{a_{i,j}\}_{i,j=1}^{m \times n}$ to denote the square matrix A of size $m \times n$ with elements $a_{i,j}$. Now, let us define operations on matrices.

Definition 8.10. Let A, B be two matrices over the field $\mathbb{F}$. The following operations are defined:
- **Matrix addition/subtraction**: $A \pm B = \{a_{i,j} \pm b_{i,j}\}_{i,j=1}^{m \times n}$. The matrices A and B must have the same size $m \times n$.
- **Scalar multiplication**: $\lambda A = \{\lambda a_{i,j}\}_{1 \leq i,j \leq n}$ for any $\lambda \in \mathbb{F}$.
- **Matrix multiplication**: $C = AB$ is a matrix $C \in \mathbb{F}^{m \times p}$ with elements $c_{i,j} = \sum_{\ell=1}^{n} a_{i,\ell} b_{\ell,j}$. The number of columns in A must be equal to the number of rows in B, that is $A \in \mathbb{F}^{m \times n}$ and $B \in \mathbb{F}^{n \times p}$.

Example. Suppose $\mathbb{F} = \mathbb{R}$. Then, consider

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 3}, \quad B = \begin{bmatrix} 2 & 1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$

We cannot add A and B since they have different sizes. However, we can multiply them:

$$AB = \begin{bmatrix} 5 & 6 \\ 7 & 9 \end{bmatrix}, \quad BA = \begin{bmatrix} 4 & 4 & 5 \\ 7 & 7 & 5 \\ 3 & 3 & 3 \end{bmatrix}$$

To see why, for example, the upper left element of AB is 5, we can calculate it as $\sum_{\ell=1}^{3} a_{1,\ell} b_{\ell,1} = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$.

Remark. Now, we add a very important remark. It just so happens that when working with vectors, we usually assume that they are **column vectors**. This means that the vector $v = (v_1, v_2, \ldots, v_n)$ is represented as a matrix:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

This is a common convention in linear algebra, and we will use it in the following sections.

One important operation we will be frequently working with is the **transpose** of the matrix. The transpose of a matrix is an operator that flips a matrix over its diagonal, that is, it switches the row and column indices of the matrix by producing another matrix denoted as $A^\top$.

Definition 8.11 (Transposition). Given a matrix $A \in \mathbb{F}^{m \times n}$, the **transpose** of A is a matrix $A^\top \in \mathbb{F}^{n \times m}$ with elements $A_{ij}^\top = A_{ji}$.

Example. For example, consider the square matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Then, the transpose of A is $A^\top = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$. However, we can transpose any matrix, for example, the matrix $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ has the transpose $B^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$. Finally, what is probably very important to us, the column vector $\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ has the transpose $\mathbf{v}^\top = [1, 2, 3]$.

Finally, is just happens that we can construct matrix from the vectors. Therefore, let us introduce the corresponding notation.

Definition 8.12 (Composing Matrix from vectors). Suppose we are given n vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \in \mathbb{F}^m$. Then, we might define matrix A as a matrix with columns $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ as follows:

$$A = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{2,1} & \ldots & v_{n,1} \\ v_{1,2} & v_{2,2} & \ldots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,m} & v_{2,m} & \ldots & v_{n,m} \end{bmatrix}$$

Alternatively, vectors might be represented as rows, and the matrix A might be defined as a matrix with rows $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$:

$$A = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{1,2} & \ldots & v_{1,m} \\ v_{2,1} & v_{2,2} & \ldots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \ldots & v_{n,m} \end{bmatrix}$$

Example. For example, consider the vectors $\mathbf{v}_1 = (1, 2, 3)$ and $\mathbf{v}_2 = (4, 5, 6)$. Then, the matrix A with columns $\mathbf{v}_1$ and $\mathbf{v}_2$ is:

$$A = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Similarly, the matrix B with rows $\mathbf{v}_1$ and $\mathbf{v}_2$ is:

$$B = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

We might go on with the vector spaces and define the **linear independence** and **basis** concepts, but for now we will skip them and move to the more important concept for us — the **inner** and **dot products**. Although **inner product** is typically introduced for ordered fields, we give a definition for our finite field $\mathbb{F}$.

Definition 8.13. Consider the vector space $\mathbb{F}^n$. The **inner product** is a function $\langle \cdot, \cdot \rangle : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$ satisfying the following conditions for all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{F}^n$:

- $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$.
- $\langle \mathbf{u}, \mathbf{v} + \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle + \langle \mathbf{u}, \mathbf{w} \rangle$.
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$ for all $\mathbf{u} \in \mathbb{F}^n$ iff $\mathbf{v} = \mathbf{0}$.
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$ for all $\mathbf{v} \in \mathbb{F}^n$ iff $\mathbf{u} = \mathbf{0}$.

Plenty of functions can be built that satisfy the inner product definition, we will use the one that is usually called **dot product**.

Definition 8.14. Consider the vector space $\mathbb{F}^n$. The **dot product** on $\mathbb{F}^n$ is a function $\langle \cdot, \cdot \rangle :$ $\mathbb{V} \times \mathbb{V} \to \mathbb{F}$, defined for every $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ as follows:

$$\langle \mathbf{u}, \mathbf{v} \rangle := \mathbf{u}^\top \mathbf{v} = \sum_{i=1}^{n} u_i v_i$$

Alternatively, the dot product can also be denoted using the dot notation as $\mathbf{u} \cdot \mathbf{v}$. That is why it is called the "dot" product.

Example. Let $\mathbf{u}, \mathbf{v}$ are vectors over the real number $\mathbb{R}$, where

$$\mathbf{u} = (1, 2, 3), \quad \mathbf{v} = (2, 4, 3)$$

Then:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{3} u_i v_i = 2 \cdot 1 + 2 \cdot 4 + 3 \cdot 3 = 2 + 8 + 9 = 19$$

Yet another product we are going to use is the **Hadamard product**. Let us see how it works.

Definition 8.15. Suppose $A, B \in \mathbb{F}^{m \times n}$. The **Hadamard product** $A \odot B$ gives a matrix C such that $C_{i,j} = A_{i,j} B_{i,j}$. Essentially, we multiply elements elementwise.

Example. Consider $A = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 0 & 3 \end{bmatrix}, B = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}$. Then, the Hadamard product is:

$$A \odot B = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 2 & 2 \cdot 1 \\ 3 \cdot 0 & 0 \cdot 2 & 3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

Finally, the final ingredient is the **outer product** and some of its properties. So here it goes!

Definition 8.16. Given two vectors $\mathbf{u} \in \mathbb{F}^n$, $\mathbf{v} \in \mathbb{F}^m$ the **outer product** is a the matrix whose entries are all products of an element in the first vector with an element in the second vector:

$$\mathbf{u} \otimes \mathbf{v} := \mathbf{u}\mathbf{v}^\top = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{bmatrix}$$

Lemma 8.17 (Properties of outer product). For any scalar $c \in \mathbb{F}$ and $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^p$:
- Transpose: $(\mathbf{u} \otimes \mathbf{v}) = (\mathbf{v} \otimes \mathbf{u})^\top$
- Distributivity: $\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = \mathbf{u} \otimes \mathbf{v} + \mathbf{u} \otimes \mathbf{w}$
- Scalar Multiplication: $c(\mathbf{v} \otimes \mathbf{u}) = (c\mathbf{v}) \otimes \mathbf{u} = \mathbf{v} \otimes (c\mathbf{u})$
- Rank: the outer product $\mathbf{u} \otimes \mathbf{v}$ is a rank-1 matrix if $\mathbf{u}$ and $\mathbf{v}$ are non-zero vectors

Example. Let $\mathbf{u}, \mathbf{v}$ are vectors over the real number $\mathbb{R}$, where

$$\mathbf{u} = (1, 2, 3), \quad \mathbf{v} = (2, 4, 3)$$

Then:

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^\top = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 2 & 4 & 3 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 & 1 \cdot 4 & 1 \cdot 3 \\ 2 \cdot 2 & 2 \cdot 4 & 2 \cdot 3 \\ 3 \cdot 2 & 3 \cdot 4 & 3 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 3 \\ 4 & 8 & 6 \\ 6 & 12 & 9 \end{bmatrix}$$

Additionally, as we can see the rows number 2 and 3 in the result matrix can be represented as a linear combination of the first row, specifically by multiplying it by 2 and 3, respectively. The same property applies to the columns. This demonstrates the property of the outer product, that the resulting matrix has a rank of 1.

8.3.2 Constraint Definition

With knowledge of the inner product of two vectors, we can now formulate a definition of the constraint in the context of an R1CS.

Definition 8.18. Each **constraint** in the Rank-1 Constraint System must be in the form:

$$\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$$

Where $\mathbf{w}$ is a vector containing all the *input*, *output*, and *intermediate* variables involved in the computation. The vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are vectors of coefficients corresponding to these variables, and they define the relationship between the linear combinations of $\mathbf{w}$ on the left-hand side and the right-hand side of the equation.

Example. Consider the most basic circuit with one multiplication gate:

$$r = x_1 \times x_2$$

Since we have 3 variables, the constraint is written as:

$$(a_1 w_1 + a_2 w_2 + a_3 w_3)(b_1 w_1 + b_2 w_2 + b_3 w_3) = c_1 w_1 + c_2 w_2 + c_3 w_3$$

Coefficients and witness vectors are: $\mathbf{w} = (r, x_1, x_2), \mathbf{a} = (0, 1, 0), \mathbf{b} = (0, 0, 1), \mathbf{c} = (1, 0, 0)$. Therefore, our expression above reduces to:

$$(0w_1 + 1w_2 + 0w_3)(0w_1 + 0w_2 + 1w_3) = (1w_1 + 0w_2 + 0w_3)$$

$$w_2 \times w_3 = w_1$$

$$x_1 \times x_2 = r$$

The interesting thing is where to take a constants from. The solution is straightforward: by placing 1 in the witness vector, so we can obtain any desired value by multiplying it by an appropriate coefficient.

Example. Now, let us consider a more complex example. Remember that we want to verify each computational step.

```
def r(x1: bool, x2: F, x3: F) -> F:
    return x2 * x3 if x1 else x2 + x3
```

We know that it can be expressed as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3)$$

However, one important consideration was overlooked. If x_1 is neither 0 nor 1, it implies that something else is being computed instead of the desired program. Since we need to add a restriction for x_1: $x_1 \times (1 - x_1) = 0$, this effectively checks that x_1 is binary.
The next constraints can be build:

$$x_1 \times x_1 = x_1 \quad \text{(binary check)} \tag{1}$$
$$x_2 \times x_3 = \text{mult} \tag{2}$$
$$x_1 \times \text{mult} = \text{selectMult} \tag{3}$$
$$(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \tag{4}$$

For every constraint we need the coefficients vectors a_i, b_i, c_i, but all of them have the same witness vector **w**.

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \text{mult}, \text{selectMult})$$

The coefficients vectors:

$$\mathbf{a}_1 = (0, 0, 1, 0, 0, 0, 0) \qquad \mathbf{b}_1 = (0, 0, 1, 0, 0, 0, 0) \qquad \mathbf{c}_1 = (0, 0, 1, 0, 0, 0, 0)$$
$$\mathbf{a}_2 = (0, 0, 0, 1, 0, 0, 0) \qquad \mathbf{b}_2 = (0, 0, 0, 0, 1, 0, 0) \qquad \mathbf{c}_2 = (0, 0, 0, 0, 0, 1, 0)$$
$$\mathbf{a}_3 = (0, 0, 1, 0, 0, 0, 0) \qquad \mathbf{b}_3 = (0, 0, 0, 0, 0, 1, 0) \qquad \mathbf{c}_3 = (0, 0, 0, 0, 0, 0, 1)$$
$$\mathbf{a}_4 = (1, 0, -1, 0, 0, 0, 0) \qquad \mathbf{b}_4 = (0, 0, 0, 1, 1, 0, 0) \qquad \mathbf{c}_4 = (0, 1, 0, 0, 0, 0, -1)$$

Now, let us use some specific values to compute an example. Using the arithmetic in a large finite field $\mathbb{F}_p$, consider the following values:

$$x_1 = 1, \quad x_2 = 3, \quad x_3 = 4$$

Verifying the constraints:
1. $x_1 \times x_1 = x_1 \quad (1 \times 1 = 1)$
2. $x_2 \times x_3 = \text{mult} \quad (3 \times 4 = 12)$
3. $x_1 \times \text{mult} = \text{selectMult} \quad (1 \times 12 = 12)$
4. $(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \quad (0 \times 7 = 12 - 12)$

Each constraint enforces that the product of the linear combinations defined by **a** and **b** must equal the linear combination defined by **c**. Collectively, these constraints describe the computation by ensuring that every step, from inputs through intermediates to outputs, satisfies the defined relationships, thus encoding the entire computational process in the form of a system

of rank-1 quadratic equations.

8.3.3 Why Rank-1?

The last unresolved question is where the "rank-1" comes from. Using the outer product we can express the constraint in another form.

> **Lemma 8.19.** Suppose we have a constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$ with coefficient vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and witness vector $\mathbf{w}$ (all from $\mathbb{F}^n$). Then it can be expressed in the form:
>
> $$\mathbf{w}^\top A \mathbf{w} + \mathbf{c}^\top \mathbf{w} = 0$$
>
> Where A is the outer product of vectors $\mathbf{a}$, $\mathbf{b}$ (denoted as $\mathbf{a} \otimes \mathbf{b}$), consequently a **rank-1** matrix.

Lemma proof. Consider the constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{w} \in \mathbb{F}^n$. Let us expand the inner products:

$$\left(\sum_{i=1}^{n} a_i w_i \right) \times \left(\sum_{j=1}^{n} b_j w_j \right) = \sum_{k=1}^{n} c_k w_k$$

Combine the products into a double sum on the left side:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j w_i w_j = \mathbf{w}^\top (\mathbf{a} \otimes \mathbf{b}) \mathbf{w} = \mathbf{w}^\top A \mathbf{w}$$

Thus, the constraint can be written as:

$$\mathbf{w}^\top A \mathbf{w} + \mathbf{c}^\top \mathbf{w} = 0$$

So, the rank-1 means the rank of the coefficients matrix A in one of the constraint formats.

8.4 Quadratic Arithmetic Program

8.4.1 R1CS in Matrix Form

While the Rank-1 Constraint System provides a powerful way to represent computations, it is not succinct at all, since the number of constraints depends linearly on the complexity of the problem being solved. In practical scenarios, this can require tens or even hundreds of thousands of constraints, sometimes even millions. The Quadratic Arithmetic Program (QAP) can address this issue.

> **Remark.** Understanding polynomials and their properties is crucial for this section. If you are not confident in this area, it is better to revisit the corresponding chapter and refresh your knowledge. See Section 1.4.

To define a constraint in the R1CS we need four vectors: three coefficient vectors ($\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$) and the witness one ($\mathbf{w}$). And that's just for one constraint. As you can imagine, many of the values in these vectors are zeros. In circuits with thousands of inputs, outputs, and auxiliary variables, where there are also thousands of constraints, you could end up with a millions of zeroes.

Remark. A matrix in which most of the elements are zero in numerical analysis is usually called **sparse matrix**.

So, we need to change the way how we manage coefficients and make the representation of such matrices and vectors succint (as required by the definition of SNARK).

Theorem 8.20. Consider a Rank-1 Constraint System (R1CS) defined by m constraints. Each constraint is associated with coefficient vectors $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$, where $i \in \{1, 2, \ldots, m\}$ and also a witness vector $\mathbf{w}$ consisting of n elements.
Then this system can also be represented using the corresponding matrices A, B, and C.

$$
A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{bmatrix} \quad C = \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \ldots & c_{mn} \end{bmatrix}
$$

such that all constraints can be reduced to the single equation:

$$
A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w}
$$

In this representation:

- Each i-th row of the matrices corresponds to the coefficients of a specific constraint.
- Each column of these matrices corresponds to the coefficients associated with a particular element of the witness vector $\mathbf{w}$.

Proof. Matrices defined this way can be expressed as

$$
A = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{b}_1^\top \\ \mathbf{b}_2^\top \\ \vdots \\ \mathbf{b}_m^\top \end{bmatrix}, \quad C = \begin{bmatrix} \mathbf{c}_1^\top \\ \mathbf{c}_2^\top \\ \vdots \\ \mathbf{c}_m^\top \end{bmatrix}
$$

Consider an expression $A\mathbf{w}$:

$$
A\mathbf{w} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{w} \\ \mathbf{a}_2^\top \mathbf{w} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{w} \end{bmatrix}
$$

The last equality is a bit tricky to observe, so let us explain how we ended up with such expression. Notice that since $A \in \mathbb{F}^{m \times n}$ and $\mathbf{w} \in \mathbb{F}^n$, the product $A\mathbf{w}$ is a vector from $\mathbb{F}^m$. Now, for j^{th} element of such vector, based on the matrix product definition, we have $(A\mathbf{w})_j = \sum_{\ell=1}^{n} a_{j,\ell} w_\ell$ which is exactly an inner product between $\mathbf{a}_j$ and $\mathbf{w}$! Therefore, we have:

$$
A\mathbf{w} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{w} \rangle \\ \langle \mathbf{a}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{a}_m, \mathbf{w} \rangle \end{bmatrix}, \quad B\mathbf{w} = \begin{bmatrix} \langle \mathbf{b}_1, \mathbf{w} \rangle \\ \langle \mathbf{b}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{b}_m, \mathbf{w} \rangle \end{bmatrix}, \quad C\mathbf{w} = \begin{bmatrix} \langle \mathbf{c}_1, \mathbf{w} \rangle \\ \langle \mathbf{c}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{c}_m, \mathbf{w} \rangle \end{bmatrix}
$$

Therefore, $A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w}$ is equivalent to the system of m constraints:

$$\langle \mathbf{a}_j, \mathbf{w} \rangle \times \langle \mathbf{b}_j, \mathbf{w} \rangle = \langle \mathbf{c}_j, \mathbf{w} \rangle, \; j \in \{1, \dots, m\}.$$

Example. The vectors $\mathbf{a}_i$ from the previous examples have the form:

$$\mathbf{a}_1 = (0, 0, 1, 0, 0, 0, 0)$$
$$\mathbf{a}_2 = (0, 0, 0, 1, 0, 0, 0)$$
$$\mathbf{a}_3 = (0, 0, 1, 0, 0, 0, 0)$$
$$\mathbf{a}_4 = (1, 0, -1, 0, 0, 0, 0)$$

This corresponds to $n = 7, m = 4$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

8.4.2 Polynomial Interpolation

OK, now is the time to define how we are going to build polynomials! Notice that the columns of these matrices (say, column $(a_{1,i}, a_{2,i}, a_{3,i}, a_{4,i})$ in matrix A from example above) represent the mappings from constraint number i to the corresponding coefficient of the j element in the witness vector!

Example. Consider the witness from the previous examples:

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \mathsf{mult}, \mathsf{selectMult})$$

For element x_1 we are interested in the third columns of the A, B and C matrices, as it's placed on the third position in the witness vector, so $j = 3$.
For matrix A:

$$\begin{bmatrix} 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{0} & 1 & 0 & 0 & 0 \\ 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 \\ 1 & 0 & \boxed{-1} & 0 & 0 & 0 & 0 \end{bmatrix}$$

Thus, for constraint number 4 ($i = 4$) the coefficient of x_1 is -1:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Good, so now we know that for the jth element in the witness vector there are m (the number of constraints) corresponding values in matrices A, B, and C. Now, we want to encode this statement in a form of a polynomial. As we know from the previous chapters, such a mapping in math can be built using the Lagrange polynomial interpolation.

Remark. As a remainder, the Lagrange interpolation polynomial for a given set of points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \subset \mathbb{F} \times \mathbb{F}$ can be built with the following formula:

$$L(x) = \sum_{i=0}^{n} y_i \ell_i(x), \quad \ell_i(x) = \prod_{j=0, j \neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

For a given column $j \in \{1, 2, \ldots, n\}$ in a matrix A the set of points that define the variable polynomial $A_j(x)$ can be defined as $\{(i, a_{ij}) : i \in \{1, 2, \ldots, m\}\}$. In other words, we want to interpolate n polynomials $A_j \in \mathbb{F}[X]$ such that:

$$A_j(i) = a_{i,j}, \ i \in \{1, 2, \ldots, m\}, \ j \in \{1, 2, \ldots, n\}$$

The same is true for matrices B and C, resulting in $3n$ polynomials, n for each of the coefficients matrices:

$$A_1(x), A_2(x), \ldots, A_n(x), B_1(x), B_2(x), \ldots, B_n(x), C_1(x), C_2(x), \ldots, C_n(x)$$

Remark. One might a reasonable question: why do we choose x-coordinates to be the indeces of the corresponding constraints? Actually, just for convenience purposes. We could have assigned any *unique* index from $\mathbb{F}$ to each constraint (say, t_i for each $i \in \{1, \ldots, m\}$) and interpolate through these points:

$$A_j(t_i) = a_{i,j}, \ i \in \{1, 2, \ldots, m\}, \ j \in \{1, 2, \ldots, n\}$$

As we will see in the subsequent lectures, we can define the x-coordinates in much more clever way to reduce the workload needed for interpolation. But for now, we will stick to this simple version.

Example. Considering the witness vector $\mathbf{w}$ and matrix A from the previous example, for the variable x_1, the next set of points can be derived:

$$\{(1, 1), (2, 0), (3, 1), (4, -1)\}$$

We can see that it is used in the 1st, 3rd, and 4th constraints as the values of the coefficients are not zero.

The Lagrange interpolation polynomial for this set of points can be built as follows (for the demonstration purposes, assume we are working in the field $\mathbb{R}$):

$$\ell_1(x) = \frac{(x - 2)(x - 3)(x - 4)}{(1 - 2)(1 - 3)(1 - 4)} = -\frac{(x - 2)(x - 3)(x - 4)}{6},$$

$$\ell_2(x) = \frac{(x - 1)(x - 3)(x - 4)}{(2 - 1)(2 - 3)(2 - 4)} = \frac{(x - 1)(x - 3)(x - 4)}{2},$$

$$\ell_3(x) = \frac{(x - 1)(x - 2)(x - 4)}{(3 - 1)(3 - 2)(3 - 4)} = -\frac{(x - 1)(x - 2)(x - 4)}{2},$$

$$\ell_4(x) = \frac{(x - 1)(x - 2)(x - 3)}{(4 - 1)(4 - 2)(4 - 3)} = \frac{(x - 1)(x - 2)(x - 3)}{6}.$$

Thus, the polynomial is given by:

$$A_1(x) = 1 \cdot \ell_1(x) + 0 \cdot \ell_2(x) + 1 \cdot \ell_3(x) + (-1) \cdot \ell_4(x)$$

$$= -\frac{(x - 2)(x - 3)(x - 4)}{6} - \frac{(x - 1)(x - 2)(x - 4)}{2} - \frac{(x - 1)(x - 2)(x - 3)}{6}$$

$$= -\frac{5}{6}x^3 + 6x^2 - \frac{79}{6}x + 9$$

Therefore, the final Lagrange interpolation polynomial is:

$$A_1(x) = -\frac{5}{6}x^3 + 6x^2 - \frac{79}{6}x + 9$$

As shown in Illustration below, the curve intersects all the given points. In this figure, the x-axis represents the constraint number, and the y-axis represents the coefficients of the x_1 witness element.
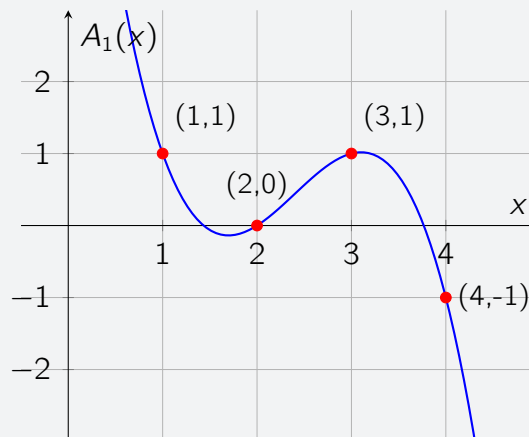

Illustration: The Lagrange interpolation polynomial for points $\{(1, 1), (2, 0), (3, 1), (4, -1)\}$ visualized over $\mathbb{R}$.

Remark. The degree of the coefficient polynomials does not exceed $m - 1$, which follows from the Lagrange interpolation properties.

8.4.3 Putting All Together!

Now, using coefficients encoded with polynomials, a constraint number $X \in \{1, \dots m\}$, from a constraint system with a witness vector $\mathbf{w}$ can be built in the next way:

$$(w_1 A_1(X) + w_2 A_2(X) + \cdots + w_n A_n(X)) \times (w_1 B_1(X) + w_2 B_2(X) + \cdots + w_n B_n(X)) =$$
$$= (w_1 C_1(X) + w_2 C_2(X) + \cdots + w_n C_n(X))$$

Or, written more concisely:

$$\left(\sum_{i=1}^{n} w_i A_i(X) \right) \times \left(\sum_{i=1}^{n} w_i B_i(X) \right) = \left(\sum_{i=1}^{n} w_i C_i(X) \right)$$

Remark. Hold on, but why does it hold? Let us substitute any $X = j$ into this equation:

$$\left(\sum_{i=1}^{n} w_i A_i(j) \right) \times \left(\sum_{i=1}^{n} w_i B_i(j) \right) = \left(\sum_{i=1}^{n} w_i C_i(j) \right) \ \forall j \in \{1, \dots, m\}$$

Recall that we interpolated polynomials to have $A_i(j) = a_{j,i}$. Therefore, the equation above can be reduced to:

$$\left(\sum_{i=1}^{n} w_i a_{j,i} \right) \times \left(\sum_{i=1}^{n} w_i b_{j,i} \right) = \left(\sum_{i=1}^{n} w_i c_{j,i} \right) \ \forall j \in \{1, \dots, m\}$$

But hold on again! Notice that $\sum_{i=1}^{n} w_i a_{j,i} = \langle \mathbf{w}, \mathbf{a}_j \rangle$ and therefore we have:

$$\langle \mathbf{w}, \mathbf{a}_j \rangle \times \langle \mathbf{w}, \mathbf{b}_j \rangle = \langle \mathbf{w}, \mathbf{c}_j \rangle \ \forall j \in \{1, \dots, m\},$$

so we ended up with the initial m constraint equations!

Now let us define polynomials $A(X)$, $B(X)$, $C(X)$ for easier notation:

$$A(X) = \sum_{i=1}^{n} w_i A_i(X), \quad B(X) = \sum_{i=1}^{n} w_i B_i(X), \quad C(X) = \sum_{i=1}^{n} w_i C_i(X)$$

Therefore, our constraint can be rewritten as $A + B = C$ — much less scary-looking than what we have written before. OK, but what does it give us?

Notice that if $A(X) + B(X) = C(X)$ for all $j \in \{1, \dots, m\}$ then polynomial, defined as $P(X) := A(X) + B(X) - C(X)$, has zeros at all elements from the set $\Omega = \{1, \dots, m\}$. Define the so-called **vanishing polynomial** on Ω as:

$$Z_\Omega(X) := \prod_{\omega \in \Omega} (X - \omega) = \prod_{i=1}^{m} (X - i)$$

Now, if $P(X)$ vanishes on all points from Ω, it means that Z_Ω must divide P, so $Z_\Omega \mid P$. But that means that P can be divided by Z_Ω without remainder! In other words, there exists

some polynomial H such that $P = Z_\Omega H$. All in all, let us give the definition of a **Quadratic Arithmetic Program**.

Definition 8.21 (Quadratic Arithmetic Program). Suppose that m R1CS constraints with a witness of size n are written in a form

$$A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w}, \ A, B, C \in \mathbb{F}^{m \times n}$$

Then, the **Quadratic Arithmetic Program** consists of $3n$ polynomials $A_1, \dots, A_n$, $B_1, \dots, B_n$, $C_1, \dots, C_n$ such that:

$$A_j(i) = a_{i,j}, \ B_j(i) = b_{i,j}, \ C_j(i) = c_{i,j}, \ \forall i \in \{1, \dots, m\} \ \forall j \in \{1, \dots, n\}$$

Then, $\mathbf{w} \in \mathbb{F}^n$ is a valid assignment for the given QAP and **target polynomial** $Z_\Omega(X) = \prod_{i=1}^m (X - i)$ if and only if there exists such a polynomial $H(X)$ such that

$$\left(\sum_{i=1}^n w_i A_i(X) \right) \left(\sum_{i=1}^n w_i B_i(X) \right) - \left(\sum_{i=1}^n w_i C_i(X) \right) = Z_\Omega(X) H(X)$$

This was our final step in representing a high-level programming language to some math primitive. We have managed to encode our computation to a single polynomial!

Remark on operations between polynomials

Remark. Some pretty obvious property should be noted. In the theorem **??** it was said about the degree of polynomials after their multiplication or addition, but what about their values?

Let $p(x), q(x) \in \mathbb{F}[x]$ be two polynomials over a field $\mathbb{F}$. Define the polynomial $r(x)$ as the sum of $p(x)$ and $q(x)$:

$$r(x) = p(x) + q(x)$$

Then, for any point $x \in \mathbb{F}$, the value of $r(x)$ is equal to the sum of the values of $p(x)$ and $q(x)$ at that point. Therefore, the set of points corresponding to the polynomial $r(x)$ is given by:

$$\{(x, y) \in \mathbb{F} \times \mathbb{F} \mid x \in \mathbb{F}, \ y = p(x) + q(x)\}$$

The same is true for product.

Example. Consider two polynomials $p(x)$ and $q(x)$ defined over the real numbers $\mathbb{R}$:

$$p(x) = -\frac{1}{2}x^2 + \frac{3}{2}x, \quad q(x) = \frac{1}{3}x^3 - 2x^2 + \frac{8}{3}x + 1.$$

The sets of points $\{(0,0),(1,1),(2,1),(3,0)\}$ and $\{(0,1),(1,2),(2,1),(3,0)\}$ lie on the graphs of $p(x)$ and $q(x)$, respectively.

The sum of these polynomials can be calculated as:

$$\begin{aligned}
r(x) &= (-\frac{1}{2}x^2 + \frac{3}{2}x) + (\frac{1}{3}x^3 - 2x^2 + \frac{8}{3}x + 1) \\
&= \frac{1}{3}x^3 - 2\frac{1}{2}x^2 + 4\frac{1}{6}x + 1
\end{aligned}$$

The resulting polynomial $r(x)$ corresponds to the set of points $\{(0,1),(1,3),(2,2),(3,0)\}$. As you can see (Figure 22), the values at each point for the corresponding x are the sum of the initial polynomials' points.
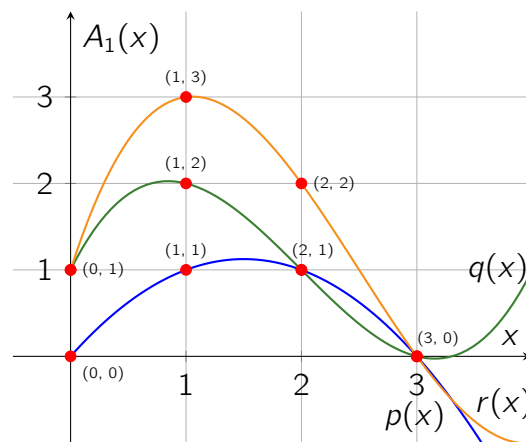

Figure 22: Addition of two polynomials