19 Sum-Check Protocol

19.1 Multivariate Polynomials

In the previous sections, we worked with univariate polynomials $\mathbb{F}[X]$, which are polynomials in a single variable. However, in a set of applications such as *Spartan*, we need to work with multivariate polynomials $\mathbb{F}[X_1, \ldots, X_v]$ where v is the number of variables.

Definition 19.1. We define a *monomial* in v variables as a product of the form: $X_1^{\alpha_1} \ldots X_v^{\alpha_v}$ where $\alpha_1, \ldots, \alpha_v$ are non-negative integers. The *degree of a monomial* is defined as $\alpha_1 + \cdots + \alpha_v$. **Multivariate polynomial** from the space $\mathbb{F}[X_1, \ldots, X_v]$ is defined as a finite linear combination of such monomials where coefficients are from $\mathbb{F}$. The **degree** of a multivariate polynomial is defined as the maximum degree of its monomials.

Example 19.1. For example, $f(X_1, X_2, X_3) = X_1^3 + 3X_1^2X_2^2 + X_3^2 + X_3$ is a linear combination of monomials $\{X_1^3, X_1^2X_2^2, X_3^2, X_3\}$, thus it is a multivariate polynomial in three variables X_1, X_2, and X_3. The maximum degree has a monomial $X_1^2X_2^2$ which is 4, thus $\deg(f) = 4$.

However, in cryptography, we won't work with arbitrary multivariate polynomials, compared to the univariate case. What we need is a *multilinear polynomial*.

Definition 19.2. A **multilinear polynomial** $f(X_1, \ldots, X_v)$ is a multivariate polynomial which is linear in each variable, meaning that each variable appears with degree at most 1. In other words, for each variable X_i, function $f(X_1, \ldots, X_v)$ is a linear function for fixed values of all other variables $X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_v$:

$$\begin{aligned} f(X_1, \ldots, X_v) = \alpha_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_v) \cdot X_i \\ + \beta_i(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_v) \end{aligned}$$

Example 19.2. For example, $f(X_1, X_2, X_3) = X_1X_2 + 3X_1X_3 + X_2X_3$ is a multilinear polynomial in three variables X_1, X_2, and X_3. Note that it is linear in X_1. Indeed, it holds

$$f(X_1, X_2, X_3) = \alpha_1(X_2, X_3)X_1 + \beta_1(X_2, X_3)$$

for $\alpha_1(X_2, X_3) = X_2 + 3X_3$ and $\beta_1(X_2, X_3) = X_2X_3$.
Similarly, $f(X_1, \ldots, X_v) = \prod_{i=1}^{v} X_i$ is also a multilinear polynomial in v variables.

19.2 Multilinear Extension

Compared to univariate polynomials, multilinear polynomials can hold information with a marge smaller degree. For example, if our interpolation domain is $\{0, 1, \ldots, n-1\}$, then we need up to n^{th} degree univariate polynomial to interpolate the values over this domain. However, for multilinear polynomials, we can encode the same domain with only $\log n$-variate multilinear polynomial. Now, the details.

Definition 19.3. Suppose we are given the function $f : \{0,1\}^v \to \mathbb{F}$, mapping the value from the v-dimensional hypercube to the field $\mathbb{F}$. We can define the **extension** of f as a v-variate polynomial $\widetilde{f} \in \mathbb{F}[X_1, \ldots, X_v]$ which agrees with f on the points of the hypercube, meaning that for each $\boldsymbol{b} \in \{0,1\}^v$ it holds that $f(\boldsymbol{b}) = \widetilde{f}(\boldsymbol{b})$.

Example 19.3. A more intuitive way to think about the extension is that given 2^v points of the hypercube, we construct the "interpolation" function $\widetilde{f}$ which agrees with the given values over the hypercube. For example, if $f(0,0) = 1$, $f(0,1) = 2$, $f(1,0) = 2$, and $f(1,1) = 3$, then the extension $\widetilde{f}$ can be defined as $\widetilde{f}(X_1, X_2) = X_1^2 + X_2^2 + 1$.

There are possibly very large number of possible extensions for a given function f. What we are interested in is the multilinear extension, which turns out to be unique.

Theorem 19.4. Any function over the v-dimensional hypercube $f : \{0,1\}^v \to \mathbb{F}$ has a unique v-variate multilinear extension $\widetilde{f} \in \mathbb{F}[X_1, \ldots, X_v]$. It is defined using the *Lagrange interpolation of multilinear polynomials* formula as follows:

$$\widetilde{f}(\mathbf{X}) = \sum_{\boldsymbol{b} \in \{0,1\}^v} f(\boldsymbol{b}) \cdot \mathsf{eq}(\mathbf{X}; \boldsymbol{b}),$$

where the set $\{\mathsf{eq}(\mathbf{X}; \boldsymbol{b})\}_{\boldsymbol{b} \in \{0,1\}^v}$ is referred to as *the set of multilinear Lagrange basis polynomials* over the set $\{0,1\}^v$. Each $\mathsf{eq}(\mathbf{X}; \boldsymbol{b})$ is defined as:

$$\mathsf{eq}(\mathbf{X}; \boldsymbol{b}) \triangleq \prod_{i=1}^{v} \{X_i b_i + (1 - X_i)(1 - b_i)\}.$$

Proof. First, let us show that the provided $\widetilde{f}$ is indeed a multilinear polynomial. Notice that each $\mathsf{eq}(\mathbf{X}; \boldsymbol{b})$ is a multilinear polynomial, since it is a product of linear polynomials in each variable X_i. Thus, $\widetilde{f}$ is a linear combination of multilinear polynomials, hence it is also a multilinear polynomial.

Why is it an extension? Notice that the formula for $\mathsf{eq}(\mathbf{X}; \boldsymbol{b})$ has the following property: $\mathsf{eq}(\mathbf{X}; \boldsymbol{b}) = 1$ if $\mathbf{X} = \boldsymbol{b}$, and 0 otherwise. Indeed, if $X_i \neq b_i$, then the term $X_i b_i + (1 - X_i)(1 - b_i)$ is equal to 0, and thus the whole product is equal to 0. If $X_i = b_i$ for each i, then each term is equal to 1 and thus the product is 1. Therefore, $\widetilde{f}(\boldsymbol{b}')$ for each $\boldsymbol{b}' \in \{0,1\}^v$ is equal to:

$$\widetilde{f}(\boldsymbol{b}') = \sum_{\boldsymbol{b} \in \{0,1\}^v} f(\boldsymbol{b}) \cdot \mathsf{eq}(\boldsymbol{b}'; \boldsymbol{b}) = f(\boldsymbol{b}') \cdot \underbrace{\mathsf{eq}(\boldsymbol{b}'; \boldsymbol{b}')}_{=1} + \sum_{\boldsymbol{b} \neq \boldsymbol{b}'} f(\boldsymbol{b}) \cdot \underbrace{\mathsf{eq}(\boldsymbol{b}'; \boldsymbol{b})}_{=0} = f(\boldsymbol{b}').$$

Example 19.4. Suppose function $f : \{0,1\}^2 \to \mathbb{F}_p$ for $p = 11$ is given by $f(0,0) = 3$, $f(0,1) = 4$, $f(1,0) = 1$, and $f(1,1) = 2$. We first build the multilinear Lagrange basis polynomials:

$$\mathsf{eq}(X_1, X_2; (0,0)) = (1 - X_1)(1 - X_2), \quad \mathsf{eq}(X_1, X_2; (0,1)) = (1 - X_1)X_2,$$
$$\mathsf{eq}(X_1, X_2; (1,0)) = X_1(1 - X_2), \quad \mathsf{eq}(X_1, X_2; (1,1)) = X_1 X_2.$$

The multilinear extension $\widetilde{f}(X_1, X_2)$ is thus computed as:

$$\widetilde{f}(X_1, X_2) = 3(1-X_1)(1-X_2) + 4(1-X_1)X_2 + X_1(1-X_2) + 2X_1X_2 = 3 - 2X_1 + X_2.$$

Now, the question is, how fast can we compute the multilinear extension $\widetilde{f}$ given 2^v values of f? Consider the following lemma.

Lemma 19.5. Fix some positive integer v and let $n = 2^v$. Given an input $f(\boldsymbol{b})$ for all $\boldsymbol{b} \in \{0,1\}^v$ and a vector $\boldsymbol{r} = (r_1, \ldots, r_v) \in \mathbb{F}^{\log n}$, one can compute $\widetilde{f}(\mathbf{r})$ in $\mathcal{O}(n)$ time and space.

19.3 The Sum-Check Protocol

19.3.1 Protocol Description

Suppose we are given the v-variate polynomial (possibly non-multilinear) $f : \{0,1\}^v \to \mathbb{F}$ over a finite field $\mathbb{F}$. The main goal of the Sum-Check protocol is to convince the verifier $\mathcal{V}$ that

$$\sum_{b_1 \in \{0,1\}} \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_v \in \{0,1\}} f(b_1, \ldots, b_v) = H$$

for the given value $H \in \mathbb{F}$. In other words, we can convince that the sum of all the values of f over the hypercube $\{0,1\}^v$ (which we further write as $\sum_{\boldsymbol{b} \in \{0,1\}^v} f(\boldsymbol{b})$ for short) is equal to the given value H. Such check might be useless at first glance, but as it turns out many protocols can be reduced to the Sum-Check protocol, similarly how any NP statement can be encoded as a high-degree univariate polynomial check (see the lecture on QAP).

Why can't the verifier just compute the sum? Typically, v is a fairly large number. According to the Definition 19.5, the verifier can compute the

multilinear extension $\widetilde{f}$ in $\mathcal{O}(2^v)$ time and space, which is infeasible for large v. The Sum-Check protocol allows the verifier to check the sum in up to $\mathcal{O}(v^2)$ time and space, which is much more adequate for large v.

How does it work? First, the prover $\mathcal{P}$ sends the value $C_1 \in \mathbb{F}$, which he claims to be the value of the sum (that is, H). The protocol proceeds in v rounds. For each round j, define the following univariate polynomial in the variable X_j:

$$f_j(X_j) = \sum_{(b_{j+1},\ldots,b_v)\in\{0,1\}^{v-j}} f(r_1,\ldots,r_{j-1},X_j,b_{j+1},\ldots,b_v),$$

where values $r_1,\ldots,r_{j-1} \in \mathbb{F}$ are fixed values of the variables $X_1,\ldots,X_{j-1}$ (which are randomnesses selected during previous rounds).

Consider the first round, when $j = 1$. In such case, according to the definition of $f_j(X_j)$, the prover $\mathcal{P}$ computes the univariate polynomial $f_1(X_1) = \sum_{(b_2,\ldots,b_v)\in\{0,1\}^{v-1}} f(X_1,b_2,\ldots,b_v)$ and sends the *claimed* polynomial $s_1(X_1)$ as the first round message. How can the verifier $\mathcal{V}$ be sure that $s_1(X_1)$ is indeed the univariate polynomial $f_1(X_1)$? Since $f_1(0) + f_1(1) = H$, the verifier can check that $s_1(0) + s_1(1) = C_1$.

However, this is not enough: there are many univariate polynomials that satisfy the condition $s_1(0) + s_1(1) = C_1$. For that reason, we are going to apply the Schwartz-Zippel lemma, which states that as long as $|\mathbb{F}| \gg \deg f_1$, it is safe to check the equality $s_1(r_1) = f_1(r_1)$ at a random point $r_1 \leftarrow\$\ \mathbb{F}$. In such case, the soundness of such check is $1 - \deg f_1/|\mathbb{F}|$.

However, can the verifier evaulate both $s_1(r_1)$ and $f_1(r_1)$ effectively to verify the equality? Good news is that $s_1(r_1)$ can be evaluated efficiently: in fact, in $\mathcal{O}(\deg s_1)$ where typically degree is small (in case of multilinear polynomials, the degree is at most 1). But what about $f_1(r_1)$? Here is the bad news: $f_1(r_1)$ is a sum of 2^{v-1} terms, and thus it cannot be computed efficiently. Another good news is that we do not need to! The idea of the sumcheck protocol is to reduce the computation of $f_1(r_1)$ to the computation of f_2, then f_3 etc. until the computation is trivial.

For concrete example, consider the second round. Now, the prover computes the polynomial $f_2(X_2) = \sum_{(b_3,\ldots,b_v)\in\{0,1\}^{v-2}} f(r_1,X_2,b_3,\ldots,b_v)$ and sends the claimed polynomial $s_2(X_2)$. The verifier now checks that $s_2(0) + s_2(1) = s_1(r_1)$ and that $s_2(r_2) = f_2(r_2)$ at random point $r_2 \leftarrow\$\ \mathbb{F}$. Since computing $f_2(r_2)$ consists in adding up 2^{v-2} terms, it is still infeasible to compute it directly, but we reduced the problem $4\times$ already.

Recursive definition. For the j-th round, the prover computes $f_j(X_j)$ and sends the claimed polynomial $s_j(X_j)$. The verifier checks whether $s_j(0)+s_j(1) = s_{j-1}(r_{j-1})$. Additionally, the verifier rejects if $\deg s_j$ is too large (for example, if $\deg s_j > \deg_j f$ where $\deg_j f$ is a degree of the polynomial in variable X_j). During the last round, the prover sends the claimed value $s_v(r_v)$ and the verifier checks whether $s_v(r_v) = f(r_1,\ldots,r_v)$[15]. If the check succeeds, then the verifier

[15]Here we assume that the verifier has the oracle access to the function f and can compute it at

accepts the claim that

$$\sum_{\boldsymbol{b}\in\{0,1\}^v} f(\boldsymbol{b}) = H.$$

One might ask: how secure is the Sum-Check protocol? Consider the following lemma.

Lemma 19.6. Let $f \in \mathbb{F}[X_1, \ldots, X_v]$ be a multivariate polynomial of degree at most d in each variable, defined over the finite field $\mathbb{F}$. For any given $H \in \mathbb{F}$, let $\mathcal{L}$ be the language of all polynomials f (given as an oracle) such that

$$H = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_v\in\{0,1\}} f(b_1, \ldots, b_v).$$

The sumcheck protocol is an IOP for $\mathcal{L}$ with the completness error $\delta_C = 0$ and the soundness error $\delta_S \leq vd/|\mathbb{F}|$.

Finally, consider the efficiency of the Sum-Check protocol.

Lemma 19.7. Suppose f is given according to the representation in Lemma 19.6. Then, the following holds:

- Communication consists of $\mathcal{O}(dv)$ field elements.
- The verifier $\mathcal{V}$ runs in $\mathcal{O}(vd) + T$ where T is the cost of the oracle access to f.
- The prover $\mathcal{P}$ runs in $\mathcal{O}(2^v T)$.

19.3.2 Sum-Check Protocol Implementation

Let us implement this algorithm in SageMath! First, define the multivariate polynomial ring and the function $f : \{0,1\}^v \to \mathbb{F}$. As an example, we will use $v = 10$ while f will be sampled randomly from $\mathbb{F}_p[X_1, \ldots, X_v]$. As a prime field, we use $p = 2^{31} - 1$. Here is the code that sets everything up:

```
# Defining the finite field GF(p) where p is a large prime
p = (1<<31) - 1
Fp = GF(p)

# Defining the multivariate polynomial ring over the finite field GF(p)
v = 10 # Degree of the polynomial
variable_names = [f'x{i}' for i in range(v)]
R = PolynomialRing(Fp, names=variable_names)
variables = R.gens()
```

any randomly selected point.

Now, let us define f and find the corresponding value $H = \sum_{\boldsymbol{b} \in \{0,1\}^v} f(\boldsymbol{b})$. For that, we simply take the random element of the polynomial ring R and manually add up all 2^v values over the boolean hypercube $\{0,1\}^v$.

```
def boolean_hypercube_sum(f: PolynomialRing) -> Fp:
    """
    Computes the sum of the polynomial f over the boolean hypercube {0,1}^v.
    The boolean hypercube is represented by evaluating the polynomial at all
    combinations of 0 and 1 for each variable.
    """

    total_sum = Fp(0)
    for i in range(1<<v):
        # Convert i to a binary representation of length v
        binary_representation = [(i >> j) & 1 for j in range(v)]
        # Evaluate the polynomial at this point
        point_value = f(*binary_representation)
        total_sum += point_value
    return total_sum

f = R.random_element(degree=v) # Random polynomial
H = boolean_hypercube_sum(f)
```

The SageMath generated the following f and H:

$$f(X_1, \ldots, X_{10}) = 226921892X_2^3X_3X_4^2X_5X_6X_8^2 + 274566947X_0X_1^2X_2^2X_4X_7^2X_8X_9 + 850030718X_1^2X_4X_6^3X_7^2X_8X_9 + 560601732X_0X_3X_5X_6X_7X_8X_9^4 - 39$$

$$H = \sum_{\boldsymbol{b} \in \{0,1\}^{10}} f(\boldsymbol{b}) = 1053620759.$$

Now, we are going to implement the prover algorithm. For that, we need to define the univariate polynomial $f_j(X_j)$ for each round j and compute the claimed polynomial $s_j(X_j)$ based on previously sampled randomness values $r_1, \ldots, r_{j-1}$. Since our protocol is a *public-coin* protocol, we can apply the *Fiat-Shamir transform* to make the randomness values $r_1, \ldots, r_v$ deterministically computable by the prover (thus making the protocol non-interactive). For that, we will compute the challenge r_j as $\mathcal{H}(s_1, s_2, \ldots, s_{j-1})$ where $\mathcal{H} : \{0,1\}^* \to \mathbb{F}_p$ is a cryptographic hash function (SHA256 in our case). That said, we define $\mathcal{H}$ as follows:

```
def fiat_shamir_sample(input_string: str) -> Fp:
    """
    Samples a random value from the finite field GF(p) using the Fiat-Shamir he
    The input string is hashed to produce a random value.
    """

    fs_hash = hashlib.sha256(input_string.encode('utf-8')).hexdigest()
```

```
    return Fp(int(fs_hash, 16) % p)
```

Finally, the verifier logic is even more simple: the verifier checks that (a) randomnesses $\{r_j\}_{j\in\{1,\ldots,10\}}$ are sampled correctly, (b) the claimed polynomials $s_j(X_j)$ satisfy the conditions $s_j(0)+s_j(1) = s_{j-1}(r_{j-1})$ for each $j \in \{1,\ldots,10\}$ (instead of s_0, we simply use the claimed value H), and (c) the final check $s_v(r_v) = f(r_1,\ldots,r_v)$ holds. Additionally, we should check the degrees of each s_j, but we omit this check for simplicity. Here is the implementation of the prover and verifier:

```
class SumCheckProtocol:
    """
    Prover and Verifier for the Sum Check protocol.
    """

    def __init__(
        self,
        polynomial: PolynomialRing,
        claimed_sum: Fp
    ) -> None:
        """
        Initializes the SumCheck protocol with a polynomial and the claimed sum

        Args:
            polynomial (PolynomialRing): The polynomial to be checked.
            claimed_sum (Fp): The sum claimed by the prover over the boolean hy
        """

        self.f = polynomial
        self.H = claimed_sum


    @staticmethod
    def fiat_shamir_from_polynomials(polynomials: list[PolynomialRing]) -> Fp:
        """
        Samples a random value from the finite field GF(p) using the Fiat-Shami
        based on the provided polynomials.

        Args:
            polynomials (list[PolynomialRing]): List of polynomials to be used

        Returns:
            Fp: A random value from the finite field GF(p).
        """

        fs_input = ''.join(str(poly) for poly in polynomials)
        return fiat_shamir_sample(fs_input)
```

```
    def prove(self) -> dict:
        """
        Prover generates the transcript of the polynomial and the claimed sum.

        Returns:
            dict: A transcript containing the polynomial, claimed sum, random v
        """

        # Initialize the transcript representing the
        # interaction between the prover and verifier.
        transcript = {
            'polynomial': self.f,
            'claimed_sum': self.H,
            'random_values': [],
            'polynomials': []
        }

        for j in range(v):
            # The protocol consists of v rounds.
            # Finding polynomial s_j that represents the sum of f over the bool
            Rj = PolynomialRing(Fp, 'x')
            x = Rj.gen()
            s_j = Rj.zero()

            for i in range(2**(v-j-1)):
                # Convert i to a binary representation of length v-j-1
                binary_representation = [(i >> k) & 1 for k in range(v-j-1)]
                # Evaluate the polynomial at this point
                point_value = self.f(*transcript['random_values'][:j], variable
                univariate_poly = point_value.subs({variables[j]: x})
                s_j += univariate_poly

            # Append the polynomial s_j to the transcript
            transcript['polynomials'].append(s_j)

            # Sample the random value using Fiat-Shamir heuristic
            random_value = SumCheckProtocol.fiat_shamir_from_polynomials(
                polynomials=transcript['polynomials']
            )
            transcript['random_values'].append(random_value)

        return transcript


    def verify(self, transcript: dict) -> bool:
        """
        Verifier checks the validity of the transcript.
```

```
        Args:
            transcript (dict): The transcript generated by the prover.

        Returns:
            bool: True if the claimed sum is valid, False otherwise.
        """

        # Decipher the transcript
        H = transcript['claimed_sum']
        f = transcript['polynomial']
        random_values = transcript['random_values']
        polynomials = transcript['polynomials']

        # Assert that random_values is formed correctly using Fiat-Shamir heuri
        if len(random_values) != v:
            print("Invalid number of random values in the transcript.")
            return False
        for i in range(v):
            if random_values[i] != self.fiat_shamir_from_polynomials(polynomial
                print(f"Random value at index {i} does not match the Fiat-Shami
                return False

        # Assert that each s_r(0) + s_r(1) matches the claimed sum H
        for j in range(v):
            # During the first round, simply check that s_1(0) + s_1(1) = H
            if j == 0:
                s_1 = polynomials[0]
                if s_1(0) + s_1(1) != H:
                    print(f"""First round sum {s_1(0) + s_1(1)}
                            does not match the claimed sum {H}.""")
                    return False
                continue

            # For subsequent rounds, check that the sum of the polynomials matc
            s, s_previous = polynomials[j], polynomials[j-1]
            if s(0) + s(1) != s_previous(random_values[j-1]):
                print(f"""Round {j} sum {s(0) + s(1)} does not match the
                        previous round's output {s_previous(random_values[j-1])
                return False

        # Final round checks whether f(r) = s_v(r_v)
        last_polynomial = polynomials[-1]
        last_random = random_values[-1]

        if last_polynomial(last_random) != f(*random_values):
            print(f"Final check failed: {last_polynomial(last_random)} != {f(*r
            return False

        return True
```

```
protocol = SumCheckProtocol(f, H)
transcript = protocol.prove()
print(f'Prover has the transcript: {transcript}')
print(f'Verifier checks the proof: {protocol.verify(transcript)}')
```

The example transcript is the following:

$$
\begin{aligned}
s_1(X) &= 763349684X^2 + 238898491X + 25686292, \quad r_1 = 493136960 \\
s_2(X) &= 1430156880X^2 + 361631142, \quad r_2 = 2831006 \\
s_3(X) &= 658473518X^3 + 208437747X^2 + 1648222287, \quad r_3 = 321757611 \\
s_4(X) &= 111234379X + 1501686780, \quad r_4 = 1835658 \\
s_5(X) &= 2135686754X^2 + 1501686780X + 1698421434, \quad r_5 = 1970078146 \\
s_6(X) &= 53232575X + 259099570, \quad r_6 = 1616339175 \\
s_7(X) &= 669263002X^3 + 514137765X + 1942401931, \quad r_7 = 887816643 \\
s_8(X) &= 1524551309X^2 + 1551518026X + 55160592, \quad r_8 = 421872749 \\
s_9(X) &= 1766229428X^2 + 826393776X + 1291949229, \quad r_9 = 1169032581 \\
s_{10}(X) &= 2030644729X^4 + 1291949229X^3 + 560585988X + 1720313399, \\
r_{10} &= 1461328437
\end{aligned}
$$

19.4 Sumcheck Applications

As of now, the Sum-Check protocol seems too abstract and not very useful: why do we even need to sum some random multivariate polynomial over the boolean hypercube? In this section, we provide some (rather theoretic) applications of the Sum-Check protocol, but as we go further, it will become more clear why it is so important.

19.4.1 The #SAT Problem

Let $\mathtt{C} : \{0,1\}^\ell \to \{0,1\}$ be any boolean formula of size $S = \mathcal{O}(\mathsf{poly}(\ell))$. In the #SAT problem, the goal is to compute the number of satisfying (boolean) assignments of $\mathtt{C}$, that is, find the value $H = \sum_{\boldsymbol{b} \in \{0,1\}^\ell} \mathtt{C}(\boldsymbol{b})$. Such problem is believed to be very difficult with the fastest known algorithm to still run in the exponential time: that is, no much better than brute-force the formula in time $\mathcal{O}(2^\ell S)$. Even determining whether $H > 0$ is widely believed to be NP-hard. But suppose we have a prover $\mathcal{P}$ who does know H and wants to prove any verifier $\mathcal{V}$ that H was computed correctly in the polynomial time.

Now, similarly to how it is done in R1CS, we *arithmetize* the circuit $\mathtt{C}$ to be computable over the finite field $\mathbb{F}$ (let us call it $\widetilde{\mathtt{C}} : \mathbb{F}^\ell \to \mathbb{F}$). Here how it goes:

- Instead of the gate $x \wedge y$, we use the multiplication $x \cdot y$.
- Instead of the gate $x \vee y$, we use the addition $x + y - x \cdot y$.

- Instead of the gate $\neg x$, we use the subtraction $1 - x$.
- Instead of the gate $x \oplus y$, we use the addition $x + y - 2xy$.

As an example, suppose we have the circuit:

$$\mathtt{C}(x_1, x_2, x_3, x_4) = (\neg x_1 \wedge x_2) \wedge (x_3 \vee x_4)$$

The extension $\widetilde{\mathtt{C}}$ of the circuit $\mathtt{C}$ is

$$\widetilde{\mathtt{C}}(X_1, X_2, X_3, X_4) = (1 - X_1)X_2(X_3 + X_4 - X_3X_4).$$

It is fairly easy to see that $\widetilde{\mathtt{C}}(\boldsymbol{b}) = \mathtt{C}(\boldsymbol{b})$ for all $\boldsymbol{b} \in \{0,1\}^\ell$ with $\ell = 4$. Then, we can apply the sum-check protocol over $\widetilde{\mathtt{C}}$ to prove that H was computed correctly. In such case, the prover runs in $\mathcal{O}(S^2 2^\ell)$ time, while the verifier in time $\mathcal{O}(S)$.

19.4.2 Matrix Multiplication Verification (MatMul Check-Sum Protocol)

Now, here is the application which gets much more interesting. Consider two matrices $A, B \in \mathbb{F}^{n \times n}$ and the goal is to verify that the product $C = A \cdot B$ is computed correctly. The naive way is to make verifier compute the product, taking $\mathcal{O}(n^3)$ time, and then compare the result with the claimed value C. However, it is possible to verify the correctness of the product in $\mathcal{O}(n^2)$ time and space using the *Freivelds'* protocol by the simple observation that we can sample a random challenge $\boldsymbol{\alpha} \leftarrow\!\!\$\ \mathbb{F}^n$ and verify that $A(B\boldsymbol{\alpha}) = C\boldsymbol{\alpha}$ (idea is pretty similar to the Schwartz-Zippel lemma where we check the polynomial equation at a random point). However, is there any better way to do that? Sumcheck protocol can help to keep the same assymptotics, but do not reveal the whole matrices A, B to the verifier.

Protocol. Here is where *multilinear extension* comes into play. Instead of perceiving matrices A, B, C as n^2 field elements, we perceive them as functions $f_A, f_B, f_C : \{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \mathbb{F}$, mapping two indices of the matrix to the corresponding value. This way, for instance:

$$f_A(\boldsymbol{i}, \boldsymbol{j}) = A_{i,j}, \quad \text{where} \quad \boldsymbol{i} = (i_1, \ldots, i_{\log n}),\ \boldsymbol{j} = (j_1, \ldots, j_{\log n}).$$

Denote by $\widetilde{f}_A, \widetilde{f}_B, \widetilde{f}_C : \mathbb{F}^{\log n} \times \mathbb{F}^{\log n} \to \mathbb{F}$ the multilinear extensions of the functions f_A, f_B, f_C. Now, how to reduce the seemingly difficult check $C = AB$ into the Sum-Check protocol? Consider the following lemma.

Lemma 19.8. $\widetilde{f}_C(\mathbf{x}, \mathbf{y}) = \sum_{\boldsymbol{b} \in \{0,1\}^{\log n}} \widetilde{f}_A(\mathbf{x}, \boldsymbol{b}) \widetilde{f}_B(\boldsymbol{b}, \mathbf{y})$.

Proof. Obviously, both left and right-hand sides are multilinear polynomials in $\mathbf{x}$ and $\mathbf{y}$. Since multilinear extension of C is unique, it suffices to check that the equality holds for all boolean assignments $\boldsymbol{i}, \boldsymbol{j} \in \{0,1\}^{\log n}$. Indeed, we have $\widetilde{f}_C(\boldsymbol{i}, \boldsymbol{j}) = \sum_{\boldsymbol{b} \in \{0,1\}^{\log n}} \widetilde{f}_A(\boldsymbol{i}, \boldsymbol{b}) \widetilde{f}_B(\boldsymbol{b}, \boldsymbol{j})$. But notice that this check literally

checks whether $C_{i,j} = \sum_{b=1}^{n} A_{i,b}B_{b,j}$ which is exactly the definition of matrix multiplication!

Now, the interactive proof is immediate: sample random $\boldsymbol{r}_1, \boldsymbol{r}_2 \xleftarrow{\$} \mathbb{F}^{\log n}$, and apply the sum-check on $g(\mathbf{z}) := \widetilde{f}_A(\boldsymbol{r}_1, \mathbf{z})\widetilde{f}_B(\mathbf{z}, \boldsymbol{r}_2)$ to prove that it equals $\widetilde{f}_C(\boldsymbol{r}_1, \boldsymbol{r}_2)$.

It can be shown that both the prover's and verifier's time is $\mathcal{O}(n^2)$, while the proof size is $\mathcal{O}(\log n)$ (compared to $\mathcal{O}(n^2)$ for the naive approach).

19.4.3 MatMul Sum-Check Protocol Implementation

Now, let us implement the matrix multiplication verification protocol in SageMath! For simplicity, we assume that the matrix has a size $n = 2^v$ so that MLE of matrices are of degree $2v$. Also, we will work over the small prime field $\mathbb{F}_{11}$ this time to make outputs somewhat more readable. So here we go:

```
# Defining the finite field GF(p) where p is a large prime
p = 11
Fp = GF(p)

# Assume for simplicity that matrices are of size 2^v x 2^v
v = 2 # Matrix of size 4x4
n = 1<<v

# Defining how MLE are computed
variable_names = [f'x{i}' for i in range(2*v)]
R = PolynomialRing(Fp, names=variable_names)
variables = R.gens()
```

Now, for concreteness, we initialize the matrices A, B and $C = AB$.

```
# Generate two random matrices A and B over the finite field GF(p)
A = Matrix(Fp, n, n, [Fp.random_element() for _ in range(n*n)])
B = Matrix(Fp, n, n, [Fp.random_element() for _ in range(n*n)])

# Find the product matrix C=A*B
C = A * B
print(f'Matrix A:\n{A}')
print(f'Matrix B:\n{B}')
print(f'Matrix C:\n{C}')
```

Here, we obtained the following matrices:

$$A = \begin{bmatrix} 1 & 1 & 3 & 10 \\ 5 & 6 & 4 & 5 \\ 1 & 8 & 3 & 4 \\ 6 & 2 & 2 & 7 \end{bmatrix}, \quad B = \begin{bmatrix} 2 & 7 & 9 & 5 \\ 5 & 10 & 0 & 10 \\ 5 & 0 & 2 & 3 \\ 2 & 6 & 6 & 6 \end{bmatrix}, \quad C = \begin{bmatrix} 9 & 0 & 9 & 7 \\ 4 & 4 & 6 & 6 \\ 10 & 1 & 6 & 8 \\ 2 & 5 & 1 & 10 \end{bmatrix}$$

Now, we need to build the multilinear extensions $\widetilde{f_A}, \widetilde{f_B}, \widetilde{f_C} : \mathbb{F}^{2\times 2} \to \mathbb{F}$. To achieve that, we first write down the function to compute the MLE given the set of values $\{(\boldsymbol{b}, f(\boldsymbol{b}))\}_{\boldsymbol{b}\in\{0,1\}^\ell}$ given interpolation formula from Theorem 19.4.

```
def mle_from_hypercube(hypercube: list) -> R:
    """
    Computes the Multivariate Linear Extension (MLE) of a hypercube.
    The hypercube is represented as a list of tuples, where each tuple
    contains the coordinates of a point in the hypercube.

    Args:
        hypercube (list): A list of tuples representing points in the hypercube
                          Each tuple should have length equal to the dimensio
    """

    # Create a polynomial for each point in the hypercube
    mle = R.zero()
    for point, value in hypercube:
        eq_poly = R(1)
        for i, bit in enumerate(point):
            eq_poly *= bit*variables[i] + (1-bit)*(1-variables[i])

        mle += eq_poly * value

    return mle
```

We can verify the correctness of this function by running the Example in the first section. Indeed:

```
print('Example MLE:', mle_from_hypercube([
    ((0, 0), Fp(3)),
    ((0, 1), Fp(4)),
    ((1, 0), Fp(1)),
    ((1, 1), Fp(2))
])) # Output: -2*x0 + x1 + 3
```

At this point, we can finally derive the MLEs. We simply iterate through all indices (i, j), bit-decompose them to get bit-vectors $\boldsymbol{i} = (i_1, i_2)$ and $\boldsymbol{j} = (j_1, j_2)$, and then interpolate the MLEs: for matrix A, for instance, we have $\{(i_1, i_2, j_1, j_2), A_{i,j}\}_{i,j\in[n]^2}$.

```
def mle_from_matrix(matrix: Matrix) -> R:
    """
    Computes the Multivariate Linear Extension (MLE) of a matrix.
    The MLE is a polynomial that represents the matrix entries as variables.
    """
```

```
    assert matrix.nrows() == matrix.ncols(), "Matrix must be square."
    assert matrix.nrows() == n, "Matrix size must be 1<<v"

    # Range over all indices, bit-decompose them, and build the mle
    hypercube = []
    for i in range(n):
        for j in range(n):
            # Convert i and j to binary representation of length v
            point = [(i >> k) & 1 for k in range(v)] + [(j >> k) & 1 for k in r
            hypercube.append((tuple(point), matrix[i, j]))

    return mle_from_hypercube(hypercube)
```

Now we find the MLEs of matrices A, B, and C:

```
A_mle = mle_from_matrix(A)
B_mle = mle_from_matrix(B)
C_mle = mle_from_matrix(C)
print(f'MLE of A:\n{A_mle}')
print(f'MLE of B:\n{B_mle}')
print(f'MLE of C:\n{C_mle}')
```

The results are the following:

$$\begin{aligned}
\widetilde{f}_A(X_1, X_2, Y_1, Y_2) = &-X_1X_2Y_1 - 3X_1X_2Y_2 + 4X_1Y_1Y_2 - 2X_2Y_1Y_2 \\
&+ X_1X_2 + X_1Y_1 - 4X_2Y_1 - 3X_1Y_2 - 4Y_1Y_2 \\
&+ 4X_1 + 2Y_2 + 1 \\
\widetilde{f}_B(X_1, X_2, Y_1, Y_2) = &-2X_1X_2Y_1Y_2 - 2X_1X_2Y_1 - 3X_1X_2Y_2 + 3X_1Y_1Y_2 \\
&+ 4X_2Y_1Y_2 + 5X_1X_2 + X_2Y_1 - X_1Y_2 + X_2Y_2 \\
&+ 2Y_1Y_2 + 3X_1 + 3X_2 + 5Y_1 - 4Y_2 + 2 \\
\widetilde{f}_C(X_1, X_2, Y_1, Y_2) = &\ 2X_1X_2Y_1Y_2 + 3X_1X_2Y_1 + X_1X_2Y_2 + 4X_1Y_1Y_2 \\
&+ 4X_2Y_1Y_2 - 3X_1X_2 - 2X_1Y_1 + 2X_1Y_2 - 4X_2Y_2 \\
&- 4Y_1Y_2 - 5X_1 + X_2 + 2Y_1 - 2
\end{aligned}$$

One can easily check that they indeed encode the matrices A, B, and C as expected. For example, it is trivial to check the upper left element in all three cases:

$$\widetilde{f}_A(0,0,0,0) = 1, \quad \widetilde{f}_B(0,0,0,0) = 2, \quad \widetilde{f}_C(0,0,0,0) = 9.$$

Now, we can implement the Sum-Check protocol to verify that $C = AB$ holds. We first sample randomnesses $\boldsymbol{r}_1, \boldsymbol{r}_2 \leftarrow\!\$\ \mathbb{F}_{11}^2$:

```
# Now, we are going to apply the sumcheck protocol. First,
```

```
# sample two random vectors r1 and r2 of size v = log(n)
r1 = [Fp.random_element() for _ in range(v)]
r2 = [Fp.random_element() for _ in range(v)]
```

Then, we define the function $g(\mathbf{z}) = \widetilde{f}_A(\boldsymbol{r}_1, \mathbf{z})\widetilde{f}_B(\mathbf{z}, \boldsymbol{r}_2)$ and apply the Sum-Check protocol to prove that it equals $\widetilde{f}_C(\boldsymbol{r}_1, \boldsymbol{r}_2)$. For that, we first define the function g:

```
claimed_sum = C_mle(*(r1 + r2))
g = A_mle.subs({
    variables[i]: r1[i] for i in range(v)
}).subs({
    variables[v+i]: variables[i] for i in range(v)
}) * B_mle.subs({
    variables[i+v]: r2[i] for i in range(v)
})
```

Turnes out that the claimed sum is $H = 6$ while the polynomial is:

$$g(X_1, X_2) = 5X_1^2X_2^2 - 5X_1^2X_2 + 3X_1X_2^2 + 3X_1^2 + 4X_1X_2 - 3X_2^2 + 4X_1 - 3X_2 + 2$$

So the only thing left is to run the already implemented Sum-Check protocol. Here how it goes:

```
# Defining a smaller polynomial ring for the protocol (since g
# is polynomial in v variables instead of 2*v)
Q = PolynomialRing(Fp, names=variable_names)
variables = Q.gens()
g = Q(g)

# Running the sum-check protocol
protocol = SumCheckProtocol(Fp, Q, g, claimed_sum, degree=v)
transcript = protocol.prove()
print_transcript(transcript)
verification_result = protocol.verify(transcript)
print(f'Verification result: {verification_result}')
```

Our interaction consisted of only two polynomials (since $v = 2$):

$$\begin{aligned} s_1(X) &= 6X^2 + 4X + 9, & r_1 &= 4 \\ s_2(X) &= X^2 + 10X, & r_2 &= 2 \end{aligned}$$

The final verification result is True, meaning prover has successfully proved that $C = AB$ holds.