The Unreasonable Power of the Sum-Check Protocol

When designing an efficient interactive proof system, there is only one hammer you need to have in your toolbox: the *sum-check protocol* of Lund, Fortnow, Karloff, and Nisan. The goal of this blog post is to describe this hammer and give a sense of why it is so useful.

This entire post will be framed in the context of interactive proofs. This means that the goal is for a verifier V to offload an expensive computation to an untrusted prover P, while achieving work-saving for the verifier. We want the verifier to run in time linear in the input size, while keeping the proof short (logarithmic size) and the prover efficient.

Since this post is appearing on the ZKProof blog, you may wonder why there is nothing said about zero-knowledge. The answer is that one can combine the ideas described in this post with cryptographic commitments to get state of the art zk-SNARKs. But that will be the subject of a future post.

Roadmap for this post. After describing the sum-check protocol, I will demonstrate its remarkable power through several applications. First, I will give simple interactive proofs (IPs) for matrix multiplication and counting triangles in graphs. A cool thing about these IPs is that the prover is super-efficient: P runs the best-known algorithm to solve the problem, and then does a low-order amount of extra work to prove the answer is correct. I don't know of any other techniques that achieve this super-efficiency with logarithmic proof length.

Second, I will re-prove the following important result of Goldwasser, Kalai, and Rothblum (GKR): all problems solvable in logarithmic space have an IP with a linear-time verifier, polynomial time prover, and polylogarithmic proof length. As I'll explain, this result is a direct consequence of the matrix-multiplication IP—this is simpler than the original treatment by GKR, who derived the result via a sophisticated IP for arithmetic circuit evaluation.

1 The Two Technical Facts Needed In This Post

To understand why the IPs in this post are sound against cheating provers, all you need to know is that any two distinct univariate polynomials of degree at most d can only agree on at most d inputs. Equivalently:

Polynomial Equality Checking Lemma. If p and q are distinct univariate polynomials of degree at most d over a field $\mathbf{F}$, then $\Pr[p(r) = q(r)] \leq d/|\mathbf{F}|$,

where the probability is over a randomly chosen element $r \in \mathbf{F}$.

The other technical notion required in this post is multilinear extensions. Here, a multivariate polynomial is said to be multilinear if it has degree at most 1 in each variable.

Multilinear Extension Lemma. Let $f\colon \{0,1\}^n \to \mathbf{F}$. Then there is a unique multilinear polynomial $\tilde{f}$ over $\mathbf{F}$ such that $\tilde{f}(x) = f(x)$ for all $x \in \{0,1\}^n$. $\tilde{f}$ is called the *multilinear extension* (MLE) of f. Given as input a list of all 2^n evaluations of f, and an arbitrary point $r \in \mathbf{F}^n$, there is an algorithm that can evaluate $\tilde{f}(r)$ in $O(2^n)$ time.

2 The Sum-Check Protocol

Suppose we are given a v-variate polynomial g defined over a finite field $\mathbf{F}$. The purpose of the sum-check protocol is to compute the sum:

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

At first blush, summing up the evaluations of a polynomial over all Boolean inputs may seem like a contrived task. But to the contrary, later sections of this post will show that many natural problems can be directly cast as an instance of Equation (1).

What does the verifier gain by using the sum-check protocol? For presentation purposes, we assume in this section of the post that the verifier has oracle access to g, i.e., the verifier can evaluate $g(r_1, \ldots, r_v)$ for any desired vector $(r_1, \ldots, r_v) \in \mathbf{F}^v$ with a single query to an oracle. (In applications of the sum-check protocol, there will be no such oracle; rather, the polynomial g will be derived from the input in some fashion). Let us further assume that g has degree at most 2 in each variable, as this will be the case in all of the applications in this post.

The verifier could clearly compute H via Equation (1) on her own with 2^v calls to the oracle, but we are thinking of 2^v as an unacceptably large runtime for the verifier. It turns out that the verifier can execute her part of the sum-check protocol in $O(1)$ time[1] per round, and then at the very end of the protocol make only *a single call* to the oracle. The protocol requires v rounds, one for each variable of g, which means the verifier's total runtime is

$$O(v + [\text{the cost of one oracle query to } g]).$$

This is much better than the 2^v oracle queries requires to compute H unassisted.[2]

[1] Throughout, we assume any addition or multiplication operation in $\mathbf{F}$ takes $O(1)$ time.

[2] Again, in applications, there will be no oracle; g will be derived from the input in some fashion, and V will have to evaluate $g(r_1, \ldots, r_v)$ at a single point $(r_1, \ldots, r_v)$ on her own.

It also turns out that the prover in the sum-check protocol can be implemented with just $O(2^v)$ calls to the oracle. This is only a constant factor more than what is required simply to compute H without proving correctness.

Description of the Start of The Protocol. At the start of the first round, the prover sends a degree-2 polynomial $g_1(X_1)$, and claims that

$$g_1(X_1) = \sum_{(x_2,\ldots,x_v)\in\{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v).$$

If g_1 is as claimed, then $H = g_1(0) + g_1(1)$. Note that g_1 can be specified with 3 field elements, for example by sending the evaluation of g_1 at each point in the set $\{0, 1, 2\}$.

The Rest of the Round 1. Let

$$s_1(X_1) = \sum_{(x_2,\ldots,x_v)\in\{0,1\}^{v-1}} g(X_1, x_2, \ldots, x_v)$$

be the polynomial that the prover *claims* g_1 equals. The idea of the sum-check protocol is that the verifier will probabilistically check this equality of polynomials holds by picking a random field element $r_1 \in \mathbf{F}$, and confirming that

$$g_1(r_1) = s_1(r_1). \tag{2}$$

Clearly, if g_1 is as claimed, then Equation (2) holds for all $r_1 \in \mathbf{F}$. Meanwhile, if $g_1 \neq s_1$, then the Polynomial Equality Checking Lemma tells us that with probability at least $1 - 2/|\mathbf{F}|$ over the verifier's choice of r_1, Equation (2) fails to hold.

The remaining issue is the following: can the verifier efficiently compute both $g_1(r_1)$ and $s_1(r_1)$, in order to check that Equation (2) holds? While the verifier can evaluate $g_1(r_1)$ in $O(1)$ time given the description of g_1 sent by P, evaluating $s_1(r_1)$ is not an easy task, as s_1 is defined as a sum over 2^{v-1} evaluations of g. Fortunately, Equation (2) expresses s_1 as the sum of the evaluations of a $(v-1)$-variate polynomial over the Boolean hypercube, the polynomial being $g(r_1, X_2, \ldots, X_v)$ that is defined over the variables $X_2, \ldots, X_v$. This is exactly the type of expression that the sum-check protocol is designed to check. Hence, rather than evaluating $s_1(r_1)$ on her own, the verifier recursively applies the sum-check protocol to evaluate $s_1(r_1)$.

Recursive Description of Rounds $2, \ldots, v$. The protocol thus proceeds in this recursive manner, with one round per recursive call. This means that in round j, variable X_j gets *bound* to a random field element r_j chosen by the verifier. This process proceeds until round v, in which the prover is forced to send a polynomial $g_v(X_v)$ claimed to equal $s_v := g(r_1, \ldots, r_{v-1}, X_v)$. When the verifier goes to check that $g_v(r_v) = s_v(r_v)$, there is no need for further recursion: the verifier can evaluate $s_v(r_v) = g(r_1, \ldots, r_v)$ with a single oracle query to g.

The soundness error of the sum-check protocol for quadratic polynomials g is at most $2v/|\mathbf{F}|$. Throughout this post, we will think of $\mathbf{F}$ as large (say, of size $\text{poly}(2^v)$), which ensures that the soundness error is very small.

3 IP for Matrix Multiplication (MatMult)

Given $n \times n$ input matrices A, B, let us denote the product matrix $A \cdot B$ by C. We will need to interpret A, B, and C as *functions* mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\mathbf{F}$ in the natural way.

That is, we will think of $(i_1, \ldots, i_{\log n}) \in \{0,1\}^{\log n}$ as the binary representation of a number i between 1 and n (let us assume n is a power of 2 for simplicity), and define:

$$A(i_1, \ldots, i_{\log n}, j_1, \ldots, j_{\log n}) \text{ to be the } (i,j)\text{'th entry of } A.$$

Once we are viewing A, B, and C as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\mathbf{F}$, it makes sense to talk about their multilinear extensions $\tilde{A}$, $\tilde{B}$, and $\tilde{C}$.

An IP for Evaluating $\tilde{C}(r_1, r_2)$ for any $(r_1, r_2) \in \mathbf{F}^{\log n} \times \mathbf{F}^{\log n}$. It is cleanest to describe the protocol for MatMult as an IP for evaluating the multilinear extension $\tilde{C}$ at any given point $(r_1, r_2) \in \mathbf{F}^{\log n} \times \mathbf{F}^{\log n}$. As we will see, evaluating $\tilde{C}$ at a single point (r_1, r_2) turns out to be sufficient for other application problems such as triangle counting.[3]

Note that this IP is only interesting when $r_1, r_2 \in \mathbf{F}^{\log n} \setminus \{0,1\}^{\log n}$. This is because for inputs $i, j \in \{0,1\}^{\log n}$, $\tilde{C}(i,j)$ is just the (i,j)'th entry of the product matrix C, and $C(i,j) = \sum_k A(i,k) \cdot B(k,j)$ can be computed directly by the verifier in $O(n)$ time. But if $r_1, r_2 \notin \mathbf{F}^{\log n}$, then $\tilde{C}(r_1, r_2)$ in fact depends on *every* entry of C, and hence the verifier cannot compute $\tilde{C}(r_1, r_2)$ on its own without knowing the product matrix.

The Key Polynomial Identity. The protocol for computing $\tilde{C}(r_1, r_2)$ exploits the following explicit representation of the polynomial $\tilde{C}(x, y)$:

$$\tilde{C}(x,y) = \sum_{b \in \{0,1\}^{\log n}} \tilde{A}(x,b) \cdot \tilde{B}(b,y). \tag{3}$$

Proof of Equation (3). The left and right hand sides of Equation (3) are both multilinear polynomials in the coordinates of x and y. Since the MLE of C is unique, we need only check that the left and right hand sides of Equation (3) agree for all *Boolean* inputs. That is, we must check that for all Boolean vectors $i, j \in \{0,1\}^{\log n}$,

$$C(i,j) = \sum_{k \in \{0,1\}^{\log n}} A(i,k) \cdot B(k,j).$$

But this is immediate from the definition of matrix multiplication.

[3] If the verifier wishes to learn the entire product matrix C, the prover can send a matrix D claimed to equal C. The verifier can evaluate $\tilde{D}$ at a random input $(r_1, r_2) \in \mathbf{F}^{\log n} \times \mathbf{F}^{\log n}$ in $O(n^2)$ time using the Multilinear Extension Lemma, and can use the IP of this section to check that $\tilde{D}(r_1, r_2) = \tilde{C}(r_1, r_2)$. If so, the Schwartz-Zippel lemma (a multivariate generalization of the Polynomial Equality Checking Lemma) implies that it is safe for the verifier to believe that the claimed answer D in fact equals the true product matrix C.

With Equation 3 in hand, the interactive protocol is immediate: compute $\tilde{C}(r_1, r_2)$ by applying the sum-check protocol to the $(\log n)$-variate polynomial $g(z) := \tilde{A}(r_1, z) \cdot \tilde{B}(z, r_2)$.

Rounds and communication cost. Since g is a $(\log n)$-variate polynomial of degree 2 in each variable, the total communication is $O(\log n)$ field elements, spread over $\log n$ rounds.

V's runtime. At end of sum-check, V must evaluate

$$g(r_3) = \tilde{A}(r_1, r_3) \cdot \tilde{B}(r_3, r_2).$$

To perform this evaluation, it suffices for V to evaluate $\tilde{A}(r_1, r_3)$ and $\tilde{B}(r_3, r_2)$. Since V is given the matrices A and B as input, the Multilinear Extension Lemma implies that both evaluations can be performed by the verifier in $O(n^2)$ time, i.e., linear in the input size.

P's runtime. Once P computes the product matrix C, P can compute all of its prescribed messages in the IP in $O(n^2)$ additional time (it is not trivial to achieve this; see Section 8.2 of this paper for details). This means that P can run the best-known algorithm to compute the product matrix C (which takes much more than n^2 time), and then do a low-order amount of extra work to prove the answer is correct.

4 Counting Triangles Protocol

To define the problem, let G be an undirected graph on n vertices with edge set E. Let $A \in \{0,1\}^{n \times n}$ be the adjacency matrix of G, i.e., $A_{i,j} = 1$ if and only if $(i,j) \in E$. In the counting triangles problem, the input is the adjacency matrix A, and the goal is to determine the number of vertex triples (i,j,k) are all connected to each other, i.e., (i,j), (j,k), and (i,k) are all edges in E. Equivalently, the goal is to compute the following quantity:

$$\Delta := \frac{1}{6} \sum_{i,j,k} A_{i,j} \cdot A_{j,k} \cdot A_{i,k} = \frac{1}{6} \sum_{i,j} (A^2)_{i,j} \cdot A_{i,j}. \tag{4}$$

Here, the factor $1/6$ comes in because the sum over *unordered* node triples (i,j,k) counts each triangle 6 times, once for each permutation of i, j, and k.

For simplicity, we will focus on computing $\Delta' := 6\Delta$, since Δ can clearly be derived from Δ' with a single division operation.

Let $\mathbf{F}$ be a finite field of size $p \geq n^3$, where p is a prime, and let us view all entries of A as elements of $\mathbf{F}$. Here, we are choosing p large enough so that Δ' is guaranteed to be in $\{0, 1, \ldots, p\}$. This ensures that, if we associate elements of $\mathbf{F}$ with integers in $\{0, 1, \ldots, p\}$ in the natural way, then $\Delta' = \sum_{i,j} (A^2)_{i,j} \cdot A_{i,j}$ even when all additions and multiplications are done in $\mathbf{F}$ rather than over the

integers. Choosing a large field to work over has the added benefit of ensuring good soundness error, as the soundness error of the sum-check protocol decreases linearly with field size.

Define the functions $f(x,y), g(x,y)$: $\{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \{0,1\}$ that interprets x and y as the binary representations of some integers i and j between 1 and n, and outputs $A_{i,j}$ and $(A^2)_{i,j}$ respectively. Let $\tilde{f}$ and $\tilde{g}$ denote the multilinear extensions of f and g over $\mathbf{F}$.

Then Δ' equals

$$\sum_{x,y \in \{0,1\}^{\log n}} \tilde{g}(x,y) \cdot \tilde{f}(x,y).$$

This quantity can be computed by applying the sum-check protocol to the quadratic polynomial $\tilde{g} \cdot \tilde{f}$. At the end of this protocol, the verifier needs to evaluate $\tilde{g}(r_1, r_2) \cdot \tilde{f}(r_1, r_2)$ for a randomly chosen input $(r_1, r_2) \in \mathbf{F}^{\log n} \times \mathbf{F}^{\log n}$. The verifier can evaluate $\tilde{f}(r_1, r_2)$ unaided in $O(n^2)$ time using the Multilinear Extension Lemma. While the verifier cannot evaluate $\tilde{g}(r_1, r_2)$ without computing the matrix A^2 (which is as hard as solving the counting triangles problem on her own), evaluating $\tilde{g}(r_1, r_2)$ is exactly the problem that the MatMult IP of Section 3 was designed to solve (as $A^2 = A \cdot A$), so we simply invoke that protocol to compute $\tilde{g}(r_1, r_2)$.

Rounds and communication cost. The total communication is $O(\log n)$ field elements, spread over $O(\log n)$ rounds.

V's runtime. V runs in $O(n^2)$ time, which is linear in the input size.

P's runtime. If P is given the squared adjacency matrix A^2, P can compute all the prescribed messages in the sum-check protocols in $O(n^2)$ time. The fastest known algorithms for counting triangles involve squaring the adjacency matrix A, which means that P can run the fastest known algorithm, and then do only $O(n^2)$ extra work to prove the answer is correct.

Comparison to Freivalds' Algorithm. In the 1970s, Freivalds gave a non-interactive protocol for verifying matrix products: the prover in Freivalds' protocol just sends the product matrix C, and the verifier picks a random vector $x \in \mathbf{F}^n$ and checks that $Cx = A \cdot (Bx)$. One can apply Freivalds' protocol in place of the MatMult IP above to count triangles, but the communication cost will be $O(n^2)$, instead of $O(\log n)$. This is the power of interaction: it allows the verifier to force the prover to correctly materialize intermediate values in a computation, without the need for the prover to explicitly send those values to the verifier.

5 Final Applications

5.1 An IP for Matrix Powers

Let A be an $n \times n$ matrix, and suppose a verifier wants to evaluate a single entry of the powered matrix A^k for a large integer k (for concreteness, let's say V is interested in learning entry $(A^k)_{n,n}$, and k and n are powers of 2). As we now explain, the MatMult IP of Section 3 gives a way to do this, with $O(\log(k) \cdot \log(n))$ rounds and communication, and a verifier that runs in $O(n^2 + \log(k)\log(n))$ time.

Clearly we can express the matrix A^k as a product of smaller powers of A:

$$A^k = A^{k/2} \cdot A^{k/2}. \tag{5}$$

Hence, letting g_ℓ denote the multilinear extension of the matrix A^ℓ, we can try to exploit Equation (5) by applying the MatMult IP to compute $(A^k)_{n,n} = g_k(\mathbf{1}, \mathbf{1})$.

But at the end of this MatMult IP, the verifier needs to evaluate $g_{k/2}$ at two points. The verifier can't do this since she doesn't know $A^{k/2}$.

Reducing two points to one. There are known interactive proof techniques that enable a verifier to reduce evaluating a polynomial $g_{k/2}$ at the two points to evaluating $g_{k/2}$ at a single point. We omit these details for brevity.

Recursion to the Rescue. After reducing two points to one, the verifier is left with the task of evaluating $g_{k/2}$ at a single input, say $(r_1, r_2) \in \mathbf{F}^{\log n} \times \mathbf{F}^{\log n}$. Since $g_{k/2}$ is the MLE of the matrix $A^{k/2}$, which can be decomposed as $A^{k/4} \cdot A^{k/4}$, the verifier can recursively apply the MatMult protocol to compute $g_{k/2}(r_1, r_2)$. This runs into the same issues as before, namely that to run the MatMult protocol, the verifier needs to evaluate $g_{k/4}$ at two points, which can in turn be reduced to the task of evaluating $g_{k/4}$ at a single point. This can again be handled recursively as above. After $\log k$ layers of recursion, there is no need to recurse further since the verifier can evaluate $g_1 = \tilde{A}$ at any desired input in $O(n^2)$ time using the Multilinear Extension Lemma.

5.2 An IP for Logarithmic Space Computations

Let M be a Turing Machine that, when run on an m-bit input, uses at most s bits of space. Let $A(x)$ be the adjacency matrix of its *configuration graph* when M is run on input $x \in \{0,1\}^m$. Here, the configuration graph has as its vertex set all of the possible states and memory configurations of the machine M, with a directed edge from vertex i to vertex j if running M for one step from configuration i on input x causes M to move to configuration j. Since M uses s bits of space, there are $O(2^s)$ many vertices of the configuration graph. This means that $A(x)$ is an $N \times N$ matrix for some $N = O(2^s)$. Note that if M never enters an infinite loop (i.e., never enters the same configuration twice), then M must trivially run in time at most N.

We can assume without loss of generality that M has a unique starting configuration and a unique accepting configuration; say for concreteness that these configurations correspond to vertices of the configuration graph with labels 1 and N. Then to determine whether M accepts input x, it is enough to determine if there is a length-N path from vertex 0 to vertex N in the configuration graph of M. This is equivalent to determining the $(1, N)$'th entry of the matrix $(A(x))^N$.

This quantity can be computed with the matrix power protocol of the previous section, which uses $O(s \cdot \log N)$ rounds and communication. At the end of the protocol, the verifier does need to evaluate the MLE of the matrix $A(x)$ at a randomly chosen input. This may seem like it should take up to $O(N^2)$ time, since A is a $N \times N$ matrix. However, the configuration matrix of any Turing Machine is highly structured, owing to the fact that at any time step, the machine only reads or writes to $O(1)$ memory cells, and only moves its read and write heads at most one cell to the left or right. This turns out to imply that the verifier can evaluate the MLE of A in $O(s \cdot m)$ time.

In total, the costs of the IP are as follows. The rounds and number of field elements communicated is $O(s \log N)$, the verifier's runtime is $O(s \log N + m \cdot s)$ and the prover's runtime is $\text{poly}(N)$. If $s = O(\log m)$, then these three costs are respectively $O(\log^2 m)$, $O(m \log m)$, and $\text{poly}(m)$. That is, the communication cost is polylogarithmic in the input size, the verifier's runtime is quasilinear, and the prover's runtime is polynomial.

By the way, if $s = \text{poly}(m)$, then the verifier's runtime in this IP is $\text{poly}(m)$, recovering the famous result of LFKN and Shamir that **IP** = **PSPACE**.

Parting Thoughts. One disappointing feature of this IP is that, if the runtime of M is significantly less than $N \geq 2^s$, the prover will still take time at least N, because the prover has to explicitly generate powers of the configuration graph's adjacency matrix. This is particularly problematic if the space bound s is superlogarithmic in the input size m, since then 2^s is not even a polynomial in m. Effectively, the IP we just presented forces the prover to explore all possible configurations of M, even though when running M on input x, the machine will only enter a tiny fraction of such configurations. A breakthrough complexity-theory result of Reingold, Rothblum, and Rothblum gave a very different IP that avoids this inefficiency for P (remarkably, their IP also requires only constantly many rounds of interaction).