Self-Testing Without the Generator Bottleneck *

FUNDA ERGÜN † S. RAVI KUMAR ‡ D. SIVAKUMAR §

Abstract

Suppose P is a program designed to compute a function f defined on a group G. The task of *self-testing* P, that is, testing if P computes f correctly on most inputs, usually involves testing explicitly if P computes f correctly on *every* generator of G. In the case of *multivariate* functions, the number of generators, and hence the number of such tests, becomes prohibitively large. We refer to this problem as the *generator bottleneck*. We develop a technique that can be used to overcome the generator bottleneck for functions that have a certain nice structure, specifically if the relationship between the values of the function on the set of generators is easily checkable. Using our technique, we build the first efficient self-testers for many linear, multilinear, and some non-linear functions. This includes the FFT, and various polynomial functions. All of the self-testers we present make only $O(1)$ calls to the program that is being tested. As a consequence of our techniques, we also obtain efficient *program result-checkers* for all these problems.

*This paper unifies the preliminary versions which appeared in the *27th Annual Symposium on Theory of Computing* [Erg95] and in the *15th Annual Foundations of Software Technology and Theoretical Computer Science* [KS95].

†Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: `ergun@cs.cornell.edu`. This work is partially supported by ONR Young Investigator Award N00014-93-1-0590, the Alfred P. Sloan Research Award, and NSF grant DMI-91157199.

‡Department of Computer Science, Cornell University, Ithaca, NY 14853. Email: `ravi@cs.cornell.edu`. This work is partially supported by ONR Young Investigator Award N00014-93-1-0590, the Alfred P. Sloan Research Award, and NSF grant DMI-91157199.

§Department of Computer Science, University of Houston, Houston, TX 77204. Email: `siva@cs.uh.edu`. Most of this work performed while the author was at SUNY/Buffalo, supported in part by K. Regan's NSF grant CCR-9409104.

1 Introduction

The notions of program result-checking, self-testing, and self-correcting as introduced in [BK89, Lip91, BLR93] are powerful tools for attacking the problem of program correctness. These methods offer both realistic and efficient tools for software verification. Various useful mathematical functions have been shown to have self-testers and self-correctors; some examples can be found in [BLR93, BF90, Lip91, CL90, GLR$^+$91, RS92, ABCG93, RS96, Rub94, BW94a]. The theoretical developments in this area are at the heart of the recent breakthrough results on probabilistically checkable proofs and the subsequent results that show non-approximability of hard combinatorial problems.

Suppose we are given a program P designed to compute a function f. Informally, a *self-tester* for f distinguishes the case where P computes f correctly always from the case where P errs frequently. A *result-checker* for a function f takes as input a program P and an input q to P, and outputs "PASS" when P correctly computes f always and outputs "FAIL" if $P(q) \neq f(q)$. Given a program P that computes f correctly on most inputs, a *self-corrector* for f is a program P_{sc} that uses P as an oracle and computes f correctly on every input with high probability.

1.1 Definitions and Basics

Before we discuss our results, we present the basic definitions of testers, checkers, etc., and state some desirable properties of these programs. Let f be a function on a domain $\mathcal{D}$ and let P be a program that purports to compute f. The testers, correctors, and checkers we define are probabilistic programs that take P as an oracle, and in addition, take one or more of the following parameters as input: an *accuracy parameter* ϵ that specifies the conditions that P is expected to meet, and a *confidence parameter* ρ that is an upper bound on the probability that the tester/corrector/checker fails to do its job. The following definitions formalize the notions of self-tester [BLR93], self-corrector [BLR93, Lip91], and result-checker [BK89].

Definition 1 (Self-Tester) *An* ϵ-self-tester *for f is a probabilistic oracle program T that, given $\rho > 0$, satisfies the following conditions:*

- $\Pr_{x \in \mathcal{D}}[P(x) = f(x)] = 1 \Rightarrow \Pr[T^P$ *outputs "PASS"* $] = 1$.
- $\Pr_{x \in \mathcal{D}}[P(x) \neq f(x)] \geq \epsilon \Rightarrow \Pr[T^P$ *outputs "FAIL"* $] \geq 1 - \rho$.

Definition 2 (Self-Corrector) *An* ϵ-self-corrector *for f is a probabilistic oracle program P_{sc} that, given any input y, and $\rho > 0$, satisfies the following condition:*

- $\Pr_{x \in \mathcal{D}}[P(x) = f(x)] \geq 1 - \epsilon \Rightarrow \Pr[P_{\mathrm{sc}}(y) = f(y)] \geq 1 - \rho$.

Definition 3 (Result-Checker) *A* checker *(or* result-checker*) for f is a probabilistic oracle program C that, given an input y and $\rho > 0$, satisfies the following conditions:*

- $\Pr_{x \in \mathcal{D}}[P(x) = f(x)] = 1 \Rightarrow C^P(y)$ *outputs* "PASS".
- $P(y) \neq f(y) \Rightarrow \Pr[C^P(y)$ *outputs* "FAIL"$] \geq 1 - \rho$.

We now list three important properties that are required of self-testers, self-correctors, and result-checkers. For definiteness, we state these for the case of self-testers. First, the self-tester T should be computationally *different* from and more *efficient* than any program that computes f [BK89]. This restriction ensures that T does not implement the obvious algorithm to compute f (and hence could harbor the same set of bugs, or be computationally inefficient). Furthermore, this ensures that the running time of T is *asymptotically better than* the running time of the best *known* algorithm for f. The second important property required of T is that it should not require the knowledge of too many correct values of f. In particular, this rules out the possibility that T merely keeps a large table of the correct values of f for all inputs. The third important property required of a self-tester is *efficiency*: an efficient self-tester should only make $O(1/\epsilon, \lg(1/\rho))$ calls to P. For constant ϵ and ρ, an efficient self-tester makes only $O(1)$ calls to the program. (In the rest of the paper, we often write $O(1)$ as a shorthand for $O(1/\epsilon, \lg(1/\rho))$, particularly when discussing the dependence on other parameters of interest.)

The following well-known proposition summarizes some relationships between the notions of self-testers, self-correctors, and result-checkers. For the reader's convenience, we sketch the idea of the proof of this proposition, suppressing the details of the accuracy and confidence parameters.

Proposition 1 ([BLR93]) *(a) If f has a self-tester and a self-corrector that make $O(1)$ calls to the program, then f has a result-checker that makes $O(1)$ calls to the program. (b) If f has a result-checker, then it has a self-tester.*

Proof. (Sketch) For part (a), suppose that f has a self-tester and a self-corrector. Given an input y and oracle access to a program P, first self-test P to ensure that it doesn't err too often. If the self-tester finds P to be too erroneous, output "FAIL." Otherwise, compute $f(y)$ by using the self-corrector for $P_{\rm sc}$ and the program P, and output "PASS" iff $P(y) = P_{\rm sc}(y)$.

Clearly a perfect program always passes. Suppose $P(y) \neq f(y)$. Then one of the following two cases must occur. The program is too erroneous, in which case the self-tester, and hence the checker, outputs "FAIL." The program is not too erroneous, in which case the self-corrector computes $f(y)$ correctly with high probability, so the checker detects that $P(y) \neq P_{\rm sc}(y)$ and outputs "FAIL."

For part (b), suppose that f has a result-checker. By using the result-checker to test if $P(x) = f(x)$ for many randomly chosen inputs x, the fraction of inputs x for which $P(x) \neq f(x)$ can be estimated. Output "PASS" iff this fraction is less than ϵ. ∎

A useful tool in constructing self-correctors is the notion of *random self-reducibility*. The fine details of this notion are beyond the scope of this paper, and we refer the reader to the papers [BF90, Lip91] (see also the survey paper [Fei93]). Informally, a function f is randomly self-reducible if evaluation of f on an input can be reduced efficiently to the evaluation of f on one or more random inputs. For a quick example, note that linear functions are randomly self-reducible: to compute $f(x)$, it suffices to pick a random r compute $f(x+r)$ and $f(r)$, and finally obtain $f(x) = f(x+r) - f(r)$. All functions that we consider in this paper are efficiently randomly self-reducible; therefore, whenever required, we will always assume that efficient self-correction is possible.

1.2 Building Self-Testers using Properties

The process of self-testing whether a program P computes a function f correctly on most inputs is usually a two-step strategy. First perform some tests to verify that P agrees on most inputs with a function g that belongs to a certain *class* $\mathcal{F}$ of functions that contains f. Then perform some additional tests to verify that the function g is, in fact, the *intended* function f.

The standard way to test whether P agrees with some function in a class $\mathcal{F}$ of functions is based on the notion of a *robust property*. Informally, property $\mathcal{I}$ is said to be a robust characterization of a function family $\mathcal{F}$ if the following two conditions hold: (1) every $f \in \mathcal{F}$ satisfies $\mathcal{I}$, and (2) if P is a function (program) that satisfies $\mathcal{I}$ for most inputs, then P must agree with some $g \in \mathcal{F}$ on most inputs. For example, Blum, Luby, and Rubinfeld [BLR93] establish that the property of linearity $f(x+y) = f(x) + f(y)$ serves as a robust property for the class of all linear functions, and use this to build self-testers for linear functions. This generic technique was first formalized in [RS96].

Definition 4 (Robust Property) *A* property *is a predicate* $\mathcal{I}^f(\vec{x} = x_1, \ldots, x_k)$. *A property* $\mathcal{I}^f(\vec{x})$ *is* (ϵ, δ)-robust *for a class of functions* $\mathcal{F}$ *over a domain* $\mathcal{D}$, *if it satisfies the following conditions:*

- $(\forall f) \left[f \in \mathcal{F} \iff (\forall \vec{x} \in \mathcal{D}^k)[\mathcal{I}^f(\vec{x}) = \text{“}TRUE\text{”}] \right]$.
- *If a function (program) P satisfies* $\Pr_{\vec{x} \in \mathcal{D}^k}[\mathcal{I}^P(\vec{x}) = \text{“}TRUE\text{”}] \geq 1 - \epsilon$ *then there is a function g such that*
    - $(\forall \vec{x} \in \mathcal{D}^k)[\mathcal{I}^g(\vec{x}) = \text{“}TRUE\text{”}]$,
    - $\Pr_{x \in \mathcal{D}}[g(x) = P(x)] \geq 1 - \delta$,

  that is, $(\exists g \in \mathcal{F})$ such that P agrees with g on all but δ fraction of inputs.

We now outline the process of building self-testers using robust properties (cf. [BLR93]). Let $\mathcal{D}$ be a (finite) group with generators $e_1, \ldots, e_n$, and let $\mathcal{F}$ denote some class of functions from $\mathcal{D}$ into some range $\mathcal{R}$. Further assume that the functions in $\mathcal{F}$ possess the property of random self-reducibility, and can hence be self-corrected efficiently. Suppose P is a program that purports to compute a specific function $f \in \mathcal{F}$. Let $\mathcal{I}^f(\vec{x})$ be a robust property that characterizes $\mathcal{F}$.

As mentioned earlier, the process of building self-testers is a two-step process. In the first step, we will ensure that that the program P agrees with some function $g \in \mathcal{F}$ on most inputs. To do this, we will use the fact that $\mathcal{I}^f$ is a robust property that characterizes $\mathcal{F}$. Specifically, the self-tester will estimate the fraction of k-tuples $\vec{x} \in \mathcal{D}^k$ for which $\mathcal{I}^f(\vec{x})$ holds. If this fraction is at least $1 - \epsilon$, then by the robustness of $\mathcal{I}^f$, it follows that there is some $g \in \mathcal{F}$ that agrees with P on all but δ fraction of $\mathcal{D}$. The required estimation can be carried out by random sampling of $\vec{x}$ and testing the property $\mathcal{I}^f$.

The next step is to verify that the function g is the same as the function f that P purports to compute. This is achieved by testing that $g(e_i) = f(e_i)$ for every *generator* of the group $\mathcal{D}$. If this is true, then by an easy induction it would follow that $g \equiv f$. An important point to be mentioned here is that the self-tester has access only to P and not to g; the function g is only guaranteed to exist. Nevertheless, the required values of g may be obtained by using a self-corrected version P_{sc} of P. Another point worth mentioning is that to carry out this step, the self-tester needs to know the values of f on every generator of $\mathcal{D}$.

1.3 The Generator Bottleneck

An immediate application of the basic method outlined above to functions whose domains are vector spaces of large dimension suffers from a major efficiency drawback. For example, if the inputs to the function f are n-dimensional vectors (or $n \times n$ matrices), then the number of generators of the domain is n (resp. n^2). The straightforward approach of exhaustively testing if P_{sc} agrees with f on each generator by making n (resp. n^2) calls to P; furthermore, the self-tester built this approach requires the knowledge of the correct value of f on n (resp. n^2) generators. When n is large, this makes the overhead in the self-testing process too high. This issue is called the *generator bottleneck* problem.

In this paper, we address the generator bottleneck problem, and solve it for a fairly large class of functions that satisfy some nice structural properties. The self-testers that we build are not only useful in themselves, but are also useful in building efficient result-checkers, which are important for practical applications.

1.4 Our Results

We present a fairly general method of overcoming the generator bottleneck and testing multivariate functions by making only $O(1)$ calls to the program being tested.

First we investigate the problem of *multivariate linear functions* (i.e., the functions f satisfying $f(\vec{x}) + f(\vec{y}) = f(\vec{x} + \vec{y})$). We show a general technique that can be applied in a natural vector space setting. The main idea is to obtain an easy and uniform way of "generating" all generators from a single generator. Using this idea, we give a simple and powerful condition for a linear function f to be efficiently self-testable on a large vector space. We then apply this scheme to obtain very efficient self-testers for many functions. This includes polynomial differentiation (of arbitrary order), polynomial integration, polynomial "mod" function, etc. We also obtain the first efficient self-tester for Fourier transforms.

We then extend this method to the case of *multilinear functions* (i.e., functions f that are linear in each variable when the other variables are fixed). We build an efficient tester for polynomial multiplication as a consequence. Another application we give is for large finite fields: we show that multilinear functions over finite field extensions of dimension n can be efficiently self-tested with $O(1)$ calls, independent of the dimension n. We also provide a new efficient self-tester for matrix multiplication.

We next extend the result to some *nonlinear functions.* We give self-testers for exponentiation functions that avoid the generator bottleneck. For example, consider the function that computes the square of a polynomial over a finite field: $f(q) = q^2$. Here we do not have the linearity property that is crucial in the proof for the linear functions. Instead, we use the fact that the Lagrange interpolation identity (cf. Fact 14) for polynomials gives a robust characterization. We exhibit a self-tester for the function $f(q) = q^d$ that makes $O(d)$ calls to the program being tested. Extending the technique when f is a constant degree exponentiation to the case when f is a constant degree polynomial (eg., $f(q) = q^d + q + 1$, where q is a polynomial over a finite field) is much harder. First we show a reduction from multiplication to the computation of low-degree polynomials. Using this reduction and the notion of a result-checker, we construct a self-tester for degree d polynomials over finite field extensions of dimension n that make $O(2^d)$ calls to the program being tested.

1.5 Related Work

One method that has been used to get around the generator bottleneck has been to exploit the property of *downward self-reducibility* [BLR93]. The self-testers that use this property, however, have to make $\Omega(\log n)$ calls to the program depending on the way the problem decomposes into smaller problems. For instance, a tester for the permanent function of $n \times n$ matrices makes $O(n)$ calls to the program, whereas a tester for polynomial multiplication that uses similar principles makes $O(\log n)$ calls. In [BLR93] a bootstrap tester for polynomial multiplication that makes $O(\log n)$ calls to the program being tested is given. It is already known that matrix multiplication can be tested (without any calls to the program) using a result-checker due to Freivalds [Fre79]. The idea of Freivalds' matrix multiplication checker can also be adapted to build testers for polynomial multiplication that make no calls to the program being tested. This approach, however, requires the underlying field to be large (have at least $(2+\gamma)n$ elements, where n is the degree of the polynomials being multiplied, and γ is a positive constant). Moreover, this scheme requires the tester to perform polynomial evaluations, whereas ours does not. For Fourier transforms, a different result-checker that uses preprocessing has been given independently in [BW94a].

A Useful Fact. The following fact, a variant of the well-known Chernoff-Hoeffding bounds, is often very useful in obtaining error-bounds in sampling 0/1 random variables [KLM89]:

Fact 2 *Let $Y_1, Y_2, \ldots$ be independently and identically distributed 0/1 random variables with means μ. Let $\theta \leq 2$. If $N \geq (1/\mu)(4\ln(2/\rho)/\theta^2)$, then $\Pr[(1-\theta)\mu \leq \widetilde{Y} \leq (1+\theta)\mu] \geq 1-\rho$, where $\widetilde{Y} = \sum_{i=1}^{N} Y_i/N$.*

Organization of the Paper. Section 2 discusses the scheme for linear functions over vector spaces; Section 3 extends the scheme for multilinear functions; Section 4 outlines the approach for non-linear functions.

2 Linear Functions over Vector Spaces

In this section, we address the problem of self-testing linear functions on a vector space without the generator bottleneck. We demonstrate a general technique to self-test without the generator bottleneck and provide several interesting applications of our technique.

Definitions. Let V be a vector space of finite dimension n over a field $\mathbb{K}$, and let f be a function from V into a ring R. We are interested in building a self-tester for the case where $f(\cdot)$ is a *linear* function, that is, $f(c\alpha+\beta) = cf(\alpha) + f(\beta)$ for all $\alpha, \beta \in V$ and $c \in \mathbb{K}$. For $1 \leq i \leq n$, let e_i denote the unit vector that has a 1 in the i-th position and 0's in the other positions. The vectors $e_1, e_2, \ldots, e_n$ form a collection of basis vectors that span V. Viewed as an Abelian group under vector addition, V is generated by $e_1, \ldots, e_n$. We assume that the field $\mathbb{K}$ is finite, since it is not clear how to choose a random element from an infinite field.

The property of linearity $\mathcal{I}^f(\alpha, \beta) \equiv [f(\alpha+\beta) = f(\alpha) + f(\beta)]$ was shown to be robust in [BLR93]. Using this and the generic construction of self-testers from robust properties, one obtains the following self-tester for the function f:

```
Property Test:
  Repeat O((1/ε) log (1/ρ)) times
    Pick α, β ∈_R V
    Verify P(α + β) = P(α) + P(β)
    Reject if the test fails

Generator Tests:
  For i = 1 to n
    Verify P_sc(e_i) = f(e_i)
```

If P passes the `Property Test` then we are guaranteed the existence of a linear function g that is close to P. There are, however, two problems with the `Generator Tests`: one is that the self-tester is inefficient—if the inputs are vectors of size n, the self-tester makes $O(n)$ calls to the program, which is not desirable. Secondly, the self-tester needs to know the correct value of f on n different points, which is also undesirable. Our primary interest is to avoid this *generator bottleneck* and solve both of the problems mentioned. The key idea is to find an easy and uniform way that "converts" one generator into the next generator. We illustrate this idea through the following example.

Example. Let $\mathcal{P}_n$ denote the additive group of all degree n polynomials over a field $\mathbb{K}$. The elements $1, x, x^2, \ldots, x^n$ generate $\mathcal{P}_n$, and multiplying any generator x^k by x gives the next generator x^{k+1}. For a polynomial $q \in \mathcal{P}_n$ and a scalar $c \in \mathbb{K}$, let $E_c(q)$ denote the function that evaluates $q(c)$. Clearly E_c is linear and satisfies the simple relation $E_c(xq) = cE_c(q)$. Suppose P is a program that purports to compute E_c, and assume that P has passed `Property Test` given above. Then we know by robustness of linearity that there is a linear function g that agrees with P on most inputs. Note that g can be computed correctly with high probability via the self-corrector (which are easy to construct for linear functions [BLR93]). Now, rather than verify that $g(x^k) = E_c(x^k)$ for all generators x^k of $\mathcal{P}_n$, we may instead verify that g satisfies the property $g(xq) = cg(q)$ everywhere. By an easy induction, this implies that g agrees with E_c at all the generators. By linearity of g, it follows that g agrees with E_c on all inputs.

We are now faced with the task of verifying $g(xq) = cg(q)$ for all $q \in \mathcal{P}_n$, which is too expensive to be tried explicitly and exhaustively. Instead, we prove that it suffices to check with $O(1)$ tests that $g(xq) = cg(q)$ *almost* everywhere that we look at. That is, pick many random $q \in \mathcal{P}_n$, ask the program P_{sc} to compute the values of $g(q)$ and $g(xq)$ and cross-check that $g(xq) = cg(q)$ holds. In other words, we prove that the property $\mathcal{J}^g(q) \equiv [g(xq) = cg(q)]$ is robust (in a restricted sense), under the assumption that g is linear. (In its most general interpretation, robustness guarantees the existence of h that satisfies $h(xq) = ch(q)$ for all $q \in \mathcal{P}_n$ and that agrees with g on a large fraction of inputs. We actually show that $h \equiv g$, hence the "restricted sense.") Notice that the number of points on which the self-tester needs to know the value of f is just one, in contrast to n as in the original approach of [BLR93].

Generalization via the Basis Rotation Function θ. We note that this idea has a natural generalization to vector spaces. Let θ denote the *basis rotation function*, i.e., the linear operator on a vector space V that "rotates" the coordinate axes that span V. θ, which can be viewed as a matrix, defines a one-to-one correspondence from the set of basis vectors to itself: for every i, $\theta(e_i) = e_{(i+1) \bmod n}$. The computational payoff is achieved when there is a simple relation between $f(\alpha)$ and $f(\theta(\alpha))$ for all vectors $\alpha \in V$. More specifically, we show that the generator bottleneck can be avoided if there is an easily computable function $h_{\theta,f}$ such that

$$f(\theta(\alpha)) = h_{\theta,f}(\alpha, f(\alpha))$$

for all $\alpha \in V$. (For instance, for polynomial evaluation E_c, $f(\theta(q)) = c \cdot f(q), \forall q \in \mathcal{P}_n$.) From here on, when obvious, we drop the suffix f and simply denote $h_{\theta,f}$ as h_θ. If the function f is linear, the linearity of θ implies that h_θ is linear in its second argument in the following sense: $h_\theta(\alpha+\beta, f(\alpha+\beta)) = h_\theta(\alpha, f(\alpha)) + h_\theta(\beta, f(\beta))$. What is more important is that h_θ be easy to compute, given just α and $f(\alpha)$. Using this scheme, we show that many natural functions f have a suitable candidate for h_θ. The `Generator Tests` of [BLR93] can now be replaced by:

```
Basis Test:
  Verify P_sc(e_1) = f(e_1)

Inductive Test:
  Repeat O((1/ε) log (1/ρ)) times
    Pick α ∈_R V
    Verify P_sc(θ(α)) = h_θ(α, P_sc(α))
    Reject if the test fails
```

The following theorem proves that this replacement is valid:

Theorem 3 *Suppose f is a linear function from the vector space V into a ring R, and suppose P is a program for f.*

(a) Let $\epsilon < 1/2$, and suppose P satisfies the following condition:

> *(1)* $\mathrm{Pr}_{\alpha,\beta \in V}[P(\alpha+\beta) \neq P(\alpha) + P(\beta)] \leq \epsilon$.

Then the function g defined by $g(\alpha) = \mathrm{majority}_{\beta \in V}\{P(\alpha+\beta) - P(\beta)\}$ is a linear function on V, and g agrees with P on at least $1 - 2\epsilon$ fraction of the inputs.

(b) Furthermore, suppose $h_\theta(\alpha, f(\alpha)) = f(\theta(\alpha))$ and g satisfies the following conditions:

> *(2)* $g(e_1) = f(e_1)$.
> *(3)* $\mathrm{Pr}_{\alpha \in V}[g(\theta(\alpha)) \neq h_\theta(\alpha, g(\alpha))] \leq \epsilon$, *where α is such that $\theta(\alpha)$ is defined.*

Then $g(\alpha) = f(\alpha)$ for all $\alpha \in V$.

Remarks. The above theorem merely lists a set of properties. The fact that this set yields a self-tester is presented in Theorem 4. Note that hypotheses (1), (2), and (3) above are *conditions* on P and g, not *tests* performed by a self-tester.

Proof. The proof that the function g is linear and P_{sc} computes g (with high probability) is due to [BLR93]. For the rest of this proof, we will assume that g is linear and that it satisfies conditions (2) and (3) above.

We first argue that it suffices to prove that if the conditions hold, then for every $\alpha \in V$, $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$. By condition (2), g agrees with f on the first basis vector. For $i > 1$, the basis vector e_i can be obtained by $\theta(e_{i-1})$. If g satisfies $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$ everywhere, it would follow that g computes f correctly on all the basis vectors. Finally, since g is linear, it computes f correctly on all of V, since the vectors in V are just linear combinations of the basis vectors.

Now we show that condition (3) implies that $\forall \alpha \in V$, $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$. Fix an arbitrary element $\alpha \in V$. We will show that the probability over a random $\beta \in V$ that $g(\theta(\alpha)) = h_\theta(\alpha, g(\alpha))$ is positive. Since the equality is independent of β and holds with non-zero probability, it must be true with probability 1. Now

$$\begin{aligned}\Pr_{\beta \in V}[g(\theta(\alpha)) &= g(\theta(\beta + \alpha - \beta)) \\ &= g(\theta(\beta) + \theta(\alpha - \beta))\end{aligned}$$

$$
\begin{aligned}
&= g(\theta(\beta)) + g(\theta(\alpha - \beta)) \\
&= h_\theta(\beta, g(\beta)) + h_\theta(\alpha - \beta, g(\alpha - \beta)) \\
&= h_\theta(\beta + \alpha - \beta, g(\beta) + g(\alpha - \beta)) \\
&= h_\theta(\alpha, g(\alpha))] \geq 1 - 2\epsilon > 0.
\end{aligned}
$$

The first equality in the above is just rewriting. The second equality follows from the linearity of θ. The third equality follows from the fact that g is linear. If the random variable β is distributed uniformly in V, the random variables β and $\alpha - \beta$ are distributed identically and uniformly in V. Therefore, by the assumption that g satisfies condition (3), the fourth equality fails with probability at most 2ϵ. The fifth equality uses the fact that h_θ is linear, and the last equality uses the fact that g is linear. ■

The foregoing theorem shows that if P (and g) satisfy certain conditions, then g, which can be computed using P, is identically equal to the function f. The self-tester comprises the following tests: `Linearity Test`, `Basis Test`, and `Inductive Test`.

Theorem 4 *For any $\rho < 1$ and $\epsilon < 1/2$, the above three tests comprise a 2ϵ-self-tester for f. That is, if a program P computes f correctly on all inputs, then the self-tester outputs "PASS" with probability 1, and if P computes f incorrectly on more than 2ϵ fraction of the inputs, then the self-tester outputs "FAIL" with probability at least $1 - \rho$.*

Proof. In performing the three tests, the self-tester is essentially estimating the probabilities listed in conditions (1), (2), and (3) of the hypothesis of Theorem 3. Note that condition (2) does not involve any probability; rather, the self-tester uses P_{sc} to compute $g(e_1)$. By choosing $O((1/\epsilon)\log(1/\rho))$ samples in `Linearity Test` and `Inductive Test` and by using the self-corrector with confidence parameter $\rho/3$ in `Basis Test` the self-tester ensures that its confidence in checking each condition is at least $1 - (\rho/3)$.

Clearly if P always computes f correctly, the tester always outputs "PASS." Conversely, suppose the tester outputs "PASS." Then with probability $1 - \rho$, the hypotheses of Theorem 3 are true. By the conclusion of Theorem 3, it follows that a function g that is identical to f exists and that g equals P on at least $1 - 2\epsilon$ fraction of the inputs. ■

2.1 Applications

We present some applications of Theorem 4. We remind the reader that a linear function f on a vector space V is efficiently self-testable without the generator bottleneck if there is a (linear) function h_θ that is easily computable and that satisfies $f(\theta(\alpha)) = h_\theta(\alpha, f(\alpha))$ for all $\alpha \in V$. In each of our applications f, we show that a suitable function $h = h_{\theta,f}$ exists that satisfies the above conditions. Recall the example of the polynomial evaluation function $E_c(q) = q(c)$, where the identity $E_c(xq) = cE_c(q)$ holds; in the applications below, we will only establish similar relationships. Also, for the sake of simplicity, we do not give all the technical parameters required; these can be computed by routine calculations following the proofs of the theorems in the last section.

Our applications concern linear functions of polynomials. We obtain self-testers for polynomial evaluation, Fourier transforms, polynomial differentiation, polynomial integration, and the *mod* function of polynomials. Moreover, the vector space setting lets us state some of these results in terms of the matrices that compute linear transforms of vector spaces.

Let $\mathcal{P}_n \subseteq \mathbb{K}[x]$ denote the group of polynomials in x of degree $\leq n$ over a field $\mathbb{K}$. The group $\mathcal{P}_n$ forms a vector space under usual polynomial addition and scalar multiplication by elements from $\mathbb{K}$. The polynomials $1, x, x^2, \ldots, x^n$ span $\mathcal{P}_n$, and a polynomial $q(x) = \sum_{i=0}^{n} q_i x^i$ has the vector representation $(q_n, q_{n-1}, \ldots, q_1, q_0)$. The basis rotation function θ in this case is just multiplication by x, thus $\theta(q) = xq$. Note that multiplying q by x results in a polynomial of degree $n+1$. To handle this minor detail, we will assume that the program works over the domain $\mathcal{P}_{n+1}$ and we conclude correctness over $\mathcal{P}_n$.

Polynomial Evaluation. For any $c \in \mathbb{K}$, let $E_c(q)$ denote, as described before, the function that returns the value $q(c)$. This function is linear. Moreover, the relation between $E_c(xq)$ and $E_c(q)$ is simple and linear: $E_c(xq) = cE_c(q)$. To self-test a program P that claims to compute E_c, the `Inductive Test` is simply to choose many random q's, and verify that $P_{\mathrm{sc}}(xq) = cP_{\mathrm{sc}}(q)$ holds.

Vandermonde Operators and the Discrete Fourier Transform. If $u_1, u_2, \ldots, u_{n+1}$ are $n+1$ distinct elements of $\mathbb{K}$, then one may wish to evaluate a polynomial $q \in \mathcal{P}_n$ simultaneously on all $n+1$ points. The ideas for E_c extend easily to this case, for $E_u(xq) = uE_u(q)$ for any $u \in \mathbb{K}$, and these relations hold simultaneously.

Let ω be a principal $(n+1)$-st root of unity in $\mathbb{K}$. The operation of converting a polynomial from its coefficient representation to pointwise evaluation at the powers of ω is known as the Discrete Fourier Transform (DFT). DFT has many fundamental applications that include fast multiplication of integers and polynomials. With our notation, the DFT of a polynomial $q \in \mathcal{P}_n$ is simply $F(q) = (E_{\omega^0}(q), E_{\omega^1}(q), \ldots, E_{\omega^n}(q))$. The DFT F is linear, and $F(xq) = (\omega^0 E_{\omega^0}(q), \ldots, \omega^n E_{\omega^n}(q))$. Notice that here the function h is really n coordinate functions h_{ω^i} for $0 \leq i \leq n$. The self-tester will simply choose q's randomly, request the program to compute $F(q)$ and $F(xq)$, and verify for each i, $0 \leq i \leq n$, that $(F(xq))[i] = \omega^i (F(q))[i]$ holds.

This suggests the following generalization (for the case of arbitrary vector spaces). Simultaneous evaluation of a polynomial at $d+1$ points $u_1, \ldots, u_{n+1}$ corresponds to multiplying the vector p by a Vandermonde matrix M, where $M_{ij} = u_i^{j-1}$. The ideas used to test simultaneous evaluation of polynomials and the DFT extend to give a self-tester for any linear transform that is represented by a Vandermonde matrix.

The matrix for the DFT can be written as a Vandermonde matrix F, where $F_{ij} = \omega^{ij}$. The inverse of the DFT, that is, converting a polynomial from pointwise representation to coefficient form, also has a Vandermonde matrix whose entries are given by $\widetilde{F}_{ij} = (1/\det F)\omega^{-ij}$. It follows that the inverse Fourier Transform can be self-tested efficiently. Another point worth mentioning here is that in carrying out the `Inductive Test` the self-tester does not have to compute $\det F$. All it needs to do is verify that for many randomly chosen q's, the identity $(\widetilde{F}(xq))[i] = \omega^{-i}(\widetilde{F}(q))[i]$ holds.

Operators in Elementary Jordan Canonical Form. A linear operator M is said to be in elementary Jordan canonical form if all the diagonal entries of M are c for some $c \in \mathbb{K}$, and all the elements to the left of the main diagonal (the first non-principal diagonal in the lower triangle of M) are 1's. It is easy to verify that $M\theta = \theta M + M'$, where M' is a matrix that has a -1 in the top left corner and a 1 in the bottom right corner and zeroes elsewhere. Therefore, for every $v = (v_n, v_{n-1}, \ldots, v_2, v_1)^T$ in the vector space, $M(\theta(v)) = \theta(M(v)) + (-v_n, 0, \ldots, 0, v_1)^T$. This gives an easy way to implement the `Inductive Test` in the self-tester.

An attempt to extend this to matrices in Jordan canonical form, or even to diagonal matrices, seems not to work. If, however, a diagonal or shifted diagonal matrix has a special structure, then we can obtain self-testers that avoid the generator problem. For example, the matrix corresponding to the differentiation of polynomials has a special structure: it contains the entries $n, n-1, \ldots, 1$ on the diagonal above the main diagonal.

Differentiation and Integration of Polynomials. Differentiation of polynomials is a linear function $D : \mathcal{P}_n \to \mathcal{P}_{n-1}$. We have the explicit form for h: $D(xq) = q + xD(q)$. Integration of polynomials is a linear function $I : \mathcal{P}_n \to \mathcal{P}_{n+1}$. The explicit form for h is $I(xq) = xI(q) - I(I(q))$. Even though this does not readily fit into our framework (since it is not of the form $I(xq) = h(q, I(q))$), the proof of Theorem 3 can be easily modified to handle this case using the linearity of I. For completeness, we spell out the details for the robustness of the `Inductive Test` which is the only change required.

Claim 5 *If $g : \mathcal{P}_n \to \mathcal{P}_{n+1}$ is a linear function that satisfies $\Pr_{q\in\mathcal{P}_n}[g(xq) \neq xg(q) - g(g(q))] = \epsilon < 1/2$, then $\Pr_{q\in\mathcal{P}_n}[g(xq) = xg(q) - g(g(q))] = 1$.*

Proof. (of claim)

$$\begin{aligned}
\Pr_{r\in\mathcal{P}_n}[g(xq) &= g(x(r+q-r)) \\
&= g(xr + x(q-r)) \\
&= g(xr) + g(x(q-r)) \\
&= xg(r) - g(g(r)) + xg(q-r) - g(g(q-r)) \\
&= x(g(r) + g(q-r)) - g(g(r) + g(q-r)) \\
&= xg(q) - g(g(q))] \geq 1 - 2\epsilon > 0.
\end{aligned}$$

Since the event $g(xq) = xg(q) - g(g(q))$ holds with positive probability and is independent of r, it holds with probability one. ∎

Thus we can avoid the generator bottleneck for these functions. This can be considered as a special case of the previous application.

Higher Order Differentiation of Polynomials. Let D^k denote the k-th differential operator. It is easy to write a recurrence-like identity for D^k in terms of $D^j, j < k$. This gives us a self-tester only in the library setting described in [BLR93, Rub90], where one assumes that there are programs to compute all these differential operators. If we wish to self-test a program that only computes D^k and have no library of lower-order differentials, this assumption is not valid. To remedy this, we will use the following lemma, which is proved in the Appendix.

Lemma 6 *If q is a polynomial in x of degree $\geq k$, then*

$$\sum_{i=0}^{k} \binom{k}{i} (-x)^{k-i} D^k(x^i q) = k! q$$

Using this identity, the self-tester can perform an `Inductive Test`. The robustness of the `Inductive Test` can be established as in the proof of Theorem 3. For completeness, we outline the key step here. Let c_i denote the coefficient of the term $D^k(x^i q)$ in the sum in Lemma 6. Thus $c_i = \binom{k}{i}(-x)^{k-i}$.

Claim 7 *If g is a linear function that satisfies* $\Pr_{q\in\mathcal{P}_n}[\sum_i c_i g(x^i q) \neq k!q] = \epsilon < 1/2$, *then* $\Pr_{q\in\mathcal{P}_n}[\sum_i c_i g(x^i q) = k!q] = 1$.

Proof. (of claim)

$$\begin{aligned}
\Pr_{r\in\mathcal{P}_n}[\sum_i c_i g(x^i q) &= \sum_i c_i g(x^i(r+q-r)) \\
&= \sum_i c_i g(x^i r) + \sum_i c_i g(x^i(q-r)) \\
&= k!r + k!(q-r) \\
&= k!q] \geq 1-2\epsilon > 0.
\end{aligned}$$

Here the first equality is rewriting, the second equality holds with probability $1-2\epsilon$ by the assumption that $\Pr_q[\sum_i c_i g(x^i q) \neq k!q] = \epsilon$ Since the event $\sum_i c_i g(x^i q) = k!q$ holds with positive probability and is independent of r, it holds with probability one. ■

Thus, testing if g satisfies this identity for most q suffices to ensure that g satisfies this identity everywhere. If g does satisfy this identity, then we know the following: if $g(q) = D^k(q), g(xq) = D^k(xq), g(x^2 q) = D^k(x^2 q), \ldots, g(x^{k-1}q) = D^k(x^{k-1}q)$, then $g(x^k q) = D^k(x^k q)$. To conclude that $g \equiv D^k$ by induction, we need to modify `Basis Test` to test k base cases: if $g(1) = D^k(1), g(x) = D^k(x), g(x^2) = D^k(x^2), \ldots, g(x^{k-1}) = D^k(x^{k-1})$, i.e., if $g(1) = g(x) = g(x^2) = \cdots = g(x^{k-1}) = 0$.

Mod Function. Let $\alpha \in \mathbb{K}[x]$ be a monic irreducible polynomial. Let $M_\alpha(q)$ denote the mod function with respect to α, that is, $M_\alpha(q) = q \bmod \alpha$. This is a linear function when the addition is interpreted as mod α addition. Since α is monic, the degree of $M_\alpha(q)$ is always less than $\deg \alpha$. If $c \in \mathbb{K}$ is the coefficient of the highest degree term in $M_\alpha(q)$, we have

$$M_\alpha(xq) = \begin{cases} xM_\alpha(q) & \text{if } \deg xM_\alpha(q) < \deg \alpha \\ xM_\alpha(q) - c\alpha & \text{if } \deg xM_\alpha(q) = \deg \alpha \end{cases}$$

As before, in testing if a program P computes the function M_α, step (3) of the self-tester is to choose many q's at random, compute $P_{\rm sc}(q)$ and $P_{\rm sc}(xq)$, and verify that one of the identities $P_{\rm sc}(xq) = xP_{\rm sc}(q)$ or $P_{\rm sc}(xq) = xP_{\rm sc}(q) - cq$ holds (depending on the degree of q).

3 Multilinear Functions

In this section we extend the ideas in Section 2 to multilinear functions. A k-variate function f is called *k-linear* if it is linear in each of its variables when the other variables are fixed, i.e., $f(\alpha_1,\ldots,\alpha_{i-1},\alpha_i+\beta_i,\alpha_{i+1},\ldots,\alpha_k) = f(\alpha_1,\ldots,\alpha_{i-1},\alpha_i,\alpha_{i+1},\ldots,\alpha_k) + f(\alpha_1,\ldots,\alpha_{i-1},\beta_i,\alpha_{i+1},\ldots,\alpha_k)$ for all $i = 1,\ldots,k$.

Our main motivating example for a multilinear function is polynomial multiplication $f : \mathcal{P}_{n-1} \times \mathcal{P}_{n-1} \to \mathcal{P}_{2n-1}$, which is bilinear. Note that the domain of f is generated by n^2 generators of the form (x^i, x^j) for $0 \leq i, j \leq n$ (i.e., pairs of generators of $\mathcal{P}_{n-1}$). Now, suppose we wish to test P that purports to compute f. The naive approach would require doing the `Generator Tests` at these n^2 generators. This requires $O(n^2)$ calls to P, rendering the self-tester highly inefficient. Blum, Luby, and Rubinfeld [BLR93] give a more efficient bootstrap self-tester that makes $O(\log^{O(k)} n)$ calls to P. It can be seen that for general k-linear functions, their method can be extended to yield a tester that makes $O(\log^{O(k)} n)$ calls to P. (In our context, it is allowable to think of k as a constant since changing k results in an entirely different function f.) We are interested in reducing the number of calls to P with respect to the problem size n for a specific function f. The complexity of the tester we present here is independent of n, and the self-tester is required to know the correct value of f at only one point. As in the previous section, our result applies to many general multilinear functions over large vector spaces.

As before, we define a set of properties depending on f, that, if satisfied by P, would necessarily imply that P must be the same as the particular multilinear function f. For simplicity, we present the following theorem for f that is bilinear. This is an analog of Theorem 3 for multilinear functions.

Theorem 8 *Suppose f is a bilinear function from V^2 into a ring R, and suppose P is a program for f.*

(a) Let $\epsilon < 1/4$, and suppose P satisfies the following condition:

(1) $\Pr_{\alpha_1,\alpha_2,\beta_1,\beta_2 \in V}[P(\alpha_1+\beta_1,\alpha_2+\beta_2) \neq P(\alpha_1,\beta_1)+P(\alpha_1,\beta_2)+P(\alpha_2,\beta_1)+P(\alpha_2,\beta_2)] \leq \epsilon.$

Then the function g defined by $g(\alpha_1,\alpha_2) = \text{majority}_{\beta_1,\beta_2\in V}\{P(\alpha_1-\beta_1,\alpha_2-\beta_2)+P(\alpha_1-\beta_1,\beta_2)+P(\beta_1,\alpha_2-\beta_2)+P(\beta_1,\beta_2)\}$ is a bilinear function on V^2, and g agrees with P on at least $1-2\epsilon$ fraction of the inputs.

(b) Furthermore, suppose g satisfies the following conditions:

(2) $g(e_1,e_1) = f(e_1,e_1)$,

(3) $\Pr_{\alpha_1,\alpha_2\in V}[g(\theta(\alpha_1),\alpha_2) \neq h_\theta^{(1)}(\alpha_1,g(\alpha_1,\alpha_2))] \leq \epsilon$, *and*

$\Pr_{\alpha_1,\alpha_2\in V}[g(\alpha_1,\theta(\alpha_2)) \neq h_\theta^{(2)}(\alpha_2,g(\alpha_1,\alpha_2))] \leq \epsilon.$

Then $g(\alpha_1,\alpha_2) = f(\alpha_1,\alpha_2)$ for all $\alpha_1,\alpha_2 \in V$.

Proof. A simple extension of the proof in [BLR93] shows that g is bilinear. (Better bounds on ϵ via a different test can be obtained by appealing to [BFL91].) As in the proof of Theorem 3, it suffices to show that given the three conditions, a stronger version of condition (3) holds: $g(\theta(\alpha_1),\alpha_2) = h_\theta^{(1)}(\alpha_1,g(\alpha_1,\alpha_2))$ and $g(\alpha_1,\theta(\alpha_2)) = h_\theta^{(2)}(\alpha_2,g(\alpha_1,\alpha_2))$ for every $\alpha_1,\alpha_2 \in V$. With the addition of this last property, it can be shown that $g \equiv f$. Taking condition (2) that $g(e_1,e_1) = f(e_1,e_1)$ as the base case and inducting by obtaining (e_{i+1},e_i) and (e_i,e_{i+1}) from (e_i,e_i) via an application of θ to either generator for all $1 \leq i < n$, it can be shown that $g(e_i,e_j) = f(e_i,e_j)$ for any bases elements e_i,e_j of V. This, combined with the bilinearity property of g, implies the correctness of g on every input.

Now we proceed to show the required intermediate result that given conditions (1) and (2), g satisfies the stronger version of condition (3) that we require above:

$$
\begin{aligned}
\Pr_{\beta_1,\beta_2\in V^2} [g(\alpha_1,\theta(\alpha_2))& \\
&= g(\beta_1+\alpha_1-\beta_1,\theta(\beta_2+\alpha_2-\beta_2)) \\
&= g(\beta_1,\theta(\beta_2+\alpha_2-\beta_2)) + g(\alpha_1-\beta_1,\theta(\beta_2+\alpha_2-\beta_2)) \\
&= g(\beta_1,\theta(\beta_2)) + g(\beta_1,\theta(\alpha_2-\beta_2)) + g(\alpha_1-\beta_1,\theta(\beta_2)) + g(\alpha_1-\beta_1,\theta(\alpha_2-\beta_2)) \\
&= h_\theta^{(2)}(\beta_2,g(\beta_1,\beta_2)) + h_\theta^{(2)}(\alpha_2-\beta_2,g(\beta_1,\alpha_2-\beta_2)) + \\
&\quad h_\theta^{(2)}(\beta_2,g(\alpha_1-\beta_1,\beta_2)) + h_\theta^{(2)}(\alpha_2-\beta_2,g(\alpha_1-\beta_1,\alpha_2-\beta_2)) \\
&= h_\theta^{(2)}(\alpha_2,g(\beta_1,\alpha_2)) + h_\theta^{(2)}(\alpha_2,g(\alpha_1-\beta_1,\alpha_2)) \\
&= h_\theta^{(2)}(\alpha_2,g(\alpha_1,\alpha_2))] \geq 1-4\epsilon > 0.
\end{aligned}
$$

The first equality is a rewriting of terms. Multilinearity of g implies the second and third equalities. If the probability $\Pr[g(\alpha_1, \theta(\alpha_2)) \neq h_\theta^{(2)}(\alpha_2, g(\alpha_1, \alpha_2))] < \epsilon$, then the fourth equality fails with probability less than 4ϵ. The rest of the equality follows from the multilinearity of $h_\theta^{(2)}$ and g. If $\epsilon < 1/4$, this probability is non-zero. Since the first and last terms are independent of β_1, β_2, and are equal with nonzero probability, the result follows. A similar approach works for $h_\theta^{(1)}$ as well. ■

```
Multilinearity Test:
  Repeat O(1/ε log 1/ρ) times
    Pick α1, α2, β1, β2 ∈_R V
    Verify P(α1 + β1, α2 + β2) = P(α1, α2) + P(β1, α2) + P(α1, β2) + P(β1, β2)
    Reject if the test fails

Basis Test:
  Verify P_sc(e1, e1) = f(e1, e1)

Inductive Test:
  Repeat O(1/ε log 1/ρ) times
    Pick α1, α2 ∈_R V
    Verify P_sc(θ(α1), α2) = h_θ^(1)(α1, P_sc(α1, α2))
    Verify P_sc(α1, θ(α2)) = h_θ^(2)(α2, P_sc(α1, α2))
    Reject if the test fails
```

Note that in the latter two tests we use a self-corrected version P_{sc} of P. The notion of self-correctors for multilinear functions over vector spaces is implied by random self-reducibility.

It is easy to see that Theorem 8 extends to an arbitrary k-linear function so long as $\epsilon < 1/2^k$. Thus, we obtain the following theorem whose proof mirrors that of Theorem 4.

Theorem 9 *If f is a k-variate linear function, then for any $\rho < 1$ and $\epsilon < 1/2^k$, the above three tests comprise a $2^k\epsilon$-self-tester for f that succeeds with probability at least $1 - \rho$.*

3.1 Applications

Let q_1, q_2 denote polynomials in x. The function $M(q_1, q_2)$ that multiplies two polynomials is symmetric and linear in each variable. Moreover, since $M(xq_1, q_2) = M(q_1, xq_2) = xM(q_1, q_2)$, polynomial multiplication has an efficient self-tester.

An interesting application of polynomial multiplication, together with the mod function described in Section 2.1, is the following. It is well-known that a degree n (finite) extension $\mathbb{K}$ of a finite field $\mathbb{F}$ is isomorphic to the field $\mathbb{F}[x]/(\alpha)$, where α is an irreducible polynomial of degree n over $\mathbb{F}$. Under this isomorphism, each element of $\mathbb{K}$ is viewed as a polynomial of degree $\leq n$ over $\mathbb{F}$, addition of two elements $q_1, q_2 \in \mathbb{K}$ is just their sum $q_1 + q_2$ as polynomials, and multiplication of $q_1, q_2 \in \mathbb{K}$ is given by $q_1 q_2 \bmod \alpha$. It follows that field arithmetic (addition and multiplication) in finite extensions of a finite field can be self-tested without the generator bottleneck, that is, the number of calls made to the program being tested is independent of the degree of the field extension.

3.2 Matrix Multiplication

Let $\mathcal{M}_n$ denote the algebra of $n \times n$ matrices over $\mathbb{Z}_p$, and let $f : \mathcal{M}_n \times \mathcal{M}_n \to \mathcal{M}_n$ denote matrix multiplication. Matrix multiplication is a bilinear function; however, since it is a matrix operation rather than a vector operation, it requires a slightly different treatment from the general multilinear functions. $\mathcal{M}_n$, viewed as an additive group, has n^2 generators; one possible set of generators is $\{E_{i,j} \mid 1 \le i, j \le n\}$, where each generator $E_{i,j}$ is a matrix that has a 1 in position (i, j) and 0's elsewhere. We note that any generator $E_{i,j}$ can be converted into any other generator $E_{k,\ell}$ via a sequence of horizontal and vertical rotations obtained by multiplications by the special *permutation matrix* Π:

$$\Pi = \begin{bmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \ldots & 0 \end{bmatrix} \quad \Rightarrow \quad E_{i,j} = \Pi^{k-i} \cdot E_{k,\ell} \cdot \Pi^{\ell-j}.$$

The rotation operations correspond to the θ operator of the model for multilinear functions. There are, however, two different kinds of rotations – horizontal and vertical – due to the two-dimensional nature of the input, and the function h defining the behavior of the function with respect to these rotations is not always easily computable short of actually performing a matrix multiplication. We therefore exploit some additional properties of the problem to come up with a set of conditions that are sufficient for P to be computing matrix multiplication f. Let $\mathcal{M}'_n$ denote the subgroup of $\mathcal{M}_n$ that contains only matrices with columns $2, \ldots, n$ all-zero.

Theorem 10 *Let P be a program for f and $\epsilon < 1/8$.*

(a) Suppose P satisfies the following:

(1) $\Pr_{X,Y,Z,W \in \mathcal{M}_n}[P(X+Z, Y+W) \ne P(X,Y) + P(X,W) + P(Z,Y) + P(Z,W)] \le \epsilon.$

Then the function g defined by $g(X,Y) = \text{majority}_{Z,W \in \mathcal{M}_n}\{P(X-Z, Y-W) + P(X-Z, W) + P(Z, Y-W) + P(Z,W)\}$ is a bilinear function on $\mathcal{M}_n^2$, and g agrees with P on at least $1 - 2\epsilon$ fraction of the inputs.

(b) Furthermore, suppose g satisfies the following conditions:

(2) $\Pr_{X \in \mathcal{M}'_n, Y \in \mathcal{M}_n}[g(X,Y) \ne f(X,Y)] \le \epsilon,$

(3) $\Pr_{X \in \mathcal{M}_n}[g(X,\Pi) \ne f(X,\Pi)] \le \epsilon$ *and* $\Pr_{X \in \mathcal{M}_n}[g(\Pi,X) \ne f(\Pi,X)] \le \epsilon,$

(4) $\Pr_{X,Y,Z \in \mathcal{M}_n}[g(X, g(Y,Z)) \ne g(g(X,Y),Z))] \le \epsilon.$

Then $g(X,Y) = f(X,Y)$ for all $X, Y \in \mathcal{M}_n$.

Proof. The bilinearity of g is as in the proof of Theorem 8. We first show that if conditions (3) and (4) on g hold for all inputs and if condition (2) can be extended such that $g(E_{i,1}, X) = f(E_{i,1}, X)$ for all $1 \le i \le n$ and $X \in \mathcal{M}_n$, then the result follows. We then proceed to show that if the conditions (2), (3), (4) hold with high probability over the inputs then they hold for all inputs.

To show that the mentioned properties are sufficient to identify g as matrix multiplication, note that from the multilinearity of g we can write,

$$g(X,Y) = g\left(\sum_{1 \le i,j \le n} x_{i,j} \cdot E_{i,j}, \sum_{1 \le k,\ell \le n} y_{k,\ell} \cdot E_{k,\ell}\right) = \sum_{1 \le i,j,k,\ell \le n} x_{i,j} \cdot y_{k,\ell} \cdot g(E_{i,j}, E_{k,\ell})$$

If $g(E_{i,j}, E_{k,\ell}) = f(E_{i,j}, E_{k,\ell})$ for all i, j, k, ℓ, then multilinearity implies that g is the same as f. Now, using our assumptions, we proceed to show the former holds:

$$\begin{aligned} g(E_{i,j}, E_{k,\ell}) &= g(f(E_{i,1}, \Pi^{j-1}), f(\Pi^{k-1}, E_{1,\ell})) \\ &= g(g(E_{i,1}, \Pi^{j-1}), g(\Pi^{k-1}, E_{1,\ell})) \\ &= g(E_{i,1}, g(\Pi^{j+k-2}), E_{1,\ell}) \\ &= g(E_{i,1}, E_{k+j-2,\ell}) \\ &= f(E_{i,1}, E_{k+j-2,\ell}) \\ &= f(E_{i,j}, E_{k,\ell}) \end{aligned}$$

The first equality is just a rewriting of the two generators in terms of other generators. The second one follows from the strengthening of condition (3) that g computes f whenever one of its arguments is equal to a power of Π. The third one follows from associativity of g, and the fourth one holds because g is the same as f when the first input is a power of Π and because of rewriting of $E_{k+j-2,\ell}$. The fifth equality is true because g computes f correctly when its first argument is $E_{i,1}$ (as the consequence of condition (2), see Lemma 11). The last one is a rewriting of the previous equality, using the associativity of multiplication. Therefore, g is the same function as f.

We now proceed to show that the three stronger conditions we used are implied by the conditions that we recount in the statement of the theorem. First, we show that condition (2) implies a stronger version of itself:

Lemma 11 *If condition (2) in Theorem 10 holds then $g(E_{i,1}, Y) = f(E_{i,1}, Y)$ for all $Y \in \mathcal{M}_n$ and $1 \leq i \leq n$.*

Proof. We in fact show something stronger. We show that $g(X, Y) = f(X, Y)$ for all $X \in \mathcal{M}'_n$ and $Y \in \mathcal{M}_n$:

$$\begin{aligned} \Pr_{Z \in \mathcal{M}'_n, W \in \mathcal{M}_n}[g(X, Y) &= g(Z, Y - W) + g(X - Z, Y - W) + g(Z, W) + g(X - Z, W) \\ &= f(Z, Y - W) + f(X - Z, Y - W) + f(Z, W) + f(X - Z, W) \\ &= f(X, Y)] > 1 - 4\epsilon > 0 \end{aligned}$$

The second equality holds with probability $\geq 1 - 4\epsilon$ by condition (2). All the rest hold from linearity of g and f. The result follows since $\epsilon < 1/4$. The lemma follows since $E_{i,1} \in \mathcal{M}'_n$. ∎

An immediate adaptation of the proof of Lemma 11 can be used to extend condition (3) to hold for all inputs.

Next, we show that the linearity of g makes it possible to conclude from hypothesis (4) that g is associative.

Lemma 12 *If condition (4) in Theorem 10 holds then g is always associative.*

Proof.

$$\begin{aligned} &\Pr_{X_1, Y_1, Z_1 \in \mathcal{M}_n}[g(g(X, Y), Z) \\ &= g(g(X_1, Y_1), Z_1) + g(g(X - X_1, Y_1), Z_1) + g(g(X_1, Y - Y_1), Z_1) + \\ &\quad g(g(X - X_1, Y - Y_1), Z_1) + g(g(X_1, Y_1), Z - Z_1) + g(g(X - X_1, Y_1), Z - Z_1) + \\ &\quad g(g(X_1, Y - Y_1), Z - Z_1) + g(g(X - X_1, Y - Y_1), Z - Z_1) \\ &= g(X_1, g(Y_1, Z_1)) + g(X - X_1, g(Y_1, Z_1)) + g(X_1, g(Y - Y_1, Z_1)) + \\ &\quad g(X - X_1, g(Y - Y_1, Z_1)) + g(X_1, g(Y_1, Z - Z_1)) + g(X - X_1, g(Y_1, Z - Z_1)) + \\ &\quad g(X_1, g(Y - Y_1, Z - Z_1)) + g(X - X_1, g(Y - Y_1, Z - Z_1)) \\ &= g(X, g(Y, Z))] \geq 1 - 8\epsilon > 0 \end{aligned}$$

The first equality holds from the linearity of g after expanding X, Y, Z as $X_1 + (X - X_1), Y_1 + (Y - Y_1), Z_1 + (Z - Z_1)$ respectively. The second one is true by the condition (4) with probability $\geq 1 - 8\epsilon$. The last one is a recombination of terms using linearity. ■

We now present the test for associativity:

> `Associativity Test:`
>   `Repeat` $O(\frac{1}{\epsilon}\log\frac{1}{\rho})$ `times`
>   `Pick` $X, Y, Z \in_R \mathcal{M}_n$
>   `Verify` $P_{\mathrm{sc}}(X, P_{\mathrm{sc}}(Y, Z)) = P_{\mathrm{sc}}(P_{\mathrm{sc}}(X, Y), Z)$
>   `Reject if the test fails`

This completes the proof of Theorem 10. ■

Thus, a self-tester can be built by testing conditions (1), (2), (3), and (4), which correspond to the `Property Test` the `Basis Test`, the `Inductive Test` and the `Associativity Test` respectively. Note that testing conditions (2) and (3) involve knowing the value of f at random inputs. These inputs, however, come from a restricted subspace which makes it possible to compute f both easily and efficiently. The following theorem is immediate.

Theorem 13 *For any $\rho < 1$ and $\epsilon < 1/8$, there is an ϵ-self-tester for matrix multiplication that succeeds with probability at least $1 - \rho$.*

4 Nonlinear Functions

In this section, we consider nonlinear functions. Specifically, we deal with exponentiation and constant degree polynomials in the ring of polynomials over the finite fields $\mathbb{Z}_p$. It is obvious that exponentiation and constant degree polynomials are clearly defined over this ring.

4.1 Constant Degree Exponentiation

We first consider the function $f(q) = q^d$ for some d (that is, raising a polynomial to the d-th power). Suppose a program P claims to perform this exponentiation for all degree n polynomials $q \in \mathcal{P}_n \subseteq \mathbb{K}[x]$. Using the low-degree test of Rubinfeld and Sudan [RS96] (see also [GLR$^+$91]) we can first test if the function computed by P is close to some degree d polynomial g. As before, using the self-corrected version P_{sc} of P, we can also verify that $g(e_1) = f(e_1)$.

The induction identity $f(xq) = x^d f(q)$ also applies, so it remains to show how to verify that $g(xq) = x^d g(q)$ for all $q \in \mathcal{P}_n$. We follow a strategy similar to the case of linear functions, this time using the Lagrange interpolation formula as the robust property that identifies a degree d polynomial. We note that this idea is similar to the use of the interpolation formula by Gemmell, *et al.* [GLR$^+$91], which extends the [BLR93] result from linear functions to low-degree polynomials. Before proceeding with the proof, we state the following fact concerning the Lagrange interpolation identity.

Fact 14 *Let g be a degree d polynomial. For any $q \in \mathcal{P}_n$, if $q_1, q_2, \ldots, q_{d+1}$ are distinct elements of $\mathcal{P}_n$,*

$$g(q) = \sum_{i=1}^{d+1} g(q_i) \prod_{j \neq i} \frac{q - q_j}{q_i - q_j}; \qquad \textit{and also} \qquad g(xq) = \sum_{i=1}^{d+1} g(xq_i) \prod_{j \neq i} \frac{q - q_j}{q_i - q_j}.$$

The self-tester for $f(q) = q^d$ comprises the following tests:

```
Degree Test:
  Verify P is close to a degree d polynomial g (low-degree test)
  Reject if the test fails

Basis Test:
  Verify P_sc(e_1) = f(e_1)

Inductive Test:
  Repeat O((1/ε) log (1/ρ)) times
    Pick α ∈_R V
    Verify P_sc(θ(α)) = x^d P_sc(α)
    Reject if the test fails
```

Let β denote the probability that the $d+1$ random choices from the domain produce distinct elements. We will assume that the domain is large enough so that β is close to 1.

Assume that P passes the `Degree Test` and $P_{\rm sc}$ passes the `Basis Test` that is, P agrees with some degree-d polynomial g on most inputs. We note that the low-degree of test of [RS96] makes $O((1/\epsilon)\log(1/\rho))$ calls to render a decision with confidence $1-(\rho/3)$. Furthermore, $P_{\rm sc}$ also requires only $O((1/\epsilon)\log(1/\rho))$ calls to compute g correctly with probability $1-(\rho/3)$. Below we sketch the proof that if $\epsilon < \beta/(d+1)$ and $P_{\rm sc}$ passes the `Inductive Test` then g satisfies $g(xq) = x^d g(q)$ everywhere. Note that $(1/\epsilon) = \Theta(d)$, so the time taken by the tester is only $\Theta(d)$ (when ρ is a constant).

$$\begin{aligned}
\Pr_{q_1,\ldots,q_{d+1}}[g(xq) &= \sum_{i=1}^{d+1} a_i g(xq_i) \\
&= \sum_{i=1}^{d+1} a_i x^d g(q_i) \\
&= x^d \sum_{i=1}^{d+1} a_i g(q_i) \\
&= x^d g(q)] \geq \beta - (d+1)\epsilon > 0.
\end{aligned}$$

Here $a_i = \prod_{j \neq i}(q-q_j)/(q_i-q_j)$. The first equality is Fact 14, and applies since g has been verified to be a degree d polynomial. Since the q_i's are uniformly and identically distributed, by `Inductive Test` the second equality fails with probability $< (d+1)\epsilon$. The third equality is just rewriting, and the fourth equality is due to Fact 14 (the interpolation identity), which can be applied so long as the q_i's are distinct, an event that occurs with probability β. Since the equality $g(xq) = x^d g(q)$ holds independent of q_i's, if $\epsilon < \beta/(d+1)$, it holds with probability 1.

Theorem 15 *The function $f(q) = q^d$ (where $q \in \mathcal{P}_n$) has an $O(1/d)$-self-tester that makes $O(d)$ queries.*

4.2 Constant Degree Polynomials

Next we consider extending the result of Section 4.1 to arbitrary degree d polynomials $f : \mathcal{P}_n \to \mathcal{P}_{n^d}$. Clearly the low-degree test and the basis test work as before. The interpolation identity is valid, too. The missing ingredient is the availability of an identity like "$f(xq) = x^d f(q)$," which, as we have shown above, is a robust property that can be efficiently tested. We show how to get around this difficulty; this idea is based on a suggestion due to R. Rubinfeld [private communication].

Suppose $f : \mathcal{P}_n \to \mathcal{P}_{n^d}$ is a degree d polynomial (eg., $f(q) = q^2 + q + 1$), and suppose a program P purports to compute f. Our strategy is to design a self-corrector R for P and to then estimate the fraction of inputs q such that $P(q) \neq R(q)$. The difficulty in implementing this idea by directly using the random self-reducibility of f is that the usefulness of the self-corrector (to compute f correctly on every input) depends critically on our ability to certify that P is correct on most inputs. Since checking whether P is correct on most inputs is precisely the task of self-testing, we seem to be going in cycles.

To circumvent this problem, we will design an intermediate *multiplication program* Q that uses P as an oracle. By the results of Section 3, Q can be efficiently self-tested without the generator bottleneck. Furthermore, if the self-tester certifies Q as correct on most inputs, we can also build a self-corrected version Q_{sc} that computes multiplication correctly on every input with high probability. Finally, using Q_{sc} to perform the d required multiplications, we can build the program R that computes f correctly on every input with high probability.

To design the program Q, we prove the following technical lemma that helps us express d-ary multiplication in terms of f—that is, we establish a reduction from the multilinear function $\prod_{i=1}^d q_i$ to the nonlinear function f. This reduction is a generalization to degree d of the elementary polarization identity $xy = ((x+y)^2 - (x-y)^2)/4$, but slightly stronger in that it works for arbitrary polynomials of degree d, not just degree-d exponentiation.

Lemma 16 *For $x \in \{0,1\}^d$, let x_i denote the i-th bit of x. For any polynomial $G(q) = \sum_i p_i q^i$ of degree d,*

$$\sum_{x \in \{0,1\}^d} \left(\prod_{i=1}^{d} (-1)^{x_i} \right) G\left(\sum_{i=1}^{d} (-1)^{x_i} q_i \right) = p_d 2^d d! \prod_{i=1}^{d} q_i.$$

Using the reduction given by Lemma 16, we will show how to construct an ϵ-self-tester T for f (for $\epsilon = \Theta(2^{-d})$), following the outline sketched.

(1) First we build a program Q that performs c-ary multiplication for any $c \leq d$ (if $c < d$, we can simply multiply by extra 1's). The program Q is then self-tested efficiently (without the generator bottleneck) by using a $(1/2^{d+1})$-self-tester for the d-variate multilinear multiplication function from Section 3. The number of queries made to P in this process is $O(1)$, where the constant depends on d (the degree of f) but not on n (the dimensionality of the domain of f). Thus if Q passes this self-testing step, then it computes multiplication correctly on all but $1/2^{d+1}$ fraction of the inputs. If Q fails the self-testing process, the self-tester T rejects.

(2) Next we build a reliable program Q_{sc} that self-corrects Q using the random self-reducibility of the multilinear multiplication function. That is, Q_{sc} can be used to compute c-ary multiplication (for any $c \leq d$) correctly for every input with probability at least $1 - \rho$ for any

constant $\rho > 0$. In particular, by making $O(2^d \log d)$ calls to Q (and hence $O(2^{2d} \log d)$ calls to P), Q_{sc} can be used to compute multiplication correctly for every input with probability at least $1 - (1/10d)$.

(3) Next, we use Q_{sc} to build the program R that computes $f(q)$ in the straightforward way by using Q_{sc} to compute the d required multiplications. If Q_{sc} computes multiplication correctly with probability $1 - (1/10d)$, then R computes f correctly for any input with probability at least 0.9.

(4) Finally, T randomly picks $N = O((1/\epsilon)\log(1/\rho))$ many samples q and checks if $P(q) = R(q)$, and outputs "PASS" iff $P(q) = R(q)$ for all the chosen values of q.

It is easy to see that if P computes f correctly on all inputs, the self-tester T will output "PASS" with probability one. For the converse, suppose that $\delta =_{\text{def}} \Pr_q[P(q) \neq f(q)] > \epsilon$, and yet T outputs "PASS." We will upper bound the probability of this event by ρ.

Since Q passes the self-testing step (Step (1)), it computes multiplication correctly on all but $1/2^{d+1}$ fraction of the inputs, and therefore, the the use of the self-corrector Q_{sc} as described in Step (2) is justified. This, in turn, implies the guarantee made of R in Step (3): for every input q, $R(q) = f(q)$ with probability at least 0.9. In Step (4), the probability that $P(q) \neq R(q)$ for a single random q is at least $(0.9)\delta$. The probability that $P(q) = R(q)$ for every random input q chosen in Step (4) is therefore at most $(1-(0.9)\delta)^N = (1-(0.9)\delta)^{O((1/\epsilon)\log(1/\rho))} < (1-(0.9)\delta)^{O((1/\delta)\log(1/\rho))} < \rho$. Thus the probability that T outputs "PASS," given that $\Pr_q[P(q) \neq f(q)] > \epsilon$, is at most ρ. The following theorem is proven, modulo the proof of Lemma 16.

Theorem 17 *The function $f(q)$, where f is a polynomial in $q \in \mathcal{P}_n$ of degree d, has an $O(1/2^d)$-self-tester that makes $O(2^d)$ queries.* ■

Even though our self-tester makes $O(2^d)$ queries to test degree-d exponentiation, the number of queries is independent of n, the dimensionality of the domain. Thus, our self-tester is attractive if n is large and d is small. In particular, in conjunction with the testers for finite field arithmetic described in Section 3.1, the self-testers described here help us to efficiently self-test constant degree polynomials on finite field extensions of large dimension.

It remains to prove Lemma 16. This lemma is a direct corollary of the following lemma, which illustrates a method to express d-ary multiplication in terms of f. The proof of the next lemma is given in the Appendix.

Lemma 18 *Let $p_1, \ldots, p_d$ be distinct variables. For $x \in \{0,1\}^d$, let x_i denote the i-th bit of x. Then*

$$\sum_{x \in \{0,1\}^d} \left(\prod_{i=1}^{d} (-1)^{x_i} \right) \left(\sum_i (-1)^{x_i} p_i \right)^c = \begin{cases} 0 & \textit{if } c < d \\ 2^d d! \prod_{i=1}^d p_i & \textit{if } c = d \end{cases}$$

Acknowledgments

We are very grateful to Ronitt Rubinfeld for her valuable suggestions and guidance. We thank Professor Manuel Blum and Mandar Mitra for useful discussions. We are grateful to the two anonymous referees for valuable comments that resulted in many improvements to our exposition. The idea of describing the proof of Lemma 18 using Fourier Transforms is also due to one of the referees.

References

[ABCG93] S. Ar, M. Blum, B. Codenotti, and P. Gemmell. Checking approximate computations over the reals. In *Proc. 25th Annual ACM Symposium on the Theory of Computing*, pages 786–795, 1993.

[BFL91] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[BF90] D. Beaver and J. Feigenbaum. Hiding instances in multioracle queries. In *Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science*, Springer LNCS 415, pages 37– 48, 1990.

[BK89] M. Blum and S. Kannan. Designing programs that check their work. In *Proc. 21st Annual ACM Symposium on the Theory of Computing*, pages 86–97, 1989.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and Systerm Sciences*, 47(3):549–595, 1993.

[BW94a] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 382–392, 1994.

[BW94b] M. Blum and H. Wasserman. Reflections on the Pentium division bug. In *Proc. 8th International Software Quality Week*, 1994.

[CR92] E. Castillo, M. R. Ruiz-Cobo. *Functional Equations and Modelling in Science and Engineering,* Marcel Dekker, Inc., 1992.

[CL90] R. Cleve and M. Luby. A note on self-testing/correcting methods for trigonometric functions. TR 90-032, ICSI, Berkeley, July 1990.

[Erg95] F. Ergün. Testing multivariate linear functions: Overcoming the generator bottleneck. In *Proc. 27th Annual ACM Symposium on the Theory of Computing*, pages 407–416, 1995.

[Fei93] J. Feigenbaum. Locally random reductions in interactive complexity theory. In J.-Y. Cai, ed., *Advances in computational complexity theory*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, pages 73–98, American Mathematical Society, Providence, RI, 1993.

[FGL+91] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–12, 1991.

[Fre79] R. Freivalds. Fast probabilistic algorithms. In *Mathematical Foundations of Computer Science*, Springer LNCS 74, pages 57–69, 1979.

[GLR+91] P. Gemmell, R. Lipton, R. Rubinfeld, M. Sudan, and A. Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proc. 23rd Annual ACM Symposium on the Theory of Computing*, pages 32–42, 1991.

[KLM89] R. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. In *Journal of Algorithms*, 10(3):429–448, 1989.

[KS95] S. R. Kumar and D. Sivakumar. On self-testing without the generator bottleneck. In *Proc. 15th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer LNCS 1026, pages 248–262, 1995.

[Lip91] R. Lipton. New directions in testing. In *Proc. of DIMACS Workshop on Distributed Computing and Cryptography*, pages 191–202, 1991.

[RS92] R. Rubinfeld and M. Sudan. Testing polynomial functions efficiently and over rational domains. In *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 23–43, 1992.

[RS96] R. Rubinfeld and M. Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal on Computing*, 25(2):252–271, 1996.

[Rub90] R. Rubinfeld. *A Mathematical Theory of Self-Checking, Self-Testing, and Self-Correcting Programs.* PhD thesis, University of California at Berkeley, 1990.

[Rub94] R. Rubinfeld. Robust functional equations with applications to self-testing/ correcting. In *Proc. 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 288–299, 1994.

[Sud94] M. Sudan. On the role of algebra in the efficient verification of proofs. In M. Agrawal, V. Arvind, and M. Mahajan, editors, *Proc. Workshop on Algebraic Methods in Complexity Theory (AMCOT)*, pages 58–68, 1994. Technical Report IMSc 94/51, The Institute of Mathematical Sciences, Madras, India.

Appendix

Proof of Lemma 6

Lemma 6 *If q is a polynomial in x of degree $\geq k$, then*

$$\sum_{i=0}^{k} \binom{k}{i} (-x)^{k-i} D^k(x^i q) = k!q$$

Proof. By induction on k. The base case ($k = 0$) is obviously true. Let $k > 0$, we have $\mathcal{S}_k = \sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^{k+1}(x^i q)$. Since $D^{k+1}(x^i q) = D^k(ix^{i-1}q + x^i D(q))$ and since differentiation is linear, we have

$$\mathcal{S}_k = \left[\sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^k(ix^{i-1}q)\right] + \left[\sum_{i=0}^{k+1} \binom{k+1}{i} (-x)^{k+1-i} D^k(x^i D(q))\right].$$

Since the first term ($i = 0$) in the first sum vanishes, and since $i\binom{k+1}{i} = (k+1)\binom{k}{i-1}$, the first sum evaluates to $(k+1)\sum_{i=1}^{k+1} \binom{k}{i-1} (-x)^{k+1-i} D^k(x^{i-1}q)$, which equals $(k+1)!q$ by the inductive hypothesis. Hence it suffices to show that the second sum evaluates to 0. This summation can be written as $\sum_{i=0}^{k} \binom{k+1}{i} (-x)^{k+1-i} D^k(x^i D(q)) + D^k(x^{k+1} D(q))$. By Pascal's identity, this sum can be split into the terms $\sum_{i=0}^{k}\binom{k}{i-1}(-x)^{k+1-i}D^k(x^iD(q))$, $\sum_{i=0}^{k}\binom{k}{i}(-x)^{k+1-i}D^k(x^iD(q))$, and $D^k(x^k\cdot xD(q))$. The second term can be seen to be $(-x)k!D(q)$ using the induction hypothesis. The first and third terms can be combined to obtain $k!(xD(q))$ again using the induction hypothesis. Thus, the entire expression equals $(-x)k!D(q)+k!(xD(q))=0$. ∎

Proof of Lemma 18

Lemma 18 *Let $p_1,\ldots,p_d$ be distinct variables. For $x\in\{0,1\}^d$, let x_i denote the i-th bit of x. Then*

$$\sum_{x\in\{0,1\}^d}\left(\prod_{i=1}^{d}(-1)^{x_i}\right)\left(\sum_i(-1)^{x_i}p_i\right)^c=\begin{cases}0 & \text{if } c<d\\ 2^dd!\prod_{i=1}^{d}p_i & \text{if } c=d\end{cases}$$

Proof. The proof uses the Fourier Transform on the boolean cube $\{0,1\}^d$ (using the standard isomorphism between $\{0,1\}^d$ and $\mathbb{Z}_2^d$). Let $\mathcal{F}$ denote the space of functions from $\mathbb{Z}_2^d$ into $\mathbb{C}$. $\mathcal{F}$ is a (finite) vector space of functions of dimension 2^d. Define the inner product between functions $f,g\in\mathcal{F}$ by $\langle f,g\rangle=2^{-d}\sum_x f(x)g(x)$. For $\alpha\in\mathbb{Z}_2^d$, define the function $\chi_\alpha:\mathbb{Z}_2^d\to\mathbb{C}$ by $\chi_\alpha(x)=\prod_{i=1}^{d}(-1)^{x_i\alpha_i}$, where x_i and α_i denote the i-th bits, respectively, of x and α. It is easy to check that $\chi_\alpha(x)\chi_\alpha(y)=\chi_\alpha(x+y)$, whence every χ_α is a character of $\mathbb{Z}_2^d$. Furthermore, it is easy to check that $\langle\chi_\alpha,\chi_\beta\rangle$ equals 0 if $\alpha\neq\beta$ and equals 1 if $\alpha=\beta$. Therefore, the characters χ_α form an orthonormal basis of $\mathcal{F}$, and every function $f:\mathbb{Z}_2^d\to\mathbb{C}$ has a unique expansion in this basis as $f=\sum_\alpha\widehat{f}_\alpha\chi_\alpha$. This is called the Fourier transform of f, and the coefficients $\widehat{f}_\alpha$ are called the Fourier coefficients of f; by the orthonormality of the basis, $\widehat{f}_\alpha=\langle\chi_\alpha,f\rangle$. An easy property of the Fourier transform is that for $\alpha\neq 0^d$, $\sum_x\chi_\alpha(x)=0$ (in fact, this is true for any non-trivial character of any group).

For the proof of the lemma, we note that it suffices to prove the lemma for all complex numbers p_i. Fix a list of complex numbers $p_1,\ldots,p_d$, and define the function $f:\mathbb{Z}_2^d\to\mathbb{C}$ by $f(x)=(\sum_i(-1)^{x_i}p_i)^c$. Then the left hand side of the statement of the lemma is just $2^d\widehat{f}_\alpha$, where $\alpha=1^d$. Thus:

$$\begin{aligned}
&\sum_{x\in\mathbb{Z}_2^d}\left(\prod_{i=1}^{d}(-1)^{x_i}\right)\left(\sum_i(-1)^{x_i}p_i\right)^c\\
&= 2^d\widehat{f}_\alpha\\
&= \sum_x\chi_\alpha(x)f(x)\\
&= \sum_x\chi_\alpha(x)\sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\ldots,n_d\le c}}\binom{c}{n_1,\ldots,n_d}\prod_{i=1}^{d}(-1)^{x_in_i}p_i^{n_i}\\
&= \sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\ldots,n_d\le c}}\binom{c}{n_1,\ldots,n_d}\prod_{i=1}^{d}p_i^{n_i}\sum_x\prod_{i=1}^{d}\chi_\alpha(x)\chi_\beta(x)\qquad\text{where }\beta_i=n_i\bmod 2\\
&= \sum_{\substack{n_1+\cdots+n_d=c\\0\le n_1,\ldots,n_d\le c}}\binom{c}{n_1,\ldots,n_d}\prod_{i=1}^{d}p_i^{n_i}\sum_x\prod_{i=1}^{d}\chi_{\alpha+\beta}(x).
\end{aligned}$$

The innermost sum is zero if $\alpha+\beta\neq 0^d$, i.e., if $\alpha\neq\beta$, and equals 2^d otherwise. Thus $\beta_i=\alpha_i=1$ for every i, that is, $n_i\equiv 1 \pmod 2$ for every i. If $c<d$, this is impossible, since $\sum_i n_i=c$. If $c=d$,

then the only way this can happen is if $n_i = 1$ for all i; otherwise, for some i, $n_i > 1$, and since $\sum_i n_i = d$, some $n_i = 0$. When $n_i = 1$ for all i, it is easy to see that we have $2^d \widehat{f}_\alpha = 2^d d! \prod_{i=1}^d p_i$. ■