Self-Testing/Correcting for Polynomials and for Approximate Functions

Peter Gemmell * Richard Lipton † Ronitt Rubinfeld ‡
Madhu Sudan § Avi Wigderson ¶

Abstract

The study of self-testing/correcting programs was introduced in [8] in order to allow one to use program P to compute function f without trusting that P works correctly. A self-tester for f estimates the fraction of x for which $P(x) = f(x)$; and a self-corrector for f takes a program that is correct on most inputs and turns it into a program that is correct on every input with high probability [1]. Both access P only as a black-box and in some precise way are not allowed to compute the function f.

Self-correcting is usually easy when the function has the random self-reducibility property. One class of such functions that has this property is the class of multivariate polynomials over finite fields [4] [12]. We extend this result in two directions. First, we show that polynomials are random self-reducible over more general domains: specifically, over the rationals and over noncommutative rings. Second, we show that one can get self-correctors even when the program satisfies weaker conditions, i.e. when the program has more errors, or when the program behaves in a more adversarial manner by changing the function it computes between successive calls.

Self-testing is a much harder task. Previously it was known how to self-test for a few special examples of functions, such as the class of linear functions. We show that one can self-test the whole class of polynomial functions over Z_p for prime p.

We initiate the study of self-testing (and self-correcting) programs which only approximately compute f. This setting captures in particular the digital computation of real valued functions. We present a rigorous framework and obtain the first results in this area: namely that the class of linear functions, the log function and floating point exponentiation can be self-tested. All of the above functions also have self-correctors.

*U.C. Berkeley. Supported by NSF Grant No. CCR 88-13632

†Princeton University. Supported in part by NSF Grant No. DCR-8420948.

‡Princeton University. Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), NSF-STC88-09648.

§U.C. Berkeley. Supported in part by NSF Grant No. CCR 88-96202. Part of this work was done while this author was visiting IBM Almaden.

¶Hebrew University and Princeton University. Partially supported by the Wolfson Research Awards administered by the Israel Academy of Sciences and Humanities.

[1] [12] independently introduces a notion which is essentially equivalent to self-correcting.

1 Introduction

Suppose someone gives us an extremely fast program P that we can call as a black box to compute a function f. Rather than trust that P works correctly, a *self-testing program* for f ([8]) verifies that program P is correct on most inputs, and a *self-correcting program* ([8] [12]) for f takes a program P that is correct on most inputs and uses it to compute f correctly on every input (with high probability). Self-testing/correcting is an extension of program result checking as defined in [5],[6], and if f has a self-tester and a self-corrector, then f has a program result checker.

To make this somewhat more precise, consider a function $P : X \to Y$ that attempts to compute f. We consider two domains, the "test" domain $D_t \subset X$ and the "safe" domain $D_s \subset X$ (usually $D_t = D_s$). We say that program P *ϵ-computes* f on D_t if $\Pr_{x \in D_t}[P(x) = f(x)] > 1 - \epsilon$. An (ϵ_1, ϵ_2)-*self-tester* $(0 \leq \epsilon_1 < \epsilon_2)$ for f on D_t must fail any program that does not ϵ_2-compute f on D_t, and must pass any program that ϵ_1-computes f on D_t (note that the behavior of the tester is not specified for all programs). The tester should satisfy these conditions with error probability at most β, where β is a confidence parameter input by the user. An *ϵ-self-corrector* for f on (D_t, D_s) is a program C that uses P as a black box, such that for every $x \in D_s$, $\Pr[C^P(x) = f(x)] \geq 2/3$, [2] for *every* P which ϵ-computes f on D_t. Furthermore, all require only a small multiplicative overhead over the running time of P and are different, simpler and faster than any correct program for f in a precise sense defined in [6].

Section 2 is devoted to self-correcting polynomial functions, and Section 3 is devoted to the self-testing of polynomial functions. Section 4 introduces self-testing/correcting and checking for programs which only approximately compute f, including the digital computation of real valued functions.

2 Self-Correcting Polynomials

In addition to the aforementioned application, self-correcting is interesting because of its independent applications to complexity theory: (1) The existence of a self-corrector for a function implies that the function has similar average and worst case complexities. (2) [4],[12] first observed that low degree polynomials over finite fields have self-correctors. This simple fact turned out to be a key ingredient in two recent results in complexity theory, mainly IP=PSPACE [14][18] and MIP=NEXPTIME [3].

Self-correcting is usually easy when the function has the random self-reducibility property [8][12]. Informally, the property of k-random self-reducibility is that $f(x)$ can be expressed as an easily computable function of $f(x_1), \ldots, f(x_k)$, where the x_i's are each uniformly distributed, and are easy to choose. Several functions have been shown to have this property: examples of such functions are integer and matrix multiplication, modular exponentiation, the mod function and low degree polynomials over finite fields.

Our first aim is to study the scope under which this phenomenon occurs. We extend the above results in various directions, and to this end, we define the following general setting.

Let R be any ring, and $X = \{x_1, x_2, \ldots x_n\}$ a set of indeterminates (we assume nothing about R, in particular the indeterminates may not commute). Let $D_c \subset R$ be the domain of coefficients (often $D_c = R$). Define a R-monomial over D_c to be an arbitrary word $w \in (D_c \cup X)^*$, and its degree, $deg(w)$, the number of occurrences of indeterminates. An R-polynomial f over D_c is simply the sum $f = \sum_i w_i$ of R-monomials, and its degree, $deg(f) = \max_i deg(w_i)$. Denote the set of all degree d polynomials (where $|X| = n$) $R_n^d(D_c)$. If $D_c = R$, we omit D_c from the above definitions, and in particular, the set of all degree d polynomials is R_n^d.

For example, if R is the ring of matrices over some field, $D_s = D_t = D_c = R$, and $A, B, C, D, E, F \in R$, then $f(X_1, X_2) = A + BX_1C + X_2DX_1EX_2 + X_1^2F$ is a polynomial of degree 2 in the two variables X_1, X_2.

If for every $f \in R_n^d(D_c)$ there is an ϵ-self-corrector on (D_t, D_s), we say that $R_n^d(D_c)$ is ϵ-resilient on (D_t, D_s). When $R = D_c = D_s = D_t$, then we say that R_n^d is ϵ-resilient. The interesting issues are of course for what rings and domains this is possible, how small should ϵ be, and what is the complexity of the corrector program C, separating its actual computation and the number of calls to the black box P.

The results of [4] [12] can be compressed to:

Theorem 1 ([4][12]) *If F is a finite field, $|F| > d+1$, then for every n, F_n^d is $\frac{1}{3d+3}$-resilient with $O(d)$ calls to P.*

The self-corrector used to prove this theorem is very simple to implement and is based on the existence of the following interpolation identity relating the function values between points: for all univariate polynomials f of degree at most d, $\forall x, t \in F$, $\sum_{i=0}^{d+1} \alpha_i f(x + a_i \cdot t) = 0$ where the a_i's are distinct elements of F, $\alpha_0 = -1$ and α_i depends only on F, d and not on x, t. [3] Then the self-correcting algorithm for evaluating $f(\bar{x}) = f(x_1, \ldots, x_n)$ is to randomly choose $\bar{t} = (t_1, \ldots, t_n) \in F^n$ with uniform distribution and output $\sum_{i=1}^{d+1} \alpha_i P(\bar{x} + i \cdot \bar{t})$. With probability at least $2/3$, all the calls to the program will return correct values, and the output will be correct.

2.1 New Domains

2.1.1 Non-Commutative Rings

Our first generalization shows, perhaps surprisingly, that the [4][12] trick works in many non-commutative rings as well. This may have complexity theoretic consequences, as many complexity classes are characterized by non-commutative group theoretic problems. Define $C(R)$, the center of the ring R, to be all elements in R which commute with every element. Define R^* to be all the elements in R which have inverses.

[2] this can be amplified to $1-\beta$ by $O(\log 1/\beta)$ independent repetitions and and a majority vote.

[3] In particular, if $F = \mathcal{Z}_p$, a_i can be chosen to be i, and then $\alpha_i = (-1)^{i+1}\binom{d+1}{i}$. Furthermore $\sum_{i=0}^{d+1} \alpha_i f(x + i \cdot t)$ can be computed in $O(d^2)$ time using only additions and comparisons by the method of successive differences in [20].

Theorem 2 *If a random element of R can be uniformly generated, and $|C(R) \cap R^*| \geq d+1$, then for every n, R_n^d is $1/(2d)$-resilient with $O(d)$ calls to P.*

We note that the condition that $|C(R) \cap R^*| \geq d+1$ is satisfied by rich classes of rings, most notably finite group algebras over finite fields of size $\geq d+1$. A special case is the algebra of square matrices of any order over a finite field (here the center is simply all diagonal matrices). Note that here uniform generation is trivial. In general, we know only a few more examples where uniform generation is possible. For any finite group, given by generators, almost uniform generation is possible by the recent Monte Carlo algorithm of [2], but whether this result extends to the group algebra is open.

The proof is based on the fact that such rings can also be shown to have interpolation identities:

Lemma 1 *For every ring R for which $|C(R)| \geq d+1$, there are weights $\alpha_1, \ldots, \alpha_{d+1}$ such that for any $f \in R_1^d$, $f(0) = \sum_{i=1}^{d+1} \alpha_i f(c_i)$ where $c_1, \ldots, c_{d+1} \in C(R)$ and are distinct.*

Proof: We will show that it is possible to select weights so that the identity is true for all monomials $m_j(x) = x a_1 x \ldots x a_j$ of degree j for $0 \leq j \leq d$. The lemma follows by linearity. The constraints on the weights are that for $j = 0, \ldots, d$, $\sum_{i=1}^{d+1} \alpha_i m_j(x) = \delta_j$ where $\delta_0 = 1$ and $\delta_j = 0$ for $j > 0$. Since $c_i \in C(R)$, we have that for all $1 \leq k \leq d+1$, $m_j(c_k) = a_1 a_2 \ldots a_d c_k^j$. Thus this is a linear system of equations in the weights. Since the matrix of the system is a Vandermonde matrix, the system of equations has a solution. ■

If c_i has an inverse, we have the property that for fixed x and uniformly distributed t, then $x + c_i t$ is uniformly distributed. The rest of the proof of the theorem is as in [4][12].

2.1.2 Fixed Point Arithmetic

Our second theorem concerns computation over domains that are not as nice as finite fields. The only similar result we know appears in [3]. Motivated by the fact that every boolean function has a multilinear representation with small integer coefficients, they considered $R = \mathcal{Z}$, $D_c = \mathcal{Z}_{2^n}$, $D_t = D_s = \mathcal{Z}_{10d}$ and showed

Theorem 3 ([3]) *$\mathcal{Z}_n^d(D_c)$ is $\frac{1}{2d}$-resilient on (D_t, D_s) with $O(d)$ calls to P.*

We consider rational domains, specifically binary fixed point arithmetic. Let $FIX_{s,r} = \{\frac{y}{s} : \text{integers } |y| \leq r\}$. We use the notation that $\tilde{x} = (x_1, \ldots, x_n)$.

A nice property of finite fields that is used to get self-correctors for polynomials over finite fields is that any fixed element of the field, when multiplied by a random uniformly distributed element of the field, gives a result that is uniformly distributed over the field. This is not a property of the fixed point domain (fixed point domains do not even have the property of "wraparound"). Though the proof of the self-corrector has the same spirit as the previous one, it is technically more involved. Here we use the generalization of the definitions of self-correcting given in [8],[12] and assume that more than one domain is tested. In addition, this is an example of a class of functions for which the test domains are larger than the safe domain: in order to get resilience on a particular precision, one should use a program that is known to be usually correct both on a slightly finer precision and over a larger range.

We have the polynomial interpolation formula $f(y) = \sum_{i=1}^{d+1} \alpha_i f(y + a_i t)$ where $a_1, \ldots, a_d$ are distinct integers and $\alpha_1, \ldots, \alpha_{d+1}$ depend only on d and not on y or t. Let $A = \operatorname{lcm}(a_1, \ldots, a_d)$, and $A^{(i)} = \frac{A}{a_i}$. Let $L = 10npd^{d+1}$, $R = FIX_{s,L}^n$, $D_s = FIX_{s,p}^n$. Then the $d+1$ test domains are $D_{t_i} = \{\frac{a_i}{A}\tilde{y} | \tilde{y} \in FIX_{s,L}^n\}$ $(1 \leq i \leq d+1)$, where the precision on the inputs is finer than that of the safe domain.

The self-corrector uses the interpolation formula over the new domain to compute $f(x)$ by picking a random $\tilde{t} \in FIX_{s,L}^n$. The domains of the $d+1$ calls to P are dependent on the input and are $D_{q_i} = \{\tilde{x} + \frac{a_i}{A}\tilde{y} | \tilde{y} \in FIX_{s,L}^n\}$ $(1 \leq i \leq d+1)$. We show that for all $1 \leq i \leq d+1$, D_{t_i} and D_{q_i} are very close (considered as uniform distributions on their respective domains), so the program returns correct answers on all inputs with probability at least $3/4$. When this happens, $f(\tilde{x})$ is computed correctly.

Theorem 4 *$R_n^d(D_c)$ is $\frac{1}{8(d+1)}$-resilient on $(D_{t_1}, \ldots, D_{t_{d+1}}, D_s)$.*

DEFINITION 2.1

$$\delta(x, y) = \begin{cases} x & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

DEFINITION 2.2 *For random variables $\bar{X}, \bar{Y}$ over domain D, $\delta(\bar{X}, \bar{Y}) = \sum_{s \in D} \delta(\Pr[\bar{X} = s], \Pr[\bar{Y} = s])$ (note that $0 \leq \delta(\bar{X}, \bar{Y}) \leq 1$).*

Lemma 2 *If $\delta(\bar{X}, \bar{Y}) \geq 1 - \epsilon_1$ and $\Pr[\bar{X} \in S] \geq 1 - \epsilon_2$ then $\Pr[\bar{Y} \in S] \geq 1 - \epsilon_1 - \epsilon_2$.*

Proof: Assume $\Pr[\bar{Y} \in S] \leq 1 - \epsilon_1 - \epsilon_2$ and $\Pr[\bar{X} \in S] \geq 1 - \epsilon_2$. Then $\sum_{s \in S} \Pr[\bar{X} = s] - \delta(\Pr[\bar{X} = s], \Pr[\bar{Y} = s]) \geq \epsilon_1$. Since $\sum_{s \notin S} \Pr[\bar{X} = s] - \delta(\Pr[\bar{X} = s], \Pr[\bar{Y} = s]) \geq 0$, we have $\delta(\bar{X}, \bar{Y}) \leq 1 - \epsilon_1$. ■

Lemma 3 *For $1 \leq i \leq d$, $\delta(D_{t_i}, D_{q_i}) \geq 1 - \frac{1}{8(d+1)}$*

Proof: For all $1 \leq j \leq n$, we have that:
$\Pr[\frac{x_j}{s} + \frac{a_i u_j^i}{As} = \frac{y_j}{A^{(i)}s}] = \Pr[x_j A^{(i)} + u_j^i = y_j]$

$$= \begin{array}{ll} \frac{1}{2L+1} & \text{if } y_j \in [-L + x_j A^{(i)}, +L + x_j A^{(i)}] \\ 0 & \text{otherwise} \end{array}$$

$\Pr[\frac{a_i u_j^i}{As} = \frac{y_j}{A^{(i)}s}] = \Pr[u_j^i = y_j]$

$$= \begin{array}{ll} \frac{1}{2L+1} & \text{if } y_j \in [-L, +L] \\ 0 & \text{otherwise} \end{array}$$

Thus $\Pr_{\tilde{x} \in D_{q_i}}[\tilde{x} = \tilde{y}] = (\frac{1}{2L+1})^n$ if $y_i \in [-L + x_i A^{(i)}, L + x_i A^{(i)}]$ for all $i = 1, \ldots, n$ and $\Pr_{\tilde{x} \in D_{t_i}}[\tilde{x} = \tilde{y}] = (\frac{1}{2L+1})^n$ if $y_i \in [-L, L]$ for all $i = 1, \ldots, n$. The claim follows by the choice of L. ■

2.2 Programs with Weaker Constraints

2.2.1 Resiliency

Note that the resiliency in Theorem 1 degrades with the degree. We show that using the theory of error correcting codes [17], constant resiliency can be achieved. This theorem was proved independently, using similar techniques by Don Coppersmith [11].

Theorem 5 *If F is a finite field, and for some m, $3d+1 < m$, m divides $|F| - 1$, then for every n, F_n^d is 1/9-resilient with $O(m)$ calls to P.*

Proof: (sketch) Let ω be a primitive m^{th} root of unity in F. The queries that C makes are $\{P(x + \omega^i r)\}_{i=0}^{m-1}$, This sequence differs from $\{f(x + \omega^i r)\}_{i=0}^{m-1}$ in at most $m/3$ places with probability $\geq 2/3$. As the second sequence is a codeword in the generalized BCH code [17], it can be recovered from the first efficiently, and then $C(x)$ can be computed again by interpolation. Note here that there are m (usually $m = O(d)$), queries to P, but as the error correction procedure requires linear algebra, the running time of C is $O(d^3)$. ■

We can also generalize this to some non-commutative rings:

Theorem 6 *If R is a finite group algebra over a finite field F, satisfying the conditions of Theorem 5, and we can uniformly sample from R, then for every n, R_n^d is 1/9-resilient.*

We do not know how to increase the resiliency over fixed point domains.

2.2.2 Memory Bounded Programs

Until now, we have assumed that the program P is a function, i.e. a non-adaptive memoryless object. Here we address the case where the program is not a function, but an adversary A. Under the assumption that there are more than one independent copies of the program (non-communicating adversaries), [7] show how to get checkers for any function that has a checker in the original model. Here we address the case of a single adversary A whose only limitation is space. The adversary can apply different programs depending on his memory state, which can be affected by previous queries. Formally, a space s adversary A (over R) is an automaton with 2^s states where state i is labeled by $P_i : R^n \to R$, and the transition alphabet is R^n. When at state i, given input $z \in R^n$, A returns $P_i(z)$ and moves to a new state j along the unique transition labeled z. We say that A ϵ-computes $f \in R_n^d$ if for every i, P_i ϵ-computes f. R_n^d is (s, ϵ)-resilient if there is a program C such that $\Pr[C^A(x) = f(x)] \geq 2/3$ for all $x \in R^n$ and for all space s adversaries A.

There were no previous results of (s, ϵ)-resiliency, even for 1-bit memory $s = 1$. We give an algorithm C, for the case of finite fields, that is $(\theta(\log|F|), \frac{1}{16d^2})$-resilient.

Theorem 7 *If $|F| > d^8$, then for every n, F_n^d is $(\frac{1}{4}\log|F|, 1/(16d^2))$-resilient with $O(d^2)$ calls to P.*

As mentioned above, for the algorithm C we give, this is optimal: it fails if $s \geq 4\log|F|$. The algorithm queries A on points $\{x + ir\}$ with $r \in_R F^n$ *and* $i \in_R F$. We use the fact that the pair (x, r) acts on i as a 2-universal hash function, and the small information A has on (x, r) enables the use of results on the distribution of hash values by [16][15] to show that the queries are fairly uniformly distributed in F^n.

We observe that this implies that bigger fields are better. When the choice of a field is ours, we have:

Corollary 4 *In polynomial time, we can achieve resiliency to any adversary that is polynomial space bounded!*

Proof: [of Theorem 7]

Let $H = \{h : U \to Z\}$ be a family of universal hash functions [9] from a domain U to range Z. This means that for any $u, u' \in U$, $z, z' \in Z$ and random $h \in_R H$, $\Pr[h(u) = z, h(u') = z' = 1/|Z|^2$.

Let s denote space, and $m = 2^s$. We shall describe a space s "tree automaton" which is stronger than the above adversary, and with which it will be convenient to work. Schematically, it will look like a $|U|$-ary tree, in which every path is a width m branching program. Thus every node (labeled by a word from U^*) will have m states, and will have a separate transition function (with alphabet Z) to each child.

Formally, for every word $w \in U^*$ and $u \in U$ let $\delta_{w,u} : [m] \times Z \to [m]$ be an arbitrary (transition) function.

Let $P^{\emptyset}$ be an arbitrary partition of H into m parts, $S_1^{\emptyset}, S_2^{\emptyset}, \cdots S_m^{\emptyset}$. This partition and the transition functions naturally define partitions $P^w = \{S_1^w, S_2^w, \cdots S_m^w\}$ for every $w \in U^*$ inductively as follows. For any $u \in U$, h is in S_j^{wu} iff $h \in S_i^w$ and $\delta_{w,u}(i, h(u)) = j$. For convenience, we denote by P^w the nodes of the tree as well as the partition of H in that node. The parts of this partition are the states of that node.

Let $S^w(h)$ denote the part (state) containing h in the partition (node) P^w. If h is chosen uniformly at random from H, then clearly its state $S^{\emptyset}(h)$ is chosen proportional to its cardinality, and within it h is uniformly distributed. It is easy to prove inductively that for every word w, the state $S^w(h)$ is reached by h on w with probability proportional to its cardinality, and within it h is uniformly distributed.

We now show that under this distribution (i.e. knowing $S^w(h)$), $h(u)$ remains essentially uniformly distributed for random $u \in_R U$.

Claim: Let P be any partition of H into m parts $S_1, S_2, \cdots S_m$. Let $B_1, B_2, \cdots B_m$ be arbitrary subsets of Z with $|B_i|/|Z| < \epsilon$, and $B(h) = B_i$ for the same i satisfying $S(h) = S_i$. Then, $\Pr[h(u) \in B(h) | S(h)] \leq \epsilon + 2|U|^{-1/4}$.

Proof: We use Lemma 10 of [15] and simple algebra to get:

$$
\begin{aligned}
& \Pr[h(u) \in B(h) | S(h)] \\
&= \sum_{i=1}^{m} (|S_i|/|U|) \Pr[h(u) \in B_i | S(h) = S_i] \\
&\leq (1/m) + \max_{\{i : |S_i| m^2 \geq |U|\}} \Pr[h(u) \in B_i] \\
&\leq |U|^{-1/4} + \epsilon + |U|^{-1/4} \\
&\leq \epsilon + 2|U|^{-1/4}
\end{aligned}
$$

■

We used $m = |U|^{1/4}$ and the [15] lemma for a subset of hash functions of density at least $|U|^{-1/2}$.

Now we can fit our algorithm and the space s bounded adversary into this framework. Our domain U is the field F, and pairs $x, r \in F^n$ map field elements $u \in F$ to $Z = F^n$ by $(x, r)(u) = x + ur$, which is universal hashing.

We want to evaluate $f(x)$, and let us assume for a moment that x is random. We pick a random $r \in F^n$, and a random sequence of field elements $u_1, u_2, \cdots$. We request from the adversary the value of $f(x + u_i r)$ allowing him to know and remember for free the value of u_i. This creates the tree structure above and guarantees that the states represent only information about the pair (x, r). We continue until we ask $d + 1$ distinct questions (the expected time until this happens is $O(d)$ as the field is large). Since it was ϵ-resilient, at every state it can err only on a subset of density ϵ of the queries $h(d)$. These are exactly the conditions in the claim which guarantee that the probability of a wrong answer is at most 2ϵ. Hence $\epsilon < 1/(4d)$ guarantees that with high probability all replies are correct and we can interpolate safely.

For the general case x is fixed and not random. We handle this by repeating the above procedure for $x + it$, with $t \in_R F^n$ and $i = 1, 2, \cdots d + 1$, from which f values we can interpolate $f(x)$. We use the same random r in all procedures. Each $(x + it, r)$ is a random pair, so each single procedure fails with probability at most $1/4d$ when $\epsilon < 1/16d^2$. To see that the correlation between different pairs cannot help the adversary, note that it was allowed to start with an arbitrary partition $P^{\emptyset}$. Hence, by telling the adversary when we are done with $(x + it, r)$ and start with $(x + (i+1)t, r)$, we let him convert in an arbitrary way his information (partition) on the old pair into another about the new pair. The details are left to the reader. ■

3 Self-Testing Polynomials

Self-testing is a much harder task than self-correcting. In [8] it is shown how to get self-testers for the two special cases of any function that is downward self-reducible in addition to being random self-reducible, and any function that is linear. In this section we show that the much richer class of all polynomial functions can be self-tested.

As in [8], our testers are of a nontraditional form: the tester is given a short specification of the function in the form of properties that the function must have, and verifies that these properties "usually" hold. We show that these properties are such that if the program "usually" satisfies these properties, then it is essentially computing the correct function.

Complementing the results of [4][12], we first show how to construct a self-tester for any univariate polynomial function provided that the value of the function is known on at least one more than the degree number of points. These results extend to give self-testers for polynomial functions of degree d with m variables, provided the function value is known on at least $(d+1)^m$ well chosen points, by combining the univariate result with the results of [3][13][19]. In the final version we show that the results can also be generalized to give self-testers and self-correctors for functions in finite dimensional function spaces that are closed under shifting and scaling.

Theorem 8 *If f is a degree d (univariate) polynomial over $\mathcal{Z}_p$, then f has an $(\frac{\epsilon}{2(d+2)}, 4\epsilon)$-self-tester on $\mathcal{Z}_p$ with $O(d \cdot \min(d^2, \frac{1}{\epsilon}))$ calls to P.*

Corollary 5 *If f is a degree d polynomial in m variables over $\mathcal{Z}_p$, then f has a $(0,\epsilon)$-self-tester on $\mathcal{Z}_p$ with $O((d+1)^m/\epsilon + poly(d,m,\frac{1}{\epsilon}))$ calls to P.*

The self-tester algorithm is almost as simple as the self-corrector algorithm, though the proof of correctness is more difficult. We present the proof of Theorem 8 at the end of this section. The self-testing is done in two phases, one verifying that the program is essentially computing a degree d polynomial function, and the other verifying that the program is computing the correct polynomial function.

When the number of variables is small, the provision that the value of the function is known on at least $(d+1)^m$ points is not very restrictive since the degree is assumed to be small with respect to the size of the field: Suppose one has a program for the RSA function $x^3 \bmod p$. Traditional testing requires that the tester know the value of $f(x)$ for random values of x. Here one only needs to know the following simple and easy to generate specification: f is a degree 3 polynomial in one variable, and $f(0)=0, f(1)=1, f(-1)=-1, f(2)=8$. These function values are the same over any field of size at least 9.

In [3][13][19], there are tests, which in our setting allows the self-tester to be convinced that the program is computing a multivariate polynomial function of low degree in polynomial time. Although the test is in polynomial time, it is somewhat complicated to perform because it involves the reconstruction of a univariate polynomial given its values at a number of points (which in turn requires multiplications and matrix inversions), and later the evaluation of the reconstructed polynomial at random points. If the number of variables in the original multivariate polynomial is relatively small, this is more difficult than computing the polynomial function from scratch, and therefore is not different than the program in the sense defined by [8].

We now present the proof of Theorem 8 for the special case of univariate polynomials of degree d over Z_p and when $\epsilon \le O(1/d^2)$.

For simplicity, in the description of our self-testing program, we assume that whenever the self-tester makes a call to P, it verifies that the answer returned by P is in the proper range, and if the answer is not in the proper range, then the program notes that there is an error.

We use $x \in_R Z_p$ to denote that x is chosen uniformly at random in Z_p.

```
program Polynomial-Self-Test(P, ε, β,
              (x_1, f(x_1)), ..., (x_{d+1}, f(x_{d+1})))

Membership Test
   Repeat O((1/ε) log(1/β)) times
      Pick x, t ∈_R Z_p and test that
         Σ_{i=0}^{d+1} α_i P(x + i * t) = 0
   Reject P if the test fails more than
      an ε fraction of the time.

Consistency Test
   for j going from 0 to d do
      Repeat O(log(d/β)) times
         Pick t ∈_R Z_p and test that
            f(x_j) = Σ_{i=1}^{d+1} α_i P(x_j + i * t).
      Reject P if the test fails more
         than 1/4th of the time.
```

Let $\delta \equiv \Pr_{x,t}[\sum_{i=0}^{d+1} \alpha_i P(x+i*t) = 0]$

DEFINITION 3.1 *We say program P is ϵ-good if $\delta \le \frac{\epsilon}{2}$ and $\forall j \in \{0,\cdots,d\}$, $\Pr_t[f(x_j) = \sum_{i=1}^{d+1} \alpha_i P(x_j+i*t)] \ge 3/4$. We say P is ϵ-bad if either $\delta > 2\epsilon$ or if $\exists j$ such that $\Pr_t[f(x_j) = \sum_{i=1}^{d+1} \alpha_i P(x_j+i*t)] < 1/2$. (Note that there are programs which are neither ϵ-good or ϵ-bad).*

The following lemma is easy to prove :

Lemma 6 *With probability at least $1-\beta$ a ϵ-good program is passed by Polynomial-Self-Test. With probability at least $1-\beta$ a ϵ-bad program is rejected by Polynomial-Self-Test.*

It is easy to see that if a program P $\frac{\epsilon}{2(d+2)}$-computes f, then it is ϵ-good. The hard part of the theorem is to show that if program P does not ϵ-compute f then it is ϵ-bad. We show the contrapositive, i.e. that if P is not ϵ-bad, then it 4ϵ-computes f.

If P is not ϵ-bad, then $\delta \le 2\epsilon$. Under this assumption, we show that there exists a function g with the following properties:

1. $g(x) = P(x)$ for most x.
2. $\forall x,t \ \sum_{i=0}^{d+1} \alpha_i g(x+it) = 0$, and thus g is a degree d polynomial.
3. $g(x_j) = f(x_j)$ for $j \in \{0,1,\cdots,d\}$.

Define $g(x)$ to be $\mathrm{maj}_{t\in Z_p} \sum_{i=1}^{d+1} \alpha_i P(x+it)$.

Lemma 7 *g and P agree on more than $1-2\delta$ fraction of the inputs from Z_p.*

Proof: Consider the set of elements x such that $\Pr_t[P(x) = \sum_{i=1}^{d+1} \alpha_i P(x+i*t)] < 1/2$. If the fraction of such elements is more than 2δ then it contradicts the condition that $\Pr_{x,t}[\sum_{i=0}^{d+1} \alpha_i P(x+i*t)=0] = \delta$. For all remaining elements, $P(x) = g(x)$. ∎

In the following lemmas, we show that the function g satisfies the interpolation formula for all x, t and is therefore a degree d polynomial. We do this by first showing that for all x, $g(x)$ is equal to the interpolation of P at x by most offsets t. We then use this to show that the interpolation formula is satisfied by g for all x, t.

Lemma 8 *For all $x \in Z_p$, $\Pr_t\left[g(x) = \sum_{i=1}^{d+1} \alpha_i P(x+i*t)\right] \geq 1-2(d+1)\delta$.*

Proof: Observe that $t_1, t_2 \in_R Z_p$ implies

$$x + i * t_1 \in_R Z_p \text{ and } x + j * t_2 \in_R Z_p$$

$$\Rightarrow \Pr_{t_1,t_2}[P(x+i*t_1) = \sum_{j=1}^{d+1} \alpha_j P(x+i*t_1+j*t_2)] \geq 1-\delta$$

$$\Rightarrow \Pr_{t_1,t_2}[P(x+j*t_2) = \sum_{i=1}^{d+1} \alpha_i P(x+i*t_1+j*t_2)] \geq 1-\delta$$

Combining the two we get

$$\begin{aligned} \Pr_{t_1,t_2}\Big[& \sum_{i=1}^{d+1} \alpha_i P(x+i*t_1) \\ = & \sum_{i=1}^{d+1}\sum_{j=1}^{d+1} \alpha_i\alpha_j P(x+i*t_1+j*t_2) \\ = & \sum_{j=1}^{d+1} \alpha_i P(x+j*t_1)\Big] \geq 1-2(d+1)\delta \end{aligned}$$

The lemma now follows from the observation that the probability that the same object is drawn twice from a set in two independent trials lower bounds the probability of drawing the most likely object in one trial. (Suppose the objects are ordered so that p_i is the probability of drawing object i, and $p_1 \geq p_2 \geq \ldots$. Then the probability of drawing the same object twice is $\sum_i p_i^2 \leq \sum_i p_1 p_i = p_1$.) ∎

Lemma 9 *For all $x, t \in Z_p$, if $\delta \leq \frac{1}{2(d+2)^2}$, then $\sum_{i=0}^{d+1} \alpha_i g(x+i*t) = 0$ (and thus g is a degree d polynomial [20]).*

Proof: Observe that, since $t_1 + it_2 \in_R Z_p$, we have for all $0 \leq i \leq d+1$

$$\Pr_{t_1,t_2}\left[g(x+i*t) = \sum_{j=1}^{d+1} \alpha_j P((x+i*t)+j*(t_1+it_2))\right]$$

$$\geq 1-2(d+1)\delta.$$

Furthermore, we have for all $1 \leq j \leq d+1$

$$\Pr_{t_1,t_2}\left[\sum_{i=0}^{d+1} \alpha_j P((x+j*t_1)+i*(t+j*t_2)) = 0\right] \geq 1-\delta$$

Putting these two together we get
$\Pr_{t_1,t_2}\left[\sum_{i=0}^{d+1} \alpha_i g(x+i*t) = \right.$
$\left.\sum_{j=1}^{d+1} \alpha_j \sum_{i=0}^{d+1} \alpha_i P((x+j*t_1)+i*(t+j*t_2)) = 0\right] > 0.$

The lemma follows since the statement in the lemma is independent of t_1, t_2, and therefore if its probability is positive, it must be 1. ∎

Lemma 10 $g(x_j) = f(x_j)$

Proof: Follows from the definition of g and the fact that P is not ϵ-bad. ∎

Theorem 9 *The program* **Polynomial-Self-Test** *is a $(\frac{\epsilon}{2(d+2)}, 4\epsilon)$-self-testing program for any degree d polynomial function over Z_p specified by its values at any $d+1$ points, if $\epsilon \leq \frac{1}{4(d+2)^2}$.*

Proof: Follows from Lemmas 6,9, and 10. ∎

4 Approximate and Real-Valued Functions

In the notions of a result checker and self-testing/correcting pair considered so far, a result of a program is considered incorrect if it is not *exactly* equal to the function value. Some programs are designed only to correctly *approximate* the value of a function. One important setting where this is the case is with functions dealing with real inputs and outputs: due to the finiteness of the representation, the program computes a function which is only an approximation to the desired function. Another example of this is the quotient function $\hat{f}(x, R) = x \operatorname{div} R$, a commonly used system function, which can be thought of as an approximation to the integer division function $f(x, R) = (x \text{ div } R, x \bmod R)$.

Below we give a formal framework to study checkers, self-testers and self-correctors in this setting. We give a general technique for self-correcting in the case that the function is approximately linear, and apply it to the quotient function, the floating point exponentiation function and the logarithm function. We assume that there is a well-defined metric space as the range of f.

We use $a \approx_\zeta b$ to denote that $|a-b| \leq \zeta$. We say that g *δ-approximates* f on D_t if for all $x \in D_t$, $g(x) \approx_\delta f(x)$. We say that P *(ϵ, δ)-approximately computes* f on D_t if there is a g such that P ϵ-computes g on D_t and g δ-approximates f on D_t.

An *approximate self-tester* verifies that that the program correctly approximates the function on most inputs: An $(\epsilon_1, \epsilon_2, \delta_1, \delta_2)$-*approximate self-tester* $(0 \leq \epsilon_1 < \epsilon_2, 0 \leq \delta_1 < \delta_2)$ must fail any program that does not (ϵ_2, δ_2)-approximately compute f on D_t, and must pass any program that (ϵ_1, δ_1)-computes f on D_t. The error probability of the self-tester should be at most β.

An *approximate self-corrector* for f takes a program P that approximately computes f on most inputs and uses it to approximately compute f on all inputs. More formally, an $(\epsilon, \delta, \delta')$-*approximate self-corrector* is a program C that uses P as a black box, such that for every $x \in D_s$, $\Pr[C^P(x) \approx_{\delta'} f(x)] \geq 2/3$, for *every* P which (ϵ, δ)-approximately computes f on D_t.

An analogous definition of an *approximate checker* can be made. An approximate result checker checks that the program correctly approximates the function on a particular input.

Intuitively, we say that a function is (k, δ_1, δ_2)-*approximate-random self-reducible* if the function value at a particular point can be approximated to within δ_2 given approximations to within δ_1 of the function at k points, where the k points are uniformly distributed and easy to choose. Regarding when we can self-correct, we have the following theorem:

Theorem 10 *If f is (k, δ_1, δ_2)-approximate-random self-reducible, then f has an $(\frac{1}{4k}, \delta_1, \delta_2)$-approximate self-corrector.*

The proof of this theorem is very similar to the proof that any random self-reducible function has a self-corrector, except that instead of taking the majority answer, the approximate self-corrector must take the "median" answer (the median is not well-defined over finite groups, but an appropriate candidate can be chosen).

4.1 Approximate Self-Testers for Linear Functions

We make the following definition:

Definition 4.1 *A function f from a group G to a group H is* Δ-approximately linear *if $\forall a, b \in G$, $f(a+b) \approx_\Delta f(a) + f(b) + E(a,b)$, where $E(a,b)$ is some easily computable function from $G \times G$ to H.* We call a function linear if it is 0-approximately linear. Notice that our definition of a linear function encompasses a much larger class of functions than the traditional definition.

In this section we assume that we have a linear function f mapping from Z_m to Z_n or Z, given by $f(0) = 0$, $f(1)$ and $f(a)$, where a is close to $\sqrt{m}$. (the property required of a is that any $x \in Z_m$ should be expressible as $b * a + c$ where $b + c \leq 3\sqrt{m}$). We show that the following self-tester tests $f(x) \approx_\Delta P(x)$.

```
Program Approximate-Self-Test(P, Δ, β, ε)

Approximate Linearity Test
   Repeat O((1/ε) log (1/β)) times
      Pick random x, y and test that
         P(x + y) ≈_Δ P(x) + P(y) + E(x, y)
   Reject P if the test fails more than
         ε/2 fraction of the time.

Neighborhood Tests
   Repeat O(log (1/β)) times
      Pick random x and test that
         P(x + 1) ≈_Δ P(x) + f(1) + E(x, 1)
         P(x + a) ≈_Δ P(x) + f(a) + E(x, a)
   Reject P if test fails more than
         1/4th of the time.
```

Definition 4.2 *We say that P is (ϵ, Δ)-good if: $\Pr_{x,y}[P(x+y) \approx_\Delta P(x) + P(y) + E(x,y)] > 1 - \epsilon$ and, if $R = Z_n$, $\Pr_x[P(x+1) \approx_\Delta P(x) + f(1) + E(x,1)] > \frac{3}{4}$ and $\Pr_x[P(x+a) \approx_\Delta P(x) + f(a) + E(x,a)] > \frac{3}{4}$*

The following lemma establishes confidence in the self-tester and can be proved using Chernoff bounds.

Lemma 11 *If P is not (ϵ, Δ)-good then the probability that P passes the basic linear test is less than β. If P is $(\frac{\epsilon}{4}, \Delta)$-good then the probability that P passes the self-test is at least $1 - \beta$.*

It is clear that if a program $(\frac{\epsilon}{12}, \Delta/3)$-approximately computes f then it is $(\frac{\epsilon}{4}, \Delta)$-good. The other direction is harder to prove. From this point on we assume that the program P is (ϵ, Δ)-good.

We define the discrepancy $disc(x) = f(x) - P(x)$. It is cleaner to work with the discrepancy of x because it allows us to ignore the effect of $E(x,y)$. We have:

$$\Pr_{x,y}[disc(x+y) \approx_\Delta disc(x) + disc(y)] > 1 - \epsilon$$

$$\Pr_x[disc(x+1) \approx_\Delta disc(x)] \geq \frac{3}{4}$$

$$\Pr_x[disc(x+a) \approx_\Delta disc(x)] \geq \frac{3}{4}$$

The rest of this section shows that $disc(x)$ is always approximately 0. This is achieved by first showing that changing the value of $disc(x)$ at only a few places gives a function h which always passes the approximate linearity test (Lemma 12). Lemma 13 shows that h is bounded well away from n. This is used in lemma 14 to show that h is actually very close to 0 everywhere, implying in turn that $disc$ is within 6Δ of 0 at most points. Finally lemma 15 shows that for most points the discrepancy is actually much smaller than 6Δ.

Lemma 12 *There exists a function h with the following properties:*

1. $\forall x, y \in Z_m, h(x+y) \approx_{6\Delta} h(x) + h(y)$

2. $\Pr_x[h(x) = disc(x)] > 1 - 2\sqrt{\epsilon}$

Proof: Let $A = \{y \in Z_m | \Pr_x[disc(x+y) \approx_\Delta disc(x) + disc(y)] > \sqrt{\epsilon}\}$ be the set of "good" values of $y \in Z_m$. For $y \in A$ define $h(y) = disc(y)$. For other values of y, pick $x, z \in A$ such that $x+z = y$ and define $h(y) = h(x)+h(z)$. It can be shown, by techniques similar to those in [8], that h so defined has the property that $\forall y, \Pr_x[disc(y+x) \approx_{2\Delta} disc(x) + h(y)] > 1 - 2\sqrt{\epsilon}$. This in turn can be used to show that h has the approximate linearity property, i.e., $h(x+y) \approx_{6\Delta} h(x) + h(y)$. The fact that the cardinality of A is large completes the proof of this lemma. ■

Lemma 13 $\forall x, h(x) \approx_{27\sqrt{m}\Delta} 0$

Proof: Since P is (ϵ, Δ)-good, $h(1) \approx_{3\Delta} 0$ and $h(a) \approx_{3\Delta} 0$. Lemma 12 now implies that for all $x \in Z_m$, $h(x+1) \approx_{9\Delta} h(x)$ and $h(x+a) \approx_{9\Delta} h(x)$. The fact that any $x \in Z_m$ can be expressed as $b*a+c$, where $b+c \leq 3\sqrt{m}$ completes the proof. ■

Lemma 14 *If* $27\sqrt{m}\Delta < n/4$ *then* $\forall x, h(x) \approx_{6\Delta} 0$ *and* $\Pr_x[disc(x) \approx_{6\Delta} 0] > 1 - \sqrt{\epsilon}$

Proof: Assume that $\exists x$ such that $dist(h(x), 0) > 6\Delta$. Pick x such that $dist(h(x), 0)$ is maximum. Since $h(x) \approx_{27\sqrt{m}\Delta} 0$ we have $dist(h(2x), 0) \geq 2dist(h(x), 0) - 6\Delta > dist(h(x), 0)$ which is a contradiction.

The second part of the assertion follows from the fact that $disc(x)$ equals $h(x)$ with probability $1 - \sqrt{\epsilon}$.

(Notice that to prove this lemma for the case where the range of f is Z, lemma 13 is not required. In turn this implies that the case where range of f is Z does not require any neighborhood tests.) ■

To get even tighter bounds on the discrepancy of most elements we use the following lemma which illustrates an important tradeoff between ϵ and Δ.

Lemma 15 *For all* $k > 0$, $\Pr[disc(x) \approx_{1+\frac{5}{2^k}\Delta} 0] > 1 - (4k+1)\sqrt{\epsilon}$

Proof: Let i be the smallest value of k such that $\Pr[(1+\frac{5}{2^k})\Delta < disc(x) < 6\Delta] > 2k\sqrt{\epsilon}$. Let $S = \{x \in Z_m | (1+\frac{5}{2^k})\Delta < disc(x) < 6\Delta\}$. Consider the set $S' = \{(x,y) | x, y \in S$, and $disc(x) + disc(y) \approx_\Delta disc(x+y)\}$. The size of this set is at least $|S|^2 - \epsilon m^2$. The size of $S'' = \{x+y | x, y \in S'\}$ is at least $(|S|^2 - \epsilon m^2)/|S|$ which is at least $(2i-1)\sqrt{\epsilon}m$. At most $\sqrt{\epsilon}m$ of these elements have $disc$ greater than 6Δ (by lemma 14). Thus at least $2(i-1)\sqrt{\epsilon}m$ elements of S'' satisfy $(1+\frac{5}{2^{(i-1)}})\Delta < disc(x) < 6\Delta$. This violates the minimality of i.

Similarly we have $\Pr[(1+\frac{5}{2^k})\Delta < n - disc(x) < 6\Delta] > 2k\sqrt{\epsilon}$. An application of lemma 14 completes the proof. ■

Thus we have proved:

Theorem 11 *If f is a linear function from Z_m to Z_n and $n > 54\sqrt{m}\Delta$, then f has a $(\epsilon/12, (4k+1)\sqrt{\epsilon}, \Delta/3, (1+\frac{5}{2^k}\Delta))$-approximate self-tester on G.*

It should be noted that, though our proof as presented here requires that $n > 54\sqrt{m}\Delta$, this condition can be easily relaxed by throwing in some more neighborhood tests. For instance, if $n = \Theta(m^{1/d})$, and $f(a_i)$ is known for $i = 1, \ldots, d+1$ (where a_i is close to $m^{i/d+1}$, then for each i we test that $P(x+a_i) \approx_\Delta P(x) + f(a_i) + E(x, a_i)$. These tests suffice to carry out the proof here with little modifications to lemma 13.

4.2 Extending the approximate-linear self-tester to other functions

4.2.1 The Quotient Function

The mod function, $f(x, R) = x \bmod R$, and the division function given by $f(x, R) = (x \text{ div } R, x \bmod R)$, are both linear, but the quotient function $f'(x, R) = x \text{ div } R$ is not. However the quotient function can be thought of as a 1-approximately linear function mapping into Z, because $(x_1 + x_2) \text{ div } R = x_1 \text{ div } R + x_2 \text{ div } R - E(x_1, x_2) + \zeta$, where $\zeta \in \{0, 1\}$, and $E(x_1, x_2) = -((x_1+x_2) \text{ div } 2^n R)2^n$ is an easy to compute function. Hence the approximate-linear self-tester applies to the quotient function and we get the following corollary. *(Note again that since the range of the quotient function is Z, no neighborhood tests are needed.)*

Corollary 16 *The quotient function has a $(\epsilon/16, \sqrt{\epsilon}, 0, 6)$ self-tester over the domain $[0 \ldots 2^n R]$.*

4.2.2 Floating Point Exponentiation

The floating point exponentiation function f computes $f(x) = 2^x$ for inputs from the domain $D_0 = \{\frac{l}{2^k} 2^{exp} | 0 \leq$

$l < 2^k, C_0 < exp < C_1\}$, where $C_0 < C_1$ are constants. (The domain here models a typical floating point word in a computer with l representing the mantissa and exp representing the exponent. Typically C_0 is negative and C_1 is positive).

We first restrict our attention to the exponentiation function working on the domain $D = \{\frac{l}{2^k} | 0 \leq l < 2^k\}$. Observe that D forms a group over modular addition defined as $x +' y = (x+y) mod 1$. Furthermore $f(x +' y) = f(x) * f(y) * E(x,y)$, making f a linear function, where $E(x,y) = 1$ if $x + y < 1$ and $1/2$ otherwise. Thus f can be self-tested over D using our approximate linearity self-tester. (D_0 on the other hand does not form a group, hence our methods are not directly applicable to it.)

Next observe that computing f over D and computing f over D_0 are equivalent in an approximate sense. A program computing f over D_0 also computes f over D. For the other direction, observe that $\forall x \in D_0$, $x \approx_{2^{-(k+1)}} a + \frac{l}{2^k}$ where $a \in Z$. Thus $f(x) \approx_{2^{-k}} 2^a f(\frac{l}{2^k})$. (Here the metric used on the range is $d(x,y) = |\log x - \log y|$.) Thus given a program computing f on the domain D we can get a program computing f in the domain D_0 approximately. The approximate equivalence of the two domains yields the following corollary to theorem 11.

Corollary 17 *The floating point exponentiation function for numbers with k bits of precision has a $(\epsilon/16, \sqrt{\epsilon}, \log(1+\frac{1}{2^{k+1}}), \log(1+\frac{9}{2^k}))$-approximate-self-tester.*

4.2.3 Floating Point Logarithm

The floating point log function f takes an input from the domain $D_0 = \{(1+\frac{l}{2^k})2^{exp} | 1 \leq l < 2^k, -C_0 < exp < C_1\}$ where C_0, C_1 are constants and computes $f(x) = \log x$. In this section we outline a self-tester for the floating point log function.

As in the case of floating point exponentiation, the problem reduces to testing on the smaller domain $D = \{(1+\frac{l}{2^k}) | 1 \leq l < 2^k\}$ (computing $\log 2^{exp}$ is easy). But unlike the floating point exponentiation case there seems to be no obvious group structure on this domain. The natural binary operation over D, given by $a \odot b = a * b$ if $a * b < 2$ and $a * b/2$ otherwise, is not associative. But, it can be shown that it satisfies the following approximate associativity property.

$$(a \odot (b \odot c)) \approx_{\frac{3}{2^k}} ((a \odot b) \odot c)$$

The tester for the log function performs the approximate linearity test and the following neighborhood tests for $j = 1, 2, 3$.

$$\texttt{for random } x \texttt{ test if } P(x \odot \frac{j}{2^k}) \approx_{\frac{1}{2^k}} P(x) + \frac{j}{2^k}$$

The correctness of this tester can be proved now in a manner similar to the proof of theorem 11 using the approximate associativity property instead of associativity. The details are omitted from this version.

Theorem 12 *The floating point logarithm function for numbers with k bits of precision has a $(\epsilon/16, \sqrt{\epsilon}, \frac{1}{2^{k+1}}, \frac{33}{2^k}) - approximate - self - tester$.*

5 Acknowledgments

We wish to thank Gary Miller for pointing out the extension of our initial results to finite-dimensional function spaces. We would also like to thank Manuel Blum for suggesting the logarithm and floating point exponentiation functions as applications of approximate self-testing/correcting, and Mike Luby for several technical suggestions. We would like to thank Shafi Goldwasser, Sampath Kannan and Umesh Vazirani for several interesting and helpful discussions.

References

[1] Babai, L., "Trading Group Theory for Randomness", *Proc. 17th ACM Symposium on Theory of Computing*, 1985, pp. 421-429.

[2] Babai, L., "Local expansion of vertex-transitive graphs and random generation of finite groups", University of Chicago TR90-31, October 16,1990.

[3] Babai, L., Fortnow, L., Lund, C., "Non-Deterministic Exponential Time has Two-Prover Interactive Protocols", *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,*, 1990.

[4] Beaver, D., Feigenbaum, J., "Hiding Instance in Multioracle Queries", *STACS 1990.*

[5] Blum, M., "Designing programs to check their work", Submitted to *CACM.*

[6] Blum, M., Kannan, S., "Program correctness checking ... and the design of programs that check their work", *Proc. 21st ACM Symposium on Theory of Computing*, 1989.

[7] Blum, M., Luby, M., Rubinfeld, R., "Program Result Checking Against Adaptive Programs and in Cryptographic Settings", *DIMACS Workshop on Distributed Computing and Cryptography*, 1989.

[8] Blum, M., Luby, M., Rubinfeld, R., "Self-Testing/Correcting with Applications to Numerical Problems," *Proc. 22th ACM Symposium on Theory of Computing*, 1990.

[9] Carter, L., Wegman, M., "Universal Hash Functions", *Journal of Comp. and Sys. Sci. 18* (1979) 143-154.

[10] Cleve, R., Luby, M.,"A Note on Self-Testing/ Correcting Methods for Trigonometric Functions", International Computer Science Institute Technical Report TR-90-032, July, 1990.

[11] Coppersmith, D., personal communication.

[12] Lipton, R., "New directions in testing", Proceeding of *DIMACS Workshop on Distributed Computing and Cryptography*, 1989.

[13] Lund, C., "The Power of Interaction", University of Chicago Technical Report 91-01, January 14, 1991.

[14] Lund, C., Fortnow, L., Karloff, H., Nisan, N., "Algebraic Methods for Interactive Proof Systems", *Proceedings of the 31st Annual Symposium on Foundations of Computer Science,*, 1990.

[15] Mansour, Y., Nisan, N., Tiwari, P., "The Computational Complexity of Universal Hashing", *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, 1990.

[16] Nisan, N., "Psuedorandom Generators for Space-Bounded Computation", *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, 1990.

[17] Peterson, W.W., Weldon, E.J., *Error Correcting Codes*, MIT Press, Cambridge, Mass.

[18] Shamir, Adi, "IP=PSPACE", *Proceedings of the 31st Annual Symposium on Foundations of Computer Science, 1990.*

[19] Szegedy, Mario, manuscript, January 1991.

[20] Van Der Waerden, B.L., *Algebra*, Vol. 1, Frederick Ungar Publishing Co., Inc., pp. 86-91, 1970.