Self-Testing/Correcting with Applications to Numerical Problems

Manuel Blum *   Michael Luby †   Ronitt Rubinfeld ‡

Abstract

Suppose someone gives us an extremely fast program P that we can call as a black box to compute a function f. Should we trust that P works correctly? A *self-testing/correcting pair* allows us to: (1) estimate the probability that $P(x) \neq f(x)$ when x is randomly chosen; (2) on *any* input x, compute $f(x)$ correctly as long as P is not too faulty on average. Furthermore, both (1) and (2) take time only slightly more than the original running time of P.

We present general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical problems, including integer multiplication, modular multiplication, matrix multiplication, inverting matrices, computing the determinant of a matrix, computing the rank of a matrix, integer division, modular exponentiation and polynomial multiplication.

*Computer Science Division, U.C. Berkeley, Berkeley, California 94720, Supported by NSF Grant No. CCR 88-13632.

†International Computer Science Institute, Berkeley, California 94704

‡Computer Science Division, U.C. Berkeley, Berkeley, California 94720, Supported by an IBM Graduate Fellowship and NSF Grant No. CCR 88-13632.

1 Introduction

Consider the task of writing a program P to evaluate a function f. One of the main difficulties is that when P is implemented it is difficult to verify that $P(x) = f(x)$ for all inputs x. There are two traditional approaches to this problem, program verification and program testing. Program verification has had fairly limited success because even relatively simple programs are hard to prove correct. Furthermore, even if the proof is correct, it only makes a statement about the program as it is written on paper, not about the compiled code nor about the hardware on which it runs. Traditional testing has two drawbacks. First, the test inputs typically do not cover all inputs encountered when the program is actually used, and thus on a particular input the user has no guarantee that the program output is correct. Second, during testing another program P' is used to compute f to compare against the answer of P, and thus there is a reliance on the correctness of another program P' that is in no quantifiable way different than the program P it is being used to test.

We introduce the notion of *self-testing/correcting*, which provides an attractive alternative to traditional approaches of verifying that a program is correct. The theory of self-testing/correcting is an extension of the theory of program checkers introduced by Manuel Blum [Blum].

We want to design a probabilistic program T_f that is able to *self-test* any program P that supposedly computes f, i.e. T_f makes calls to P to estimate the probability that $P(x) \neq f(x)$ for a random input x. We call this probability the error probability of P. We insist that T_f be *different* than any correct program for computing f, in the sense that the running time of T_f, not counting the time for calls to P, must be faster than the running time of any correct program for computing f. This ensures that T_f must be doing something *quantifiably* different than computing f directly, because there is not enough time for this. A self-testing program is in this sense an "independent" verification step for a program P supposedly computing f. In addition, although it is hard to quantify, the self-testers we develop also have the property that the resulting code is aesthetically simple. We would like T_f to be *efficient*, in the sense that the running time of T_f, counting the time for calls to P, is within a constant multiplicative factor of the running time of P. This ensures that the advantages we gain by using T_f to self-test P are not overwhelmed by an inordinate running time slowdown.

In conjunction with a self-testing program, we want to design a probabilistic program C_f that is able to *self-correct* any program P as long as the error probability of P is sufficiently low, i.e., for any input x, C_f makes calls to P to compute $f(x)$ correctly as long as the error probability of P is small enough. As for self-testing programs and for the same reasons, we want C_f to be both *different* and *efficient*.

A self-testing/correcting pair (T_f, C_f) for a function f is a powerful tool. A user can take *any* program P that purportedly computes f and self-test it with T_f. If P passes the self-test then, on any input x, the user can call C_f, which in turn makes calls to P, to correctly compute $f(x)$. Even a program P that computes f incorrectly for a small but significant fraction of the inputs can be used with confidence to correctly compute $f(x)$ for any input x. In addition, if in the future somebody designs a faster program P' for computing f then the same pair (T_f, C_f) can be used to self-test/correct P' without any further modifications. Thus, it makes sense to spend a reasonable amount of time designing self-testing/correcting pairs for functions commonly used in practice and for which a lot of effort is spent writing super-fast programs.

We develop general techniques for constructing simple to program self-testing/correcting pairs for a variety of numerical problems. The following table summarizes the running time behavior of our self-testing/correcting pairs as a function of the problem input size n. The second column is the running time not counting time for calls to P and the third column is the total running time counting time for calls to P, where $M(n)$ is the running time of P on inputs of size n. These times exclude a constant multiplicative factor and they also exclude the running time dependence on the confidence parameter β, which is typically $O(\log(1/\beta))$.

Problem	Without P	Total
Integer Mult.	n	$M(n)$
Mod	n	$M(n)$
Mod Mult.	n	$M(n)$
Integer Div.	$n \log n$	$M(n) \log n$
Poly. Mult.	n	$M(n)$
Mod Exp., ϕ	n	$M(n)$
Mod Exp., no ϕ	$n \log^4 n$	$M(n) \log^3 n$
Matrix Mult.	n	$M(n)$
Determinant	n	$M(n)$
Matrix Inv.	n	$M(n)$
Matrix Rank: C	n	$M(n)$
Matrix Rank: T	$n\sqrt{n}$	$M(n)\sqrt{n}$

1.1 Related Work

[Blum Micali] construct a pseudo-random generator, where a crucial ingredient of the construction can be thought of as a self-correcting program for the discrete log problem. [Rubinfeld] introduces checking for parallel programs, and uses self-testing to design a constant depth circuit to check the majority function.

A self-testing/correcting pair for a function f implies a program result checker for f. A program result checker for f implies a self-tester for f, but it is not known whether a program result checker also implies a self-corrector. Previous to our work, [Kaminski] gives program result checkers for integer and polynomial multiplication. Independently of our work, [Adleman Huang Kompella] give program result checkers for integer multiplication and modular exponentiation. Both of these papers use very different techniques than ours. Previous to our work, [Freivalds] introduces a program result checker for matrix multiplication over a finite field. We make use of this checker when designing the self-testing/correcting pair for matrix multiplication over a finite field.

[Lipton], independently of our work, discusses the concept of self-correcting programs and for several problems uses it to construct a testing program with respect to any distribution *assuming that the programs are not too faulty with respect to a particular distribution.* To highlight the importance of being able to self-test, consider the mod function. To self-correct on input x and modulus R, the assumption in [Lipton] and here is that the program is correct for most inputs x *with respect to the particular modulus* R. This requires a different assumption for each distinct modulus R. Our self-testing algorithm for the mod function on input R can be used to efficiently either validate or refute this assumption.

Previously, [Kannan] provides an elegant program result checker for computing the determinant of a matrix, but it is not efficient. Our self-correcting/testing pair for determinant is efficient, but it relies heavily on allowing the pair to call a library of linear algebra programs instead of restricting calls to a single program that supposedly computes determinant.

In this paper, we assume that the program's answer on a particular input does not depend on previous inputs. [Blum Luby Rubinfeld] considers the case when the program adaptively decides its answer based on previous inputs.

2 A More Formal Overview

For expository purposes, we restrict ourselves to the case when f is a function of one input from some universe I. Let $I_1, I_2, \ldots$ be a sequence of subsets of I such that $I = \cup_{n \in \mathcal{N}} I_n$. The subscript n indicates the "size" of the problem. Let $\mathcal{D} = \{D_n | n \in \mathcal{N}\}$ be an ensemble of probability distributions such that D_n is a distribution on I_n. Let P be a program that supposedly computes f. Let $error(P, f, D_n)$ be the probability that $P(x) \neq f(x)$ when x is randomly chosen in I_n according to D_n. Let $\beta > 0$ be a confidence parameter.

Definition (probabilistic oracle program) : A probabilistic program M is an *oracle* program if it makes calls to another program that is specified at run time. We let M^A denote M making calls to program A.

Definition: Let $0 \leq \epsilon_1 < \epsilon_2 \leq 1$. An (ϵ_1, ϵ_2)-*self-testing program* for f with respect to $\mathcal{D}$ is a probabilistic oracle program T_f that has the following properties for any program P on input n and β.

1. If $error(P, f, D_n) \leq \epsilon_1$ then T_f^P outputs "PASS" with probability at least $1 - \beta$.

2. If $error(P, f, D_n) \geq \epsilon_2$ then T_f^P outputs "FAIL" with probability at least $1 - \beta$.

The value of ϵ_1 should be as close as possible to ϵ_2 to allow as faulty as possible programs P to pass test T_f^P and still have self-corrector C_f^P work correctly.

Definition: Let $0 \leq \epsilon < 1$. An *ϵ-self-correcting program* for f with respect to $\mathcal{D}$ is a probabilistic oracle program C_f that has the following property on input n, $x \in I_n$ and β. If $error(P, f, D_n) \leq \epsilon$ then $C_f^P(x) = f(x)$ with probability at least $1 - \beta$.

We would like T_f and C_f to be both *different* and *efficient* as discussed in the introduction, although sometimes we are forced to relax the efficiency requirement somewhat. In the definitions of *different* and *efficient*, we ignore the running time dependence on the confidence parameter β, which is typically a multiplicative factor of $O(\log(1/\beta))$.

Definition: A *self-testing/correcting pair* for f is a pair of probabilistic programs (T_f, C_f) such that there are constants $0 \leq \epsilon_1 < \epsilon_2 \leq \epsilon < 1$ and an ensemble of distributions $\mathcal{D}$ such that T_f is an (ϵ_1, ϵ_2)-self-testing program for f with respect to $\mathcal{D}$ and C_f is an ϵ-self-correcting program for f with respect to $\mathcal{D}$.

3 Self-Correcting

Because self-testers must be *different*, the strategy used by T_f^P cannot be the naive technique of choosing $x \in I_n$ according to D_n and seeing if $P(x) = f(x)$, because this requires computation of $f(x)$. Similarly, C_f^P is forced to make various calls to P to help it compute $f(x)$ correctly. Many of the self-testers and self-correctors we introduce exploit the following property.

Random Self-Reducibility : Let $x \in I_n$. Let $c > 1$ be an integer. The property is that $f(x)$ can be expressed as an easily computable function F of x, $a_1, \ldots, a_c$ and $f(a_1), \ldots, f(a_c)$, where $a_1, \ldots, a_c$ are easily computable given x and each a_i is randomly distributed in I_n according to D_n.[1]

The strength of this property is that it can be used to transform a program that is correct on a large enough

[1]However, no independence between these random variables is needed, e.g. given the value of a_1 it is not necessary that a_2 be randomly distributed in I_n according to D_n.

fraction of the inputs into a program that computes $f(x)$ correctly with high probability for *any* input x. The following program is a $\frac{1}{4c}$-self-correcting program for f with respect to D_n. The input to the program is n, $x \in I_n$ and a confidence parameter β.

Gen Self-Correcting Program (n, x, β)

> Do for $m = 1, \ldots, 12\ln(1/\beta)$
> randomly generate $a_1, \ldots, a_c$ based on x
> For $i = 1, \ldots, c$, $\alpha_i \leftarrow P(a_i)$.
> $ans_m \leftarrow F(x, a_1, \ldots, a_c, \alpha_1, \ldots, \alpha_c)$
> Output the most common answer among $\{ans_m : m = 1, \ldots, 12\ln(1/\beta)\}$

Lemma 1: The above program is a $\frac{1}{4c}$-self-correcting program for f with respect to D_n.

Proof: Because $error(P, f, D_n) \leq \frac{1}{4c}$ and because, for each $k = 1, \ldots, c$, a_k is randomly distributed in I_n according to D_n, all c outputs of P are correct with probability at least $3/4$ each time through the loop. If all c outputs of P are correct, then by the random self-reducibility property, $ans_m = f(x)$. A straightforward calculation shows that after $12\ln(1/\beta)$ executions, the majority of the answers are equal to $f(x)$ with probability at least $1 - \beta$. ■

As an example, consider the function $f_R(x) = x \bmod R$. (We are viewing this as a function of one input x, where R is a fixed but arbitrary positive integer.) Let $I_n = \mathcal{Z}_{R2^n} = \{0, \ldots, R2^n - 1\}$, let $x \in \mathcal{Z}_{R2^n}$ be the input to the self-corrector, let $D_n = U_{\mathcal{Z}_{R2^n}}$ be the uniform distribution on $\mathcal{Z}_{R2^n}$, let $+_R$ denote addition mod$R$ and let $+_{R2^n}$ denote addition $\bmod R2^n$.

Suppose $error(P_R, f_R, U_{\mathcal{Z}_{R2^n}}) \leq 1/8$. At each step the self-corrector randomly and uniformly chooses $x_1 \in \mathcal{Z}_{R2^n}$, computes $x_2 \leftarrow (x - x_1) \bmod R2^n$ and lets the answer from the step be $P_R(x_1) +_R P_R(x_2)$. Note that $f_R(x) = f_R(x_1) +_R f_R(x_2)$ and that x_2 is uniformly distributed in $\mathcal{Z}_{R2^n}$. Thus, the probability that the answer for a particular step is equal to $f_R(x)$ is at least $3/4$. The self-corrector repeats this step $12\ln(1/\beta)$ times and outputs the most common answer. It is easy to see that the most common answer is correct with probability at least $1 - \beta$. The code for the self-corrector is simple, involving only calls to P_R on random inputs, integer additions and comparisons. Furthermore the self-corrector is both efficient and different.

In addition to the mod function, this generic self-correcting program can be implemented for integer multiplication, modular multiplication and modular exponentiation, matrix multiplication over a finite field, multiplication of polynomials over a finite field and several other problems. In all cases, the resulting self-correcting program is both different and efficient. [Lipton] uses the same basic outline to develop a self-correcting program for several of these problems and also for evaluating a polynomial over a finite field.

4 Linearity and Self-Testing

Although the most interesting of our self-testing methods leads to self-testers that are almost as simple to code as the self-correctors described above, the proofs that they meet their specifications are more difficult and interesting and involve some probability theory on groups that may have other applications. This method applies to integer multiplication, the mod function, modular multiplication, and modular exponentiation when the ϕ function of the modulus is known. The resulting self-testers are simple to code, and are both different and efficient.

4.1 Self-Testing Mod

To give some idea of how the method works, we concentrate on the mod function. For positive integers x and R, let $f_R(x) = x \bmod R$. (As before, we are viewing this as a function of one input x, where R is a fixed but arbitrary positive integer.) Because the self-correcting program for the mod function relies on a program that is correct for most inputs with respect to a particular modulos R, the self-testing program for the mod function is designed to self-test with respect to an input modulus R. This is an important motivation for constructing efficient self-testing programs, because the self-testing program is executed each time a new modulus is used. Similar remarks hold for modular multiplication and modular exponentiation.

There are two critical tests performed by the self-tester. Let x_1 and x_2 be randomly, independently and uniformly chosen in $\mathcal{Z}_{R2^n}$, and set $x \leftarrow x_1 +_{R2^n} x_2$. Note that $f_R(x) = f_R(x_1) +_R f_R(x_2)$, i.e. f_R is a (modular) linear function of its inputs. The *linear consistency test* is

$$\text{“Does } P_R(x) = P_R(x_1) +_R P_R(x_2)\text{?”},$$

and the *linear consistency error* is the probability that the answer to the linear consistency test is “no”. Let z be randomly chosen in $\mathcal{Z}_{R2^n}$ according to $U_{\mathcal{Z}_{R2^n}}$, and set $z' \leftarrow z +_{R2^n} 1$. Note that $f_R(z') = f_R(z) +_R 1$, i.e. in addition to being linear f_R also has (modular) slope one. The *neighbor consistency test* is

$$\text{“Does } P_R(z') = P_R(z) +_R 1\text{?”},$$

and the *neighbor consistency error* is the probability that the answer to the neighbor consistency test is “no”.

Our main theorem with respect to the self-tester for f_R is that there are constants $0 < \psi < 1$ and $\psi' > 1$ such that $error(P_R, f_R, U_{\mathcal{Z}_{R2^n}})$ is at least ψ times the minimum of the linear consistency error and the neighbor consistency error, and that $error(P_R, f_R, U_{\mathcal{Z}_{R2^n}})$ is at most ψ' times the maximum of the linear consistency error and the neighbor consistency error. Thus, we can *indirectly* approximate $error(P_R, f_R, U_{\mathcal{Z}_{R2^n}})$ by instead estimating the linear and neighbor consistency errors.

Without loss of generality, the output of P is always in the range $\{0, \ldots, R-1\}$. To enforce this condition, we can simply check that each answer that P supplies is in this range, and if it isn't then set the answer to zero. This trivially modifies P to another program P' that is at least as correct as P. The same modification needs to be made in the self-correcting program. For simplicity of exposition, we omit these lines of code.

Mod Function Self-Testing Program (n, R, β)

> $N = \lceil 576 \ln(4/\beta) \rceil$
> $t \leftarrow 0$
> Do for $m = 1, \ldots, N$
>   Call **Mod_Linear_Test** (n, R, ans)
> $t \leftarrow t + ans$
> If $t/N > 1/72$ then "FAIL"
>
> $N' = \lceil 32 \ln(4/\beta) \rceil$
> $t' \leftarrow 0$
> Do for $m = 1, \ldots, N'$
>   Call **Mod_Neigh_Test** (n, R, ans)
> $t' \leftarrow t' + ans$
> If $t'/N' > 1/4$ then "FAIL" else "PASS"

Mod_Linear_Test (n, R, ans)

> $ans \leftarrow 0$
> Choose x_1 randomly in $\{0, \ldots, R2^n - 1\}$
> Choose x_2 randomly in $\{0, \ldots, R2^n - 1\}$
> $x \leftarrow x_1 +_{R2^n} x_2$
> If $P(x_1, R) +_R P(x_2, R) \neq P(x, R)$ then $ans \leftarrow 1$

Mod_Neigh_Test (n, R, ans):

> $ans \leftarrow 0$
> Choose z randomly in $\{0, \ldots, R2^n - 1\}$
> $z' \leftarrow z +_{R2^n} 1$
> If $P(z, R) +_R 1 \neq P(z', R)$ then $ans \leftarrow 1$

Theorem 2.2: The above program is an $(1/288, 1/8)$-self-testing program for the mod function with any modulus R.

Proof: This is a corollary of Theorem 2 from the next subsection. ■

The only non-trivial lines of code in the self-testing program are generation of random numbers, calls to the program P, integer additions and integer comparisons.

4.2 Generic Linear Self-Testing

In this section, we describe a generalization of the mod function self-tester to functions f mapping a group G into another group G'. In addition to the mod function, this generic self-tester can be applied to integer multiplication, modular multiplication and modular exponentiation. In all cases, the resulting self-testing program is extremely simple to code, different and efficient. The proof of the two main theorems for the generic self-tester, Theorems A and B, appear in the appendix. A more detailed account of these theorems can be found in [Ben-Or Coppersmith Luby Rubinfeld].

For this version of the paper, we assume that all groups are abelian; in the final version of the paper we generalize to non-abelian groups. Let G be a finite group with group operation $\circ$ and with generators $g_1, \ldots, g_c$ and identity element 0. For $y \in G$, let y^{-1} denote the inverse of y. Let G' be a (finite or countable) group with group operation $\circ'$ and identity element $0'$. For $\alpha \in G'$, let α^{-1} denote the inverse of α. Let $f : G \to G'$ be a function. Intuitively, f is hard to compute compared to either $\circ$ or $\circ'$.

Let $\mathcal{U}_G$ be the uniform probability distribution on G. We say that f has the *linearity property* if:

(1) It is easy to choose random elements of G according to $\mathcal{U}_G$.

(2) F is an easily computable function with the property that, for any pair $x_1, x_2 \in G$, $F(x_1, x_2) \in G'$ and furthermore $f(x_1 \circ x_2) = f(x_1) \circ' f(x_2) \circ' F(x_1, x_2)$. We call this property *linear consistency.* In all of our applications except for integer multiplication, $F(x_1, x_2) = 0'$ for all inputs x_1, x_2, in which case f is a group homomorphism.

(3) For each generator $g_i \in G$, F_i is an easily computable function with the property that, for any $z \in G$, $F_i(z) \in G'$ and furthermore $f(z \circ g_i) = f(z) \circ' F_i(z)$. We call this property *neighbor consistency.* This property is not needed for integer multiplication. For all of the other applications, both G and G' are generated by a single element denoted 1 and $1'$, respectively, (i.e. they are both cyclic groups), and for all $z \in G$, $f(z \circ 1) = f(z) \circ' 1'$.

The linearity property is a special case of random self-reducibility.

Let P be a program that supposedly computes f such that, for all $y \in G$, $P(y) \in G'$. Gen Self-Testing Program 1 is an $(\epsilon/36, \epsilon)$-self-testing program for f with respect to $\mathcal{U}_G$ when G' is an infinite group that has no finite subgroups except $\{0'\}$. The self-tester for integer multiplication is based on Gen

Self-Testing Program 1, where $G = \{0, \ldots, 2^n - 1\}$ with addition mod 2^n as the group operation, and $G' = \mathcal{Z}$ with addition as the group operation. Gen Self-Testing Program 2 is an $(\epsilon/36, \epsilon)$-self-testing program for f with respect to $\mathcal{U}_G$ for all other G'. The self-tester for the mod function described in subsection 4.1, for modular multiplication and for modular exponentiation are all based on Gen Self-Testing Program 2.

Gen Self-Testing Program 1 (ϵ, β)

$N \leftarrow \lceil \frac{72}{\epsilon} \ln(2/\beta) \rceil$
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call **Gen_Linear_Test** (ans)
  $t \leftarrow t + ans$
If $t/N > \epsilon/9$ then "FAIL" else "PASS"

Gen Self-Testing Program 2 (ϵ, β)

$N \leftarrow \lceil \frac{72}{\epsilon} \ln(4/\beta) \rceil$
$t \leftarrow 0$
Do for $m = 1, \ldots, N$
  Call **Gen_Linear_Test** (ans)
  $t \leftarrow t + ans$
If $t/N > \epsilon/9$ then "FAIL"

$N' \leftarrow \lceil 32 \ln(4c/\beta) \rceil$
$t' \leftarrow 0$
Do for $m = 1, \ldots, N'$
  $ans \leftarrow 0$
  Do for $i = 1, \ldots, c$
    Call **Gen_Neigh_Test** (i, ans)
  $t' \leftarrow t' + ans$
If $t'/N' > 1/4$ then "FAIL" else "PASS"

Gen_Linear_Test (ans)

randomly choose $x_1 \in G$ according to $\mathcal{U}_G$.
randomly choose $x_2 \in G$ according to $\mathcal{U}_G$.
If $P(x_1 \circ x_2) \neq P(x_1) \circ' P(x_2) \circ' F(x_1, x_2)$ then $ans \leftarrow 1$ else $ans \leftarrow 0$

Gen_Neigh_Test (i, ans)

randomly choose $z \in G$ according to $\mathcal{U}_G$.
If $P(z \circ g_i) \neq P(z) \circ' F_i(z)$ then $ans \leftarrow 1$

We introduce some notation and provide motivation for why the self-testers work. For each $y \in G$, define the *discrepancy of* y to be

$$disc(y) = f(y) \circ' P(y)^{-1}.$$

Note that P computes f correctly for all inputs if and only if the discrepancy function defines a homomorphism from G into $\{0'\}$.

Because of the linearity property, part (2), and because the self-testing program computes $F(x_1, x_2)$ correctly on its own, $P(x_1 \circ x_2) = P(x_1) \circ' P(x_2) \circ' F(x_1, x_2)$ if and only if

$$disc(x_1 \circ x_2) = disc(x_1) \circ' disc(x_2).$$

If this equality holds for all $x_1, x_2 \in G$ then the discrepancy function defines a homomorphism h from G into G'. Gen_Linear_Test verifies that the discrepancy function is close to some h. If G' is infinite with no non-trivial finite subgroups then, because G is finite, $h = \{0'\}$.

Now suppose G' has a finite subgroup not equal to $\{0'\}$. Because of the linearity property, part (3), and because the self-testing program computes $F_i(z)$ correctly on its own, $P(z \circ g_i) = P(z) \circ' F_i(z)$ if and only if

$$disc(z \circ g_i) = disc(z).$$

If, for all $z \in G$ and for all $i = 1, \ldots, c$, $disc(z \circ g_i) = disc(z)$ then $h = \{0'\}$. Gen_Neigh_Test verifies that $h = \{0'\}$.

The following notation is used throughout the rest of this section.

Notation:

- $\delta = \Pr[disc(x_1 \circ x_2) \neq disc(x_1) \circ' disc(x_2)]$ when x_1 and x_2 are randomly and independently chosen according to $\mathcal{U}_G$.
- For all $i = 1, \ldots, c$, $\delta_i = \Pr[disc(z) \neq disc(z \circ g_i)]$ when z is randomly chosen in G according to $\mathcal{U}_G$.
- $\psi = \Pr[disc(y) \neq 0']$ when y is randomly chosen in G according to $\mathcal{U}_G$.

Theorems A and B are proved in the appendix. These two theorems are the heart of the proof that Generic Self-Testing Programs 1 and 2 meet their specifications.

Theorem A: Let G' be an infinite countable group with no finite subgroups except for the trivial subgroup $\{0'\}$. Then, $\delta \geq 2\psi/9$. ∎

Theorem B: Let G' be any (finite or countable) group. If, for all $i = 1, \cdots, c$, $\delta_i < 1/2$, then $\delta \geq 2\psi/9$. ∎

Lemma 1: Let G' be any (finite or countable) group. Then, $\psi \geq \delta/2$.

Proof: Because $1 - \psi = \Pr[disc(y) = 0']$, $\Pr[disc(x_1 \circ x_2) = 0'] \geq (1 - \psi)^2$, and consequently $\delta \leq 1 - (1 - \psi)^2 = \psi(2 - \psi)$. Because $\psi \geq 0$, this implies that $\psi \geq \delta/2$. ∎

Lemma 2: Let G' be any (finite or countable) group. Then, for all $i = 1, \ldots, c$, $\psi \geq \delta_i/2$.

Proof: For all $i = 1, \ldots, c$, if $disc(z \circ g_i) \neq disc(z)$ then either $disc(z \circ g_i) \neq 0'$ or $disc(z) \neq 0'$. Thus, $\psi \geq \delta_i/2$. ∎

Theorem 1: Generic Self-Testing Program 1 is $(\epsilon/36, \epsilon)$-self-testing for any $0 \leq \epsilon \leq 1$.

Proof: Use Theorem A and Lemma 1. ∎

Theorem 2: Generic Self-Testing Program 2 is $(\epsilon/36, \epsilon)$-self-testing for any $0 \le \epsilon \le 1$.

Proof: Use Theorem B and Lemma 2. ■

5 Libraries and Linear Algebra

Often programs for related problems are grouped in packages; common examples include packages that solve statistics problems or packages that do matrix manipulations. It is reasonable therefore to use programs in these packages to help test and correct each other. We extend the theory proposed in [Blum] to allow the use of several programs, or a *library*, to aid in testing and correcting. We show that this allows one to construct self-testing/correcting pairs for functions which did not previously have efficient self-testing or self-correcting programs, or even program result checkers. Thus, the self-testing/correcting pair is given a collection of programs, all of which are possibly faulty, and may call any one of them in order to test or correct a particular program.

As an example, we show how to self-test/correct a library of possibly fallible programs for matrix multiplication, matrix inverse, determinant and rank. Working with a library of programs rather than with just a single program is a key idea: enormous difficulties arise in attempts to check a determinant program in the absence of programs for matrix multiplication and inverse.

The Linear Algebra Library:

Matrix Multiplication
Input: $n \times n$ matrices A, B over finite field F
Output: $A \cdot B$

Matrix Inverse
Input: $n \times n$ matrix A over finite field F
Output: A^{-1} if it exists, "NO" otherwise

Determinant
Input: $n \times n$ matrix A over finite field F
Output: determinant(A)

Rank
Input: $n \times n$ matrix A over finite field F
Output: rank(A)

For the analysis of the running time, we assume that field operations can be performed in constant time. Let U_n be the distribution on pairs of $n \times n$ matrices where each entry is chosen independently and uniformly from the finite field F. Freivalds_Checker described below is due to [Freivalds].

Self_Correcting

Matrix_Mult(A, B, β)

Specifications: If $error(P, f, U_n) \leq 1/8$ then the probability that the output is equal to $A \times B$ is at least $1 - \beta$.

For $i = 1, \ldots, \infty$ do:
  $A_1 \leftarrow$ a random $n \times n$ matrix
  $B_1 \leftarrow$ a random $n \times n$ matrix
  $A_2 \leftarrow A - A_1$
  $B_2 \leftarrow B - B_1$
  $C \leftarrow P(A_1, B_1) + P(A_1, B_2)$
   $+P(A_2, B_1) + P(A_2, B_2)$
  If
   Freivalds_Checker(A, B, C, β)
   $=$ "PASS" then output C
   and STOP

Freivalds_Checker(A, B, C, β)

Specifications: If $C \neq A \times B$ then output "FAIL" with probability at least $1 - \beta$. If $C = A \times B$ then output "PASS". The running time is $O(n^2 \lceil \log(1/\beta) \rceil)$.

For $j = 1, \ldots, \lceil \log(1/\beta) \rceil$ do
  $R \leftarrow$ random $(n \times 1)$ $0/1$ vector from F
  If $C \cdot R \neq A \cdot (B \cdot R)$ then
   output "FAIL"
output "PASS"

Claim: Self_Correcting Matrix_Mult meets the specifications. Furthermore, the expected total running time is $O(M(n) + n^2 \log(1/\beta))$, where $M(n)$ is the running time of P on $n \times n$ matrices.

Proof: A_1 and A_2 are uniformly distributed (though dependent) matrices. B_1 and B_2 are uniformly distributed (though dependent) matrices that are independent from A_1 and A_2. Hence $P(A_i, B_j) \neq A_i \times B_j$ with probability at most $1/8$, and thus $C = A \times B$ with probability at least $1/2$ at each iteration. Let p be the probability that the final output of Self_Correcting Matrix_Mult is equal to $A \times B$. With probability at least $1/2$ in the first iteration $C = A \times B$, in which case Freivalds_Checker returns "PASS". With probability at most $1/2$ in the first iteration, $C \neq A \times B$, in which case Freivalds_Checker returns "FAIL" with probability at least $1 - \beta$, and the second iteration starts. Thus $p \geq \frac{1}{2} + \frac{1}{2}(1 - \beta)p$. From this, it can be verified that $1 - p$ is at most β.

The expected running time of Self_Correcting Matrix_Mult is at most $M(n) + n^2 \lceil \log(1/\beta) \rceil$ times the expected number of iterations until $C = A \times B$, which is at most two. ■

The self-testing program for matrix multiplication program is simple. The following step is executed $O(\log(1/\beta))$ times to obtain a good estimate $error(P, f, U_n)$. Randomly and independently choose matrices A and B and set $C \leftarrow P(A, B)$. If the output of Freivalds_Checker$(A, B, C, 1/4)$ is "PASS", then the answer is 0 from the step, and if the output is "FAIL" then the answer is 1. It is easy to verify that if $error(P, f, U_n) \geq 1/8$ then the fraction of 1 answers is at least $1/16$ with probability at least $1 - \beta$, and if $error(P, f, U_n) \leq 1/32$ then the fraction of 1 answers is at most $1/16$ with probability at least $1 - \beta$. This yields a $(1/32, 1/8)$-self-tester for matrix multiplication.

We next design a self-correcting program for matrix inversion. Hereafter, we call Self_Correcting Matrix_Mult (abbreviated SCMM) whenever we want to multiply matrices together. The assumption is that SCMM uses a program P_1 has already been self-tested and "PASSED" to compute matrix multiplications. To avoid cluttering the explanation with messy details, we assume that P_1 "PASSED" for good reason, i.e. it has error probability at most $1/8$, and thus SCMM does self-correct.

Let I be the $n \times n$ identity matrix.

Let U'_n be the uniform distribution on $n \times n$ invertible matrices over F. **Randall_Gen_Matrix**(n) randomly generates an $n \times n$ matrix over F according to U'_n, and is due to [Randall].

Self_Correcting Matrix_Inv(A, β)

Specifications: If $error(P, f, U'_n) \leq 1/8$ and A is invertible then the output is A^{-1} with probability at least $1 - \beta$. If A is not invertible then the output is "NO" with probability at least $1 - \beta$.

For $i = 1, \ldots, 12 \ln(1/\beta)$ do:
  $R \leftarrow$Randall_Gen_Matrix(n)
  $R' \leftarrow$ SCMM$(A, R, 1/32)$
  If $P(R')$ = "NO" then
    $ans_i \leftarrow$ "NO"
  Else $A' \leftarrow$ SCMM$(R, P(R'), 1/32)$
    If $I \neq$ SCMM$(A, A', 1/32)$ then
      $ans_i \leftarrow$ "NO"
    Else $ans_i \leftarrow A'$
Output the most common answer

Claim: Self_Correcting Matrix_Inv meets the specifications.

Proof: Suppose that A is invertible. Then, because R is a random invertible matrix, $A \times R$ is a random invertible matrix. If the first call to SCMM is correct then $R' = A \times R$. Because the first call is correct with probability at least $31/32$, the distance between the distribution on R' and U'_n is at most $1/32$. Consequently $P(R') = R'^{-1} = R^{-1} \times A^{-1}$ with probability at least $7/8 - 1/32$. If $P(R') = R^{-1} \times A^{-1}$ and the second call to SCMM is correct then $A' = A^{-1}$. If the third call to SCMM is correct then $ans_i = A^{-1}$. Since these last two calls to SCMM are both correct with probability at least $15/16$, $ans_i = A^{-1}$ with probability at least $3/4$. Now suppose that A is not invertible. Then, for every A', $I \neq A' \times A$. Since the last call to SCMM is wrong with probability at most $1/32$, it follows that $ans_i =$ "NO" with probability at least $31/32$. ■

As was the case for the self-testing program for matrix multiplication, the self-tester for matrix inversion is simple. Notice that inputs need only be self-tested with respect to U'_n. The following step is executed $O(\log(1/\beta))$ times to obtain a good estimate $error(P, f, U'_n)$. Set $R \leftarrow$ Randall_Gen_Matrix(n), and set $R' \leftarrow P(R)$. If $I = \text{SCMM}(R, R', 1/64)$ then the answer is 0 from the step, and otherwise the answer is 1. It is easy to verify that if $error(P, f, U'_n) \geq 1/8$ then the fraction of 1 answers is at least $1/16$ with probability at least $1 - \beta$, and if $error(P, f, U_n) \leq 1/32$ then the fraction of 1 answers is at most $1/16$ with probability at least $1 - \beta$. This yields a $(1/32, 1/8)$-self-tester for matrix inversion.

In the final paper, we will give a self-testing/correcting pair for matrix determinant using calls to the self-testing/correcting pairs for matrix multiplication and matrix inversion, and a pair for matrix rank based on the pairs for all the other problems. The resulting pairs are all *different*, and all but the pair for matrix rank are *efficient*. The matrix rank self-corrector is efficient, but the self-tester is not. However, the self-tester is the more efficient than the program result checker given in [Kannan], [Blum Kannan].

6 Another General Technique for Numerical Problems

In this section we introduce another method of designing self-testers. It is easier to prove that this method of self-testing meets its specifications than it is for self-testing based on linearity. This method works for all the problems that the linear self-testing works for, as well as for polynomial multiplication, matrix multiplication, modular exponentiation when the ϕ function of the modulus is not known, and integer division. The drawback is that this method is often less efficient and that the code is slightly more complicated.

The two requirements for this method to work are random self-reducibility and:

Self-Reducibility to Smaller Inputs: The property is that there is a constant c such that for all $x \in I_n$, $f(x)$ can be expressed as an easily computable function F of x, $a_1, \ldots, a_c$ and $f(a_1), \ldots, f(a_c)$, where $a_1, \ldots, a_c$ are each in I_{n-1}.

For example, for integer multiplication, this condition is fulfilled as follows: Let $x = (x_1, x_2)$. Let x_1^L be the most significant 2^{n-1} bits of x_1

and let x_1^R be the least significant 2^{n-1} bits of x_1. Define x_2^L and x_2^R analogously with respect to x_2. Let $a_1 = (x_1^R, x_2^R)$, $a_2 = (x_1^L, x_2^R)$, $a_3 = (x_1^R, x_2^L)$ and $a_4 = (x_1^L, x_2^L)$. Then, $f(x) = F(x, a_1, \ldots, a_c, f(a_1), \ldots, f(a_c)) = f(a_1) + (f(a_2) + f(a_3))2^{2^{n-1}} + f(a_4)2^{2^n}$.

The overall idea behind this method is that once smaller size inputs have been self-tested, larger inputs can be self-tested by choosing a random input x, decomposing x into smaller inputs, self-correcting the smaller inputs using random self-reducibility (which works because smaller inputs have been self-tested), and then comparing the answer against the answer the program gives on input x. This method of bootstrapping can be continued until the desired problem size is reached. We now give more specific details.

We say that $x \in I_n$ is *bad* if $P(x) \neq f(x)$, and otherwise x is *good*. Program Self_Correct is the generic self-correcting program described in Section 3.

Subroutine Rec_Self_Test (n, β): Rec_Self_Test verifies that most of the inputs in I_n are good given that most of the inputs in I_{n-1} are good.

Specification:

(1) If at least a fraction of $\frac{1}{4c}$ of the inputs in I_n are bad and at most a fraction of $\frac{1}{4c}$ of the inputs in I_{n-1} are bad then Rec_Self_Test outputs "FAIL" with probability at least $1 - \beta$.

(2) If at most a fraction of $\frac{1}{16c}$ of the inputs in I_n are bad and at most a fraction of $\frac{1}{4c}$ of the inputs in I_{n-1} are bad then Rec_Self_Test outputs "PASS" with probability at least $1 - \beta$.

$l \leftarrow O(c \ln(1/\beta))$
Do for $m = 1, \ldots, l$
  $ans_m \leftarrow 0$
  randomly choose $x \in I_n$.
  If $n = 1$ then:
    compute $f(x)$ directly
    If $f(x) \neq P(x)$ then $ans_m \leftarrow 1$
  Else $n > 1$ then:
    randomly generate $a_1, \ldots, a_c$ from x.
    For $k = 1, \ldots, c$,
      $y_k \leftarrow$Self_Correct$(n-1, a_k, \frac{1}{16c^2})$.
    If $F(x, a_1, \ldots, a_c, y_1, \ldots, y_c) \neq P(x)$
      then $ans_m \leftarrow 1$
If $\Sigma_{k=1}^{l} ans_k / l \geq \frac{3}{16c}$ then "FAIL"
Else "PASS"

Claim: Rec_Self_Test meets the specification.

Proof:

(1) Because Self_Correct is called with confidence parameter $\frac{1}{16c^2}$, the probability that there is an incorrect y_k for $k = 1, \ldots, c$ is at most $\frac{1}{16c}$. Therefore, in each iteration $\Pr[ans_m = 1] \geq \frac{1}{4c}(1 - \frac{1}{16c}) \geq \frac{15}{64c}$.

(2) In each iteration $\Pr[ans_m = 1] \leq \frac{1}{16c} + \frac{1}{16c} = \frac{2}{16c}$.

Thus, the average of ans_m over $O(c \ln(1/\beta))$ iterations is at least $\frac{3}{16c}$ with probability at least $1 - \beta$ in case 1 and at most $\frac{3}{16c}$ with probability at least $1 - \beta$ in case 2. ■

Program Self_Test (l, x, β): We make the convention that if any call to one of the subroutines returns "FAIL" then final output is "FAIL" and otherwise the output is "PASS".

Specification:

(1) If there is an i, $1 \leq i \leq l$, such that the fraction of of bad inputs in I_i is at least $\frac{1}{4c}$, then output "FAIL" with probability at least $1 - \beta$.

(2) If for all i, $1 \leq i \leq l$, the fraction of bad inputs in I_i is at most $\frac{1}{16c}$ then output "PASS" with probability at least $1 - \beta$.

For $i = 1, \ldots, l$, call
Rec_Self_Test$(i, \beta/l)$.

Theorem: Self_Test meets the specifications.

Proof:

(1) If there is an i, $1 \leq i \leq l$ such that for all $1 \leq j \leq i-1$, the fraction of bad inputs in I_j is at most $\frac{1}{4c}$ and the fraction of bad inputs in I_i is at least $\frac{1}{4c}$ then Rec_Self_Test$(i, \beta/l)$ outputs "FAIL" with probability at least $1 - \beta/l \geq 1 - \beta$.

(2) If, for all i, $1 \leq i \leq l$, the fraction of bad inputs in I_i is at most $\frac{1}{16c}$ then Rec_Self_Test$(i, \beta/l)$ outputs "FAIL" with probability at most β/l. Thus, over the l calls, the probability that all answers are "PASS" is at least $1 - \beta$. ■

7 Future Work

- Are there self-testing/correcting pairs for other important functions? Some additional results can be found in [Beaver Feigenbaum], [Lipton]. Sorting is a candidate problem, the hard part seems to be the design of a self-corrector.

- Is it possible to show that some functions are not going to have a self-testing/correcting pair? Some partial progress can be found in [Feigenbaum Kannan Nisan], [Yao].

- Are there applications of the convolutions theorems in other areas? We also propose to develop more probabilistic tools long these lines.

- One area of practical concern for self-testing/correcting pairs is the overhead incurred by running the self-tester and self-corrector. Recently a *batch self-corrector* for the mod function has been designed which reduces the overhead to a small additive factor if it is infrequent that P answers incorrectly for some input in a batch [Rubinfeld]. We would like to design batch self-correctors for other important functions.

8 Acknowledgements

We thank Silvio Micali for pointing out the general applicability of our methods, for his enthusiastic support and for numerous illuminating technical discussions. We thank Oded Goldreich, Shafi Goldwasser, Sampath Kannan, Richard Cleve, Don Coppersmith, Michael Ben-Or, Russell Impagliazzo and Steve Omohundro for very helpful discussions.

9 References

Adleman, L., Huang, M., Kompella, K., "Efficient Checkers for Number-Theoretic Computations", Submitted to *Information and Computation.*

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

Ben-Or, M., Coppersmith, D., Luby, M., Rubinfeld, R., "Convolutions on Groups", rough manuscript.

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

Blum, M., Kannan, S., "Program correctness checking ... and the design of programs that check their work", *STOC 1989.*

Blum, M., Luby, M., Rubinfeld, R., "Program Result Checking Against Adaptive Programs and in Cryptographic Settings", *DIMACS workshop on cryptography and distributed systems*, 1989.

Blum, M., and Micali, S., "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits", *SIAM J. on Computing*, Vol. 13, 1984, pp. 850-864, *FOCS 1982.*

Feigenbaum, J., Kannan, S., Nisan, N., "Lower Bounds on Random-Self-Reducibility", *Structures in Complexity Theory*, 1990.

Freivalds, R., "Fast Probabilistic Algorithms", Springer Verlag Lecture Notes in CS No. 74, Mathematical Foundations of CS, 57-69 (1979).

Kaminski, Michael, "A note on probabilistically verifying integer and polynomial products," *JACM*, Vol. 36, No. 1, January 1989, pp.142-149.

Kannan, S., "Program Result Checking with Applications", Ph.D. thesis, U.C. Berkeley, 1990.

Lipton, R., "New directions in testing", manuscript.

Randall, D., "Efficient Random Generation of Invertible Matrices", personal communication.

Rubinfeld, R., "Designing Checkers for Programs that Run in Parallel", manuscript, 1989.

Rubinfeld, R. "Batch Checking for the Mod Function", manuscript, 1990.

Yao, A., "Coherent Functions and Program Checking", these proceedings.

A Convolution Theorems

In this appendix, we prove Theorems A and B. These are the main two theorems with respect to self-testers based on the linearity property. The specific proofs given in this appendix, due largely to Don Coppersmith, are simpler than our original proofs. A full exposition of some related general probability results will appear in [Ben-Or Coppersmith Luby Rubinfeld].

We retain the notation of Subsection 4.2. Uncapitalized letters from the end of the alphabet denote elements chosen randomly from G according to $\mathcal{U}_G$, e.g. x,y, z, whereas uncapitalized letters from the beginning of the alphabet denote fixed elements of G, e.g. a, b, c.

Theorem A: Let G' be an infinite countable group with no finite subgroups except for the trivial subgroup $\{0'\}$. Then, $\delta \geq 2\psi/9$.

Theorem B: Let G' be any (finite or countable) group. If, for all $i = 1, \cdots, c$, $\delta_i < 1/2$, then $\delta \geq 2\psi/9$.

Before giving the proofs of these two theorems, we prove some intermediate lemmas. For Lemma A1, Lemma A2, Lemma A3 and Lemma A4, we assume that $\delta < 2/9$. Let δ' be defined as the solution to the equality $\delta'(1-\delta') = \delta$. Because $\delta < 2/9$, $\delta' < 1/3$.

Lemma A1: $\forall a \in G$, $\exists a' \in G'$ such that $\Pr[disc(x \circ a) = disc(x) \circ' a'] \geq 1 - \delta'$.

Proof: By the definition of δ and because $x \circ a$ is distributed in G according to $\mathcal{U}_G$ and $a \circ y$ is distributed in G according to $\mathcal{U}_G$,

$$\Pr[disc(x \circ a) \circ' disc(y) = disc(x \circ a \circ y)$$

$$= disc(x) \circ' disc(a \circ y)] \geq 1 - 2\delta.$$

So

$$\Pr[disc(x \circ a) \circ' disc(x)^{-1} = disc(y \circ a) \circ' disc(y)^{-1}]$$

$$\geq 1 - 2\delta.$$

This is the sum, over all $a' \in G'$, of the square of the probability

$$\Pr[disc(x \circ a) \circ' disc(x)^{-1} = a'].$$

Since $\delta < 2/9$, this sum exceeds $5/9$ and thus there must be one value a' with

$$\Pr[disc(x \circ a) \circ' disc(x)^{-1} = a'] \geq 1 - \delta'$$

where $(1-\delta')^2 + \delta'^2 = 1 - 2\delta$ and $\delta' < 1/2$. This leads to $\delta'(1-\delta') = \delta$. ■

Lemma A1 leads to the definition of the function h from G to G' defined as follows: For all $a \in G$, let $h(a) = a'$, where a' is the element of G' described in Lemma A1.

Lemma A2: The function h is a group homomorphism from G to G', i.e. for all $a, b \in G$, $h(a \circ b) = h(a) \circ' h(b)$.

Proof: Using Lemma A1 three times, for all $a, b \in G$,

$$\Pr[disc(x)\circ' h(a)\circ' h(b) = disc(x \circ a)\circ' h(b) =$$

$$disc(x \circ a \circ b) = disc(x)\circ' h(a \circ b)] \geq 1 - 3\delta'.$$

This probability is strictly greater than zero because $\delta' < 1/3$, and thus $h(a \circ b) = h(a) \circ' h(b)$. ■

Lemma A3:

(1) If G' is an infinite countable group with no finite subgroups except for the trivial subgroup $\{0'\}$ then for all $a \in G$, $h(a) = 0'$.

(2) If G' is any (finite or countable) group and, for all $i = 1, \cdots, c$, $\delta_i < 1/2$, then for all $a \in G$, $h(a) = 0'$.

Proof: By Lemma A2, h is a group homomorphism and thus the image of h is a finite subgroup of G'. In case (1), the only finite subgroup of G' is $\{0'\}$. In case (2), consider a fixed $i \in \{1, \ldots, c\}$. Because $1 - \delta_i > 1/2$ and using Lemma A1 and the fact that $1 - \delta' > 2/3$,

$$\Pr[disc(x) = disc(x \circ g_i) = disc(x)\circ' h(g_i)] > 1/6,$$

and thus there is some $x \in G$ such that $disc(x) = disc(x) \circ' h(g_i)$ which implies that $h(g_i) = 0'$. Thus, for all $i = 1, \ldots, c$, $h(g_i) = 0'$. Because $g_1, \ldots, g_c$ are generators for G it follows that for all $a \in G$, $h(a) = 0'$. ■

Lemma A4: Under the same conditions as (1) and (2) in Lemma A3, $\Pr[disc(x) = disc(x \circ y)] \geq 1 - \delta'$.

Proof: By Lemma A3, $h(a) = 0'$ for all $a \in G$. On the other hand, Lemma A1 says that

$$\Pr[disc(x \circ a) = disc(x) \circ' h(a)] \geq 1 - \delta'$$

for every $a \in G$, and thus certainly this is true when a is replaced with a random y. Thus, $\Pr[disc(x \circ y) = disc(x)] \geq 1 - \delta'$. ■

Proof of Theorem A: Assume first that $\delta < 2/9$. By definition of δ and using Lemma A4, $\Pr[disc(x) = disc(x \circ y) = disc(x) \circ' disc(y)] \geq 1 - \delta' - \delta$, and thus $\Pr[disc(y) = 0'] \geq 1 - \delta' - \delta$ which implies that $\psi \leq \delta + \delta'$. Because $\delta' < 1/3$, $1 - \delta' > 2/3$ which implies that $\delta' \leq 3\delta/2$. This implies that $\delta \geq 2\psi/5$. On the other hand, if $\delta \geq 2/9$, then because $\psi \leq 1$ it follows that $\delta \geq 2\psi/9$. ■

Proof of Theorem B: Analogous to the proof of Theorem A. ■