A Self-Tester for Linear Functions over the Integers with an Elementary Proof of Correctness

Sheela Devadas* and Ronitt Rubinfeld†

December 18, 2014

Abstract

We present simple, self-contained proofs of correctness for algorithms for linearity testing and program checking of linear functions on finite subsets of integers represented as $n$-bit numbers. In addition we explore two generalizations of self-testing to multiple variables - the case of multilinear functions and homomorphisms on a multidimensional vector space. We show that our self-testing algorithm for the univariate case can be directly generalized to vector space domains. The number of queries made by our algorithms are independent of domain size. However, linearity testing for multilinear functions requires a different testing algorithm. We give an algorithm for the $k$-linearity problem with queries independent of the size of the domain.

1 Introduction

In this paper, we consider the problem of linearity testing, as in the program checking, self-testing and property testing frameworks of [6], [5], [14], [8]. A function $f$ is linear if for any $x, y$ in the domain, $f(x) + f(y) = f(x+y)$. In the case where the domain and range are both the set of integers, it is easily shown (by induction) that the linear functions are exactly the functions that multiply by a constant: i.e., $f(x) = bx$ for some $b$.

Given a program $P_b$ that computes multiplication by $b$, our self-tester should pass $P_b$ if it gives the correct answer for all inputs, and should fail $P_b$ with high probability if it is incorrect on a large enough fraction of the $n$-bit integer inputs.

We present efficient algorithms for linearity testing and program checking of functions on integers represented as $n$-bit numbers as considered in [5]. Our query complexities are of the same order of magnitude, though the constants we achieve are not as good as in [2], [10], [11]. However, our proofs are more elementary; in contrast to previous proofs, our proofs use little “algebraic structure”. We then show that our techniques can be extended to multivariate linear functions and multilinear functions on the set of integers.

Previous Work

Linearity testing was first considered in [5] and used to give algorithms for self-testing and program checking programs that compute linear functions. Linearity testing for multivariate functions over various finite groups has been considered in several works, including [2], [10], [1], [9], [13], [10], [2], [17], [4], [13]. In [5], [13], linearity testing for multivariate functions over finite subsets of infinite groups is considered. Testers for real-valued multilinear functions defined over finite domains have been studied in [12]. Linearity testing for functions over finite subsets of rational domains is considered in [9], [13].

Several works have considered the problem of testing low-degree polynomials. Specifically relevant to this work are results on testing polynomials over rational domains, such as [14].
Outline of Paper  In Section 2, we give definitions of property testing, self-testing, and checking algorithms from [6, 5, 14, 8], and give a general overview of the other definitions and techniques necessary for our proofs. In Section 3, we present self-testing and property testing algorithms for the univariate case and proofs that they catch errors in all programs whose output function differs from any linear function in a significant fraction of locations. In Section 4, we discuss how to extend the results we have for the univariate case to two distinct multivariate cases: linear homomorphisms and multilinear functions. Finally, in the Appendix we present an additional checking algorithm for the univariate case.

2  Preliminaries

A checking algorithm, as defined in [6] for a function \( f \), is an algorithm \( C \) that gets as input a specific input \( x \) and program \( P \). The goal is to determine whether program \( P \) is correct on input \( x \) or has a fault. \( C \) may make calls to \( P \) on any input. If \( P = f \) for all inputs, the checking algorithm \( C \) should pass with probability at least 2/3; but if \( P(x) \neq f(x) \) at the chosen input \( x \), algorithm \( C \) returns FAIL with probability 2/3, where the probability is over the coin tosses of \( C \), and not on any assumption on the input distribution. Note that if \( P(x) = f(x) \), but there is a \( y \neq x \) such that \( P(y) \neq f(y) \), then \( P \) may either output PASS or FAIL.

A self-testing algorithm, as defined in [6] for a function \( f \) over a finite domain \( D \), is an algorithm \( T \) that gets as input a program \( P \) and a parameter \( \epsilon \). The goal is to determine whether program \( P \) is correct on most inputs in \( D \). \( T \) may make calls to \( P \) on any input. If \( P = f \) for all inputs in \( D \), the self-testing algorithm \( T \) returns PASS with probability at least 2/3; but if \( P(x) \neq f(x) \) on \( \geq \epsilon \) fraction of inputs in \( D \), algorithm \( T \) returns FAIL with probability 2/3. Note that if there exists an \( x \) such that \( P(x) \neq f(x) \), but \( P(x) \neq f(x) \) on at most \( \epsilon \) fraction of inputs in \( D \), then \( T \) may either output PASS or FAIL.

In general, given a finite domain \( D \), we say that \( P \) is \( \epsilon \)-close to \( f \) if the probability that \( P(x) = f(x) \) over \( x \) chosen uniformly from \( D \) is \( \geq 1 - \epsilon \), and that \( P \) is \( \epsilon \)-far from \( f \) if it is not \( \epsilon \)-close. Therefore our self-testing algorithm returns FAIL with high probability (2/3) if \( P \) is \( \epsilon \)-far from \( f \).

A property testing algorithm for a function family \( F \), as defined in [14] (there described as an \( \epsilon \)-function-family-tester), is an algorithm \( T \) that gets as input a program \( P \) and a parameter \( \epsilon \). \( T \) may make calls to \( P \) on any input. If for some \( f \in F \), \( P = f \) on all inputs, the testing algorithm \( T \) returns PASS with probability at least 2/3, but if \( P \) is \( \epsilon \)-far from all \( f \in F \), then \( T \) returns FAIL with probability 2/3.

For checking, self-testing, and property testing algorithms, the parameters to optimize are the number of queries to the program \( P \) and the additional computation time the algorithm needs to perform, where we define the additional computation time as the running time of the algorithm not including time spent by \( P \) in answering queries. In this paper we focus on algorithms that make a constant number of queries to \( P \) and incur an additional computation time that is only linear in the input size. Note that for functions such as integer multiplication, no linear time algorithm is known; the best known algorithm is \( O(n \log n \log \log n) \), in [7]. Thus it is not known how to test multiplication by comparing results with another known multiplication program using only an additional cost of linear time.

Here we focus on programs \( P \) purporting to compute not just some linear function but a specific linear function \( f \): for these programs, we give a self-testing algorithm. In the univariate case, such a function must be \( f(x) = bx \) for some \( b \). Similar constraints are true for the linear homomorphism and multivariate cases. We rely on being able to compute \( f(x) \) more quickly for inputs \( x \) of a specific form: in particular, \( b \cdot 2^n \) can be computed in linear time by shifting \( b \) by \( n \) spaces. Note that multiplication and division by \( 2^n \) can be done in time linear in \( n \).

\footnote{Note that by repeating \( C \) at least \( 1/\epsilon \) times and outputting the majority answer, error probability \( \leq \epsilon \) can be achieved.}
3 Testing Algorithm

We describe an algorithm that tests the correctness of a program $P_b$ that purports to compute a function $f_b(x) = b \cdot x$. Let $P_b(x)$ indicate the program’s output when given the input $x$. We refer to the domain of integers represented as $n$-bit numbers as $D_n$; the size of $D_n$ is $2^n$.

**Theorem 3.1.** There exists a self-testing algorithm for the linear function $f_b$ over the domain $D_n$ with $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries to the program $P_b$.

We will also describe an algorithm that tests the correctness of $P_b$ on a specific input:

**Theorem 3.2.** There exists a checking algorithm for the linear function $f_b$ over the domain $D_n$ with $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries to the program $P_b$.

The proof of Theorem 3.2 is given in the appendix.

To prove Theorem 3.1, we first define a function RandSplit that is useful in our testing and checking algorithms and will be redefined in the multilinear case. The idea of this function is to use the distributive property of multiplication: $b(a + c) = ba + bc$. It should be the case that $P_b(a + c) = P_b(a) + P_b(c)$, so we will check that this is the case for random $a$ and $c$. One issue is that $a + c$ can be bigger than $2^n$, though not bigger than $2^n + 1$. We can keep track of which is the case, and if $(a + c) = 2^n + x$ we can check that $P_b(a) + P_b(x) + b \cdot 2^n = P_b(a + c)$ instead. We can use this ‘wraparound’ property and our ability to calculate $b \cdot 2^n$ easily, via a linear time shift operation, to verify that $P_b$ satisfies this distributive property.

$$
\text{input} : \text{An } n\text{-bit number } x, \text{an integer } b, \text{a program } P_b \text{ for computing multiplication by } b \\
\text{output}: \text{FAIL or PASS}
$$

1. $x_1 \leftarrow \text{Rand}(0,2^n)$;
2. if $x_1 < x$ then $\delta \leftarrow 0$ else $\delta \leftarrow 1$
3. $x_2 \leftarrow \delta \cdot 2^n + x - x_1$
4. if $P_b(x_1) + P_b(x_2) \neq b \cdot \delta \cdot 2^n + P_b(x)$ then
5. return FAIL;
6. end
7. return PASS;

**Algorithm 1: RandSplit**

We now use Algorithm 1 to create a testing algorithm. Before running Algorithm 1 at least $k_2$ times, we test $n$-bit inputs summing to $2^n$ for the same distributive property $k_1$ times. The combination of these two tests will allow us to detect errors in $P_b$. We will show that it suffices for $k_1 + k_2$ to be set to $O(1/\epsilon)$.

$$
\text{input} : \text{A program } P_b \text{ for computing multiplication by } b \\
\text{output}: \text{FAIL or PASS}
$$

1. for $i \leftarrow 1$ to $k_1$ do
2. $x \leftarrow \text{Rand}(0,2^n)$;
3. if $P_b(x) + P_b(2^n - x) \neq b \cdot 2^n$ then
4. return FAIL;
5. end
6. end
7. for $i \leftarrow 1$ to $k_2$ do
8. $x \leftarrow \text{Rand}(0,2^n)$;
9. if RandSplit $(x, b, P_b) = \text{FAIL}$ then
10. return FAIL;
11. end
12. end
13. return PASS;

**Algorithm 2: Test**
Lemma 3.3 (Main Lemma). If $P_b$ is correct on $\leq 1 - \epsilon$ fraction of inputs in $D_n$, then Algorithm returns FAIL with probability $\geq 3/4$.

Proof. We first define the discrepancy of an input:

Definition 3.4. The discrepancy of $x$ is $d(x) = P_b(x) - x \cdot b$.

If $P_b(x)$ is correct, then $d(x) = 0$. By our assumption, $d(x) \neq 0$ for at least $\epsilon$ fraction of the inputs.

We now show that if a function is likely to pass the test, the discrepancy function must have a certain form; the number of inputs with positive discrepancy and the number of inputs with negative discrepancy must be about the same.

Proposition 3.5. If less that $\epsilon/2 - \beta$ fraction of the numbers in the domain $D_n$ have discrepancy $> 0$ ($< 0$) then Line 4 will output FAIL with probability at least $2 \beta$.

Proof. Consider ordered pairs of the form $(x, 2^n - x)$. The test in Line 3 pairs the numbers $(x, 2^n - x)$ and detects an error if their discrepancies do not sum to 0. Therefore if no error is detected but $x$ has positive discrepancy, $2^n - x$ must have discrepancy strictly less than 0. If $\leq \epsilon/2 - \beta$ fraction of the numbers have discrepancy strictly less than 0, then since we know that $\leq 1 - \epsilon$ fraction of the numbers have discrepancy 0 (since $P_b$ is correct on those numbers), we see that $\geq \epsilon/2 + \beta$ of the numbers have positive discrepancy. When pairing the numbers into $(x, 2^n - x)$, we see that an error will be detected if the discrepancies do not sum to 0. However, by our assumption, there are fewer numbers with discrepancy strictly less than 0; specifically, $\epsilon/2 + \beta - (\epsilon/2 - \beta) = 2 \beta$ fraction of the numbers more have $d(x) > 0$ than have $d(x) < 0$. Therefore $2 \beta$ of the $(x, 2^n - x)$ pairs have $d(x) > 0$ and $d(2^n - x) \geq 0$. Therefore one iteration of the first ‘for loop’ gives an error with probability $2 \beta$ because the discrepancies need to sum to 0. The same proof works for showing that $> \epsilon/2 - \beta$ fraction of the numbers must have discrepancy strictly greater than 0 or there will be an error with probability $2 \beta$.

This means that we can assume that if the loop in Lines 16 in Algorithm 2 passes with probability $2 \beta$, then the fraction of numbers with discrepancy greater than 0 is $> \epsilon/2 - \beta$ but $< \epsilon/2 + \beta$. The same is true for the fraction of numbers with discrepancy strictly less than 0.

We now define a $\delta_x$ function related to the discrepancy function.

Definition 3.6. $\delta_x(x_1) = 1$ if $d(x_1) > 0$ and $d(x - x_1) < 0$. $\delta_x(x_1) = 0$ otherwise.

Definition 3.7. The number of opposite-sign matches is $\sum_{0 \leq x_1 \leq 2^n} \delta_x(x_1)$.

Proposition 3.8. The expected size of the number of opposite-sign matches is $\leq (\epsilon/2 + \beta)^2 2^n$.

Proof. Using the fact that $x_1$ gets matched to any $x_2$ with probability $1/2^n$,

$$E[\text{number of opposite - sign matches}] = E_x \left[ \sum_{0 \leq x_1 \leq 2^n} \delta_x(x_1) \right]$$

$$= \sum_{0 \leq x_1 \leq 2^n} E_x[\delta_x(x_1)]$$

$$= \sum_{0 \leq x_1 \leq 2^n} Pr_x[\delta_x(x_1)]$$

$$\leq \sum_{x_1 | d(x_1) > 0} Pr_x[\delta_x(x_1)]$$

$$\leq (\epsilon/2 + \beta)^2 2^n Pr_x[\delta_x(x_1)]$$

$$\leq (\epsilon/2 + \beta)^2 2^n$$

\[\square\]
Definition 3.9. $x$ is good if the number of opposite-sign matches is $\leq \frac{1}{\alpha}(\epsilon/2 + \beta)^2 n$ and bad otherwise.

Proposition 3.10. The probability that a bad $x$ is picked in Line 8 is $\leq \frac{1}{\alpha^2}$. 

Proof. Apply Markov’s inequality to the definition of a good $x$. 

Proposition 3.11. If a good $x$ is picked in Line 8 then with probability $\geq \epsilon/2 - \frac{1}{\alpha}(\epsilon/2 + \beta)^2$, Line 11 outputs FAIL.

Proof. The probability of picking any $(x_1, x - x_1)$ pair during verification is $\frac{1}{2^n}$. 

If we have $d(x_1) > 0$ and $d(x_2) \geq 0$, then $d(x_1) + d(x_1 - x_1) > 0$. We see that the probability that $d(x_1) > 0$ is $\geq \epsilon/2 - \beta$ and the number of matches with $d(x_1) > 0$ and $d(x_1 - x_1) < 0$ is $\leq \frac{1}{\alpha}(\epsilon/2 + \beta)^2$. Therefore the probability that $d(x_1) > 0$ and $d(x_1 - x_1) \geq 0$ is $\geq \epsilon/2 - \beta - \frac{1}{\alpha}(\epsilon/2 + \beta)^2$. A similar proof shows that the probability that $d(x_1) < 0$ and $d(x - x_1) \leq 0$ is also $\geq \epsilon/2 - \beta - \frac{1}{\alpha}(\epsilon/2 + \beta)^2$.

In the first case we note that $d(x_1) + d(x_1 - x_1) > 0$, while in the second case $d(x_1) + d(x_1 - x_1) < 0$. Therefore the probability that $d(x_1) + d(x_1 - x_1) > 0$ is $\geq \epsilon/2 - \beta - \frac{1}{\alpha}(\epsilon/2 + \beta)^2$, as is the probability that $d(x_1) + d(x_1 - x_1) < 0$. Since the test only passes if $d(x) = d(x_1) + d(x_1 - x_1)$ and $d(x)$ cannot be both positive and negative, a mistake must be found with probability at least $\geq \epsilon/2 - \beta - \frac{1}{\alpha}(\epsilon/2 + \beta)^2$.

Since we know the probability of finding an error in each part of the algorithm, we need only repeat the parts enough times to catch the error in order to output FAIL with high probability.

3.1 Putting It Together

We now consider possible specific values for $\alpha, \beta$. Let $\beta = \epsilon/4$ and $\alpha = 3/4$.

What is the runtime of Algorithm 2? Applying the Chernoff bound as given in the Appendix, we see that if we want to be able to say that $\geq \epsilon/2 - \beta$ of the inputs have discrepancy $< 0$ or $> 0$ with probability $\geq 7/8$ we need to have $k_1 = O(1/\beta) = O(1/\epsilon)$. The probability of detecting an error in $P_b$ in the loop in Lines 7-12 is $\geq (1 - \alpha)(\epsilon/2 - \beta - \frac{1}{\alpha}(\epsilon/2 + \beta)^2) = \epsilon/4 - \frac{3\epsilon^2}{4}$, we therefore need to run the loop $k_2 = O(\frac{1}{\epsilon - 3\epsilon^2})$ times in order to expect to see this error with probability $7/8$; for sufficiently small $\epsilon$, this is $O(1/\epsilon)$ as well. Therefore $k_1 + k_2$ is $O(1/\epsilon)$ as desired. If we see errors in both parts of the algorithm with probability $7/8$, by a union bound the probability that we fail to catch an error in $P_b$ in both parts is $1/4$, so we output FAIL with probability $3/4$ as desired.

If $\epsilon = 1/8$, then using the specific values for the Chernoff bound given in Theorem 3.1 in the Appendix, we see that $k_1 = 96$ and $k_2 \approx 1200$ will give us our desired result: Algorithm 2 will return FAIL with probability $\geq 3/4$.

Proof of Theorem 3.1. If $P_b$ is correct on all inputs in $D_n$, then Algorithm 2 will pass. If $P_b(x) \neq f(x)$ on at least $\epsilon$ fraction of the inputs in $D_n$, then we have shown in Lemma 3.3 that Algorithm 2 returns FAIL with probability $\geq 3/4$. We have also shown that setting $k_1 + k_2$ to be $O(1/\epsilon)$ lets Algorithm 2 return FAIL with probability $\geq 3/4$, so the number of queries to $P_b$ will also be $O(1/\epsilon)$. The extra computation done by the algorithm consists only of linear time operations such as the shift allowing us to compute $b \cdot 2^n$; therefore the additional computation time is $O(n(k_1 + k_2)) = O(n/\epsilon)$. Thus Algorithm 2 is a valid self-testing algorithm for the function $f_b$.

3.2 General Linear Function

We include a property testing algorithm for the case where we do not know what linear function $P$ claims to compute; only that it claims to compute a linear function. In this section, the property tester needs to pass any linear function and fail any function that is not $\epsilon$-close to some linear function.
Theorem 3.12. There exists a property testing algorithm for linear functions on $n$-bit inputs with $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries to the program $P_b$.

The idea of this general linear testing algorithm, or property testing algorithm, is similar to Algorithm 2 above in that we run two tests several times each. The first test here is to check that for various pairs of inputs $x, 2^n - x$ that sum to $2^n$ that $P(x) + P(2^n - x)$ is always equal to $P(2^n)$, which must be $b \cdot 2^n$ for some integer $b$. This will allow us to reduce to the case of a specific linear function for the second part of this test, which is just running Algorithm 1 on various inputs $x$ for the value of $b$ that we found in the first part. In much the same way as above, this will allow us to detect errors in $P$. We note that this algorithm essentially learns the value of $b$.

Algorithm 3: GeneralLinearTest

```
input: A program $P$ claiming to compute a linear function
output: FAIL or PASS
1    $a \leftarrow P(2^n)$;
2    if $2^n \not| a$ then
3        return FAIL;
4    end
5    for $i \leftarrow 1$ to $k_1$ do
6        $x \leftarrow \text{Rand}(0, 2^n)$;
7        $a_i \leftarrow P(x) + P(2^n - x)$;
8        if $a_i \neq a$ then
9            return FAIL;
10       end
11    end
12    $b \leftarrow a/2^n$;
13    for $i \leftarrow 1$ to $k_2$ do
14        $x \leftarrow \text{Rand}(0, 2^n)$;
15        if $\text{RandSplit}(x, b, P) = \text{FAIL}$ then
16            return FAIL;
17        end
18    end
19    return PASS;
```

Lemma 3.13. If $P$ is $\epsilon$-far from linear, then Algorithm 3 returns FAIL with probability $\geq 3/4$.

Proof. If $P$ passes the test in Line 2 that $2^n \mid P(2^n)$ we can reduce to the case where $P = P_b$ for $b = a/2^n$, and the testing algorithm is equivalent to the original testing algorithm. Since $P$ is $\epsilon$-far from linear, it is $\epsilon$-far from the function $b \cdot x$, so then by Lemma 3.13 since Algorithm 2 will return FAIL with probability $\geq 3/4$, so will Algorithm 3.

Proof of Theorem 3.12 If $P$ is linear, Algorithm 3 will output PASS with probability 1. From Lemma 3.13 it is clear that Algorithm 3 will output FAIL with probability $\geq 3/4$ if we set $k_1 + k_2 = O(1/\epsilon)$ as in Algorithm 2. We see that it makes the same number of queries and has extra running time on the same order as Algorithm 2. Therefore it is a linearity property tester with additional computation time $O(n/\epsilon)$ and queries $O(1/\epsilon)$.

4 Multivariate Linear Functions

In this section we give self-testers for multivariate linear functions - both linear homomorphisms on a vector space and multilinear functions.
For a vector space $V$ of dimension $n$, we say $f$ is a linear homomorphism on $V$ if for any $x_1, x_2 \in V$, $f(x_1) + f(x_2) = f(x_1 + x_2)$. An example of a linear homomorphism in two variables is $f(x,y) = x + y$.

We say that $f$ is a multilinear function in $m$ variables or an $m$-linear function, if for all $i \in [m]$, $x_1, \ldots, x_m$, and $y$, $f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m) + f(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_m) = f(x_1, \ldots, x_{i-1}, x_i + y, x_{i+1}, \ldots, x_m)$. In this case the domain is $D_m^n$. An example of a multilinear function in 2 variables is $f(x,y) = xy$.

To see an example of the difference between a multilinear function and a linear homomorphism, note that for a linear homomorphism in two variables $f(1,3) + f(2,3) = f(3,6)$, but for a 2-linear function, $f(1,3) + f(2,3) = f(3,3)$.

### 4.1 Linear Homomorphisms

To test multivariate linear homomorphisms, we make the assumption that for the vector $a = (2^n, \ldots, 2^n)$ with $n$ entries we know $f(a)$. Note that this is a generalization of the univariate case, in which we need to assume we know the value of $f(2^n)$.

Let $P$ be a program that purports to compute $f$. We can modify the algorithm above to replace $2^n$ with our vector $a$ and modify the random-split function to split one vector into a random pair of vectors (by using the usual random-split function component-wise). Instead of calling Algorithm 1 on a number $x$, integer $b$, and program $P$, we call it on a vector $x$ and program $P$ and verify that for a random vector $x_1$ we have $f(x_1) + f(x - x_1) = f(x)$. Then the proof above still holds if we replace $2^n$ with the number of vectors in the vector space - which is $2^{kn}$ if our domain is $k$ dimensional vectors of $n$-bit numbers. Because there was no dependence on the size of the domain, the same error bounds hold; therefore we can get the same bound that if $P$ is correct on $\leq 3/4$ of the inputs, the program will return FAIL with high probability.

We therefore have the following theorem given a linear homomorphism $f$ on a $k$-dimensional vector space $V$ with a vector $a$ for which we can easily compute $f(a)$. We assume vectors in $V$ have coordinates that are $n$-bit integers. Therefore $|V| = 2^{kn}$.

**Theorem 4.1.** There exists a self-testing algorithm for the linear homomorphism $f$ over the domain $V$ with $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries to the program $P$.

We redefine the algorithms for this case. We define that a vector $x$ is less than another vector $y$ if any of the coordinates of $x$ is less than the corresponding coordinate in $y$.

```
input : A $k$ dimensional vector of $n$-bit numbers $x \in V$, the value $f(a)$, a program $P$ for computing multiplication by $f$
output: FAIL or PASS
1 $x_1 \leftarrow \text{Rand} (V)$;
2 if $x_1 < x$ then $\delta \leftarrow 0$ else $\delta \leftarrow 1$ $x_2 \leftarrow \delta \cdot a + x - x_1$;
3 if $P(x_1) + P(x_2) \neq \delta \cdot f(a) + P(x)$ then
4 return FAIL;
5 end
6 return PASS;
```

**Algorithm 4: RandSplitTwo**

We now use Algorithm 4 to create a self-testing algorithm for this linear homomorphism just as we did with the univariate case.
A program $P$ for computing the function $f$ on a domain $V$ and the value $f(a)$

Output: FAIL or PASS

1. for $i \leftarrow 1$ to $k_1$ do
   2. $x \leftarrow \text{Rand}(V)$;
   3. if $P(x) + P(a - x) \neq f(a)$ then
      4. return FAIL;
   5. end
6. end

7. for $i \leftarrow 1$ to $k_2$ do
   8. $x \leftarrow \text{Rand}(V)$;
   9. if $\text{RandSplitTwo}(x, f(a), P) = \text{FAIL}$ then
      10. return FAIL;
   11. end
12. end
13. return PASS;

Algorithm 5: LinearHomomorphismTest

Lemma 4.2. If $P$ is correct on $\leq 1 - \epsilon$ fraction of inputs in $V$, then Algorithm 5 returns FAIL with probability $\geq 3/4$.

Proof. We note that this is exactly equivalent to the case of Lemma 3.3 and Algorithm 2 only with a larger domain. Since the error bounds did not depend on the size of the domain, we see that by Lemma 3.3 we again get that Algorithm 5 returns FAIL with probability $\geq 3/4$.

Now that we have the algorithm we can show that it is our desired self-testing algorithm.

Proof of Theorem 4.1. We see easily that if $P$ is correct on all inputs in $V$ that Algorithm 5 always outputs PASS. Then by Lemma 4.2 we see that if $P$ is $\epsilon$-far from $f$ that Algorithm 5 outputs FAIL with probability $\geq 3/4$.

By the same Chernoff bounds from Section 3.1 we see that we can set $k_1, k_2$ so that $k_1 + k_2 = O(1/\epsilon)$. The additional computation time depends on how long it takes to compute $f(a)$ in general. If we assume it is $O(n)$, then the additional computation time is $O(n/\epsilon)$ as desired. Therefore $P$ is a valid self-testing algorithm for the function $f$ as desired.

4.2 Multilinear Functions

We first consider the 2-linear case as an example and then generalize to all multilinear functions. Again we have a program $P$ purporting to compute $f(x, y)$: this time the tests are on “cross-sections”. The main idea of our algorithm is to hold $x$ constant at a random value and then run the tests described in the first section as though $f$ was a function of one variable, $y$; we repeat this several times for different values of $x$. Then we hold $y$ constant at a random value and run the tests on the single variable $x$; we repeat this several times as well. We call holding one variable constant and testing the other a cross-section test. In order to be able to do this there must exist $x_0, y_0$ such that we can quickly and easily compute $f(x_0, y), f(x, y_0)$ for any $x, y$. This is analogous to the vector $a$ or $b \cdot 2^n$ in Section 3.

We consider the probability of error for each part of each test. For a given $x$, let $\gamma_x$ be the fraction of $y$-values in $D_n$ so that $f(x, y)$ is correct, and let $\Gamma_x$ be the fraction of $y$ values in $D_n$ for which $f(y, x)$ is correct.

We note that the average value of both $\gamma_x$ and $\Gamma_x$ is just $1 - \epsilon$, the fraction correct of the total. What we want to do is find a the probability that $\gamma_x$ or $\Gamma_x$ is less than $7/8 = 1 - 1/8$: we are able to give good bounds on the probability of outputting FAIL if $\gamma_x$ or $\Gamma_x$ is less than $7/8$. We see that by Markov’s inequality, the probability that $\gamma_x < 7/8$ is $\geq 1 - \frac{8(1-\epsilon)}{\epsilon}$. Therefore the total probability of getting an error on one
cross-section test is $\geq 3/4(1 - \frac{8(1-\epsilon)}{7}) = 6\epsilon/7 - 3/28$. If we do $k$ cross-section tests, our probability of getting an error is then $\geq 1 - (1 - (6\epsilon/7 - 3/28))^k$.

The same technique can be used for a function that is linear in each of $m$ variables when the remaining $m - 1$ are held constant. In this case, a cross-section test corresponds to holding $m - 1$ variables constant and letting the remaining variable vary. We will need $a_1, \ldots, a_m$ values such that we can quickly and easily compute $f$ where the $i$th variable is $a_i$ and the other $m - 1$ variables are any values in our domain. In Section 3, we used the fact that for $a_i = 2^n$ we could easily compute $P_b(2^n) = b \cdot 2^n$.

We will get the following theorem:

**Theorem 4.3.** There exists a self-testing algorithm for the multilinear function $f$ over the domain $D_n^m$ with $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries to the program $P$.

Assume we are testing a program $P([x])$, where the argument $[x]$ is used to denote a list of $m$ values $x_k$. We can write out an algorithm for this multilinear case similarly to the testing algorithm above; we just need different definitions for the RandSplit function and a Test function using calls to the tester algorithm of section 3; we do this by writing the tester algorithm as a function $T(i, [x])$, where $[x]$ is a list of $m - 1$ numbers and $i$ represents the position we are allowing to vary. Then $[x].append(x, i)$ represents inserting the value $x$ into the $i$th position in the list, giving us $m$ values and a possible input to $P$ or $f$: this means that if $[x]$ is the list of length $m - 1$ $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m$ that $[x].append(x, i)$ returns the list of length $m \cdot x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_m$. We also need the $a_i$ from the previous paragraph.

We begin by redefining the RandSplit function for the multilinear case.

```plaintext
input : An n-bit number $x$, a vector $[x]$ of $m - 1$ n-bit numbers, a position $i$, a program $P$
computing a multilinear function on $m$ inputs
output: FAIL or PASS
1 $y \leftarrow$ Rand $(0, 2^n)$;
2 if $y < x$ then $\delta \leftarrow 0$ else $\delta \leftarrow 1$
3 $z \leftarrow \delta a_i + x - y$;
4 if $P([x].append(y, i)) + P([x].append(z, i)) \neq \delta \cdot f([x].append(a_i, i)) + P([x].append(x, i))$ then
5 return FAIL;
6 return PASS;
```

**Algorithm 6: MultiRandSplit**

We now use Algorithm 6 to rewrite the univariate testing algorithm as a function CrossSectionTest where we hold $m - 1$ of the inputs constant and vary the remaining input:
input: A vector \([x_j]\) of \(m-1\) \(n\)-bit numbers, a position \(i\), a program \(P\) computing a multilinear function on \(m\) inputs
output: FAIL or PASS

1 for \(i \leftarrow 1\) to \(k_1\) do
2 \quad \(x \leftarrow \text{Rand}(0,2^n)\);
3 \quad \text{if} \ P([x_j].\text{append}(x,i)) + P([x_j].\text{append}(a_i - x, i)) \neq f([x_j].\text{append}(a_i, i)) \text{ then}
4 \quad \quad \text{return FAIL;}
5 \quad \end{condition}
6 \end{for}
7 for \(i \leftarrow 1\) to \(k_2\) do
8 \quad \(x \leftarrow \text{Rand}(0,2^n)\);
9 \quad \text{if} \ \text{MultiRandSplit} (x, [x_j], i, P) = \text{FAIL} \text{ then}
10 \quad \quad \text{return FAIL;}
11 \quad \end{condition}
12 \end{for}
13 \text{return PASS;}

Algorithm 7: CROSSSECTIONTEST

Finally we can use Algorithm 7 to create a multilinear testing algorithm:

input: A program \(P\) computing a multilinear function on \(m\) inputs
output: FAIL or PASS

1 for \(i \leftarrow 1\) to \(k\) do
2 \quad for \(j \leftarrow 0\) to \(m-2\) do
3 \quad \quad \(x_j \leftarrow \text{Rand}(0,2^n)\);
4 \quad \end{condition}
5 \quad \(i \leftarrow \text{Rand}(0,m-1)\);
6 \quad \text{if} \ \text{CrossSectionTest} ([x_j], i, P) = \text{FAIL} \text{ then}
7 \quad \quad \text{return FAIL;}
8 \quad \end{condition}
9 \end{for}
10 \text{return PASS;}

Algorithm 8: MULTISET
Lemma 4.4. If $P$ is correct on $1 - \epsilon$ fraction of inputs, Algorithm 8 returns FAIL with probability $\geq 3/4$.

Proof. Let $\gamma_i[x_j]$ represent the fraction of inputs $x \in D_i$ such that $P([x_j].append(x, i)) = f([x_j].append(x, i))$. Since the total fraction correct is $1 - \epsilon$, we see that $E_{[x_j]}[\gamma_i[x_j]] = 1 - \epsilon$ for all $i$. We see that by Markov's inequality, the probability taken over the choice of $[x_j]$ that $\gamma_i[x_j] < 7/8$ is $\geq 1 - \frac{8(1-\epsilon)}{7}$. By the proof of Theorem 3.1 we see that if $\gamma_i[x_j] < 7/8$, the probability that Test([x_j], i) returns FAIL is $\geq 3/4$. Therefore the probability that CrossSectionTest([x_j], i) returns FAIL is $\geq 3/4(1 - \frac{8(1-\epsilon)}{7}) = 6\epsilon/7 - 3/28$; since we run CrossSectionTest([x_j], i) several times, we see that this program returns FAIL with high probability, as desired.

Specifically, if we run the main program (or CrossSectionTest([x_j], i)) $k$ times, the probability of error is $\geq 1 - (6\epsilon/7 - 3/28)^k$. We therefore see that it is enough to pick $k$ to be a constant depending only on $\epsilon$ and not $m$ or $n$. 

We note that we can choose $k$ so that $k = O(1/\epsilon)$. In each call to CrossSectionTest, we have a constant number of queries and $O(n)$ additional computation time since we have set $k_1, k_2$ as constants. Therefore the number of queries to $P$ and additional computation time for the multilinearity self-tester will be $O(1/\epsilon), O(n/\epsilon)$ respectively.

Proof of Theorem 4.3. We see that if $P$ is correct on all inputs, Algorithm 8 returns PASS with probability 1. We note by Lemma 4.4 that if $P$ is $\epsilon$-far from $f$ that Algorithm 8 returns FAIL with probability $\geq 3/4$. We also noted above that the number of queries to $P$ is $O(1/\epsilon)$ and the additional computation time is $O(n/\epsilon)$. Therefore Algorithm 8 is a valid self-tester for $f$.

References

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

[2] M. Bellare, D. Coppersmith, J. Håstad, M. Kiwi, and M. Sudan. Linearity testing over characteristic two. *IEEE Transactions on Information Theory*, 42(6):1781–1795, 1996.

[3] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, pages 294–304, 1993.

[4] E. Ben-Sasson, M. Sudan, S. Vadhan, and A. Wigderson. Randomness-efficient low degree tests and short pcps via epsilon-biased sets. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 612–621, 2003.

[5] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *JCSS*, 47:549–595, 1993.

[6] Manuel Blum and Sampath Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, 1995.

[7] Martin Furer. Faster integer multiplication. *SIAM J. Comput*, 39(3):979–1005, 2009.

[8] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *JACM*, 45(4):653–750, 1998.

[9] J. Hastad and A. Wigderson. Simple analysis of graph tests for linearity and pcp. *Random Structures and Algorithms*, 22(2):139–160, 2003.
Appendices

A Checking Algorithm

We now describe an algorithm that checks whether the program is correct when multiplying two $n$-bit numbers $a, b$, rather than if the program is correct in general, in order to prove Theorem 3.2. The function RandSplit is defined as before:

```
input : An $n$-bit number $a$, a program $P_b$ computing multiplication by $b$
output: FAIL or PASS
1 if Test $(P_b) = FAIL$ then return FAIL;
2 return RandSplit ($a, P_b$)
```

Algorithm 9: Checker
Theorem A.1. Algorithm 9 correctly checks $a \cdot b$ for $a \in D_n$ and outputs FAIL if $P_b$ is incorrect on input $a$ with probability $\geq 3/4$.

Proof. It is clear that if the program always answers correctly on the domain $D_n$ then the checker will output CORRECT. There are now two cases where $P_b(a)$ is incorrect: the first is $P_b$ is 1/8-far from $f_b(x) = bx$ and the second is when it is 1/8-close. If it is 1/8-far from $f_b$, then we know from Theorem 3.1 that when we call Algorithm 2 it will return FAIL with probability $\geq 3/4$. If instead $P_b$ is 1/8-close to $f_b$, then the second test - RandSplit on $a$ in Line 2 - will be checking $P_b(x) + P_b(a-x) = P_b(a)$ for some $x$. Since $P_b$ is 1/8-close to $f_b$, we see that the probability that $P_b(x) \neq f_b(x)$ is $\leq 1/8$, and the same for the probability that $P_b(a-x) \neq f_b(a-x)$. Therefore by the union bound the probability that $P_b + P_b(a-x) = f_b + f_b(a-x) = f_b(a) \neq P_b(a)$ is $\geq 3/4$, so the second test will return FAIL with probability $\geq 3/4$, as desired.

We also note that the running time and queries of this algorithm are on the same order as those of Algorithm 2, or $O(n/\epsilon)$ additional computation time and $O(1/\epsilon)$ queries.

B Chernoff Bounds

We use Chernoff bounds often to describe the error probabilities of our algorithms. We use the following bound specifically, where $X_1, \ldots, X_n$ are random Bernoulli variables with expectation $p$ and $X = \sum X_i$:

$$Pr[X < (1-\delta)np] \leq e^{-\delta^2 np/2}$$

where $0 < \delta < 1$. The theorem we use for our algorithms is the following:

Theorem B.1. If the probability that a test correctly detects an error in $P_b$ is $p$, by running the test $6/p = O(1/p)$ times we will detect the error with probability 7/8.

Proof. Assume we run the test $n$ times. For the $i$th time we run the test, we let $X_i = 1$ if an error is detected and $X_i = 0$ otherwise. If we run the test $n$ times without detecting error, this means that $\sum X_i = X = 0 < 1$. By the Chernoff bound above, we see that by letting $\delta = (np - 1)/np$ that $Pr[X < 1] \leq e^{-(np-1)^2/(2np)}$. The probability that $X < 1$ is the same as the probability that we fail to detect an error, which we wish to be $\leq 1/8$. Therefore we want $e^{-(np-1)^2/(2np)} \leq 1/8$.

$$e^{-(np-1)^2/(2np)} \leq 1/8$$

$$-\frac{(np - 1)^2}{2np} \leq \log 1/8$$

$$\frac{(np - 1)^2}{np} \geq \log 64$$

If $np = 6$, then $\frac{(np - 1)^2}{np} = 25/6 > \log 64$, as desired. Therefore by letting $n = 6/p$, the probability that $X < 1$ is $\leq 1/8$, so we detect an error with probability 7/8.