Verifying Membership in NP-languages, or How to Avoid Reading Long Proofs

Leonid Reyzin*

April 1, 1996

1 Introduction

1.1 General Overview

The Classes NP, IP, MIP and PCP

The class NP was defined by Cook ([Coo71]) and independently by Levin ([Lev73]) as the class of languages that are accepted in non-deterministic polynomial time. If $L \in NP$ and $x \in L$, then there is a "proof" that $x \in L$ which can be verified in deterministic polynomial time (the "proof" consists merely of the choices made by the non-deterministic Turing machine). For example, for the problem of satisfiability of a Boolean formula, the "proof" can be a satisfying truth-assignment. Then it can be verified by simply plugging the truth-values into the formula.

The class IP is defined as the class of languages which have efficient interactive proofs of membership, and was introduced in [GMR89] and in [Bab85]. In the IP model for a language L, given input x, a verifier V is communicating with a prover P who is trying to convince V that $x \in L$. The Prover can have unlimited computational powers, whereas the Verifier is restricted to randomized polynomial-time computations. $L \in$IP if for all $x \in L$, the Verifier can be convinced that $x \in L$; and if $x \notin L$, then no Prover P' can convince V otherwise with a non-negligible probability. Building on the ideas of Lund, Fortnow, Karloff and Nisan ([LFKN92]), Shamir, in [Sha90], proved that IP=PSPACE. That is, the languages for which the membership can be established in polynomial space (and any amount of time) are the same as the languages for which membership can be interactively proved in polynomial time.

The IP model lead to the definition of the class MIP (for multi-prover interactive proofs) in [BGKW88]. In the MIP model, two or more provers that cannot communicate with each other interact with the Verifier. In [FRS88], MIP was proved equivalent to the following model.

Let M be a random polynomial-time Turing machine that can interact with an oracle O_x. The oracle is trying to convince M that $x \in L$. An oracle, unlike a prover, is memoryless,

*Harvard University, 426 Quincy Mail Center, Cambridge, MA 02138; reyzin@das.harvard.edu; 617-493-7669

i.e., its responses are fixed ahead of time. Thus, an oracle can be viewed as a written "proof," a bit-string any bit of which can be queried by the verifier. $L \in$MIP if and only if there is M such that if $x \in L$, then there exists O_x such that M will accept x; and if $x \notin L$, then for any oracle O', M will reject x with high probability. That is, a MIP language is such that membership proofs for it can be written down and then probabilistically verified (without any interaction) in polynomial time. Babai, Fortnow and Lund in [BFL91], building on ideas of [LFKN92], [BF91] and [Sha90], prove that MIP=NEXPTIME (where NEXPTIME is the class of languages that are acceptable in non-deterministic exponential time). This result demonstrates the power of randomness: if the polynomial-time Verifier is deterministic, then, by definition of NP, it can only verify membership proofs for languages in NP; however, if we allow the Verifier to be probabilistic and put up with a small probability of error, it can verify membership proofs for languages in a bigger class NEXPTIME (NP$\neq$NEXPTIME, as shown in [SFM78]).

In this paper, I concentrate on the described oracle model of MIP. Of interest in this model is the number of random bits used by M and the number of bits that M requests from the oracle. Hence, PCP$(r(k), q(k))$ (for probabilistically checkable proofs) model was introduced in [AS92], which is the same as the model described above, except that M can use no more than $O(r(k))$ random bits in its computation, and request no more than $O(q(k))$ bits from the oracle (where k is the length of the input x). There was much research devoted to establishing the place for NP in the PCP hierarchy. In [BFLS91], it was proven that NP=PCP$(\log k, (\log k)^c)$, for a small constant $c > 1$; moreover, the verifier of [BFLS91] runs in polylogarithmic time if the input is provided in a certain error-correcting code. Independently, in [FGL$^+$91], it was shown that NP$\subset$PCP$(\log k \log\log k, \log k \log\log k)$; the verifier of [FGL$^+$91] runs in polynomial time. Both papers built on the ideas contained in the MIP=NEXPTIME result of [BFL91], scaling the result down from NEXPTIME to NP.

The authors of [AS92], building on the previous results and introducing a new idea of recursive proof checking, showed that NP=PCP$(\log k, \sqrt{\log k})$; this result was improved upon in [ALM$^+$92], where it was shown that NP=PCP$(\log k, 1)$. This surprising result means that for any language $L \in$NP, and for any string x, a proof that $x \in L$ can be probabilistically verified by looking at only a constant number of bits from the proof, independent of the length of x. This result again demonstrates the power of randomness: unless NP-complete problems are deterministically decidable in subexponential time, a *deterministic* Verifier needs to read polynomially many bits of the proof (otherwise, the Verifier could, in subexponential time, generate all the possible proofs by itself to decide if $x \in L$). A *probabilistic* Verifier, on the other hand, using $O(\log k)$ random bits, needs to read only a constant number of bits of the proof.

PCP and Approximation Problems

The results about the MIP and PCP models had surprising implications for a number of NP-approximation problems. Take an NP-complete problem, such as, for example, *clique* (computing the size of a maximum complete subgraph for a given graph ([Kar72])), and ask how hard it is to approximate the result. The connection between PCP and the NP-approximation problems was first established in [FGL$^+$91]. The authors showed that ap-

proximating clique is "almost NP-complete." The authors of [ALM+92] (more specifically, Mario Szegedy and Madhu Sudan) extended the result a to whole class of problems known as MAXSNP ([PY91]), showing that they do not have polynomial-time approximation schemes unless P=NP.

A lower characterization of NP in the PCP hierarchy would improve some of these non-approximability results. However, it is unlikely that NP$\subset$PCP$(o(\log k), 1)$. Indeed, if that is the case, the Verifier can only possibly come up with $2^{o(\log k)} \subset o(k)$ different random strings, and for each string, it will read a constant number of bits from the proof. Thus, the number of relevant bits of the proof is $o(k)$, which means that all the possible proofs can be generated in $2^{o(k)}$ time and each can be then deterministically verified in polynomial time (since to verify a proof deterministically, we could just generate all the $o(k)$ random strings and do the verification for each string in polynomial time). Thus, this would imply that languages in NP can be deterministically decided in subexponential time.

To get better non-approximability results, a number of people worked on reducing the constants in the PCP$(\log k, 1)$ protocol and looking at other characteristics of these protocols, such as the proof length, reliability, and the so-called "free bits". They also developed new techniques for getting the non-approximability results. A host of papers followed: [BGLR93, BS92, Zuc93, FK94, BS94, LY94, KLS93, BGS95] and others. See [BGS95] for a good survey and some new results. Due to this work, for many NP-approximation problems the gap between the upper and lower bounds is closing.

1.2 Focus of This Paper

This paper brings together the ideas that lead up to the proof in [ALM+92] of the following theorem:

Theorem 1 *NP=PCP*$(\log k, 1)$.

However, due to time and space constraints, I do not present the proof of one of the steps needed to prove Theorem 1 (namely, a result of [ALM+92] that NP$\subset$PCP$(\text{poly}(k), 1)$), although I do show how to prove Theorem 1 assuming the missing step. Therefore, the lowest characterization of NP achieved in this paper is somewhat higher in the PCP hierarchy. It is due to [AS92] and [ALM+92]:

Theorem 2 *NP=PCP*$(\log k, \text{polylog} \log k)$.

The presentation combines the ideas of [BFL91, BFLS91, FGL+91, AS92, ALM+92]. In Section 2, I present some simple facts whose understanding is a prerequisite for understanding the rest of the paper. Section 3 is devoted to a technique due to [LFKN92], augmented by [BFL91], that was crucial to the MIP=NEXPTIME proof and its scaling down to NP. Understanding the basic ideas of this technique is necessary to get a clear picture of the non-recursive verification procedure, described in Section 4. The technique of recursive proof checking is described in Section 5, and requires an understanding of Section 4. While the result of Section 6 is necessary for the procedures to work and also of value as an independent result, the rest of the paper can be understood without knowing the ideas of Section 6—hence it is presented at the end.

2 Preliminaries

2.1 Definition of PCP

The following is the definition of PCP given by [AS92].

Let M be a random polynomial-time Turing machine that has random access to a string Π, known as "the membership proof"—M can query any bit of Π. We will call M a *verifier*, or a *randomized oracle machine*.

Definition 1 *A language L is in PCP$(r(k), q(k))$ if there exists a verifier M such that:*

- *If given an input x of length k, M uses no more than $O(r(k))$ random bits and queries no more than $O(q(k))$ bits from Π.*
- *If $x \in L$, then there is a proof Π_x such that M accepts with probability 1.*
- *If $x \in L$, then for any fixed proof Π', M rejects with probability greater than $1/2$.*

Note that by definition, NP=PCP$(0, poly(n))$. Also note that the value of $1/2$ for a rejection probability is not crucial—it could as well be $1/4$, for example. To get the probability of rejection greater, and thus improve the reliability of the Verifier, just repeat the Verification procedure a number of times. More specifically, to get rejection probability $1-\delta$, just repeat the procedure $\log(1/\delta)$ times.

It is clear from the definition of NP that

$$\mathrm{PCP}(\log k, \mathrm{poly}(k)) \subset \mathrm{NP}.$$

Indeed, if $L \in$PCP$(\log k, \mathrm{poly}(k))$, then the randomized verification procedure for $x \in L$ can be simulated in deterministic polynomial time by coming up with all the possible values of the random bits, thus giving us a deterministic, polynomial-time verification procedure for x. Hence, from this point on, it is all right to state that NP=PCP$(r(k), q(k))$ whenever it is proven that NP$\subset$PCP$(r(k), q(k))$, as long as $r(k) \in O(\log k)$ ($q(k)$ is always polynomial, since the Verifier can only take polynomial time).

2.2 Multivariate Lagrange Interpolation

Unless explicitly stated otherwise, by "degree" of a multivariate polynomial I mean its maximum degree in each variable.

Suppose we are given a field F, its subset $H \subset F$, and a function $f : H^m \to F$ for some m. Let $h = |H| - 1$.

Definition 2 *An m-variable polynomial $\hat{f} : F^m \to F$ is called a* degree-h extension *of f if $\hat{f}$ is of degree at most h and f agrees with $\hat{f}$ on $|H|^m$.*

Lemma 1 *Every function $f : H^m \to F$ has a unique degree-h extension $\hat{f}$ (where $h = |H|-1$). Given f, $\hat{f}(\overline{y})$ is computable in $O(m(h+1)^{m+1})$ operations over F for any arbitrary $\overline{y} \in H^m$.*

Proof This lemma is a simple extension of the well-known univariate Lagrange interpolation. Indeed, if $m = 1$, this lemma is the univariate Lagrange interpolation.

For $u \in H$, let L_u be the univariate polynomial of degree h such that $L_u(u) = 1$ and $L_u(y) = 0$ for $y \in H - \{u\}$:

$$L_u(y) = \left(\prod_{v \in H - \{u\}} (y - v) \right) \Big/ \left(\prod_{v \in H - \{u\}} (u - v) \right).$$

Note that for an arbitrary $y \in F$, $L_u(y)$ takes $O(h)$ operations over F to compute (simply using the defining formula).

Then, for $u_1, u_2, \ldots, u_m \in H$, let $M_{u_1,u_2,\ldots,u_m}$ be the m-variable polynomial of degree h such that $M_{u_1,u_2,\ldots,u_m}(u_1, u_2, \ldots, u_m) = 1$ and $M_{u_1,u_2,\ldots,u_m}(y_1, y_2, \ldots, y_m) = 0$ on the rest of H^m:

$$M_{u_1,u_2,\ldots,u_m}(y_1, y_2, \ldots, y_m) = \prod_{i=1}^{m} L_{u_i}(y_i).$$

Note that for a given m-tuple $(y_1, y_2, \ldots, y_m) \in F^m$, $M_{u_1,u_2,\ldots,u_m}(y_1, y_2, \ldots, y_m)$ takes $O(mh)$ operations over F to compute (using the defining formulas for L_u and $M_{u_1,u_2,\ldots,u_m}$).

Define $\hat{f}$ as follows:

$$\hat{f}(y_1, y_2, \ldots y_m) = \sum_{u_1,u_2,\ldots,u_m \in H} f(u_1, u_2, \ldots, u_m) M_{u_1,u_2,\ldots,u_m}(y_1, y_2, \ldots, y_m).$$

Clearly, $\hat{f}$ is a degree-h extension of f. To compute $\hat{f}(\overline{y})$ for arbitrary $\overline{y} \in F^m$, we need $(h+1)^m$ multiplications and evaluations of $M_{u_1,u_2,\ldots,u_m}$, so it takes $O(mh(h+1)^m)$ operations over F.

Uniqueness of $\hat{f}$ can be proved by a simple induction on m. Suppose, for $m = 1$, there are two degree-h extensions of f: g_1 and g_2. Then $g_1 - g_2$ is a univariate polynomial of degree at most h which has at least $|H| = h + 1$ roots, which is impossible unless $g_1 - g_2 = 0$. Thus, $g_1 = g_2$.

Suppose, degree-h extensions are unique for $m = k$. Then, for $m = k + 1$, let g_1 and g_2 be two degree-h extensions of f. If $g_1 \neq g_2$, then for some $y_1, y_2, \ldots, y_m \in F$, $g_1(y_1, y_2, \ldots, y_m) \neq g_2(y_1, y_2, \ldots, y_m)$. Let f', g_1' and g_2' by the k-variable polynomials obtained from f, g_1 and g_2, respectively, by fixing the m-th variable to y_m. We have that $g_1' \neq g_2'$, since $g_1'(y_1, y_2, \ldots, y_{m-1}) \neq g_2'(y_1, y_2, \ldots, y_{m-1})$. But g_1' and g_2' are both degree-h extensions of f', hence, by the inductive hypothesis, $g_1' = g_2'$, which is a contradiction. Hence, $g_1 = g_2$. $\square$

In particular, the above theorem implies that a function over $\{0,1\}^m$ can be extended to a multi-linear (of degree 1 in each variable) function over any field F.

Remark 1 *The result of Lemma 1 can be trivially extended to a function f defined on $H_1 \times H_2 \times \ldots \times H_m$, where $H_i \subset F$ and $|H_i| = h + 1$ for $1 \leq i \leq m$.*

2.3 Distance Between Functions

Definition 3 *Given a finite field F and two functions, $f : F^m \to F$ and $g : F^m \to F$, the distance Δ between them is defined as*

$$\Delta(f,g) = \frac{|\{\overline{x} \in F^m | f(x) \neq g(x)\}|}{|F^m|}.$$

*That is, $\Delta(f,g)$ is the fraction of the points on which f and g differ. Two functions f and g are δ-*close *if $\Delta(f,g) \leq \delta$. Define $\Delta_h(f)$ as the distance from f to the nearest polynomial of degree at most h.*

Clearly, Δ is a metric. A lemma due to Schwartz ([Sch80]) states that for two m-variable polynomials f and g of degree at most h, $\Delta(f,g) \geq 1 - mh/|F|$. Hence, if we pick a field $|F| > 2mh$, then two polynomials cannot be .5-close. Hence, by triangle inequality, if for a function f, $\Delta_h(f) < .25$, then there is a unique polynomial within .25 of f.

I will sometimes use the informal statement "f and g are close" to indicate that that they are δ-close for a small enough δ.

3 Vanishing Test

This section presents a result that plays a pivotal role in the verification procedure of Section 4. Suppose, we are given a polynomial f in m variables $x_1, x_2, \ldots, x_m$ over a field F; suppose further that the degree of the polynomial in each variable is no greater than d. We would like to verify that this polynomial vanishes on some subset of F^m. More specifically, let $H \subset F$; we would like to present a probabilistically checkable proof that shows that if $x_1, x_2, \ldots, x_m \in H$, then $f(x_1, x_2, \ldots, x_m) = 0$.

3.1 Sum-Check

First, I will describe a protocol to check a weaker condition. Namely, it will check that the sum over H^m of the values of f is equal to some number $c \in F$. While it may not seem useful at first, it is the major ingredient of the Vanishing Test.

The protocol itself was designed in [BFL91], and relies on the ideas contained in [LFKN92] (a similar protocol, although for a different purpose, was designed by [Sha90], and also relies on the ideas of [LFKN92]). This particular form of the protocol appears in [BFLS91] and [AS92].

Theorem 3 *Let F be a finite field, let $f : F^m \to F$ be a polynomial in m variables of degree at most d, and let $H \subset F$. Suppose further that $|F| > 2md$. Then there exists a randomized oracle machine that, given f and $c \in F$, verifies the statement*

$$\sum_{x_1 \in H} \sum_{x_2 \in H} \cdots \sum_{x_m \in H} f(x_1, x_2, \ldots, x_m) = c.$$

The machine uses $m \log |F|$ random bits, receives $m(d+1) \log |F|$ bits of information from the proof, and uses $O(|H|md)$ operations over F.

Remark 2 *A more precise definition of what we mean by "given f" is necessary. As we will see, the Verifier needs only one value of f, at one randomly chosen point. Thus, f can be provided in any way as long as the Verifier can reliably compute its value at one random point in the time allotted.*

Proof For $0 \leq i \leq m$, define a partial sum

$$f_i(x_1, x_2, \ldots, x_i) = \sum_{x_{i+1} \in H} \sum_{x_{i+2} \in H} \cdots \sum_{x_m \in H} f(x_1, x_2, \ldots, x_m).$$

Thus, $f = f_m$ and the task is to verify that $f_0 = c$. Note that for $0 \leq i < m$,

$$f_i(x_1, x_2, \ldots, x_i) = \sum_{h \in H} f_{i+1}(x_1, x_2, \ldots, x_i, h)$$

Also note that $f_i(a_1, a_2, \ldots, a_{i-1}, x_i)$ can be viewed as a univariate degree-d polynomial in x_i. The proof is required to contain the coefficients of all such polynomials. Of course, the Prover can be dishonest, and put incorrect coefficients in the proof. Let $g_i(a_1, a_2, \ldots, a_{i-1}, x_i)$ be the polynomial that the Prover puts in place of $f_i(a_1, a_2, \ldots, a_{i-1}, x_i)$. Define $g_0 = c$ (g_0 need not be part of the proof since it is given as the input to the Verifier).

The protocol is as follows. The Verifier picks m random elements $r_1, r_2, \ldots, r_m$ of F. The verification procedure takes $m + 1$ rounds. For $0 \leq i < m$, the Verifier tests that

$$g_i(r_1, r_2, \ldots, r_i) = \sum_{h \in H} g_{i+1}(r_1, r_2, \ldots, r_i, h).$$

In the last round, the Verifier performs the *final test*:

$$g_m(r_1, r_2, \ldots, r_m) = f(r_1, r_2, \ldots, r_m).$$

The Verifier accepts if and only if all the tests passed.

Clearly, if the sum is indeed c and the Prover is honest, the Verifier will accept. Now, suppose the sum is not c. That is, suppose $g_0 \neq f_0$. Then, in order for the first round to pass, the polynomial $g_1(x_1)$ cannot be equal to $f_1(x_1)$. If $g_1 \neq f_1$, then, as two polynomials of degree d, they cannot agree on more than d points. Thus, with probability at least $1 - d/|F|$, $f_1(r_1) \neq g_1(r_1)$. But in that case, in order for the second round to pass, the polynomials $g_2(r_1, x_2)$ and $f_2(r_1, x_2)$ cannot be equal. By the same reasoning, if $g_{i-1}(r_1, r_2, \ldots, r_{i-2}, x_{i-1})$ is not equal to the polynomial $f_{i-1}(r_1, r_2, \ldots, r_{i-2}, x_{i-1})$, then with probability at least $1 - d/|F|$, $g_{i-1}(r_1, r_2, \ldots, r_{i-2}, r_{i-1}) \neq f_{i-1}(r_1, r_2, \ldots, r_{i-2}, r_{i-1})$, and therefore, in order for the i'th round to pass, the polynomials $g_i(r_1, r_2, \ldots, r_{i-1}, x_i)$ and $f_i(r_1, r_2, \ldots, r_{i-1}, x_i)$ can not be equal. Thus, a lie in each round most likely leads to a lie in the next round, and therefore if the rounds 0 through $m - 1$ pass, with probability at least $1 - (m-1)d/|F|$, the polynomials $g_m(r_1, r_2, \ldots, r_{m-1}, x_m)$ and $f_m(r_1, r_2, \ldots, r_{m-1}, x_m)$ are not equal. But this will be discovered with probability at least $1 - d/|F|$ by the final test (because $f_m(r_1, r_2, \ldots, r_{m-1}, x_m)$ and $g_m(r_1, r_2, \ldots, r_{m-1}, x_m)$ cannot agree on more than d points). Thus, the total probability of discovering an error is at least $1 - md/|F| > 1 - md/(2md) = 1/2$.

Random bits are used to come up with m random elements of F, and hence we need $m \log |F|$ of them. The query bits are used for coefficients of m degree-d polynomials, so

$m(d+1)\log|F|$ of them are needed. The verification consists of computing $|H|+1$ values of each of the polynomials (and $|H|-1$ additions for each of the first m rounds), and hence takes $O(|H|md)$ operations over $|F|$. $\square$

Note that as the protocol is described above, the number of random bits the Verifier needs grows with the size of F. Since this is sometimes undesirable, I will modify the protocol slightly to make the number of random bits independent of $|F|$. Namely, the Verifier chooses a subset $I \subset F$ such that $H \subset I$ and $|I| = 2md+1$. The random choices $r_1, r_2, \ldots, r_m$ are made from I rather than all of F. The rest of the proof works with no changes.

Corollary 1 *In Theorem 3, the number of random bits used can be changed to* $m\log(2md+1) \in O(m\log md)$.

Remark 3 *If, for* $1 \le i \le m$, $H_i \subset F$, *the protocol can be trivially extended to test that*

$$\sum_{x_1 \in H_1} \sum_{x_2 \in H_2} \cdots \sum_{x_m \in H_m} f(x_1, x_2, \ldots, x_m) = c.$$

3.2 From Sum-Check to Vanishing Test

Now, knowing the sum-check protocol, I describe how to verify that a polynomial f vanishes over H. There has been a number of ways proposed to turn the above protocol into a Vanishing Test. If working over the integers or the rationals, one could just perform the Sum-Check test for the polynomial f^2—if

$$\sum_{x_1 \in H} \sum_{x_2 \in H} \cdots \sum_{x_m \in H} (f(x_1, x_2, \ldots, x_m))^2 = 0,$$

then f vanishes over H (this idea appears in [BFL91]). However, it does not work over finite fields. The following is due to [BFL91], with slight modifications by [BFLS91].

Here is the rough idea. Suppose, f is not identically 0 on H^m. Let $R = \{R_t\}$ be a family of polynomials. Pick a random polynomial from this family. If the family is well-chosen, it is likely that

$$\sum_{\overline{x} \in H^m} R_t(\overline{x}) f(\overline{x}) \neq 0$$

(since for at least one $\overline{x} \in H^m$, $f(\overline{x})$ is non-zero). If all the polynomials in R are of degree at most d_1 in each variable, then the degree of Rf is at most $d+d_1$, and the protocol from the previous section can be applied to Rf if we replace d by $d+d_1$. Of course, the proof-string would have to contain the proof for each possible choice of R_t by the Verifier.

So, we need to find such a family R. Assume for now that $H = [0..h-1]$. Then $\overline{x} = (x_1, x_2, \ldots, x_m)$ can be viewed as a number base h. Let E be an extension of the field F of size at least $4h^m$. Consider the polynomial

$$g(t) = \sum_{\overline{x} \in H^m} f(\overline{x}) t^{\overline{x}}.$$

The polynomial is of degree at most h^m, so unless it is identically 0, for at least three quarters of $t \in E$, $g(t) \neq 0$. If we check that $g(t) = 0$ at a random location, we can be assured with probability at least 3/4 that $f(\overline{x}) = 0$ for all $\overline{x} \in H^m$.

We now need to exhibit a polynomial $R_t : F^m \to E$ that agrees with the function $t^{\overline{x}}$ for $\overline{x} \in H^m$. By multivariate Lagrange interpolation, there is such a polynomial, and it can be made of degree at most $h-1$ in each variable. The problem is that Lagrange interpolation takes too many operations to compute (and in the sum-check protocol, the Verifier needs to compute a value of the polynomial). But $t^{\overline{x}}$ is a special function, and happens to have a faster interpolation method.

More specifically, given $t \in E$, let $t_i = t^{h^i}$. Then

$$t^{\overline{x}} = \prod_{i=1}^{m-1} t_i^{x_{m-i}}.$$

Let L_u be the degree-$(h-1)$ univariate Lagrange interpolation polynomial over H: namely $L_u(v) = 0$ if $v \neq u$ and $L_u(v) = 1$ if $v = u$. Then the above can be re-written as

$$t^{\overline{x}} = \prod_{i=1}^{m-1} \left(\sum_{v=0}^{h-1} t_i^v L_v\left(x_{m-i}\right) \right).$$

Thus, the protocol is as follows. The field $E \supset F$ is chosen in advance. The Verifier chooses a random $t \in E$ and then uses the sum-check protocol for the $(h-1+d)$-degree polynomial $f(\overline{x})R_t(\overline{x})$ (where

$$R_t(\overline{x}) = \prod_{i=1}^{m-1} \left(\sum_{v=0}^{h-1} t_i^v L_v\left(x_{m-i}\right) \right)$$

and is defined over all of E).

An error in testing can come from a bad pick of t or from the error in the sum-check protocol. The probability of a bad pick of t is at most $1/4$; to get the overall error probability under $1/2$, in the sum-check protocol, the random values are selected from a subset of F of size at least $4m(h-1+d)+1$ (rather than $2m(h-1+d)+1$, as required by Corollary 1).

Note that t does not need to be chosen from all of E, but only from a large enough subset of it, of size $4h^m$. The choice of t thus requires $m \log 4h$ random bits; and the sum-check protocol requires $m \log(4m(h-1+d)+1)$ random bits; thus, the total number of random bits is in $O(m \log m(d+h))$.

Since the values of the polynomial are now in the field E, the number of bits queried from the proof is $m(d+h)\log|E|$. Note that the value of $R_t(\overline{x})$ can be computed in $O(mh^2)$ operations over E.

What happens if $H \neq [0..h-1]$? Let $|H| = h$. Let $\sigma : [0..h-1] \to H$ be a bijection; let $\rho = \sigma^{-1}$. Then in the formula for R_t, replace L_v with $L_{\sigma(v)}$. The same proof works, since $R_t(x_1, x_2, \ldots, x_m)$ computes $t^{\overline{\rho(x_1)\rho(x_2)\ldots\rho(x_m)}}$.

Thus, I have proved the following theorem.

Theorem 4 *Let F be a finite field, let $f : F^m \to F$ be a polynomial in m variables of degree at most d, and let $H \subset F$. Suppose further that $|F| > 4m(|H|-1+d)$. Then there exists a randomized oracle machine that, given f, verifies the statement*

$$(\forall \overline{x} \in H^m) f(\overline{x}) = 0.$$

The machine uses $O(m \log m(|H| + d))$ random bits, reads $m(d + |H|) \log |E|$ bits from the proof, and uses $O(m(|H|d + |H|^2))$ operations over E (where E is an extension of F of size at least $4|H|^m$).

The authors of [FGL+91] suggest using a projection of $t^x \in E$ into F in order to avoid working in a larger field E. They do not provide the details of their construction, however.

4 The Procedure

In this section, I prove the following theorem:

Theorem 5 *NP=PCP*$(\log k, \log^3 k / \log \log k)$

My proof combines the ideas of [BFLS91, FGL+91, AS92]. In Section 4.5, I show how to extend the result of Theorem 5 using the ideas of [BFLS91]. A part of the extended result is essential for the recursive proof checking, described in Section 5.

4.1 Arithmetization of Boolean Formulas

As was noticed in [BF91] and [Sha90], it is often beneficial to consider Boolean formulas as arithmetic, rather than Boolean, expressions. This often allows us to rely on properties of polynomials when verifying statements about formulas. In particular, [BF91] and [Sha90] use arithmetization of Boolean formulas to prove that IP=PSPACE, and [BFL91] uses it to prove that MIP=NEXPTIME.

The authors of [BFL91] define arithmetization as follows:

Definition 4 *A polynomial $f(x_1, \ldots, x_n)$ is said to be an* arithmetization *of a Boolean function $B : \{0,1\}^n \to \{0,1\}$ if on all $(0,1)$-substitutions, the (Boolean) value of B and the (arithmetic) value of f agree.*

The advantage of this definition is that if a Boolean function is given by a Boolean formula, it is easy to arithmetize: first one needs to eliminate disjunctions using DeMorgan's laws, then replace every conjunction with a multiplication and every negation of a subformula of the form $\neg s$ with $(1 - s)$. It is clear that the result satisfies the definition. Moreover, if the original Boolean formula was of length l, then the resulting polynomial is of total degree at most $l/2$ (since the total degree is no more than the number of conjunctions and disjunctions plus one). I will use this method of arithmetization later, in Section 4.5.

For now, however, I will need a different method of representing Boolean formulas with arithmetic ones. The reason is that Definition 4 only allows us to represent B as a polynomial of n variables. Thus, computations with it will have to take at least n operations. It is unlikely to work when we are trying to produce algorithms that have sub-linear constraints.

This problem is addressed in similar ways in [BFLS91] and [FGL+91]. The following is the observation of [FGL+91].

Let φ be a 3-CNF formula of no more than n clauses and no more than n variables. Then every clause and every variable can be represented by numbers from $[0..n-1]$, or by $\log n$

bits. Let $m = \log n$. The formula, then, can be completely described by the following family of functions: $\chi_j : \{0,1\}^{2m} \to \{0,1\}$ takes two m-digit numbers, c and v, in binary notation, and outputs 1 if and only if the variable number v is the j-th variable of the clause number c; $s_j : \{0,1\}^m \to \{0,1\}$ takes a number c in binary notation and outputs 1 if and only if the j-th variable of clause c is unnegated (since we are dealing with a 3-CNF, $j = 1,2,3$).

Thus, through a sort of indirect referencing of variables and clauses, we described the formula completely with 6 functions of $O(\log n)$ Boolean variables each. Now, let $A : \{0,1\}^m \to \{0,1\}$ be a truth assignment: it takes v and assigns 0 or 1 to the variable number v. Then A satisfies φ if and only if for all $c, v_1, v_2, v_3 \in \{0,1\}^m$,

$$\prod_{j=1}^{3} \chi_j(c, v_j)(s_j(c) - A(v_j)) = 0.$$

By Lemma 1, the functions χ_j, s_j and A can be extended to multi-linear polynomials over any field F. Let their extensions be $\hat{\chi_j}, \hat{s_j}$ and $\hat{A}$. We can then consider the polynomial in $4m$ variables over F, of degree at most 6 in each variable:

$$\varphi^A(c, v_1, v_2, v_3) = \prod_{j=1}^{3} \hat{\chi_j}(c, v_j)(\hat{s_j}(c) - \hat{A}(v_j))$$

(c, v_1, v_2 and v_3 are vectors of m elements each).

We have thus reduced the question of A being a satisfying assignment for a formula to the question of whether a certain low-degree multivariate polynomial in $\log n$ variables vanishes on all $\{0,1\}$ substitutions. We can now apply the result of Section 3 in order to test if φ^A indeed vanishes on $\{0,1\}^{4m}$, provided we are able to compute φ^A and one random point over F^{4m}. We have to pick F to be of size greater than $4 \cdot 4m \cdot (2 - 1 + 6) = 112 \log n$, and E to be its extension of size at least $4 \cdot 2^{4m} = 4n^4$. The protocol will use $O(\log n \log\log n)$ random bits and $O(\log^2 n)$ bits from the proof.

4.2 Using Systems Other than Binary

The reduction performed in the previous section was possible because we used binary notation to name clauses and variables in a Boolean formula. The authors of [BFLS91] suggested using a different base for the encoding.

Let h and m be such that $(h+1)^m \geq n$ (where n is still the number of variables and clauses in φ). In the previous section, h was equal to 1 and m was equal to $\log n$. We define the functions $\chi_j(c, v)$, $s_j(c)$ and $A(v)$ just as before, except that instead of taking $\log n$ $\{0,1\}$-variables to identify c and v, they now take using m $[0..h]$-variables. Then, by multivariate Lagrange interpolation they can be extended to any field F ($|F| > h$) to become polynomials $\hat{\chi_j}$, $\hat{s_j}$ and $\hat{A}$ of degree h (we have to identify some subset of F with $[0..h]$). If we define $\varphi^A(c, v_1, v_2, v_3)$ as above in terms of $\hat{\chi_j}$, $\hat{s_j}$ and $\hat{A}$, we will get that φ^A is a polynomial in $4m$ variables of degree at most $6h$. It vanishes on all $[0..h]$ substitutions if and only if A is a satisfying assignment for φ.

Just as above, we can apply the result of Section 3 in order to test if φ^A indeed vanishes on $[0..h]^{4m}$, provided we are able to compute φ^A and one random point over F^{4m}. We have

to pick F to be of size greater than $4 \cdot 4m \cdot ((h+1) - 1 + 6h) = 112mh$, and E to be its extension of size at least $4 \cdot h^{4m} \leq 4n^4$. The protocol will use $O(m \log mh)$ random bits and $O(mh \log n)$ bits from the proof.

Varying m and h allows some flexibility in the size of F and the number of random and query bits. In particular, if we set $m = \log n / \log\log n$ and $h = \log n$ (in which case $(h+1)^m > (\log n)^{\log n / \log\log n} = n$), we will need only $O(\log n)$ random bits and $O(\log^3 n / \log\log n)$ query bits.

4.3 The Protocol

Given a language $L \in NP$, we need to construct a Verifier that, for a given x, verifies the proof that $x \in L$. Without loss of generality we can assume that the alphabet for L is $\{0,1\}$ and hence $x \in \{0,1\}^k$. Let $x_1, x_2, \ldots, x_k$ be the bits of x.

From the proof of the Cook-Levin Theorem ([Coo71], [Lev73]), it follows that for every k, there exists a polynomial-time constructible 3-CNF formula $\varphi_k(z_1, z_2, \ldots, z_n)$ such that $x \in L$ if and only if there exists a truth assignment A such that

$$A(z_1) = x_1, A(z_2) = x_2, \ldots, A(z_k) = x_k$$

and

$$\varphi_k(A(z_1), A(z_2), \ldots, A(z_n)) = 1$$

(note that n is polynomial in k).

We can view A as a function $A : [0..h]^m \to \{0,1\}$. We require the prover to provide the table of values of $\hat{A}$. Suppose, the prover instead provides the table of a function $B : F^m \to F$ (if the prover is honest, then $B = \hat{A}$). Then, we need to verify three things:

1. B is equal to a degree-h polynomial $\hat{A}$ over F
2. φ^A vanishes on $[0..h]^{4m}$
3. $\hat{A}(i) - x_i = 0$ for $1 \leq i \leq k$

We need the first condition in order to be able to apply the result of Section 3 for the verification of the second condition (because the result relies on polynomiality of φ^A which relies on polynomiality of $\hat{A}$). Clearly, however, the first condition is impossible to verify without reading the entire table for B (since an error in a single entry could make B not equal to a degree-h polynomial), which takes too long. We need a better version of the condition.

Note that we will use the table of values of B only three times in the verification of the second condition—in order to evaluate φ^A at one random point. Thus, if up to 10% of the values of B are incorrect, the chance of getting the correct value for φ^A is still at least 70%. Hence, if φ^A does not vanish, the Vanishing Test will still detect it with probability at least 35%. Running the Vanishing Test a small constant number of times (or increasing the reliability of the test by making F bigger), we can get any desired constant reliability.

In view of the above, we can replace the first condition with the following:

1. $\Delta_h(B) \leq .1$

This can be verified by the so-called "Low-Degree" test, which is the subject of Section 6.1.

The third condition can be verified similarly to the second one. If we assume that $k = (h+1)^l$ for some l, and view x as a function $x : [0..h]^l \to \{0,1\}$, then we can construct its degree-h extension $\hat{x}$ over F^l. In that case, the verification that $\hat{A}(i) - x_i = 0$ is merely a vanishing test for a polynomial in l variables of degree h in each. If $k \neq (h+1)^l$ for some l, then let $l = \lceil \log_{h+1} k \rceil$ and pad x with zeroes to get the length of $(h+1)^l$.

Remark 4 *Note that $\hat{A}$ is meant to be an extension of a* Boolean *function over $[0..h]^m$. However, there is no need to verify that its values are Boolean on $[0..h]^m$. Indeed, if $\hat{A}(v) \notin \{0,1\}$ for some $v \in [0..h]^m$, and φ^A still vanishes on $[0..h]^m$, then the assignment A is satisfying regardless of the value of the variable v (a simple examination of the construction of φ^A shows this).*

4.4 Complexity of the Protocol

In order for the Low-Degree Test of the first condition to work, we need F to be of the size at least $c(m^2h^3)$ for some constant c (see Section 6.1 for details). This F will also work for the Vanishing Tests of the second and third conditions (since it only needs to be of size bigger than $112mh$ for the Vanishing Test of φ^A and even less for the Vanishing Test of $\hat{A}(i) - x_i$. In order for the Vanishing Tests to work, we pick E to be an extension of F of size at least $4n^4$.

Then, the protocol uses $O(m \log mh)$ random bits for the two Vanishing Tests and the same for the Low-Degree Test. Each Vanishing Test requires $O(mh \log n)$ bits of the proof; the Low-Degree test requires only $O(mh \log |F|) = O(mh \log mh)$ bits.

Aside from computing the value of the function at a random point, each Vanishing Test uses $O(mh^2)$ operations over E; the Low-Degree test also uses poly(mh)) operations. It is computing the values of φ^A and $\hat{A}(i) - \hat{x}(i)$ at a random point that takes the most time. Indeed, if we are given just the input x and the functions χ_j and s_j, we need $O(m(h+1)^{m+1}) = O(mhn)$ operations to compute their extensions at one random point.

In order to get Theorem 5, just set $m = \log n / \log \log n$ and $h = \log n$, and remember that n is polynomial in the length of input k.

4.5 Making the Procedure Faster

Note that most of the time in the verification procedure is spent computing extensions of functions. The authors of [BFLS91] address this problem. Their method used to reduce the time is based on the following observations:

- If a Boolean function is given by a Boolean formula, then it is easy to arithmetize it according to Definition 4.
- If a function over H^m is given by a polynomial formula, it is easy to compute its extension over F.

- The input x is only used to construct its extension $\hat{x}$.

The specifics of how to apply these observation are described below.

For now, I will again be using the binary system to encode the names of variables and clauses in φ (i.e., $h = 1$ and $m = \log n$). As described above, the formula φ and hence the functions χ_j and s_j depend only on the language L and the length k of the input x. Note that if we are using the binary system, then χ_j and s_j are Boolean functions of $O(\log k)$ variables (since n is polynomial in k). Suppose that the language L is such that the Boolean formulas for χ_j and s_j can be computed in time polylogarithmic in k. Then the formulas are of length at most polylogarithmic in k; hence, they can be arithmetized by the method of Definition 4 to become polynomials p_{χ_j} and p_{s_j} of total degree at most polylog$k$.

Thus, the functions χ_j and s_j are given over $\{0,1\}^{2m}$ by polynomial formulas. Then their extensions over F are also given by these polynomial formulas, except that the extensions are of degree at most polylog$k$ in each variable rather than multi-linear. Using these extensions rather than $\hat{\chi_j}$ and $\hat{s_j}$ slows down the Vanishing and the Low-Degree tests, but speeds up the computation of the extensions dramatically.

Now we go back to working in base $h+1$. We will have to assume $h+1$ is a power of 2, i.e., $h+1 = 2^l$ for some l. In that case, every base-$(h+1)$ digit $d \in [0..h]$ is representing l binary digits. Let $\pi_j(d)$ $(1 \leq j \leq l)$ be equal to the j-th digit in the binary representation of d. Then $\pi_j(d)$ can be represented by a univariate degree-h polynomial, whose value at a point is computable in time $O(h^2)$ (simply by univariate Lagrange interpolation).

Then, $s_j(v_1, v_2, \ldots, v_m)$ (where $v_i \in [0..h]$) is equal to

$$p_{s_j}(\pi_1(v_1), \pi_2(v_1), \ldots \pi_l(v_1), \pi_1(v_2), \pi_2(v_2), \ldots, \pi_l(v_2), \ldots, \pi_1(v_m), \pi_2(v_m), \ldots, \pi_l(v_m)).$$

We thus represented s_j explicitly by a polynomial of degree at most $h$polylog$k$. We can now use this polynomial to extend s_j over all of F, except that the extension is not of degree h, but rather of degree $h$polylog$k$. However, it takes only $O(h^2 \log k + \text{polylog}k)$ operations to compute its value at a point ($O(h^2)$ operations each for $\pi_j(v_i)$, and there are $\log n = O(\log k)$ of those; plus the number of operations required to compute p_{s_j}).

The same method works for χ_j. However, it does not work for computing $\hat{x}$ if we are given just x. But if we stipulate that the input be provided as its extension $\hat{x}$ rather than x, then we do not need to construct $\hat{x}$. In fact, we do not even need to read the whole input, since it is only used to evaluate $\hat{x}(i)$ at a few random points. If the input is not trustworthy, we can first use the Low-Degree Test to verify that it differs from a polynomial on no more than, say, 10% of the points (in which case such a polynomial is unique—see Section 2.3). In fact the extension of the input can be stored as a part of the proof, in which case the Verifier, after running the Low-Degree Test, is assured that there is a unique string x such that $\hat{x}$ differs on more that 10% of the points from what is stored in the proof. Since both the Vanishing Test and the Low-Degree Test use only a constant number of values of $\hat{x}$, the whole procedure will only read a constant number of values of the input (note, however, that each value takes more than a constant (namely, $\log |F|$) number of bits).

Thus, we can use the same protocol as above, except that we work with p_{χ_j} and p_{s_j} rather than $\hat{\chi_j}$ and $\hat{s_j}$. Of course, the field F would have to be bigger in order for the Low-Degree and Vanishing Test to work (E can stay the same as long as it is still an extension

of F). The protocol will then use $O(m \log(mh \log k))$ random bits for each of the Vanishing Tests and the Low-Degree Tests; the number of bits from the proof is $O(m(h\text{polylog}k) \log n)$ for the the Vanishing Tests and $O(m(h\text{polylog}k) \log(mh \log k))$ for the Low-Degree Tests. Aside from computing the value of φ^A at one random point, the Vanishing Tests will use $O(mh^2\text{polylog}k)$ operations over E, and the Low-Degree Tests will use $O(\text{poly}(mh \log k))$ operations. Computing a value of φ^A now takes only $O(h^2\text{polylog}k)$ operations.

If we now set $m = \log n / \log \log n$, and $h = \log n$ (and remember that n is polynomial in k), we get the following extension of Theorem 5.

Theorem 6 *If the input x is given to the Verifier as its degree-h extension $\hat{x}$, then the Verifier needs to read only a constant number of the values of the input. Suppose further that L is such that given the length k of x, the reduction to 3-SAT (in the form of the Boolean formulas for χ_j and s_j) can be computed in time polylogarithmic in k. Then the Verification procedure runs in time polylogarithmic in k. The number of query bits goes up to $O(\text{polylog}k)$.*

5 Recursive Proof Checking

The idea of recursive proof checking was introduced in [AS92], and generalized in [ALM+92]. In this section I present it along the lines of [ALM+92].

5.1 The Idea of Recursion

In the verification procedure of Section 4 the bits of proof that are being queried depend only on the random bits selected by the Verifier (i.e., the locations of the bits queried do not depend on the results of the previous queries). That is, once a random string r is selected by the Verifier, the Verifier can query the appropriate bits of the proof, get a string y_r, perform a polynomial-time computation on x, r and y_r and then decide whether to accept or reject. Thus, for each x and r, we can consider a language $L_{x,r}$ of all such y_r that the Verifier will accept. The idea of recursion is to run a verification procedure for $L_{x,r}$ rather than for L.

The question of membership in $L_{x,r}$ is in P, since the computation involving the query bits is polynomial in their number. Hence, $L_{x,r} \in$NP, and hence there is a proof for it that can be verified with $O(\log(|y_r|))$ random bits and $O(\text{polylog}(|y_r|))$ query bits. Note that $|y_r| = O(\text{polylog}|x|)$, so the number of random and query bits is small. The new (recursive) proof will contain, for each possible random string r selected by the Verifier, the degree-h extension $\hat{y_r}$ of y_r as well as the proof that $y_r \in L_{x,r}$. Then the Verifier selects a random string r, chooses the appropriate segment of the new proof, and verifies (using the protocol of Section 4.5) that indeed $y_r \in L_{x,r}$

One problem with the above method is that there is nothing forcing the Prover to present y_r's that are consistent with each other. That is, in the non-recursive proof, y_{r_1} and y_{r_2} can have some bits in common. In the recursive version as presented above, however, the proofs for r_1 and r_2 are disjoint, and hence y_{r_1} and y_{r_2} can be inconsistent. It is thus possible for the Prover to "fool" the Verifier by presenting a recursive "proof" for each r, even though no non-recursive proof exists. This problem is addressed below.

In Section 5.4 I show how the proof required by the verification procedure of Section 4 can be transformed into the following form: it will consist of l disjoint segments $y_1, y_2, \ldots, y_l$, of size $O(\text{polylog}k)$ each, such that the Verifier reads a constant number of segments in the verification procedure. Given a random string r, the Verifier of Section 4 will read the segments $y_{r,1}, y_{r,2}, \ldots, y_{r,c}$ for some constant c (for notational convenience, when it creates no ambiguity, I will use $y_1, y_2, \ldots, y_c$ for $y_{r,1}, y_{r,2}, \ldots, y_{r,c}$).

Also note that the method of encoding the input and making it be a part of the proof, presented in Section 4.5, can be extended as follows: we can break up the input into a constant number of pieces $x^1, x^2, \ldots, x^c$ and present the degree-h extension of each of those pieces separately. Then the Vanishing Test for $\hat{A}(i) - x_i$ will become c Vanishing Tests: for $\hat{A}(i) - x_i^1, \hat{A}(i) - x_i^2, \ldots, \hat{A}(i) - x_i^c$; and if we conduct c Low-Degree Tests, one for each $\hat{x^j}$, then we can be assured that there is a unique collection set of segments $x^1, x^2, \ldots, x^c$ such that what is stored in the proof is close to their degree-h extensions, and their concatenation x is in the language L (we may have to conduct those test a small constant number of times in order to get the failure probability under $1/2$).

Using the two observations above, we now make the structure of the recursive proof as follows. It contains, for each segment y_i, $1 \leq i \leq l$, its degree-h extension $\hat{y_i}$. It also contains, for each r selected by the Verifier, the proof that the concatenation of segments $y_{r,1}, y_{r,2}, \ldots, y_{r,c}$ is in the language $L_{x,r}$.

The verification procedure is clear: the verifier picks a random string r, constructs, accordingly, the verification procedure for the language $L_{x,r}$, and runs it on what the Prover claims are degree-h extensions of $y_{r,1}, y_{r,2}, \ldots, y_{r,c}$. An honest Prover will clearly be able to come up with a proof that is always accepted. Suppose now that $x \notin L$. Then, for any proof y, for more than half of the random strings r, the concatenation of $y_{r,1}, y_{r,2}, \ldots, y_{r,c}$ will not be accepted by the Verifier of Section 4; i.e., it is not in $L_{x,r}$. In that case, the verification procedure for $L_{x,r}$ will reject with probability $1/2$. Thus, a total probability of rejecting if $x \notin L$ is $1/4$. We can run the verification procedure 3 times to get the probability of rejecting over $1/2$.

5.2 Recursion Generalized

Let us consider again what we need in order for the recursion procedure to work. First of all, the proof and the verification procedure for the statement $x \in L$ have to comply with certain restrictions: the proof has to consist of a number of disjoint segments, each of which is of size $O(q(k))$ (where the number of query bits is $q(k)$); given a random string r, the Verifier has to first select a constant number of segments to verify (independent of the contents of the proof), and then perform a computation on the segments, r and x in order to decide whether to accept or reject. Also, any computation involving the proof segments has to be polynomial in their size (another way to put it is that the Verifier can spend time polynomial in $|x|$ before getting the segments, but can only spend time polynomial in $q(k)$ afterwards). Let us call the subclass of PCP$(r(k), q(k))$ for which the proofs and the verification procedures have this property PCP-SEG$(r(k), q(k))$ (the authors of [ALM+92] use OPT$(r(k), q(k))$ for this subclass).

Second, the language $L_{x,r}$, consisting of tuples $(y_1, y_2, \ldots, y_c)$, has to be in PCP$(s(k), t(k))$

(now k is not the length of x, but the length of the tuple). Moreover, if the segments are provided in a certain encoded form $y'_1, y'_2, \ldots, y'_c$, there is a way to verify that there are unique strings $y_1, y_2, \ldots, y_c$ such that y'_i is close to $\hat{y_i}$, and $(y_1, y_2, \ldots, y_c) \in L_{x,r}$, without using more than $O(s(k))$ query bits and more than $O(t(k))$ random bits. Also, no more than a constant number of values of $\hat{y_i}$ is read from each segment. Let us call a subclass of PCP that has this property PCP-ENC$(s(k), t(k))$.

The recursive verification procedure, thus, takes a language from PCP-SEG, stipulates that all of its proof segments be encoded, comes up with a random r and uses the PCP-ENC-type procedure to verify that the string $(y_{r,1}, y_{r,2}, \ldots, y_{r,c})$ is acceptable for the PCP-SEG verifier. Note that the roles of PCP-ENC and PCP-SEG here are different: the procedure itself runs only a PCP-ENC-type verifier, and not the PCP-SEG-type. The PCP-SEG-type verifier is not run, but only used to construct the PCP-ENC verifier for the language of c-tuples. Thus, this is not recursion in the strict sense of the word, since we use different types of verifiers, and for different purposes. However, their similarity justifies the term.

Given a language $L \in$PCP-SEG$(r(k), q(k))$, we need that the language $L_{x,r}$ of c-tuples be in PCP-ENC$(s(l), t(l))$, where l is the number of query bits for L. The language is clearly in P (because of the condition on PCP-SEG that the computation with the query bits be polynomial in their number), hence a sufficient condition would be that P$\subset$PCP-ENC$(s(l), t(l))$ (clearly, P$\in$PCP$(0, 0)$, since no proof is needed at all; however, remember that in PCP-ENC is a restricted subclass of PCP).

In view of the above, I state the following theorem:

Theorem 7 *Suppose, P$\subset$PCP-ENC$(s(k), t(k))$. Let L be in PCP-SEG$(r(k), q(k))$. Then there exists a constant d such that $L \in$PCP$(r(k) + s(q(k)^d), t(q(k)^d))$.*

Proof Use the recursive procedure, as described above. The only things I need to show are the polynomiality of the procedure and how to get the bounds on the random and query bits.

Let x be the input, and r be the random string. Define $L_{x,r}$ as above. We need to *construct*, in time polynomial in $|x|$, a verification procedure for $L_{x,r}$ (it cannot be "programmed" in advance, since x can be of any size). There are multiple ways to do so; one of them is to reduce the question of membership in $L_{x,r}$ to a question of membership in some single canonical language, for which the verification procedure is known in advance.

Specifically, let K be the language of $c+1$-tuples such that the first element of the tuple is a c-input circuit C. A tuple $(C, y_1, y_2, \ldots, y_c)$ is in the language if C accepts the inputs $(y_1, y_2, \ldots, y_c)$. Clearly, the language K is in P; hence, $K \in$PCP-ENC$(s(k), t(k))$. Note that given a verification procedure for L, the question of membership in $L_{x,r}$ can be answered by a circuit $C_{x,r}$, constructible in time polynomial in $|x|$ (since the verification procedure for L runs in time polynomial in x). The circuit itself has size polynomial in $q(k)$ (because of a condition imposed on PCP-SEG). Thus, the question of membership in $L_{x,r}$ becomes a question of membership of the tuple $(C_{x,r}, y_1, y_2, \ldots, y_c)$ in the language K. The size of the tuple is polynomial in $q(|x|)$, i.e., $q(|x|)^d$ for some d.

Therefore, the input to the PCP-ENC-type verifier is of length $q(|x|)^d$. Recall that by definition $k = |x|$. Thus, the PCP-ENC-type verifier uses an order of $s(q(k)^d)$ random bits

and $t(q(k)^d)$ query bits. Also, an order of $r(k)$ random bits are used to select the segments on which to run the PCP-ENC-type verifier. □

In order to be able apply recursion more than once, we need a stronger result: namely, we need that the resulting procedure also obey the restrictions imposed on PCP-SEG. We therefore define a class PCP-ENC-SEG, which is a subclass of PCP-ENC that also obeys the restrictions of PCP-SEG.

Theorem 8 *Suppose, P⊂PCP-ENC-SEG$(s(k), t(k))$. Let L be in PCP-SEG$(r(k), q(k))$. Then there exists a constant d such that L ∈PCP-SEG$(r(k) + s(q(k)^d), t(q(k)^d))$.*

Proof We perform the same procedure as in Theorem 7. Note that the resulting proof consists of two parts: the encodings of each of the y_i's and the PCP-ENC-SEG-type proof for each $L_{x,r}$. The PCP-ENC-SEG-type proofs are already segmented; the encodings of each of the y_i's can be broken up into segments of one value of $\hat{y_i}$ each (since only a constant number of values of $\hat{y_i}$ is used during the verification procedure). It is clear that the resulting procedure obeys the conditions of PCP-SEG. □

5.3 The Payoff

Now that we have the powerful recursion machinery formalized, we can apply it to get results about NP.

I have shown in Section 4 that NP⊂PCP$(\log k, \log^3 k/\log\log k)$. In Section 5.4, it is shown how to transform the unsegmented proof I presented into a segmented one. The computation involving the query bits is still polynomial in their number. As a result, we get that NP⊂PCP-SEG$(\log k, \text{polylog}k)$. Using the result of Section 4.5, and applying the same techniques, we also get that P⊂NP⊂PCP-ENC-SEG$(\log k, \text{polylog}k)$.

Now, applying Theorem 8 we get the Theorem 2; actually, we get a slightly stronger statement that

$$\text{NP} = \text{PCP} - \text{SEG}(\log k, \text{polylog}\log k),$$

We could repeat the recursion, applying Theorem 8 to Theorem 2, to get that

$$\text{NP} = \text{PCP} - \text{SEG}(\log k, \text{polylog}\log\log k),$$

and keep proceeding in this fashion.

The authors of [ALM+92] achieve an even better result. They show that NP⊂PCP-ENC$(\text{poly}(k), 1)$, and then apply Theorem 7 to Theorem 2, to get the statement of Theorem 1:

$$\text{NP} = \text{PCP}(\log k, 1).$$

5.4 Segmenting the Proof

In this section, I show how to transform the unsegmented proof of Section 4 into a segmented one. This method is due to [ALM+92]

Consider the procedure of Section 4. Let the proof string be y, let $l = |y|$ and let q be the maximum number of query bits that the Verifier requests during the procedure. Without loss of generality, we can assume that the Verifier always requests q bits. Let Q_r be the set of indices into y that the Verifer requests given a random string r (we assume that the input to the Verifier, x, is fixed). We call Q_r a *query set*. Note that $|Q_r| = q$.

The idea is to encode y as its degree-h extension, and to represent all the possible query sets as polynomials; after verifying some polynomial identities, we can just read off the values of the relevant query set. The details follow.

Pick m and h such that $(h+1)^m > l$; thus, y is a map from $[0..h]^m \to \{0,1\}$. Pick a large enough prime p (we will need that $p > \text{poly}(q \log l)$). We will be working in the field $F = F_p$. Let $\hat{y}$ be the degree-h extension of y over F. Thus, the value of the i-th bit of y is given by $\hat{y}(i)$.

For each query set $Q_r = \{b_1, b_2, \ldots, b_q\}$, we would like to present a polynomial p_r such that $p_r(i) = \hat{y}(b_i)$. Then, after verifying that p_r indeed agrees with $\hat{y}$, all we need to do is read off the values of p_r to get the query bits. Since for each r, p_r can be presented in a separate segment, the proof will be segmented (provided its other parts are segmented, of course). For reasons to become apparent later, we will actually need to have $|F|^m$ polynomials for each r: for each $\alpha \in F^m$, we will have $p_{r,\alpha}$ such that $p_{r,\alpha}(0) = \hat{y}(\alpha)$ and $p_{r,\alpha}(i) = \hat{y}(b_i)$ for $1 \le i \le q$.

How to construct such $p_{r,\alpha}$? First, by interpolation, for each r and α, we can construct a degree-q curve $C_{r,\alpha} : F \to F^m$ such that $C_{r,\alpha}(0) = \alpha$ and $C_{r,\alpha}(i) = b_i$ for $1 \le i \le q$ (of course, each b_i is represented as a number base $h+1$, i.e. as m elements of F). Then, by combining $C_{r,\alpha}$ with the polynomial for $\hat{y}$, we get $p_{r,\alpha}(i) = \hat{y}(C_{r,\alpha}(i))$; $p_{r,\alpha}$ is a univariate polynomial of degree at most qmh (since $C_{r,\alpha}$ is of degree q and $\hat{y}$ is of total degree hm).

The verification procedure is now as follows. The Verifier stipulates that the proof contain $\hat{y}$ together with the proof of its polynomiality (we only care about the total degree of $\hat{y}$, not its degree in each variable), and, for each r and α, the coefficients of the univariate polynomial $p_{r,\alpha}$.

Let Y be the table that the Prover claims is $\hat{y}$, and $g_{r,\alpha}$ be the polynomial that the Prover claims is $p_{r,\alpha}$. Fix a random string r. The Verifier first checks that Y is close to a polynomial, using the total-degree test of Section 6.2. Then it picks a random $\alpha \in F^m$ and a random $t \in F_p - [0..q]$. It computes $C_{r,\alpha}(t)$, and verifies that $Y(C_{r,\alpha}(t)) = g_{r,\alpha}(t)$. If not, the Verifier rejects; otherwise it gets $g_{r,\alpha}(b_i)$ for $1 \le i \le q$ as its query bits, and proceeds with the Verification procedure of Section 4 using these bits.

Clearly, if $Y = \hat{y}$ and $g_{r,\alpha} = p_{r,\alpha}$, the Verifier will accept.

The polynomiality assures us that Y is δ-close to some polynomial P of total degree mh. Suppose, $g_{r,\alpha} \neq p_{r,\alpha}$. Then they disagree for at least $p - (q+1) - qmh$ values of t. We would be done, except that the Verifier doesn't directly compute $p_{r,\alpha}$, but uses Y to compute it. Note that, for any $t > q$, if α is uniformly distributed, so is $C_{r,\alpha}(t)$. Thus, when the Verifier computes $Y(C_{r,\alpha}(t))$, it most likely (with probability $1-\delta$) is computing $P(C_{r,\alpha}(t))$. Hence, with probability $1-\delta$, what the verifier computes is actually $p_{r,\alpha}(t)$ (i.e., a degree-qmh univariate polynomial). Therefore, if $g_{r,\alpha} \neq p_{r,\alpha}$, the Verfier will catch the error with probability $1 - \delta - qmh/(p-q-1)$.

If we now set $m = \log n / \log\log n$, and $h = \log n$, and recall that the size of y is polynomial

in the size k of the input x, we have shown that

$$\mathrm{NP} \subset \mathrm{PCP} - \mathrm{ENC} - \mathrm{SEG}(\log k, \mathrm{polylog} k).$$

6 Testing for Polynomiality

In this section I present a procedure that, given a function $B : F^m \to F$ as a table of values held by an oracle, efficiently verifies the proof that B is close to a polynomial of degree at most h over F. The first such procedure was presented in [BFL91] and then improved upon by [BFLS91] and [FGL+91] (although [FGL+91] presents a multinearity test only). Independently and concurrently with the last two papers, the authors of [GLR+91] and [RS92] came up with more polynomiality-testing results. The authors of [AS92] improved the efficiency of the testers of [BFLS91] and [FGL+91]. I present their test below, in Section 6.1.

An different tester was presented in [ALM+92]. It was based on the work of [RS92] and [AS92], and is needed for segmentation in recursive proof checking. As opposed to the test of Section 6.1, it tests the *total* degree of a multivariate polynomial. I present it in Section 6.2.

6.1 Using $O(mh \log mh)$ Queries

We would like to test that B is a polynomial of degree h, i.e., $\Delta_h(B) = 0$. However, this is impossible without reading all the values of B, because a single incorrect value could make B non-polynomial. Hence, we *allow* the Verifier to accept as long as $\Delta_h(B) < \delta$ for some small δ, but the Verifier is not *obliged* to accept unless $\Delta_h(B) = 0$.

Theorem 9 *Let h be a natural number, F be a finite field, $B : F^m \to F$ be a function, and $\delta > 0$. Suppose $|F| > c(m^2h^3)$ (where is c is some fixed constant). Then there exists a randomized oracle machine that accepts if $\Delta_h(B) = 0$, rejects with probability over 1/2 if $\Delta_h(B) > \delta$, and is free to either accept or reject if $0 < \Delta_h(B) \leq \delta$. The machine queries a constant number of values of B, uses $O(m \log |F|)$ random bits and $O(mh \log |F|)$ bits from the proof. It runs in time polynomial in m, h and $1/\delta$.*

Proof *(sketch)* Define a *line* in F^m as the subset of F^m where all but one variables are fixed. That is, for $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_m \in F$, define

$$L_{[a_1,\ldots,a_{i-1},*,a_{i+1},\ldots,a_m]} = \{(x_1, x_2, \ldots, x_m) | x_1 = a_1, \ldots, x_{i-1} = a_{i-1}, x_{i+1} = a_{i+1}, \ldots, x_m = a_m\}$$

We say that $L_{[a_1,\ldots,a_{i-1},*,a_{i+1},\ldots,a_m]}$ is a line in the i-th direction.

Clearly, if B is a polynomial of degree-h in each variable, then its restriction to each line is a univariate degree-h polynomial. Hence, a restriction can be described by simply listing its $h+1$ coefficients. We require that the proof contain all such polynomials for all such lines. Of course, the prover can be dishonest and put an incorrect polynomial. Let $g^i_{[a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_m]}$ be the polynomial that the prover claims is the restriction of B on the line $L_{[a_1,\ldots,a_{i-1},*,a_{i+1},\ldots,a_m]}$.

The verification procedure is as follows:

1. Pick $\lceil 4/\delta \rceil + 1$ random points $(a_1, a_2 \ldots, a_m) \in F^m$ and verify that

$$B(a_1, a_2, \ldots, a_m) = g^1_{[a_2,\ldots,a_m]}(a_1).$$

2. Pick $O(m/\delta)$ random points $(a_1, a_2, \ldots, a_m) \in F^m$, and for each random point, a random $i \in [2..m]$, and verify that

$$g^i_{[a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_m]}(a_i) = g^{i-1}[a_1, \ldots, a_{i-2}, a_i, \ldots, a_m](a_{i-1}).$$

Clearly, if $\Delta_h(B) = 0$ and the Prover is honest in presenting g's, then the Verifier will accept. The hard part is proving that if $\Delta_h(B) > \delta$, then the Verifier will reject.

All the functions $g^1_{[a_2,\ldots,a_m]}$ put together define a map from $F^m \to F$. Call that map g^1. Similarly, define maps g^2, g^3, and so on. If the Prover is honest, $B = g^1 = g^2 = \ldots = g^m$. What we are testing is that B is close to g^1, g^1 is close to g^2, ..., g^{m-1} is close to g^m. Note that if $\Delta(B, g^1) > \delta/2$, then with probability greater than $1/2$ the first part of the test will fail. We need to show that if $\Delta_h(g^1) > \delta/2$, then the second part of the test will fail with probability greater than $1/2$. Then, if $\Delta_h(B) > \delta$, it cannot be that both $\Delta(B, g^1) \leq \delta/2$ and $\Delta_h(g^1) \leq \delta/2$ (by triangle inequality). Hence, no matter how the Prover chooses g^1, the test will fail with probability greater than $1/2$.

Thus, all I need to show is the following proposition:

Proposition 1 *If $\Delta_h(g_1) > \delta/2$, then the second half of the test will fail with probability greater than $1/2$.*

Proof *(sketch)* We can consider each g^i as an $m \times m \times \ldots \times m$ hypercube of values. Recall that g^i is a polynomial in the i-th variable, which means that all the "rows" in the i-th direction in this hypercube are polynomials. The following Lemma says that if only a few of the hyperplanes orthogonal to these rows are close to polynomial, then the whole hypercube is close to polynomial. It thus allows to infer a global property of g^i from its local properties.

Lemma 2 *Let $g : F^m \to F$ be a function. Suppose that, for some i, g restricted to any line in the i-th direction is a univariate degree-h polynomial. If there are $2h$ points $b_1, b_2, \ldots, b_{2h} \in F$ such that g restricted to a hyperplane $a_i = b_j$ is .1-close to a degree-h polynomial in $m-1$ variables, then $\Delta_h(g) \leq .2$.*

Given this Lemma, the proof would go by induction as follows. Divide g^1 into F hyperplanes that are orthogonal to the first direction. If g^1 is too far from a polynomial, then, by the Lemma, $F - 2h$ of the hyperplanes of g^1 have to also be far from polynomials Each hyperplane represents an $m-1$ variable function, so applying the inductive hypothesis to each hyperplane would yield a large number of inconsistencies.

Unfortunately, the inductive argument is not given in [AS92]. Of the many sketchy arguments whose details I was able to fill in in this paper, this one turned out to be more problematic than expected. The work on it is still ongoing. □

6.2 Using $O(h \log F)$ Queries and a Segmented Proof

In this section, I present the polynomiality test required for segmentation of [ALM+92] to work (see Section 5.4). I omit the proof of the correctness of the test. As opposed to the test presented above, this test ensures us that a function B is close to a polynomial of a *total* degree h.

The idea is similar to the proof presented above, except that the notion of a line is more general. More specifically, the proof is required to contain, for each $\alpha, \beta \in F^m$, a univariate degree-h polynomial $f_{\alpha,\beta}(t)$. The verifier picks a constant number of random triples (α, β, t) (where $\alpha, \beta \in F^m$ and $t \in F$) and checks that $B(t\alpha + (1-t)\beta) = f_{\alpha,\beta}(t)$.

The statement on which the correctness of the proof rests is the following: if

$$Pr(B(t\alpha + (1-t)\beta) \neq f_{\alpha,\beta}(t)) < \delta$$

then B is 2δ-close to a polynomial of total degree at most h (provided $|F| > \text{poly}(hm)$).

Note that the proof is segmented, as required by Section 5.4.

Acknowledgement

I am greatly indebted to Michael O. Rabin for the many hours of discussion and valuable insights with which he provided me.

References

[ALM+92] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd IEEE Symposium on Foundation of Computer Science*, pages 14–23, 1992.

[AS92] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs; A new characterization of NP. In *Proceedings of the 33rd IEEE Symposium on Foundation of Computer Science*, pages 2–13, 1992.

[Bab85] László Babai. Trading group theory for randomness. In *Proceedings of the 17th ACM Symposium on the Theory of Computing*, pages 421–429, 1985.

[BF91] László Babai and Lance Fortnow. Arithmetization: A new method in structural complexity theory. *Computational Complexity*, 1(1):41–66, 1991.

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

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, pages 21–31, 1991.

[BGKW88] L. Ben-Or, S. Goldwasser, J. Kilian, and A. Wigderson. Multi-prover interactive proofs: How to remove intractability. In *Proceedings of the 20th ACM Symposium on the Theory of Computing*, pages 113–131, 1988.

[BGLR93] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *Proceedings of the 25th ACM Symposium on the Theory of Computing*, 1993. See also Errata Sheet in *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994.

[BGS95] Mihir Bellare, Oded Goldreich, and Madhu Sudan. Free bits, PCPs and non-approximability—Towards tight results (Version 3). Technical Report TR95-024, ECCC (*Electronic Colloquium on Computational Complexity*, `http://www.eccc.uni-trier.de/eccc`), 1995. Version 1 appears in *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, 1995.

[BS92] P. Berman and G. Schnitger. On the complexity of approximating the independent set problem. *Information and Computation*, 96:77–94, 1992.

[BS94] Mihir Bellare and Madhu Sudan. Improved non-approximability results. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994.

[Coo71] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

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

[FK94] U. Feige and J. Kilian. Two prover protocols—Low error at affordable rates. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, 1994.

[FRS88] L. Fortnow, J. Rompel, and M. Sipser. On the power of multi-prover interactive protocols. In *Proceedings of the 3rd IEEE Symposium on Structure in Complexity Theory*, pages 156–161, 1988.

[GLR+91] Peter Gemmell, Richard Lipton, Ronitt Rubinfeld, Madhu Sudan, and Avi Wigderson. Self-testing/correcting for polynomials and for approximate functions. In *Proceedings of the 23rd ACM Symposium on the Theory of Computing*, pages 32–42, 1991.

[GMR89] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal of Computing*, 18:186–208, 1989.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KLS93] S. Khanna, N. Linial, and S. Safra. On the hardness of approximating the chromatic number. In *Proceedings of the Second Israel Symposium on Theory and Computing Systems*, 1993.

[Lev73] Leonid A. Levin. Universal'nyĭe perebornyĭe zadachi (Universal search problems, in Russian). *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. A corrected English translation appears in an appendix to [Tra84].

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proofs systems. *Journal of the ACM*, 39(4):859–868, 1992.

[LY94] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *Journal of the ACM*, 41:960–981, 1994.

[PY91] C.H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

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

[Sch80] J.T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27:701–717, 1980.

[SFM78] J. Seiferas, M. Fischer, and A. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25:146–167, 1978.

[Sha90] Adi Shamir. IP=PSPACE. In *Proceedings of the 22nd ACM Symposium on the Theory of Computing*, pages 11–15, 1990.

[Tra84] B. A. Trakhtenbrot. A survey of Russian approaches to *Perebor* (brute-force search) algorithms. *Annals of the History of Computing*, 6:384–500, 1984.

[Zuc93] D. Zuckerman. NP-complete problems have a version that is hard to approximate. In *Proceedings of the 8th IEEE Conference on Structure in Complexity Theory*, 1993.