Probabilistically checkable proofs (PCP's) (also called *transparent* or *holographic* proofs) are proofs that can be verified with high probability by looking at the proof in only a few randomly chosen places. This note is intended to be a relatively accessible introduction to PCP's, explaining how such proofs can exist. For this purpose, we present a relatively simple self-contained proof, essentially due to [1], that any satisfiable Boolean formula has a PCP of its satisfiability that can be checked by querying a constant number of bits. Formally, we show

Theorem 1

$$\mathbf{SAT} \in \mathbf{PCP}(52, 2^{n\times n})$$

SAT is the well-known NP-complete problem "Given a multi-variate Boolean Formula, does any assignment of the variables satisfy it?" The parameter n is the number of literals and operators in the formula. $\mathbf{PCP}(q,s)$ is the set of languages L such that some randomized algorithm V (the *verifier*) exists such that for all w

$$\begin{array}{lcl} w \in L & \Rightarrow & (\exists y)\ V(w,y) \text{ accepts} \\ w \notin L & \Rightarrow & (\forall y)\ \Pr[V(w,y) \text{ rejects}] \geq 1/2 \end{array} \tag{1}$$

where V examines only q bits of any y and runs in time polynomial in $|w|$; y ranges over the $s(|w|)$-bit binary strings.

$$(v_1 \,\mathbf{or}\, v_2) \,\mathbf{and}\, (\overline{v_1} \,\mathbf{or}\, \overline{v_2}) \quad \Rightarrow \quad \begin{array}{lcl} v_3 & = & v_1 \,\mathbf{or}\, v_2 \\ v_4 & = & \overline{v_1} \,\mathbf{or}\, \overline{v_2} \\ v_5 & = & v_3 \,\mathbf{and}\, v_4 \\ v_5 & = & \mathbf{true} \end{array} \quad \Rightarrow \quad \begin{array}{lcl} b_3 & = & b_1 + b_2 - b_1 b_2 \\ b_4 & = & 1 - b_1 b_2 \\ b_5 & = & b_3 b_4 \\ b_5 & = & 1 \end{array}$$

Figure 1: Encoding Circuit Satisfiability in Equations Over Z_2

The Verifier — Freivalds' Trick. How could the Verifier possibly have evidence for *any* global structure after querying only a few bits of a proof? Given a Boolean formula, the Verifier constructs a set of equations over Z_2 that are mutually satisfiable only if the Boolean formula is. (Z_2 is the integers with arithmetic modulo 2. This is a standard construction. See Figure 1.) The Verifier takes a *random subset* R of those equations and adds them together to get a final equation of the form $f(b) = \alpha$ where $f : \mathrm{Z}_2^n \rightarrow \mathrm{Z}_2$. The Verifier checks this final equation. Clearly, if a given value for b does satisfy the original set of equations, then $f(b) = \alpha$. Conversely if any $b' \in \mathbf{Z}_2^n$ does not satisfy one of the original equations, then $\Pr_f[f(b') = \alpha] \leq 1/2$. (We leave this as an exercise for the reader.)

The PCP. To check this equation, the Verifier will consult the string y. In fact, the string y will consist of an explicit list, for some fixed b corresponding to a satisfying assignment, of the values $f_1(b), f_2(b), \ldots, f_N(b)$, where the functions $\{f_i\}$ include all the possible functions that the Verifier might be interested in. Note that the Verifier knows f but not b, whereas the string y "knows" b but not f.

However, the Verifier will not simply consult the string y at the location encoding $f(b)$ and accept if that value is α. If it did, there would always be some y that caused it to accept — each bit of y could be chosen to be the bit that would cause the Verifier to accept. In this case, the string y is "cheating" by giving inconsistent answers for different queries. Somehow, the Verifier also needs evidence that the values encoded by y for different f's are *consistent with one fixed* b. In fact, this is the meat of the problem. For an answer, we turn to the theory of self-correcting/testing functions.

Self-Testing, Self-Correcting Functions. A similar issue arises in testing software. Given a program for computing some function g, is it possible to efficiently check that the program is correct? In general, this problem is quite difficult. The recently developed theory of self-testing/correcting functions takes the following approach: instead of testing *exact* correctness (which is not efficiently testable), test *approximate* correctness. Then, show that an *approximately correct* program can be *corrected* — converted into a randomized program that computes correctly on any input with high probability. For certain classes of functions (especially *randomly self-reducible* functions), both of these steps can sometimes be done quite efficiently.

Let's give a concrete example. Suppose we are given a procedure computing some function g that is supposed to be a *linear* function. We want an efficiently testable condition on g that ensures that g at least approximates some (in our case unknown) linear function g'. If g passes this test, we want to be able to efficiently and reliably compute g' on any input by querying g.

Definition 1 *A function g from a finite group to a finite group is* approximately linear *if, for r and s chosen uniformly at random,*

$$\Pr[g(r+s) = g(r) + g'(s)] \geq 91/100.$$

The group operator is "+". Approximate linearity is efficiently testable:

Claim 1 *If g is not approximately linear, then testing 8 random pairs will discover $\langle x, y\rangle$ such that $g(x)+g(y) \neq g(x+y)$ with probability at least $1/2$.*

(This is simply because $(91/100)^8 < 1/2$.) Remarkably, approximate linearity ensures the existence of the desired g'!

Claim 2 *If g is approximately linear, then there exists a* linear *function g' such that*

$$(\forall x) \quad \Pr_r[g'(x) = g(x+r) - g(r)] \geq 9/10.$$

Claim 2 is proved in the Appendix. The proof essentially comes from "Self-Testing/Correcting with Applications to Numerical Problems" by Blum, Luby, and Rubinfeld [2].

In summary, a supposedly linear function can be checked for approximate linearity using only 24 queries. If the function is not approximately linear, then the test will prove non-linearity with probability at least 1/2. Otherwise, the function in fact encodes a unique linear function which can be efficiently be computed on any given input with probability at least 9/10 by making only 2 queries to the original function.

Using Self-Testing/Correcting to Prove Consistency. Recall that our dilemma was that the Verifier needs evidence that different answers to different queries would be answered consistently. More precisely, the Verifier needs evidence that the function values encoded in the proposed PCP (the string y) are consistent with a single fixed b. If they are, then the Verifier can use Freivalds' trick to check that b corresponds to a satisfying assignment.

Recall that the string y (if it is an acceptable PCP) will consist of an explicit list, for a fixed b corresponding to a satisfying assignment, of the values $f_1(b), f_2(b), \ldots, f_N(b)$, where the functions $\{f_i\}$ include all the possible functions that the Verifier might want to query. There are at most $2^{n\times n}$ such functions, because any f that the Verifier might query has the form:

$$\sum_{1\leq i,j\leq n} x_{ij} b_i b_j \tag{2}$$

where the x_{ij}'s are the coefficients of the terms in f obtained by adding a randomly chosen subset of the Verifier's equations (see Figure 1). The key point is that, for any fixed b, expression (2) is a *linear* function of x. Because of this, the Verifier can use self-testing to make sure the function g that y encodes is approximately linear. As long as it does, by the self-testing/correcting results, y implicitly encodes a truly linear function g'. The Verifier queries this function g', which is guaranteed to be consistent.

Now we give an overview of the Verifier. Let w and y be the two inputs to the Verifier (w encodes the Boolean formula, y encodes the proposed PCP). The Verifier interprets the string y as a function $g(x)$, allegedly encoding a linear function of the form (2) for some fixed b.

1. For 8 random pairs (r, s), the Verifier checks $g(r + s) = g(r) + g(s)$. If any pair fails the test, the Verifier rejects. By Claim 1, if g is not approximately linear, the Verifier will reject with probability at least 1/2. Thus, if this test passes, the Verifier can assume that g is approximately linear and encodes a linear function g' such that the value of g' on any input can be computed with probability 9/10 by two queries to g as described in Claim 2.

2. At this point the Verifier can assume that g' is linear — that is, that it has the form

$$g'(x) = \sum_{1\leq i,j\leq n} x_{ij} C_{ij} \tag{3}$$

for some C. Next the Verifier checks that g' also has the more restricted form shown in (2). It suffices to check

$$C = bb^T$$

(i.e., each $C_{ij} = b_i b_j$) where b is a column vector with $b_i = C_{ii}$. The Verifier chooses two random vectors r and s and checks

$$r^T C s = (r^T b)(b^T s).$$

If $C \neq bb^T$, then this equality will fail with probability at least 1/4 (this is Freivald's trick applied twice). The Verifier can check the above equation because $r^T C s$, $r^T b$, and $b^T s$ each correspond to $g'(x)$ for an appropriately chosen x. This check works with probability at least $(1 - 3/10) \cdot (1/4) = 7/40$ (each of the three self-correcting queries is correct and Frievald's trick succeeds). The Verifier repeats this check an additional 3 times, so that any function g' not having the restricted form will be discovered with probability at least 1/2. If any test fails, the Verifier rejects.

3. Now the Verifier can assume that g' in fact encodes a fixed vector b. To test that b corresponds to a satisfying assignment, the Verifier chooses a random subset of equations, adds them together, and checks the single resulting equation $f(b) = \alpha$ (as described on page 1). It does this by querying the function g' on the value x representing f (see expression 2). If b does not correspond to a satisfying assignment, this check will discover an error with probability

$9/10 \cdot 1/2 = 9/20$. The Verifier repeats this check once more, rejecting if equality fails to hold either time. If b does not correspond to a satisfying assignment, the probability of rejecting is at least $1-(1-9/20)^2 \geq 1/2$.

If the given Boolean formula is satisfiable, then clearly some string y exists for which all of the Verifier's checks will succeed. Thus, the formula will be accepted by the Verifier.

Conversely, if the formula is not satisfiable, then any string y will either not encode an approximately linear function, or it will but the corresponding linear function g' will not have the correct restricted form, or g' will have the correct form, but the corresponding b will not represent a satisfying assignment. In each case the Verifier will reject with probability at least 1/2.

A careful count shows that this Verifier makes at most 52 queries to the string y. The Verifier runs in time polynomial in the size of the Boolean formula. Thus, this Verifier demonstrates that $\mathbf{SAT} \in \mathbf{PCP}(52, 2^{n \times n})$.

Remark. The result proven here is relatively weak and ignores the Verifier's use of random bits, which is important for applications to hardness of approximation results. In fact, the "PCP theorem" is usually stated in the following form:

$$\mathbf{NP} = \mathbf{PCP}(O(logn), O(1))$$

where the first parameter, $O(\log n)$, is the number of random bits used by the verifier and the second, $O(1)$, is the number of query bits. In this case the proof size is polynomial, and in fact can be reduced to $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 14–23, 1992.

[2] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 73–83, 1990.

Appendix

Proof of Claim 2:

Let g be any approximately linear function and let r and s be randomly chosen inputs. Define δ so that $\Pr[g(r+s) = g(r)+g(s)] = 1-\delta$. By assumption $\delta \leq 9/100$.

Claim 3

$$(\forall x) \Pr[g(x+r) - g(r) = g(x+s) - g(s)] \geq 1-2\delta.$$

proof: It suffices that

$$g(r+x) + g(s) = g(r+x+s) = g(r) + g(x+s).$$

Since $r+x$ and $x+s$ are themselves random vectors, by the definitiion of δ, each of the two equalities holds with probabability at least $1-\delta$. Thus, they both hold with probability at least $1-2\delta$.

Claim 4 *Choose $\delta' < 1/2$ such that $\delta'(1-\delta') = \delta$.*

$$(\forall x)(\exists g'(x)) \quad \Pr[g'(x) = g(r+x) - g(r)] \geq 1 - \delta'$$

Note that since $\delta \leq 91/100$, $\delta' \leq 1/10$.

Claim 5 *The function g' defined in the previous claim is linear.*

proof: For any x and y, it suffices that for some r,

$$\begin{aligned} g'(x+y) &= g(x+y+r) - g(r) \\ &= g'(x) + g(y+r) - g(r) \\ &= g'(x) + g'(y) + g(r) - g(r) \end{aligned}$$

If r is a random vector, then so are $x+y+r$ and $y+r$. By the previous claim, for random r, each of the three equalities holds with probability at least $1-\delta'$. Thus, all three hold simultaneously with probability at least $1-3\delta'$. By the choice of δ' this is positive, so some such r exists. QED