Locally Testable Codes

By

Mahdi Cheraghchi-Bashi-Astaneh

SUBMITTED TO THE
DEPARTMENT OF COMPUTER AND COMMUNICATION SCIENCES
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
LAUSANNE, SWITZERLAND
JULY 2005

Thesis Supervisor: Mohammad Amin Shokrollahi
Professor of Computer Science and Mathematics
EPFL – ALGO/LMA

To my parents.

Contents

Abstract **v**

Acknowledgements **vi**

1 Introduction **1**

2 Preliminaries and Notation **6**

2.1 Error correcting codes 6

2.2 The tradeoff between rate and distance 9

2.3 Linear codes 11

2.4 Composition of the codes 13

2.4.1 Concatenated codes 14

2.4.2 Tensor product 14

2.4.3 Tanner product 15

2.5 Reed-Solomon codes and related constructions 16

2.6 Oracle machines 19

3 A Toolbox of Polynomials **21**

3.1 Basic notions and facts 21

3.2 Low-degree tests 25

3.2.1 Linearity test 26

3.2.2 Univariate low-degree test 27

3.2.3 Multivariate low-degree test 29

4 Efficient Verification of Codewords **32**
4.1 Definition . . . 32
4.2 Constructions . . . 37
4.2.1 Testing Reed-Muller codes . . . 37
4.2.2 Polynomial-Line codes . . . 38
4.2.3 Tensor product codes . . . 42

5 Codeword Test by Proof Checking **45**
5.1 Probabilistically Checkable Proofs . . . 47
5.1.1 Definitions and facts . . . 47
5.1.2 Relationship with locally testable codes . . . 49
5.2 Hardness of approximation and efficient PCPs . . . 50
5.3 PCPs of proximity . . . 54
5.3.1 Formal definition . . . 55
5.3.2 Robustness and composition . . . 56
5.3.3 Nearly linear-sized PCPPs . . . 58
5.3.4 PCPPs and locally testable codes . . . 61
5.4 Combinatorial constructions . . . 62

6 Conclusions and Related Work **67**

Bibliography **72**

Abstract

Error correcting codes are combinatorial objects that allow reliable recovery of information in presence of errors by cleverly augmenting the original information with a certain amount of redundancy.

The availability of efficient means of error detection is considered as a fundamental criterion for error correcting codes. Locally testable codes are families of error correcting codes that are highly robust against transmission errors and in addition provide super-efficient (sublinear time) probabilistic algorithms for error detection. In particular, the error detection algorithm probes the received sequence only at a small (or even constant) number of locations.

There seems to be a trade-off between the amount of redundancy and the number of probes for the error detection procedure in locally testable codes. Even though currently best constructions allow reduction of redundancy to a nearly linear amount, it is not clear whether this can be further reduced to linear while preserving a constant number of probes.

We study the formal notion of locally testable codes and survey several major results in this area. We also investigate closely related concepts, and in particular, polynomial low-degree tests and probabilistically checkable proofs.

Keywords: Error Correcting Codes, Locally Testable Codes, Probabilistically Checkable Proofs (PCP), Low-Degree Tests, Hardness of Approximation.

Acknowledgements

I am so grateful to my supervisor, Amin Shokrollahi, for introducing the problem of locally testable codes to me and for his advice and encouragement. Thanks to Luca Trevisan for co-refereeing the thesis. Besides Amin, I also thank all other members of ALGO and LMA laboratories for creating such an active and joyful environment: Andrew Brown, Bertrand Ndzanandzana, Christina Fragouli, Frédérique Oggier, Giovanni Cangiani, Lorenz Minder, Mahdi Molkaraie, Natascha Fontana, and Payam Pakzad. Additional thanks to my friend and office-mate, Soheil Mohajer, who was always ready to talk about the most nonsense topics!

The last but not the least, thanks to Don Knuth and Leslie Lamport for bringing quality typography to homes with their wonderful TEX and LATEX!

Chapter 1

Introduction

Consider the following four problems regarding Alice and Bob, stated in informal terms:

Problem 1. Bob is a teaching assistant in an undergraduate programming course. In one of the exercises, the students are supposed to implement an efficient (and weird) algorithm for the evaluation of very large multivariate polynomials of a certain degree. Now Bob wants to grade the exercises, but there are a couple of problems. First, the number of students is huge and he doesn't want to waste his time reading all the source codes. On the other hand, he doesn't quite understand the algorithm himself! So he is looking for a workaround. Alice suggests writing a program to automatically verify the correctness of programs by inspecting their behavior, i.e., providing them with test cases (i.e., a sequence of points over the domain of polynomials) and verifying if the obtained answers are correct. However, to be certain about the correctness of each program, a huge number of test cases are required, which makes even an automatic verification so time consuming. Moreover, Bob is not willing to do the same exercise himself! Is there a simple and efficient method to automatically grade the exercises with reasonable confidence?

Problem 2. Alice wants to send a message (say represented as a binary string) to Bob over a noisy communication channel that may flip individual bits in the message. Because of the noise, Alice adds a fair amount of redundancy to the message using an error correcting code. Bob receives the encoded message. The decoding algorithm for the code is rather time consuming, and it makes little sense to apply the algorithm when the message is too corrupted, rather than asking Alice to send her message again. Is there any "lightning fast" method for Bob to check whether the message he has received is worth decoding or not?

Problem 3. Alice asserts that she has just discovered that $\mathsf{P} = \mathsf{NP}$[1]. Bob asks

[1] You can replace this with your favorite difficult problem, e.g., Riemann hypothesis.

for a proof, and she gives her recent 200-page "paper" containing a proof to him. Bob is not eager to read the whole paper. Is it possible for him to get convinced of the correctness of the proof (if it is indeed correct) or otherwise find a flaw with a reasonable chance by reading only a small portion (say two pages) of the proof?

Problem 4. Bob is working as a scientist in a company and is developing an efficient algorithm for a certain problem, as a part of which he needs to solve the following problem within some small error bound (say one percent): Given an undirected graph, partition the vertices into two groups such that the number of edges going from one group to another is maximized. This optimization problem is denoted by MAX-CUT. Will Bob succeed to come up with an efficient algorithm this way?

The first problem is related to the area of computer programming and software testing, whereas the second is in the are of coding theory, the third is about mathematics, and the fourth fits in the area of approximation algorithms. In this thesis, we study the formal framework of these problems and examine their relationships. But before going through the solutions, we wonder how these seemingly different problems can ever be related? In fact this is true, and the key observation is that they all involve making *fast approximate* decisions.

Solution 1. The answer is *yes.* When the degree of the polynomial is small enough compared to the size of the underlying field, Bob can use a *low-degree test* (which is the main topic of chapter 3) to obtain an efficient automatic tester for the exercises[2].

Polynomials are inherently redundant objects. In fact, it is well known that a nonzero low-degree polynomial cannot have many roots [Sch80]. This immediately implies that two distinct low-degree polynomials cannot agree at too many points, and hence, low-degree polynomials are far apart from each other. This makes polynomials *resilient* against *errors.* Furthermore, if the evaluations of a function are far enough from that of a low-degree polynomial, this would likely be detected by only a few probes. As an intuitive example, consider a degree one multivariate polynomial, i.e., the linear function

$$f(x_1, \dots, x_n) = \alpha_1 x_1 + \cdots + \alpha_n x_n. \tag{1.1}$$

Where the coefficients and variables take values over the binary field $\mathbb{F}_2$. Note that a function f is in the above form if and only if for all n-tuples $\mathbf{x}$ and $\mathbf{y}$ we have that

$$f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y}). \tag{1.2}$$

This suggest the following intuitive test: Pick a pair of random points $\mathbf{x}$ and $\mathbf{y}$ and check if they satisfy (1.2). In fact, this test turns out to act very robustly upon testing

[2]A caveat: Using a low-degree test Bob can only get convinced that the program evaluates *some* low-degree polynomial, and not necessarily the one given as its input. This however makes sense, as Bob does not want to evaluate the polynomial himself!

linear functions [BCH$^+$96], and is able to detect nonlinear functions with a reasonable confidence. In addition, it only makes queries the given function thrice and is very efficient with respect to the running time.

Solution 2. The answer is *yes*, if they use a *proper* error-correcting code. In fact, the answer to the question "what codes are considered to be *proper*?" which is *locally testable codes* is the main topic of this thesis and in particular chapter 4.

Informally, a family of locally testable codes allows highly efficient (probabilistic) error detection besides providing robustness against a large number of transmission errors. Namely, any locally testable code comes with a randomized algorithm called the *tester*. A tester randomly probes the received sequence in only a very small (ideally constant) number of locations and decides whether it corresponds to a valid codeword or is highly corrupted. The decision is such that if the sequence is indeed a codeword the tester decides correctly and if it is considerably corrupted (e.g., in at least one percent of the locations) it detects the error with high probability (say at least 99 percent). Moreover, the running time of the tester is normally a very slowly growing (and clearly polynomially bounded) function of the codeword length, e.g., poly-logarithmic or even logarithmic.

There is an interesting relationship between locally testable codes and low-degree tests. Having solved the first problem, Alice can now consider her message as the coefficients of a low-degree polynomial, say the one given in (1.1), and transmit the evaluation of such a polynomial (using one of the correct solutions of the programming exercise!) over all possible points. Now Bob can use his automatic exercise tester to decide whether the message he has received is too corrupted. This is in fact one of the classical ideas for construction of locally testable codes and is formally discussed in section 2.5.

The problem with the simple approach above is that of efficiency. Evaluating the polynomial over all possible points imposes a great amount of redundancy and results in a locally testable code with poor rate. In particular, using (1.1), we get a code with exponentially large block length (namely, the *Hadamard* code). Now the next challenge for Alice and Bob would be to design locally testable codes with better rates.

It turns out that there is a seemingly inherent trade-off between the number of probes made by the tester (known as the *query complexity* of the tester) and the length of the codewords (or equivalently, the amount of *randomness* used by the tester). As we decrease the codeword length, the amount of queries needed by the tester grows up.

The ultimate goal is to design locally testable binary (and hopefully linear) codes that are testable with a constant query complexity and a constant rate (i.e., the ratio between the length of the codeword and the length of the original message). However, no such construction is known to date, and in fact it is not even clear if such codes exist. However, the best known constructions come very close to the idealistic case. Namely, locally testable binary codes with constant query complexity and *almost linear* block length have been constructed. We will review such constructions in chapter 4 and 5.

Solution 3. The answer is again *yes*, if Alice writes up her proof in a *proper* format. Mathematical proofs are often considered as intuitive and informal counterparts of NP-proofs. Now consider a problem in NP, say the classical problem 3SAT. It is formally defined as the set of satisfiable boolean formulae in conjunctive normal form such that each clause only depends on three literals, namely, 3CNF formulae. Given a satisfiable 3CNF formula, an NP-proof for its membership in 3SAT is simply a set of assignments to the boolean variables it depends on. Such a proof it easy to verify: a deterministic verifier can simply plug the assignment into the formula and see if it is satisfied or not.

Note that if we consider an unsatisfiable 3CNF formula and any assignment of variables as a proof, the proof cannot make the verifier accept simply because it does not satisfy the formula. However, there is at least one clause in the formula that remains unsatisfied by the given assignment. As every clause depends on only three literals, it follows that there is a three-bit evidence in the proof for unsatisfiability of the circuit by the given assignment. However, the verifier needs to read the entire proof to find such an evidence. Now, what if we give the power of *randomness* to a standard NP verifier? That is, if we allow verifier to inspect randomly chosen locations in the proof, and restrict it to read only three (or a constant number of) bits in the proof. Even in that case, the verifier can only find the evidence by sheer luck, as there might be only one clause remained unsatisfied by the assignment.

The above question is answered by a remarkable result known as the *PCP theorem* [AS98, ALM$^+$98]. This theorem implies that every NP-proof for an assertion (i.e., a member of a language in NP, such as a satisfiable formula for 3SAT) can be rewritten in a new form verifiable by a probabilistic verifier that probes the proof at only a constant number of locations. Followed by this fact, the proof in its new form is called *probabilistically checkable proof* (PCP).

More precisely, a PCP verifier has the power of tossing coins and probing the proof at randomly chosen locations, but only for a constant number of times. Moreover, if the assertion is correct (e.g., the 3CNF formula is satisfiable), the transformation of a valid NP-proof (e.g., a satisfying assignment) into the PCP format makes the verifier accept. Otherwise, the it rejects with high probability, no matter what the content of the proof is. It is also guaranteed that the transformation of the NP-proofs into PCPs only incurs a polynomially bounded blow-up in their lengths. This is indeed an exquisite characterization of the classical complexity class NP.

We will investigate PCPs and related notions in chapter 5. Interestingly, certain locally testable codes can be obtained from certain PCP constructions and vice versa.

This is in particular highlighted in section 5.1.2 and section 5.3.4. Likewise locally testable codes, constructing PCPs with short length and low query complexity is a fundamental problem in complexity theory, not only due to their relationships with locally testable codes, but also for their remarkable applications in other theoretical areas, and in particular, approximation algorithms.

Solution 4. This time, the answer is *no*, of course unless Alice's proof on the equality of P and NP in the last problem turns out to be correct!

There is a close, surprising and rather unexpected relationship between PCP systems and approximating NP-optimization problems. The key observation is that by the PCP theorem we know that the PCP verifier only consumes a logarithmic amount of random bits. Hence, there is only a polynomial number of possible executions for the verifier. This means that given an assertion and a probabilistically checkable proof for the assertion, the action of the verifier can be described as a polynomially long boolean $3CNF$ formula (which is simply the conjunction of the verifier's verdict over all possible executions). By the PCP theorem, is it guaranteed that such a formula is either satisfiable (if the assertion is true) or far from being satisfiable (if the assertion is false). Therefore, any problem in NP can be reduced to an approximation of MAX-SAT within some constant factor. This implies that there are constant factors within which approximation of MAX-SAT is NP-complete.

On the other hand, there is a reduction from MAX-SAT to MAX-CUT that preserves the approximation factor up to some multiplicative constant. So, Bob cannot hope to be able to solve his problem efficiently by a nearly precise (even though within a constant ratio) approximation of MAX-CUT.

We will have a more detailed discussion of the relationship between PCPs and approximation problems in section 5.2. In fact, the area of approximation algorithms is greatly evolved under the light of PCP constructions. Moreover, the impact of locally testable codes on such constructions naturally clarifies the connection between these codes and approximation algorithms.

Prerequisites: No specific academic background is assumed for this thesis, except the standard undergraduate coursework in the theory of computation. All necessary knowledge regarding the coding theory (as well as our notational conventions) is covered in chapter 2. For chapter 5 however, a familiarity with the basic notions of the NP-completeness theory and approximation algorithms is assumed (refer to, say, [Pap94, Vaz01] for a thorough study of such notions).

Chapter 2

Preliminaries and Notation

We begin by reviewing certain basic definitions and facts in standard coding theory that will become handy in later chapters. For a comprehensive study of the coding theory refer to many standard texts available, say, [MS81]. Then we continue with the notion of oracle machines whose role is central in our work.

2.1 Error correcting codes

For a finite set Σ (where Σ is typically the set of elements in a finite field) a *string* of length n is any ordered sequence of n elements (i.e., an n-tuple) in Σ. We will refer to the i^{th} element of a string x as x_i, where $i \in [n]$ (we use the short form $[n]$ for the set $\{1, \ldots, n\}$), and n is the length of x. When Σ is a finite field, we shall occasionally consider strings of length n as row vectors or n-dimensional points in the vector space Σ^n.

Consider a *sender* that communicates through a *noisy channel* with a *receiver*. The sender has a *message* m which is simply a string of length k over a finite *alphabet* Σ of q elements. We usually restrict Σ to be the set of elements in a finite field $\mathbb{F}_q$ (for q being a prime power). Because of the noisy nature of the communication channel, transmission errors may occur and the message received at the receiver's side might be slightly different from the original message sent by the sender. Hence before the transmission, sender *encodes* m to an encoded message $\mathcal{C}(m)$ of length n by adding redundant information to it. Clearly, we have $n \geq k$. The encoding scheme should be designed in a way that if few transmission errors occur, the receiver would be able to exactly recover the original message from what he receives. The following definition formalizes the scenario:

Definition 2.1. An *error correcting code* $\mathcal{C}$ is an injective mapping $\mathcal{C}\colon \Sigma^k \to \Sigma^n$, for some finite alphabet Σ of q elements. The parameters k and n, where $n \geq k$, are known as the *message length* and the *block length*, respectively. For any message string m of length k, The string $\mathcal{C}(m)$ is called the *codeword* associated with m. Thus, the block length of a code specifies the length of the codewords. The ratio $\mathcal{R}(\mathcal{C}) = \frac{k}{n}$ is referred to as the *rate* of the code. A code with the alphabet size q is called a q-ary (binary for the special case $q = 2$) code.

It usually does not matter to which message string a particular codeword is assigned to. Therefore, we may occasionally refer to a code as the set of all possible codewords that it defines. That is, we may define an error correcting code $\mathcal{C}$ as a subset of Σ^n, where the *size* of the code, $|\mathcal{C}|$, is $|\Sigma|^k$. The rate of a code $\mathcal{C}$ can also be defined as $\mathcal{R}(\mathcal{C}) = \frac{\lg\lceil|C|\rceil}{n}$. (We use the notation $\lg(\cdot)$ for the logarithm function on base 2.) Furthermore, note that the input alphabet (over which the messages are defined) and the output alphabet (that defines the codewords) of a code might be different.

We are typically interested in the asymptotic behavior of codes, where the parameter k above tends to infinity. To that end, we shall need our codes to be defined for arbitrary block lengths. Hence, we define a *family of codes* as follows:

Definition 2.2. A *family of error correcting codes* $\mathcal{C}$ is a set $\{\mathcal{C}_k\}$ defined over infinitely many values of $k \in \mathbb{Z}^+$, where each C_k is an error correcting code with message length k and block length $n(k)$, for some fixed function $n\colon \mathbb{Z}^+ \to \mathbb{Z}^+$. In general, the alphabet size of $\mathcal{C}_k$ is also allowed to depend on k.

We are interested in the families for which the set of possibilities for the message length is dense enough. Ideally, a family should be defined for all possible values of $k \in \mathbb{Z}^+$; However, certain constructions that we will review in later chapters only define codes for values of k satisfying a special property (but still with infinitely many possibilities). In a dense enough construction, for each message length k, the next largest possible length is bounded by $k^{1+o(1)}$. It is worth noting that most known constructions can be easily modified (e.g., by a simple padding) to work for all $k \in Z^+$; However, maintaining various properties of the code with such a modification is not as straightforward.

To analyze the error correcting capabilities of a code we need a distance measure. We use the traditional convention of Hamming distance, defined as follows:

Definition 2.3. For two strings $\vec{x}, \vec{y} \in \Sigma^n$, let x_i (resp. y_i) denote the value at the i^{th} coordinate of $\vec{x}$ (resp. $\vec{y}$). Then the *absolute Hamming distance* of $\vec{x}$ and $\vec{y}$, denoted by $\boldsymbol{\Delta}(\vec{x}, \vec{y})$, is defined as the number of coordinates at which the two strings differ, i.e., $\boldsymbol{\Delta}(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} |\{i \mid x_i \neq y_i\}|$. The *relative Hamming distance* (that we will simply refer to as the distance) of $\vec{x}$ and $\vec{y}$, denoted by $\boldsymbol{\delta}(\vec{x}, \vec{y})$, is also defined as $\boldsymbol{\delta}(\vec{x}, \vec{y}) = \frac{\boldsymbol{\Delta}(\vec{x}, \vec{y})}{n}$. The absolute Hamming distance of a vector $\vec{x}$ from the zero vector $\vec{0}$ is called the *Hamming weight* (or simply weight) of $\vec{x}$ and is denoted by $\mathrm{wt}(\vec{x})$.

Definition 2.4. The absolute Hamming distance of a vector $\vec{x}$ from a set $\mathcal{C} \subseteq \Sigma^n$, denoted by $\boldsymbol{\Delta}(\vec{x}, \mathcal{C})$, is defined as the minimum absolute Hamming distance of $\vec{x}$ from the elements of $\mathcal{C}$. That is, $\boldsymbol{\Delta}(\vec{x}, \mathcal{C}) \stackrel{\text{def}}{=} \min\{\boldsymbol{\Delta}(\vec{x}, \vec{w}) \mid \vec{w} \in \mathcal{C}\}$. Similarly, the relative Hamming distance (or simply distance) of $\vec{x}$ from $\mathcal{C}$ is defined as $\boldsymbol{\delta}(\vec{x}, \mathcal{C}) \stackrel{\text{def}}{=} \min\{\boldsymbol{\delta}(\vec{x}, \vec{w}) \mid \vec{w} \in \mathcal{C}\}$.

Definition 2.5. A vector $\vec{x}$ is considered δ-close to a set $\mathcal{C}$ if $\boldsymbol{\delta}(\vec{x}, \mathcal{C}) \leq \delta$. Similarly, $\vec{x}$ is δ-far from $\mathcal{C}$ if $\boldsymbol{\delta}(\vec{x}, \mathcal{C}) \geq \delta$.

Definition 2.6. The absolute Hamming distance of a set $\mathcal{C} \subseteq \Sigma^n$, denoted by $\boldsymbol{\Delta}(\mathcal{C})$, is defined as the minimum absolute Hamming distance of pairs of vectors in $\mathcal{C}$. That is, $\boldsymbol{\Delta}(\mathcal{C}) \stackrel{\text{def}}{=} \min_{\vec{x}, \vec{y} \in \mathcal{C}}\{\boldsymbol{\Delta}(\vec{x}, \vec{y})\}$. The relative Hamming distance (or simply distance) of $\mathcal{C}$ is defined similarly, i.e., $\boldsymbol{\delta}(\mathcal{C}) \stackrel{\text{def}}{=} \min_{\vec{x}, \vec{y} \in \mathcal{C}}\{\boldsymbol{\delta}(\vec{x}, \vec{y})\}$.

Followed by the above definition, we can define minimum distance of a code, looking at it as a subset of the ambient space. We will use the shorthand form $(n, k, d)_q$-code while referring to a q-ary code $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ with a minimum absolute distance of d. Moreover, a family of codes $\mathcal{C} = \{\mathcal{C}_k\}$ where each C_k is a $(\frac{k}{r(k)}, k, \frac{k}{r(k)} \cdot \delta(k))_{q(k)}$ code will be denoted as an $(r(k), \delta(k))_{q(k)}$-family of codes.

Definition 2.7. The *ball* of *radius* r around a point $\mathbf{x} \in \Sigma^n$, $\mathcal{B}_r(\mathbf{x})$, is defined as the set of all points $\mathbf{w} \in \Sigma^n$ such that $\boldsymbol{\Delta}(\mathbf{x}, \mathbf{w}) \leq r$. The number of such points is called the *volume* of the ball.

Note that the volume of an n-dimensional ball $\mathcal{B}_r(\mathbf{x})$, as defined above, is independent of $\mathbf{x}$ and is equal to $\sum_{i=0}^{r} \binom{n}{i}(|\Sigma| - 1)^i$. We shall denote this quantity by $\mathrm{Vol}(r, n, q)$.

Definition 2.8. A (minimum distance) decoder for a code $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ is the mapping $\mathcal{C}^{-1} \colon \Sigma^n \to \Sigma^k$ that maps a vector $\vec{y} \in \Sigma^n$ to the vector $\vec{x} \in \Sigma^k$ such that $\mathcal{C}(\vec{x})$ is the codeword with the minimum distance to $\vec{y}$. Ties are broken arbitrarily.

Note that for an error correcting code $\mathcal{C}$ with the absolute minimum distance of d, the minimum distance decoder can correct up to $t = \lfloor \frac{d-1}{2} \rfloor$ errors. (To see this, consider a ball of radius t around each codeword, and observe that the balls are disjoint by the distance assumption.)

Another less-popular parameter of a code is its *covering radius*, which is defined as follows:

Definition 2.9. The *normalized covering radius* of a code $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ is the smallest fraction δ such that every vector $\vec{w} \in \Sigma^n$ is δ-close to $\mathcal{C}$. The *covering radius* is the normalized covering radius of the code multiplied by n.

It is worth noting that the covering radius of a code is at least half its minimum distance.

2.2 The tradeoff between rate and distance

By definition, the higher the minimum distance of a code is, the more resilient it becomes upon the transmission errors. So a general aim is to design codes with high minimum distance. However, this is only achievable for a cost of losing rate, which is a measure of the efficiency of the channel use by the code. It turns out that there is an inevitable trade-off between these two parameters, as the following propositions suggest:

Proposition 2.10. *(Known as the sphere packing bound, the Hamming bound, and also the balls bound) For any $(n, k, d)_q$-code $\mathcal{C}$, we have $k = \log_q |\mathcal{C}| \leq \log_q \left\lceil \frac{q^n}{\mathrm{Vol}(\lfloor \frac{d-1}{2} \rfloor, n, q)} \right\rceil$. That is, the rate of a code with a high minimum distance has to be low.*

Proof: As the minimum distance of the code is d, the balls of radius $\lfloor \frac{d-1}{2} \rfloor$ around each codeword have to be disjoint. The bound follows noting that the size of the ambient space is q^n. ∎

There are a few families of codes that achieve the sphere packing bound with equality (e.g., Hamming codes). Such codes are known as $perfect codes$.

Let us call an $(r, \delta)_q$-family of codes where r and δ are (lower bounded by) absolute positive constants an *asymptotically good* family of codes. That is, both absolute minimum distance and codeword length of an asymptotically good code are linear functions of the message length k. Followed by the above proposition, one may wonder if asymptotically good codes exist. In fact, a positive answer can be easily shown. Consider the following exponential time greedy algorithm that constructs a family of codes:

Algorithm 2.11	GREEDY-CODE(n, d, Σ)
Input	A block length n, Distance value δ, The alphabet Σ of size q.
Output	The set of codewords $\mathcal{C}_k$.

1. Set $S \leftarrow \Sigma^n$ and $\mathcal{C}_k \leftarrow \emptyset$.
2. **while**$S \neq \emptyset$ **do**
   (a) Pick a point $\mathbf{x}$ in S, and add it to $\mathcal{C}_k$.
   (b) Remove all the points in S that are within the distance δ from $\mathbf{x}$. ∎

Proposition 2.12. *Algorithm 2.15 produces a code $\mathcal{C}_k$ with $|\mathcal{C}_k| \geq \frac{q^n}{\mathrm{Vol}(n\delta, n, q)}$.*

Proof: At most $\mathrm{Vol}(n\delta, n, q)$ points are removed from Σ^n in each iteration. Thus, the number of codewords that is equal to the number of iterations is at least $\frac{q^n}{\mathrm{Vol}(n\delta, n, q)}$.
∎

Corollary 2.13. *Algorithm 2.15 constructs an asymptotically good family of codes.*

Proof: It is easy to see that for $\delta \leq \frac{q-1}{q}$ and large enough n, the quantity $\mathrm{Vol}(n\delta, n, q)$ can be approximated by (and is upper bounded by) the exponential value $q^{nH_q(\delta)}$, where H_q is the q-ary entropy function, defined as

$$H_q(p) \stackrel{\text{def}}{=} -p \log_q p - (1-p)\log_q(1-p) + p\log_q(q-1).$$

Let r be the rate of the code. Followed by Proposition 2.12 we get,

$$\begin{aligned} q^{nR} &= |\mathcal{C}_k| \\ &\geq \frac{q^n}{\mathrm{Vol}(n\delta, n, q)} \\ &\geq \frac{q^n}{q^{nH_q(\delta)}} = q^{n(1-H_q(\delta))}. \end{aligned}$$

It follows that for large block length and $\delta \leq \frac{q-1}{q}$ we have $r \geq 1 - H_q(\delta)$. Taking an absolute constant for δ, the rate of the code constructed by Algorithm 2.15 will also be an absolute constant, making the constructed code asymptotically good. ∎

As a result, we have the following *positive* bound on the set of achievable rates:

Corollary 2.14. *(Gilbert-Varshamov Bound) For every distance value δ, Asymptotically good q-ary families of code exist that achieve a rate of at least $1 - H_q(\delta)$.* ∎

Now, consider the following randomized construction of a family of codes with the minimum distance δ:

Algorithm 2.15	RANDOM-CODE(n, d, Σ)
Input	A block length n, Distance value δ, The alphabet Σ of size q.
Output	The set of codewords $\mathcal{C}_k$.

1. Initialize $\mathcal{C}_k$ with a random subset of Σ^n of size $q^{n(1-H_q(\delta))}$.
2. Let the *bad set* B be the set $\{\vec{x} \in \mathcal{C}_k \mid \boldsymbol{\delta}(\vec{x}, \mathcal{C}_k \setminus \vec{x}) < \delta\}$.
3. Set $\mathcal{C}_k \leftarrow \mathcal{C}_k \setminus B$. ∎

One can show that the expected number of codewords in the above construction is still $\Theta(q^{n(1-H_q(\delta))})$ and hence a random code also achieves the Gilbert-Varshamov bound.

Now we give another proposition that puts an upper bound on the minimum distance of any code, when the alphabet size is not a fixed parameter:

Proposition 2.16. *(Singleton Bound) Let $\mathcal{C}$ be an $(n,k,d)_q$-code. Then $d \leq n-k+1$ (no matter what q is).*

Proof: Let Σ be the alphabet of $\mathcal{C}$. Consider a projection $\pi\colon \Sigma^n \to \Sigma^{k-1}$ that maps any codeword to its first $k-1$ coordinates. As the number of codewords is Σ^k, it follows by the pigeonhole principle that there must be two distinct codewords $\vec{x}$ and $\vec{y}$ that are projected to the same vector. As the first $k-1$ coordinates of $\vec{x}$ and $\vec{y}$ are the same, the absolute distance of these two vectors cannot exceed $n-k+1$. Hence, the absolute minimum distance of $\mathcal{C}$ is at most $n-k+1$. ∎

The following is a slightly more precise form of the Singleton bound:

Proposition 2.17. *(Plotkin Bound) Let $\mathcal{C}$ be an $(n,k,d)_q$-code. Then for any $1 \leq j \leq k$ we have $d \leq (n-k+j)(1-\frac{1}{q})\frac{q^j}{q^j-1}$.* ∎

Any code that meets the Singleton bound is called a *Maximum Distance Separable (MDS)* code. On can mention Reed-Solomon codes (that we introduce later) as an example of MDS codes. Two trivial examples of MDS codes include the identity code (i.e., $\mathcal{C} = \Sigma^n$) where $n = k$ and $d = 1$ and also the repetition code for the message length one (where the codeword is essentially the message symbol repeated n times, thus, $k = 1$ and $d = n$).

2.3 Linear codes

Ideally, the process of designing a family of codes involves giving a description of the underlying structure of the code, or more precisely, defining the mapping $\mathcal{C}_k$ for every possibility of k in a uniform and constructive manner. An interesting structural property of certain families of codes is *linearity* that makes them easier to describe and analyze. The formal definition follows.

Definition 2.18. A code $\mathcal{C}\colon \Sigma^k \to \Sigma^n$, where the alphabet Σ is associated to the elements of a finite field $\mathbb{F}_q$ (and $q = |\Sigma|$ is a prime power) is called *linear* if the property $\mathcal{C}(\alpha\vec{x} + \beta\vec{y}) = \alpha\mathcal{C}(\vec{x}) + \beta\mathcal{C}(\vec{y})$ holds for every $\alpha, \beta \in \mathbb{F}_q$ and $\vec{x}, \vec{y} \in \mathbb{F}_q^k$. In other words, a code is linear if it defines a linear subspace of the ambient space Σ^n. The dimension of such a subspace is called the dimension of the code and equals the message length k.

As in any linear transformation, one can specify the mapping associated to a linear code by a $k \times n$ matrix $\mathcal{G}(\mathcal{C})$ over the field $\mathbb{F}_q$, which is known as the *generator matrix* of the code. Considering strings as row vectors, the codeword $\vec{y} \in \Sigma^n$ associated to a message vector $\vec{x} \in \Sigma^k$ can be written as $\vec{y} = \vec{x} \cdot \mathcal{G}(\mathcal{C})$. Note that the zero vector is a codeword of any linear code, and it is easy to show that the absolute minimum distance of a linear code equals to the minimum Hamming weight of a nonzero codeword.

The interesting point about the linear codes is that they inherit all properties of linear spaces known in linear algebra. Thus we already have an extensive set of tools at our disposal to analyze the linear codes. Moreover, the generator matrix of the code provides a natural and efficient means of specifying a linear code and computing the codewords.

Linear codes also provide a natural procedure for error detection. For a linear code $\mathcal{C}$, let $\mathcal{C}^{\perp}$ be its dual space. Note that $\mathcal{C}^{\perp}$ is also a linear code mapping the the strings in Σ^{n-k} to Σ^{n}. We call the code $\mathcal{C}^{\perp}$ as the *dual code* of $\mathcal{C}$. Moreover, any codeword of $\mathcal{C}$ is perpendicular to all the codewords of $\mathcal{C}^{\perp}$ and vice versa. So, the dual code of $\mathcal{C}^{\perp}$ is $\mathcal{C}$ itself. Let the $n \times (n-k)$ matrix $\mathcal{P}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{G}(\mathcal{C}^{\perp})$ be the generator matrix of the dual code. We shall refer to the matrix $\mathcal{P}(\mathcal{C})$ as the *parity check* matrix of $\mathcal{C}$. One can look at the codewords of $\mathcal{C}$, i.e., the image of the transformation defined by $\mathcal{G}(C)$, as the kernel (null space) of the dual code, defined by $\mathcal{P}(\mathcal{C})$. That is, the row vector $\vec{y} \in \Sigma^{n}$ is a codeword of $\mathcal{C}$ if and only if

$$\vec{y} \cdot \mathcal{P}(\mathcal{C}) = \vec{0}.$$

This gives us a straightforward procedure to detect errors in any linear code, i.e., for a candidate codeword $\vec{y}$, simply multiply it by the parity check matrix of the code and check if the result is the zero vector or not.

Another interesting property of the parity check matrix is that it defines the minimum distance of the code. The following proposition is easy to verify:

Proposition 2.19. *The absolute minimum distance of a code is the smallest positive integer d such that there exists d rows of the parity check matrix of the code that are linearly dependent.* ∎

The parity check matrix of a linear code gives us a graph-theoretic interpretation of the code. For a $(n, k, d)_q$-code $\mathcal{C}$, consider the bipartite graph G with n (ordered) left vertices and $n-k$ (ordered) right vertices. Let $P \in \mathbb{F}_q^{n\times(n-k)}$ be the parity-check matrix of $\mathcal{C}$. Note that each column of P defines a *constraint* that must be satisfied by the codewords of $\mathcal{C}$. Thus the parity check matrix defines $n-k$ constraints. Now, for any nonzero entry p_{ij} of P, put an edge between the i^{th} left vertex and the j^{th} right vertex of G, weighted by the entry p_{ij}. (Note that weights are ignored in the case of $q = 2$ as they are always 1.) Note that G precisely defines the parity check matrix of $\mathcal{C}$. The left vertices of G correspond to codeword coordinates and its right vertices correspond to the constraints. Such a graph representation is known as the *Tanner graph* of the code.

It is notable that certain families of linear codes with constant rates whose parity check matrices are sparse (i.e., have a linear number of nonzero entries) are of particular interest. Such families, introduced and analyzed by Gallagar [Gal63] are known as *Low-Density Parity Check (LDPC)* codes. A special category of LDPC codes are known as regular LDPC codes. For absolute constants c and d, a linear code is called a (c,d)-regular LDPC code if its Tanner graph is (c,d)-regular, i.e., the parity check matrix of the code has c nonzero entries in each row and d nonzero entries in each column. It can be shown that a randomly constructed family of regular LDPC codes is expected to be asymptotically good.

Likewise generic linear codes, decoding LDPC codes turns out to be an NP-complete problem in general; However, Gallagar proposed two practically efficient decoding algorithms in his PhD thesis for decoding special families of LDPC codes. This makes LDPC codes interesting objects to study.

Occasionally, an assumption that any message is a part of its encoding by a code may ease the analysis of the code. Indeed codes with such a property exist, and are known as *systematic codes*. The following is a precise definition of these codes:

Definition 2.20. An $(n,k,d)_q$-code $\mathcal{C}$ over a q-ary alphabet Σ is called *systematic* if for any message $\vec{x} \in \Sigma^k$ and $\vec{y} = \mathcal{C}(\vec{x}) \in \Sigma^n$ we have that $y_1 = x_1, y_2 = x_2, \ldots, y_k = x_k$. In such a case, the first k coordinates of $\vec{y}$ are called *message symbols* (or *message bits* in case of $q = 2$) and the rest of them are called *check symbols* (or *check bits* for $q = 2$).

Observe that for a linear systematic code, the generator matrix is of the form $[\mathbf{I} \mid \mathbf{A}]$, where $\mathbf{I}$ is the $k \times k$ identity matrix, and $\mathbf{A}$ is some $k \times (n-k)$ matrix. Moreover, any linear code can be transformed to a systematic linear code (without a change in the rate and minimum distance) by a sequence of primitive row and column operations on its generator matrix. In fact, such a transformation is nothing but a change of basis.

Clearly, the negative bounds on the rate and distance of the codes (namely, the sphere packing and Singleton bounds) also hold for linear codes. Moreover, one can show that the Gilbert-Varshamov bound can be achieved by linear codes. Namely, this can be done by choosing the entries of the generator matrix uniformly and independently at random and analyzing the (expected) rate of the resulting linear code.

2.4 Composition of the codes

Normally, the process of designing a family of error-correcting codes that satisfies certain properties is difficult if carried out in one step. A typical approach is to start from a *bad* code that does not satisfy or weakly satisfies the desired criteria, and improve its different parameters step by step, usually in a recursive manner. Code composition is normally a major ingredient of such constructions. Roughly, in a composition, one constructs a new error correcting code from a pair of known to exist or already been constructed codes in a certain manner such that the new code inherits all *good* properties of the two. In this section we examine a few composition operations that turn out to be useful in construction of locally testable codes.

2.4.1 Concatenated codes

Code concatenation is normally used to reduce the alphabet size of a code. The idea behind concatenated codes is to encode the output alphabet of an *outer* code by using a "small" *inner code*, hence obtaining a new code with the desired output alphabet. This naturally deteriorates the rate of the outer code, but such a loss can be moderated by a proper choice of the inner code. The more precise definition of a concatenated code follows:

Definition 2.21. Let $\mathcal{C}$ and $\mathcal{C}'$ be error-correcting codes with the encoding functions $E(\mathcal{C})\colon \mathbb{F}_{q^{k'}}^{k} \to \mathbb{F}_{q^{k'}}^{n}$ and $E(\mathcal{C}')\colon \mathbb{F}_q^{k'} \to \mathbb{F}_q^{n'}$. Then *concatenation* of $\mathcal{C}$ and $\mathcal{C}'$, denoted by $\mathcal{C} \diamond \mathcal{C}'$, is a code with encoding function $E(\mathcal{C} \diamond \mathcal{C}')\colon \mathbb{F}_q^{kk'} \to \mathbb{F}_q^{nn'}$ that operates as follows:

- An input message $w \in \mathbb{F}^{kk'}$ is partitioned into k sub-messages, each of length k'.
- The partitioned message can be considered as a string of length k over the alphabet $q^{k'}$ and is encoded into a message w' of length n over the same alphabet using the encoding function of $\mathcal{C}$.
- Each coordinate of w' represents a string of length k' over the alphabet q and can be encoded into a string of length n' over the same alphabet using the encoding function of $\mathcal{C}'$. Using this transformation on each coordinate of w' an encoded string of length nn' is obtained, which is the output of the encoding function $E(\mathcal{C} \diamond \mathcal{C}')$.

The following lemma easily follows from the above definition:

Lemma 2.22. *Let $\mathcal{C}$ be an $(n,k,d)_{q^{k'}}$-code and $\mathcal{C}'$ be an $(n',k',d')_q$-code. Then $\mathcal{C} \diamond \mathcal{C}'$ is an $(nn',kk',dd')_q$-code. Moreover, if $\mathcal{C}$ and $\mathcal{C}'$ are linear, then so is their concatenation.* ∎

2.4.2 Tensor product

The tensor product (also known as the *direct product*) is another way of constructing a new code from a pair of existing ones. In contrast with the concatenated code, the underlying alphabet is not "actively" involved in the product. Let us define the product first:

Definition 2.23. Let $\mathcal{C}$ be an $(n,k,d)_q$ and $\mathcal{C}'$ be an $(n',k',d')_q$-code over the same alphabet $\mathbb{F}_q$. Then the tensor product of $\mathcal{C}$ and $\mathcal{C}'$, denoted by $\mathcal{C} \otimes \mathcal{C}'$, is an error-correcting code with the encoding function $E(\mathcal{C} \otimes \mathcal{C}')\colon \mathbb{F}_q^{kk'} \to \mathbb{F}_q^{nn'}$ that operates as follows:

- Consider the message string as a $k' \times k$ matrix over $\mathbb{F}_q$.
- Apply the encoding function of $\mathcal{C}$ on each row of the matrix to obtain an intermediate $k' \times n$ matrix.
- Apply the encoding function of $\mathcal{C}'$ on each column of the intermediate matrix to obtain an $n' \times n$ matrix, which is the output of the encoding function.

Note that the above definition does not require $\mathcal{C}$ and $\mathcal{C}'$ to be linear codes. Similar to concatenated codes, we have the following regarding the tensor product codes:

Lemma 2.24. *Let $\mathcal{C}$ be an $(n,k,d)_q$-code and $\mathcal{C}'$ be an $(n',k',d')_q$-code. Then $\mathcal{C} \otimes \mathcal{C}'$ is an $(nn',kk',dd')_q$-code.* ∎

It is easy to see that an $n' \times n$ matrix is a codeword of $\mathcal{C} \otimes \mathcal{C}'$ if and only if all its rows are codewords of $\mathcal{C}$ and all its columns are codewords of $\mathcal{C}'$. Moreover, when $\mathcal{C}$ and $\mathcal{C}'$ are linear codes with generator matrices M and M', respectively, the encoding function of $\mathcal{C} \otimes \mathcal{C}'$ can be considered as $Y = M'^T X M$, where Y is the $n' \times n$ matrix corresponding to the message string X, viewed as a $k' \times k$ matrix. In that case, the tensor product is itself a linear code.

2.4.3 Tanner product

Consider putting a string w of length n on the left vertices of the Tanner graph of a binary linear code $\mathcal{C}$, where n is as usual the block length of the code. Assume that the Tanner graph is right-regular, i.e., all right vertices have the same degree. By construction, w will be a codeword if and only if each constraint on the right vertices is satisfied, i.e., the coordinates of w defined by the right vertex sum to zero. This idea can be generalized in the following way. Consider a *small* code $\mathcal{C}'$ that need not be binary nor linear. Then we define the string w to be a codeword of $\mathcal{C}$ if and only if the coordinates corresponding to every right node define a codeword of $\mathcal{C}'$.

The above construction is known as the Tanner product code of $\mathcal{C}$ and $\mathcal{C}'$, first analyzed by Tanner [Tan81]. It can also be considered as product of a bipartite graph (the Tanner graph of $\mathcal{C}$) and a code ($\mathcal{C}'$), as we look at the outer code as a bipartite constraint graph rather than an error-correcting code.

2.5 Reed-Solomon codes and related constructions

Reed-Solomon codes are very popular families of linear error-correcting codes based on the univariate polynomials over finite fields. They are extensively applied in practical situations (e.g., storage devices such as compact discs) and have been source of inspiration for many other interesting coding schemes. These codes are essentially based on the following simple observation:

Observation 2.25. *A nonzero univariate polynomial of degree d has at most d roots.*

Let $\mathcal{R}_{k,\vec{\alpha}}$, where the parameter $\vec{\alpha}$ is a fixed vector of size n over some finite field $\mathbb{F}_q$, denote the Reed-Solomon code for the message length k. The the encoding function of $\mathcal{R}_{k,\vec{\alpha}}$ is defined by the following algorithm:

Algorithm 2.26	Reed-Solomon-Encode($\vec{u}$, $\vec{\alpha}$, q)
Input	A field size q, a message $\vec{u} \in \mathbb{F}_q^k$, and a vector $\vec{\alpha} \in \mathbb{F}_q^n$.
Output	The codeword $\vec{w} = \mathcal{R}_{k,\vec{\alpha}}(\vec{u}) \in \mathbb{F}_q^n$.

1. Let $U(x)$ be the univariate polynomial of degree $k-1$ with the coefficients taken from the message $\vec{u}$. That is,

$$U(x) \stackrel{\text{def}}{=} u_k x^{k-1} + u_{k-1} x^{k-2} + \cdots + u_2 x + u_1.$$

2. **for** $i \leftarrow 1$ **to** n, set $w_i \leftarrow U(\alpha_i)$. ∎

In other words, the encoding function of a q-ary Reed-Solomon interprets the input message as the coefficients of a univariate polynomial and evaluates such a polynomial in a fixed sample of the elements of the finite field $\mathbb{F}_q$. However, for that procedure to be well defined one needs a finite field of size at least equal to the block length. In other words, the size of the finite field has to grow with the block length of the code.

It immediately follows from the definition that Reed-Solomon codes are linear. Now let us examine their distance.

Proposition 2.27. *The absolute minimum distance of an $(n, k, d)_q$ Reed-Solomon code equals to $n - k + 1$. That is, Reed-Solomon codes are maximum distance separable.*

Proof: From Observation 2.25 it follows that a nonzero codeword cannot be zero in more than $k - 1$ coordinates. Thus the weight of a nonzero codeword is at least $n - k + 1$. Hence, the minimum distance of the code is lower bounded by $n - k + 1$. Combined with the Singleton bound, we get that the minimum distance of the code is precisely $n - k + 1$. ∎

The minimum distance of a Reed-Solomon code is the largest one can hope for, but this comes at the expense of the large alphabet needed by the code. This drawback can be resolved by using a straightforward generalization of the Reed-Solomon codes, namely, Reed-Muller codes, that allow a more flexible choice of parameters.

Reed-Muller codes are based on the same idea as Reed-Solomon codes. Instead of univariate polynomials, Reed-Muller codes consider m-variate polynomials, for an arbitrary positive integer m. A univariate Reed-Muller code will essentially be a Reed-Solomon code.

In a Reed-Muller code, the message vector is interpreted as the vector of the coefficients of an m-variate polynomial of total degree[1] ℓ, where the positive integer ℓ is another parameter of the code and is normally restricted to be less than the alphabet size. Then the codeword is constructed by evaluating such a polynomial over all m-dimensional points in the vector space $\mathbb{F}_q^m$. In other words, the codeword is the evaluation map of the polynomial whose coefficients are described by the message vector. The encoding process for such a code (denoted by $\mathcal{RM}_{m,\ell}$) is described in the following algorithm:

Algorithm 2.28	Reed-Muller-Encode($\vec{u}$, m, d, q)
Input	A field size q, a message $\vec{u} \in \mathbb{F}_q^k$ (k to be resolved later), the number of variables m, and a total degree parameter d.
Output	The codeword $\vec{w} = \mathcal{RM}_{m,\ell}(\vec{u}) \in \mathbb{F}_q^n$ whose coordinates are indexed by the elements of $\mathbb{F}_q^m$.

1. Let $U(\vec{x})$ be the univariate polynomial of total degree ℓ with the coefficients taken from the message $\vec{u}$.

2. **for all** $\vec{i} \in \mathbb{F}_q^m$, set $w_{\vec{i}} \leftarrow U(\vec{i})$. ∎

Clearly, the block length of an $(n, k, d)_q$ Reed-Muller $\mathcal{RM}_{m,\ell}$ code is equal to q^m. Moreover, the message length k is the number of coefficients in an m-variate polynomial of total degree ℓ. This is analogous to the number of sequences of m nonnegative integers that sum to at most ℓ, which is $\binom{m+\ell}{m}$. (To see this, think of cutting a *ruler* of length $m + \ell$ into $m + 1$ pieces.) The following lemma helps us analyze the minimum distance of the code:

Lemma 2.29. [Sch80] *(Schwartz-Zippel Lemma) A nonzero m-variate polynomial of total degree ℓ over the q-ary field evaluates to zero on at most an $\frac{\ell}{q}$ fraction of the points in $\mathbb{F}_q^m$.* ∎

[1] In a polynomial of total degree ℓ, summation of the exponents in any monomial is at most ℓ.

Using the Schwartz-Zippel lemma, it is straightforward to show that the minimum distance of the code is exactly $(1 - \frac{\ell}{q})$, in a similar way to the Proposition 2.27. In particular, a special choice of $\ell = 1$ and $q = 2$ gives a code of minimum distance half. This particular family of the Reed-Muller codes is known as *the Hadamard code*.

As the standard definition of the Reed-Muller codes suggests, there is an affine constant (a monomial of total degree zero) in the polynomial described by the message vector. Thus for the case of the Hadamard code, the polynomial corresponding to the message vector is an affine linear function of m variables. However, the affine constant could be a bit of mess because the linearity of the underlying polynomial turns out to be a very useful property. Therefore, we shall deviate a bit from the standard definition of the Reed-Muller codes and describe the the encoding function of the Hadamard code as follows:

Algorithm 2.30	HADAMARD-ENCODE($\vec{u} \in \mathbb{F}_2^k$)
Input	A binary message $\vec{u}$ of length k.
Output	The codeword $\vec{w} = \mathcal{H}_k(\vec{u})$ whose coordinates are indexed by the elements of $\mathbb{F}_q^k$.

1. Let $\ell_u(\vec{x})$ be the k-variate linear form (i.e., a k-variate polynomial of total degree one with no monomial of total degree zero) with the coefficients taken from the message $\vec{u}$,
$$\ell_{\vec{u}}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \vec{u} \cdot \vec{x} = u_1 x_1 + \cdots + u_n x_n.$$
2. **for all** $\vec{i} \in \mathbb{F}_q^k$, set $w_{\vec{i}} \leftarrow \ell_u(\vec{i})$. ▮

Hence, the Hadamard code is a $(2^k, k, 2^{k-1})_2$-code and its rate, $\frac{k}{2^k}$, is quasi-exponentially small. In fact, it can be shown that this is almost the best possible rate for a binary code with a distance of half. (The best possible rate is $\frac{k}{2^{k-1}}$, which is negligibly better and is achieved by the binary Reed-Muller code $\mathcal{RM}_{k-1,1}$.)

Many interesting properties of the Hadamard code (notably its binary alphabet and the existence of efficient test and correction procedures, as we shall see later) makes it an important ingredient of many constructions of the locally-testable codes. However, because of the exponentially large block length of the Hadamard code, it is normally used in the deepest level of a concatenation, where the message length is already small and the exponential blow-up is acceptable.

The operator $\ell_{\vec{u}}$ defined in Algorithm 2.30 above has special properties. First, it is a linear operator, i.e.,
$$\ell_{\vec{u}+\vec{v}}(\vec{x}) = \ell_{\vec{u}}(\vec{x}) + \ell_{\vec{u}}(\vec{x}).$$
This is the property that makes the Hadamard code linear. Second, $\ell_{\vec{u}}(\vec{x}) = \ell_{\vec{x}}(\vec{u})$. The latter property gives us an alternative view of the Hadamard code. Instead of

thinking the Hadamard code as the evaluation of a linear multivariate polynomial on all possible points, one can look at the code as the evaluation of *all* possible linear polynomials on a *single* point specified by the message vector. Such alternative interpretation becomes handy in certain analyses.

The idea of looking at the Hadamard code as the evaluation of all linear functions over a single point can be extended to construct codes that evaluate all quadratic polynomials on the specified point, or even all possible *functions* on certain point. The latter construction, known as the *long code*, leads to a code with double-exponentially long block length, hence the name! Even though it is not a priori clear, both these extensions turn out to be useful in several constructions of locally testable codes and proofs.

2.6 Oracle machines

In a locally testable code, the number of symbols in the alleged codeword read by the tester algorithm, known as its query complexity, is a major efficiency measurement. One can consider the running time of the tester as an obvious bound on its query complexity but such a measurements happens to be too rough. Instead, we need to extend the classical notion of the Turing machine to formally capture the new complexity parameter. To that end, we introduce the notion of oracle machines that we will extensively use later.

Definition 2.31. An *oracle machine* is a Turing machine with an extra random access and read-only tape called the *oracle tape*. Upon startup, a certain *oracle* string is assumed to be available on the oracle tape. During its operation, the machine can transit to a query state and read one symbol on the query tape. The total number of queries made by the Turing machine is an extra complexity measure of its complexity and is denoted by the *query complexity* of the machine. Like other complexity measurements, query complexity is generally a function of the input length. More precisely, the query complexity of the machine is defined as the function $q\colon \mathbb{Z}^+ \to \mathbb{Z}^+$ such that $q(n)$ is the the maximum, over all choices of the input strings of length n, of the number of queries made into the oracle tape. We use the shorthand form $M^\pi(x)$ to denote the boolean decision of an oracle machine M on input x and oracle π.

One may think of the oracle as a function provided as a *black box* to the Turing machine. The machine can query the oracle by providing the black box with a proper input, and receiving the evaluation of the function at the desired point in one query step. Then the total number of evaluation requests made by the machine will determine its query complexity. As Turing machines are formalizations of algorithms, oracle machines are also known as *oracle algorithms*.

Like a classical Turing machine, an oracle machine can be probabilistic, i.e., receive an external binary random string (a sequence of independent random bits, each uniformly distributed over the binary alphabet) along with its input but on a separate read-only tape. In that case, the query complexity is considered as the maximum of the number of queries over all choices of input strings of specific length *and* all the choices of the random bits for each input. The number of distinct random bits read by the Turing machine is considered as its *randomness complexity.* For our purposes, we shall need probabilistic oracle machines.

Definition 2.32. An oracle machine is called *adaptive* if each position of the oracle tape it queries is independent of the outcome of all previous queries (but it can of course depend on the input and the random bits, if available). In other words, the machine receives the outcome of all its queries at once, or equivalently, the positions at which the oracle is queried are independent of the content of the oracle.

Intuitively, adaptive machines seem to be more difficult to analyze. Thus we are particularly interested in probabilistic non-adaptive oracle machines. In fact, it is not clear if allowing adaptive oracle machines makes can lead to essentially more efficient constructions. Most known constructions of locally testable codes and proofs are based on non-adaptive machines. However, a negative result will be more convincing if it can be extended to the case of adaptive machines.

Chapter 3

A Toolbox of Polynomials

Most known constructions of the locally testable codes are algebraic and heavily rely on the properties of low-degree polynomial. In this chapter we introduce a few notations related to the polynomials and list a few useful facts regarding low degree polynomials.

We also examine the notion of low-degree tests that is implicit in most constructions of the locally testable codes and proof. We will review a few efficient methods for low-degree polynomial testing. Refer to Sudan's PhD thesis [Sud92] for a comprehensive study of polynomial checking.

3.1 Basic notions and facts

Informally, an m-variate *polynomial* over a commutative ring R (that we usually consider to be a finite field) is a summation of finitely many *monomials*, each of the form $\alpha X_1^{\alpha_1} X_2^{\alpha_2} \dots X_m^{\alpha_m}$, where $\alpha \in R$ is a constant, $\alpha_1, \dots, \alpha_m$ are nonnegative integers, and $X_1, \dots, X_m$ are indeterminates (i.e., variables) taking values from R. The total degree of such a monomial is defined as $\alpha_1 + \cdots + \alpha_m$, where α_i is referred to as the individual degree of X_i. A monomial is zero if its constant multiplier α is zero.

More precisely, for a commutative ring R, the univariate *polynomial ring* $R[X]$ has finite-length sequences $\alpha = \langle \alpha_0 + \cdots + \alpha_\ell \rangle$ as its elements that is interpreted as the summation $\sum_{i=0}^{\ell} \alpha_i X^i$, where X is an indeterminate in R. Addition and multiplication over $R[X]$ are defined in the natural way, i.e., if $\alpha = \langle \alpha_0, \dots, \alpha_k \rangle$ and $\beta = \langle \beta_0, \dots, \beta_\ell \rangle$ ($\ell \leq k$) are elements of $R[X]$, the summation $\alpha + \beta$ will be the sequence

$$\alpha + \beta = \langle \alpha_0 + \beta_0, \dots, \alpha_\ell + \beta_\ell, \alpha_{\ell+1}, \dots, \alpha_k \rangle,$$

whereas the multiplication $\gamma = \alpha \cdot \beta$ is the polynomial $\gamma = \langle \gamma_0, \ldots, \gamma_{k+\ell} \rangle$, where

$$\gamma_i = \sum_{j=0}^{i} \alpha_j \beta_{i-j}.$$

As the univariate polynomial ring is a commutative ring itself, we can construct the bivariate polynomial ring $R[X, Y]$ as $R[X][Y]$. In general, the m-variate polynomial ring is defined inductively as $R[X_1, \ldots, X_m] \stackrel{\text{def}}{=} R[X_1, \ldots, X_{m-1}][X_m]$.

A polynomial is the zero polynomial iff it has no nonzero monomials. The total degree of a polynomial p (denoted by $\deg(p)$) is defined as the smallest nonnegative integer d such that it has no nonzero monomial of total degree larger than d. For the special case of the zero polynomial, its degree is defined to be zero. The individual degree of the polynomial in any of its variables is defined similarly. The degree of a polynomial p in the variable X_i is denoted by $\deg_{X_i}(p)$. We will use the notation $R^{(d)}[X_1, \ldots, X_m]$ for the set of all m-variate polynomials of total degree at most d over R.

Definition 3.1. A nonzero univariate polynomial over a field is called *monic* if the monomial corresponding to its degree has a constant multiplier 1.

Definition 3.2. A polynomial $p \in R[X]$ over a commutative ring R is *reducible* if there exists $f, g \in R[X]$, each of degree at least one, such that $p = f \cdot g$. The polynomial p is called *irreducible* if it is not reducible.

Definition 3.3. For a commutative ring R, the *evaluation* of a polynomial $p = \langle p_0, \ldots, p_\ell \rangle \in R[X]$ at a point $\alpha \in R$, denoted by $p(\alpha)$, is the value of the summation $\sum_{i=0}^{\ell} p_i \alpha^i$.

Note that for any pair of polynomials f and g, we have $(f + g)(\alpha) = f(\alpha) + g(\alpha)$ and $(f \cdot g)(\alpha) = f(\alpha) \cdot g(\alpha)$.

Proposition 3.4. *(Division Algorithm) Let $f, g \in \mathbb{F}[X]$ be a pair of polynomials over the field $\mathbb{F}$. Then there exists a unique pair of polynomials $q, r \in F[X]$, denoted by* quotient *and* remainder*, respectively, such that* $\deg(r) < \deg(q)$ *and* $f = g \cdot q + r$. *Furthermore, if* $\deg(f) \geq \deg(g)$, *we have that* $\deg(q) = \deg(f) - \deg(g)$. ∎

When the remainder r in the division algorithm is the zero polynomial, we say that f *divides* g (or g is a *factor* of f). Plugging $g = X - \alpha$, we see that the polynomial $X - \alpha$ divides f if and only if $f(\alpha) = 0$, or in other words, α is a root of f.

Definition 3.5. Let R be a commutative ring. A sequence $\langle \alpha_1, \ldots \alpha_m \rangle \in R^m$ is a *root* of an m-variate polynomial $p \in R[X_1, \ldots, X_m]$ if $p(\alpha_1, \ldots, \alpha_m) = 0$.

Proposition 3.6. (Unique Factorization) *Let $\mathbb{F}$ be a field. A polynomial $p \in \mathbb{F}[X]$ has a* unique factorization *into a field element times a product of monic irreducible polynomials over the field, each of degree at least one.* ∎

Combining the above facts, one can easily prove Observation 2.25, i.e., a nonzero degree d univariate polynomial has at most d roots. Then this fact can be inductively generalized to the multivariate case that we have already seen as the Schwartz-Zippel lemma (Lemma 2.29), saying that a nonzero m-variate polynomial of total degree d evaluates to zero on at most a $\frac{d}{q}$ fraction of the points, where q is the field size over which the polynomial is defined.

Definition 3.7. For a commutative ring R of a finite size q, the *evaluation vector* of a polynomial $p \in R[X_1, \ldots, X_m]$, denoted by $\mathrm{eval}(p)$, is the vector $\vec{\eta}$ of length q^m indexed by the elements of R^m where for every $\vec{x} \in R^m$, $\eta_{\vec{x}} \stackrel{\text{def}}{=} p(\vec{x})$.

One can extend the above definition to define evaluation vectors of arbitrary m-variate functions. In fact, every vector of proper length can be considered as the evaluation vector of some function. Such a correspondence turns out to be useful in our applications.

Definition 3.8. The distance of two polynomials is defined as the distance of their evaluation vectors. In particular, a vector $\vec{w}$ of the proper length is said to be δ-close to (resp., δ-far from) a polynomial p if $\boldsymbol{\delta}(\vec{w}, \mathrm{eval}(p)) \leq \delta$ (resp., $\boldsymbol{\delta}(\vec{w}, \mathrm{eval}(p)) \geq \delta$).

Considering the above definition, an alternative and useful interpretation of the Schwartz-Zippel lemma is that every two distinct polynomials of total degree d over finite field $\mathbb{F}_q$ are $(1 - \frac{d}{q})$-far. This shows that (low-degree) polynomials are highly redundant objects, and thus *resilient* to errors. This is the key idea behind low-degree tests, that we will see later. The following corollary simply follows from the Schwartz-Zippel lemma and occasionally simplifies analyses:

Corollary 3.9. *If a function $f \colon \mathbb{F}_q^m \to \mathbb{F}_q$ is δ-close to a degree d and m-variate polynomial g, for some $\delta < \frac{1}{2}(1 - \frac{d}{q})$, then g is unique, in the sense that there is no other degree d polynomial that is δ-close to f.*

Proof: Suppose there is another polynomial $g' \neq g$ that is δ-close to f. By the triangle property of the distance function, the distance of g and g' can be at most 2δ, which is less than $1 - \frac{d}{q}$. But this is a contradiction by the Schwartz-Zippel lemma. ∎

Lemma 3.10. *(Lagrange Interpolation) For every sequence of n tuples of the elements in a field, there is a univariate polynomial of degree at most $n - 1$ that passes through all of them.*

Proof: Let $(x_1, y_1), \ldots, (x_n, y_n)$ be the desired sequence. Consider the polynomial $p(x) = \sum_{i=1}^{n} p_i(x)$, where

$$p_i(x) \stackrel{\text{def}}{=} y_i \prod_{\substack{j=1 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

It is easy to see that the polynomial $p(x)$ satisfies the condition of the lemma, i.e., $p(x_i) = y_i$. Note that if x_i's are chosen to be distinct, the interpolating polynomial of degree $n-1$ will be unique. ∎

Polynomial *curves* are also key objects in certain constructions of locally testable codes and proofs. In particular, they serve as basic tools in reduction of the number of queries which is normally carried out by *paralellization* of multiple queries into one. Firstly, we define general curves.

Definition 3.11. A *curve* C through a vector space $\mathbb{F}^m$ is a mapping $C\colon \mathbb{F} \to \mathbb{F}^m$ that takes a parameter t and returns a point $C(t) \in \mathbb{F}^m$. One can think of such a mapping as a collection of m functions, each of which a mapping from $\mathbb{F}$ to $\mathbb{F}$.

When the individual functions that specify a curve are polynomials, we get a *polynomial curve*:

Definition 3.12. A curve $C\colon \mathbb{F} \to \mathbb{F}^m$ is called a *polynomial curve* of degree d if it is a collection $\{c_i\}_{i=1,\ldots,m}$ of univariate mappings, each of which a univariate polynomial, such that the maximum degree of the polynomials in the collection is d.

The following fact is an immediate consequence of the polynomial interpolation lemma:

Fact 3.13. *Any set of $d+1$ points $x_1, \ldots, x_{d+1} \in \mathbb{F}^m$ can be interpolated by a curve of degree d.* ∎

One can restrict functions to a curve, in the following manner:

Definition 3.14. Let $f\colon \mathbb{F}^m \to \mathbb{F}$ be an m-variate function and $C\colon \mathbb{F} \to \mathbb{F}^m$ be a curve. The *restriction* of f to the curve C, denoted by $f|_C$, is a univariate function over $\mathbb{F}$ defined as $f|_C(t) \stackrel{\text{def}}{=} f(C(t))$.

It can be easily verified that the restriction of any degree d_1 polynomial by a degree d_2 polynomial curve is a univariate polynomial of degree $d_1 d_2$. The special case $d_2 = 1$ is of particular interest. A polynomial curve of degree one $C(t) = \mathbf{x} + t \cdot \vec{h}$ can be specified by a point $\mathbf{x}$ and a *slope* (also known as *offset*) vector $\vec{h}$. We call such a curve the *line* of slope $\vec{h}$ through $\mathbf{x}$ and denote it as $\ell_{\mathbf{x},\vec{h}}$. Note that any degree d polynomial restricted to a line becomes a univariate polynomial of degree d. Moreover, different values of $\mathbf{x}$ and $\vec{h}$ may correspond to the same line. For instance, the lines $\ell_{\mathbf{x},\vec{h}}$ and $\ell_{\mathbf{x},c\cdot\vec{h}}$ are the same for nonzero c.

3.2 Low-degree tests

The problem of low-degree testing is precisely the following:

Problem 3.15. *(Low-Degree Test) For some (finite) field $\mathbb{F}$, given a function $f : \mathbb{F}^m \to \mathbb{F}$ as an oracle and inputs $d \in \mathbb{Z}^+$ and $\epsilon \in (0, 1)$, distinguish (with a possibly randomized algorithm known as the* low-degree tester*) in polynomial time between the following two cases: (i.e., output* `PASS` *for the first case and* `FAIL` *for the second.)*

> **Case 1:** $f \in \mathbb{F}^{(d)}[X_1, \ldots, X_n]$,
>
> **Case 2:** $\boldsymbol{\delta}(f, \mathbb{F}^{(d)}[X_1, \ldots, X_n]) > \epsilon$.

The term *low-degree* is chosen to emphasize that the degree d is an independent parameter and in particular can be an absolute constant. However, the main application of the low-degree tests is on the occasions where the degree is a small enough function of the field size and the length of the evaluation vector of the given function.

The above problem can be formulated in several different (equivalent or closely related) ways. For instance, one may consider the problem of approximating the distance $\boldsymbol{\delta}(f, \mathbb{F}^{(d)}[X_1, \ldots, X_n])$ to within an additive factor ϵ. Alternatively, one may accept a small probability of making error on the second case. One may also assume presence of some *help* in a certain form from an additional *proof oracle*. We will clarify the latter idea later. It is also possible to pose the problem of recovering the coefficients of the polynomial as a part of the low-degree test.

Another problem which is related to low-degree testing is the problem of *self-correcting* low-degree polynomials. Assume that an evaluation vector suspected to correspond to a low-degree polynomial has passed a low-degree test and we are sure that it is close to some low-degree polynomial. Now we may need the evaluation of the corresponding polynomial on a certain point. Observe that it is a poor idea to simply probe the evaluation oracle at the corresponding point, as the vector might be corrupted just at that specific point. Instead, we would like to devise a reliable method to recover the correct evaluation with high probability, no matter at which specific point we need to evaluate the polynomial. It is interesting to note that certain low-degree testers can be modified in a straightforward manner to provide a low-degree correction mechanism. This also shows the resilience of polynomials upon errors.

Of course one can construct a low-degree tester by reading the given function entirely and check if it fits to a polynomial of degree d or not. However, the challenge is to construct highly efficient low-degree testers that probes the function oracle in very few positions and decides correctly (or *almost* correctly). In fact we will see that such efficient testers do exist.

3.2.1 Linearity test

The lowest possible degree parameter in a low-degree test is one, i.e., when we wish to test whether a given vector is the evaluation of a linear multivariate polynomial. This particular case is of special interest because testing linearity is a major ingredient in various code and proof testers. Note that a binary function f is linear if and only if $f(x+y) = f(x) + f(y)$, for all x and y.

Now we introduce a highly efficient linearity tester due to [BLR93]. The construction of the tester is very simple and natural, but analyzing its behavior turns out to be fairly complex. The following algorithm describes the testing procedure for the binary field that requires only three queries: (We use the notion $x \in_{\mathrm{R}} \Omega$ to indicate that x is a uniformly random sample of the set Ω.)

Algorithm 3.16	LINEARITY-TEST(f, n)
Input	An oracle function $f \colon \mathbb{F}_2^n \to \mathbb{F}_2$.
Output	A verdict `ACCEPT/REJECT`.

1. Pick $\mathbf{x}, \mathbf{y} \in_{\mathrm{R}} \mathbb{F}_2^n$.
2. Accept if and only if $f(\mathbf{x}) + f(\mathbf{y}) = f(\mathbf{x} + \mathbf{y})$. ∎

The following proposition shows the effectiveness of the above test:

Proposition 3.17. [BCH+96] *The above test is complete and sound. Precisely, we have the following:*

Completeness: *If f is linear then the test accepts with probability* 1.

Soundness: *If f is δ-far from being linear for some $\delta \le \frac{1}{2}$, that is,*

$$\boldsymbol{\delta}(f, \mathbb{F}^{(1)}[X_1, \ldots, X_n]) \ge \delta,$$

then the test rejects with probability at least $\Gamma(\delta)$, where the function Γ is defined as

$$\Gamma(\delta) \stackrel{\mathrm{def}}{=} \begin{cases} 3\delta - 6\delta^2 & \text{for } 0 \le \delta \le \frac{5}{16}, \\ \frac{45}{128} & \text{for } \frac{5}{16} \le \delta \le \frac{45}{128}, \\ \delta & \text{for } \frac{45}{128} \le \delta \le \frac{1}{2}. \end{cases}$$ ∎

The three-case function Γ above is rather inconvenient. As we are mainly concerned with small values of δ, it is more convenient to simplify the above proposition as the following corollary:

Corollary 3.18. *In the linearity test, if f is a linear function then the test accepts with probability* 1*. Furthermore, if for some $\delta \leq 0.2$, the probability that the test accepts is more than $1 - \delta$ then f is 2δ-close to being linear.* ∎

Linear functions are also self correctable in a similar fashion. The following linear self-corrector is also due to [BLR93]:

Algorithm 3.19	LINEAR-SELF-CORRECT$(f, \mathbf{x}, n)$
Input	An oracle function $f\colon \mathbb{F}_2^n \to \mathbb{F}_2$, and a point $\mathbf{x} \in \mathbb{F}_2^n$.
Output	The corrected value of f at point $\mathbf{x}$.

1. Pick $\mathbf{y} \in_{\mathrm{R}} \mathbb{F}_2^n$.
2. Output $f(\mathbf{x} + \mathbf{y}) - f(\mathbf{y})$. ∎

Proposition 3.20. [BLR93] *The behavior of the self-correction algorithm above is as follows:*

Completeness: *If f is linear then the corrector outputs $f(\mathbf{x})$ correctly with probability* 1*.*

Soundness: *If f is δ-close to a linear function g, the above procedure outputs $g(\mathbf{x})$ with probability at least $1 - 2\delta$.* ∎

3.2.2 Univariate low-degree test

Now we consider the another extreme of the low-degree test, where the number of variables is one. Here we consider tests based on [RS96, RS92]. The first test uses a basic interpolation technique and the following characterization of the low-degree polynomials:

Proposition 3.21. *A function f is a univariate polynomial of degree at most d over a field of order at least $d + 2$ if and only if for all choices of $d + 2$ points $x_0, \ldots, x_{d+1}$, there exists a polynomial $g_{x_0,\ldots,x_{d+1}}$ of degree at most d that agrees with f at all the x_i's, i.e., $g_{x_0,\ldots,x_{d+1}}(x_i) = f(x_i)$ for all $i = 0, \ldots, d + 1$.*

Algorithm 3.22	UNIVARIATE-TEST(f, d)
Input	An oracle function $f\colon \mathbb{F} \to \mathbb{F}$, and a degree parameter d.
Output	A verdict `ACCEPT/REJECT`.

1. Repeat the following for a constant number of times:

   (a) Pick distinct points $x_0, \ldots, x_{d+1} \in_{\mathrm{R}} \mathbb{F}$.
   (b) Reject if f is not a degree d polynomial on $x_0, \ldots, x_{d+1}$.

2. Accept. ∎

The effectiveness of the above test is confirmed by the following proposition:

Proposition 3.23. [Sud92] *The above test has perfect completeness (i.e., if f is a polynomial of degree d it accepts with probability 1). Moreover, for a field $\mathbb{F}$ of size at least $d+2$, if f is δ-far from being a degree d polynomial, the test rejects with probability at least δ. Moreover, the test queries f in only $O(d)$ places.* ∎

The second test is based on *evenly spaces* points, defined as follows:

Definition 3.24. A set of points $\{\mathbf{x_0}, \ldots, \mathbf{x_n}\}$ in a vector space $\mathbb{F}^m$ is said to be *evenly spaced* if there exists an offset $\vec{h} \in F^m$ such that $\mathbf{x_i} = \mathbf{x_0} + i \cdot \vec{h}$ for $i = 0, \ldots, n$.

With the help of evenly spaced points, polynomials over the field $\mathbb{Z}_p$ (for some prime p) can be characterized in the following well-known way:

Characterization 3.25. [RS96] *A function $f\colon \mathbb{Z}_p \to \mathbb{Z}_p$ is a polynomial of degree at most d if and only if for all $x, h \in \mathbb{Z}_p$ we have $\sum_{i=0}^{d+1} \alpha_i f(x + i \cdot h) = 0$, where the coefficients α_i are defined as $\alpha_i \stackrel{\text{def}}{=} (-1)^{(i+1)} \binom{d+1}{i}$.* ∎

The above characterization naturally implies the following test that can be implemented with $O(d^2)$ additions and subtractions in each iteration and no use of the multiplication and division operations (cf. [Sud92]):

Algorithm 3.26	UNIVARIATE-EVENLY-SPACED-TEST(f, d, β)
Input	An oracle function $f\colon \mathbb{Z}_p \to \mathbb{Z}_p$, a degree parameter d, and a confidence fraction β.
Output	A verdict `ACCEPT/REJECT`.

1. Repeat the following for $O(d^2 \log(\frac{1}{\beta}))$ number of times:
   (a) Pick $x, h \in_{\mathrm{R}} \mathbb{Z}_p$.
   (b) Reject if $\sum_{i=0}^{d+1} \alpha_i f(x + i \cdot h) \neq 0$.

2. Accept. ∎

Proposition 3.27. [RS96] *Algorithm 3.26 has perfect completeness. Moreover, if f is not $O(\frac{1}{d^2})$-close to a degree d polynomial, the algorithm rejects with probability at least $1 - \beta$.* ∎

3.2.3 Multivariate low-degree test

We finally consider the case of an arbitrary degree and number of variables. The evenly-spaced test that we introduced for univariate polynomials extends to the multivariate case in a straightforward manner.

Algorithm 3.28	MULTIVARIATE-EVENLY-SPACED-TEST(f, d, β)
Input	An oracle function $f\colon \mathbb{Z}_p^m \to \mathbb{Z}_p$, a degree parameter d, and a confidence fraction β.
Output	A verdict `ACCEPT/REJECT`.

1. Repeat the following for $O(d^2 \log(\frac{1}{\beta}))$ number of times:
   (a) Pick $\mathbf{x}, \vec{h} \in_{\mathrm{R}} \mathbb{Z}_p^m$.
   (b) Reject if $\sum_{i=0}^{d+1} \alpha_i f(\mathbf{x} + i \cdot \vec{h}) \neq 0$.
2. Accept. ▮

Proposition 3.29. [RS96] *The above algorithm has perfect completeness. Moreover, if f is not $O(\frac{1}{d^2})$-close to a degree d polynomial, the algorithm rejects with probability at least $1 - \beta$.* ▮

We have already seen that the restriction of a polynomial of degree d on any line is a univariate polynomial of degree d. The following characterization shows that the converse is also true:

Characterization 3.30. [FS95] *Let $\mathbb{F}$ be a finite field of order q and characteristic p such that $q - \frac{q}{p} - 1 \geq d$. Then a function $f\colon \mathbb{F}^m \to \mathbb{F}$ is a polynomial of degree at most d if and only if for all $\mathbf{x}, \vec{h} \in \mathbb{F}^m$, the function f restricted to the line $\ell_{\mathbf{x},\vec{h}}$ is a univariate polynomial of degree d.* ▮

Even more, one can show that if the restriction of f on *most* lines is *close* to a polynomial of degree d, then f itself is close to a multivariate polynomial of total degree at most d. In other words, the characterization is *robust*.

Inspired by these facts, we propose a *proof-assisted* low-degree test for multivariate polynomials. Such a test is different from the ones we have already studied in the sense that it assumes the presence of an auxiliary oracle that is supposed to help the tester become convinced that the function oracle is indeed a low-degree polynomial more efficiently and in particular with less queries. One may think of the auxiliary oracle as an external *prover* with unlimited computational power trying to *convince* the tester in an *interactive* manner. Such an interpretation is known as *interactive proof* schemes.

As with any other oracle, the queries made into the auxiliary oracle is also counted as a part of the query complexity of the tester. However, the size of the auxiliary oracle expected by the tester is considered a drawback and should be analyzed separately.

Note that for the tester it is does not matter how the auxiliary oracle is constructed or obtained. The existence of such an oracle might seem to be an odd and intangible requirement; However, in the applications of low-degree tests the extra oracle is *embedded* in the construction (e.g., as a part of the codeword in a locally testable cod) in a natural manner. We will see such constructions in later chapters.

The following proof-assisted multivariate test is also proposed in [RS96] and has also been appeared in [AS98, ALM$^+$98]:

Algorithm 3.31	Polynomial-Line-Test$(f, B, m, d, \mathbb{F})$
Input	An oracle function $f\colon \mathbb{F}^m \to \mathbb{F}$, a degree parameter d, and an auxiliary oracle function $B\colon \mathbb{F}^{2m} \to \mathbb{F}^{(d)}[X]$.
Output	A verdict `ACCEPT/REJECT`.

1. Pick $\mathbf{x}, \vec{h} \in_{\mathrm{R}} \mathbb{F}^m$ and $t \in_{\mathrm{R}} \mathbb{F}$.
2. Accept if and only if $f(\mathbf{x} + t \cdot \vec{h}) = B(\mathbf{x}, \vec{h})(t)$. ∎

The following proposition is a direct consequence of Characterization 3.30:

Proposition 3.32. [RS96] *The test in Algorithm 3.31 is complete and sound, and it makes only one query to each oracle. Precisely, we have the following:*

Completeness: *If f is a polynomial of degree d then there exists an auxiliary oracle with respect to which the test accepts with probability* 1*.*

Soundness: *For every $\epsilon > 0$, there exists absolute constant c such that if $\mathbb{F}$ is a field of size at least cd, then if for some auxiliary oracle the above test rejects with probability $\delta \leq \frac{1}{8} - \epsilon$, the function f is 2δ-close to a degree d polynomial.* ∎

Hence, if the given function is δ-far from being a low-degree polynomial, the above test rejects with probability $\Omega(\delta)$, no matter how we choose the evaluations of the auxiliary oracle. One can transform this to a high-probability statement by repeating the test for $O(\frac{1}{\delta^2})$ times (cf. [Sud92]).

Proposition 3.32 asserts that if the function f is in fact a degree d polynomial, then there exists an auxiliary oracle B that makes the test always accept. It is clear to see that such an oracle is in fact the restrictions of f to all possible lines in $\mathbb{F}^m$, known as the *lines representation* of f. Let us denote the lines representation by the short form lines_f. Note that the number all possible (but not necessarily distinct) lines is $\mathbb{F}^{2m}$ and the restriction of f to each of them is a degree d univariate polynomial expressible by its $d+1$ coefficients. Hence, the size of the auxiliary oracle needed is $(d+1)\,|\mathbb{F}|^{2m}\log|\mathbb{F}|$ bits.

Based on the same idea, it is possible to propose the following self-correction algorithm for multivariate low-degree polynomials. This self-corrector is allowed to fail in certain occasions.

Algorithm 3.33	$\textsc{Polynomial-Line-Correct}(f, B, \mathbf{x}, m, d, \mathbb{F})$
Input	An oracle function $f\colon \mathbb{F}^m \to \mathbb{F}$, a degree parameter d, an auxiliary oracle function $B\colon \mathbb{F}^{2m} \to \mathbb{F}^{(d)}[X]$, and a point $\mathbf{x} \in \mathbb{F}^m$.
Output	The corrected value of f at point $\mathbf{x}$.

1. Pick $\vec{h} \in_{\mathrm{R}} \mathbb{F}^m$ and $t \in_{\mathrm{R}} \mathbb{F} \setminus \{0\}$.
2. Let $q = B(\mathbf{x}, \vec{h})$.
3. **if** $q(t) \neq f(\mathbf{x} + t \cdot \vec{h})$ **then** reject **else** output q(0). ∎

Proposition 3.34. [ALM+98, GLR+91] *If f is indeed a degree d polynomial, the above procedure always outputs the correct value of $f(\mathbf{x})$ when provided with lines_f as the auxiliary oracle. Moreover, if f is δ-close to a degree d polynomial g, then the probability (over random choices of h and t in the algorithm) that the corrector does not output $g(\mathbf{x})$ is at most $2\sqrt{\delta} + \frac{d}{|\mathbb{F}|-1}$.* ∎

Chapter 4

Efficient Verification of Codewords

In this chapter we examine the formal notion of locally testable codes and review several constructions of such codes. A natural problem in coding theory is efficient detection of errors in a code. That is, for a given string of the proper length, deciding if the string is a valid codeword or not.

It is desirable to construct families of codes that allow efficient detection of errors. Linear codes for instance, have a universal procedure for error detection. Namely, one can multiply the given vector by the parity check matrix of the code and verify if the result is zero or not. However, such a procedure requires reading all the information in the suspected codeword and normally much more than linear (with respect to the block length) time.

Informally, a locally testable code are families of codes that allow a super-efficient error detection procedure. Namely, in a locally testable code errors can be detected in sublinear time and by only reading a negligible fraction (say constant) of the given string. Clearly one can never be sure of the correctness of the final decision without reading the entire information, as in the extreme case error can occur only at one coordinate, and finding such a coordinate needs reading the entire string. Hence, we allow the error detection to be probabilistic and err with small probability.

4.1 Definition

The notion of locally testable codes independently appeared in [FS95, RS96, Aro94, Spi95] under different names. Here we describe the precise formulation of locally testable codes and various parameters related to them.

Definition 4.1. Let $\mathcal{C}$ be an (n, k, d)-code over the alphabet Σ, and let $q \in \mathbb{N}$, and $\delta, \epsilon \in (0, 1)$. A (q, δ, ϵ)-tester (also known as a q-local δ-tester with error probability ϵ) is a probabilistic non-adaptive oracle machine M with query complexity at most q that runs in polynomial time and satisfies the following conditions:

Completeness. The oracle machine always accepts codewords of $\mathcal{C}$, i.e., for any $\vec{x} \in \Sigma^k$, $\Pr[M^{\mathcal{C}(\vec{x})}(1^k) = 1] = 1$.

Soundness. The oracle machine rejects non-codewords that are δ-far from $\mathcal{C}$ with bounded probability. That is, for any $\vec{w}$ that is δ-far from $\mathcal{C}$, we have $\Pr[M^{\vec{w}}(1^k) = 0] \geq 1 - \epsilon$. The probability is taken over the choices of the random bits read by the machine.

The operation of the codeword tester can be thought of as follows: Having received a random binary string $\vec{r}$ as input, the oracle machine produces in polynomial time a sequence of queries $i_1, \ldots, i_q$ and a boolean circuit C that receives q inputs. It then queries the oracle string $\vec{w}$ and accepts if and only if $C(w_{i_1}, \ldots, w_{i_q}) = 1$.

The parameter δ is called the proximity parameter of the tester. We will omit the error parameter ϵ when it is an absolute constant, say, $\frac{1}{2}$. Note that any absolute constant in $(0, 1)$ will do, because once the the error probability is bounded by an absolute constant, one can repeat the test for a constant number of times to reduce it to any desired constant. Such an error reduction comes at the expense of constant factors in the complexity of the oracle machine which is negligible. The complement of the error probability, $1 - \epsilon$, is called the *detection probability* of the tester and is more handy in certain occasions.

Now we extend the above definition to a family of codes, albeit we shall occasionally abuse the notation and use the terms code and family of codes interchangeably, whenever it is clear from the context:

Definition 4.2. Let $\mathcal{C} = \{\mathcal{C}_k\}$ be a family of codes where each $\mathcal{C}_k$ is an $(n(k), k, d(k))$-code. For functions $q\colon \mathbb{N} \to \mathbb{N}$ and $\delta, \epsilon\colon \mathbb{N} \to (0, 1)$, a probabilistic non-adaptive oracle machine M is called a (q, δ, ϵ)-tester for $\mathcal{C}$ if it is a $(q(k), \delta(k), \epsilon(k))$-tester for every $\mathcal{C}_k$. In other words, we require the whole family to be uniformly testable with the same oracle machine.

We are finally at a position to formally define locally testable codes:

Definition 4.3. A family of codes is (q, δ)-locally testable if there exists a (q, δ)-tester for it.

Note that the definition of the locally testable codes is essentially meaningless if we allow an arbitrary minimum distance for the code. In particular, for a code with too many codewords (say the identity code), a trivial tester that always accepts will do. Hence, we are only interested in codes with acceptably high (and in particular constant) minimum distance.

Moreover, we need the proximity parameter δ to be smaller than the covering radius of the code, as otherwise, no string can be δ-far from the code! We are also particularly interested in locally testable codes with constant query complexities. However, a query complexity being a very slowly growing function of the message length is also acceptable. So we shall follow the convention that all locally testable codes have constant query complexities, unless specified otherwise.

Meeting the above conditions is enough for a locally testable code; However, locally testable codes are also defined by the following slightly stronger definition of the codeword tester:

Definition 4.4. Let $\mathcal{C}$ be an (n, k, d)-code over the alphabet Σ, and let $q \in \mathbb{N}$, and $\delta, \epsilon \in (0, 1)$. A q-tester is a probabilistic non-adaptive oracle machine M with query complexity at most q that runs in polynomial time and satisfies the following conditions:

Completeness. The oracle machine always accepts codewords of $\mathcal{C}$, i.e., for any $\vec{x} \in \Sigma^k$, $\Pr[M^{\mathcal{C}(\vec{x})}(1^k) = 1] = 1$.

Soundness. The oracle machine rejects non-codewords with a probability proportional to their distance from the code. That is, for any $\vec{w}$ that is δ-far from $\mathcal{C}$, we have $\Pr[M^{\vec{w}}(1^k) = 0] = \Omega(\delta)$.

Definition 4.5. A family of codes is q-locally testable if there exists a q-tester for it.

The latter definition in stronger in the sense that for a constant δ, every q-tester can be converted into a $(O(q), \delta, \frac{1}{2})$-tester by a simple repetition of the test for a constant number of times. Thus, any q-locally testable code is also $(O(q), \delta, \frac{1}{2})$-locally testable. On the other hand, the strong tester has a nonzero rejection probability for every non-codeword, while this is not a requirement in the initial definition. However, many known constructions of the locally testable codes also satisfy the strong conditions.

It is also plausible to allow two-sided errors. That is, we let the oracle machine to err for the "completeness" case. Moreover, the standard definition of locally testable codes does not require anything when the oracle not a codeword, but is close to one. In fact, it also makes sense to extend the "completeness margin." Such an extension is formalized in [GR05] under the notion of *tolerant locally testable codes*. Their definition involves two proximity parameters instead of one and requires that if the oracle is close to (resp. far from) a codeword, the tester must accept (resp. reject) with probability $\frac{2}{3}$. They also show how to adapt several known constructions to work with the modified notion.

In construction of locally testable codes, the following parameters are of specific interest and need to be optimized:

- Rate of the code,

- The alphabet size of the code,
- Query complexity of the oracle machine,
- Randomness complexity of the oracle machine.

Note that the minimum distance of the code is typically restricted to a constant value, as we argued earlier. It turns out that there is an inevitable trade-off between the above parameters and the challenge is to construct (or at least, show the existence of) codes that achieve asymptotically optimal (or near-optimal) values for all criteria. Ideally, we would like to construct a code with constant rate, constant query complexity and logarithmic randomness. Moreover, the alphabet size should be constant and we will be most content with a binary alphabet.

It turns out that as the query complexity of the code decreases, its rate has to increase, and it is not clear if all parameters above can be simultaneously optimized. Thus, a negative result (showing that certain nontrivial combinations are not achievable) would also be interesting. In fact, such negative results do exist. In particular, it is shown in [BGS03] that a locally testable code with constant distance defined over a constant-sized alphabet (say binary) that is testable with two queries can only have a constant number of codewords.

At the time of this write-up, the following is the major open problem in the area of locally testable codes:

Question 4.6. *Do asymptotically good locally testable codes with constant query complexity exist?*

There is a negative evidence regarding this question, due to Ben-Sasson and Harsha [BSHR03]. They show that with high probability, every tester for a randomly constructed regular LDPC code needs a linear amount of query, even if we allow adaptiveness and two-sided errors. Of course, this does not imply the impossibility of asymptotically good locally testable codes, even among regular LDPC codes, but it indicates non-triviality of such constructions, if they exist.

As we will see later, even though the above question is not yet answered, locally testable codes with constant distance and *nearly linear* block length have been explicitly constructed, even over the binary alphabet. Now one might wonder what block lengths are considered "nearly linear?" Inspired by [Gol05], we consider the following classifications of the nearly liner term, ordered from loose to tight:

Constant nearly linear: In this most liberal case, a sequence of functions $f_\epsilon \colon \mathbb{N} \to \mathbb{N}$ is considered to be nearly-linear if for every constant $\epsilon > 0$ it holds that $f_\epsilon(n) = O(n^{1+\epsilon})$. In the context of codes, this would mean that for every constant $\epsilon > 0$ we can construct a family $\{\mathcal{C}_k\}$ of codes with the block length at most $O(k^{1+\epsilon})$.

Tight nearly linear: In the tight case, a function $f\colon \mathbb{N} \to \mathbb{N}$ is said to be nearly linear if we have $f(n) = O(n^{1+o(1)})$. This is a broad class with several particularly interesting sub-cases, namely, for the sub-constant term in the exponent being $\frac{1}{\log\log n}$, $\frac{1}{(\log n)^c}$, or $\frac{\exp((\log\log\log n)^c)}{\log n}$, for some constant $c \in (0,1)$.

Polylog nearly linear: This strongest form is in fact a special case of the second category that deserves special attention. It is obtained by letting the sub-constant function in the exponent of $n^{1+o(1)}$ be $\frac{\mathrm{poly}(\log\log n)}{\log n}$, which yields $f(n) = O(n \cdot \mathrm{poly}(\log n))$. For the case of locally testable code, this corresponds to codes with poly-logarithmic rate.

Now, consider a family $\{\mathcal{C}_k\}$ of locally testable codes with query complexity $q(k)$, randomness complexity $r(k)$, and block length $n(k)$. Note that once the tester is provided with an oracle of length $n(k)$, it has $2^{r(k)}$ possible executions, depending on the choice of random bits fed to it. Moreover, in each execution it probes the oracle on at most $q(k)$ positions. Hence (based on the assumption that the tester is non-adaptive) the length of the oracle need not be more than $2^{r(k)} \cdot q(k)$, as otherwise certain parts of it are not ever accessed by the tester and would be essentially redundant.

Thus, we have the upper bound $2^{r(k)} \cdot q(k)$ on the block length of a locally testable code. This bound turns out to be tight enough for most known constructions and hence one can only focus on the trade-off between randomness and query complexity of the code on a certain alphabet. In particular, a separate analyses for the block length and rate is not normally needed.

Recall that the soundness condition of the codeword tester implies that if the given oracle is far from the code, it must be rejected by the tester with a sufficiently high probability. It is often useful to add an additional requirement that in such cases the tester should *strongly reject* non-codewords. Suppose that given a random string s as input, the tester produces a circuit C that gets the response to the queries made by the tester as input and outputs a verdict. Then the additional requirement, known as the *robustness* of the tester, implies that if an oracle is far from the code, it should also be far from satisfying the circuit C. This notion was proposed in [BS04], motivated by robust proof verifiers of [BGH$^+$04].

More precisely, suppose that for a random string $\vec{s}$ received as input, the tester produces a decision circuit C, and q queries $i_1, \ldots, i_q \in [n]$, where n is the block length of the code. For oracle $\vec{r}$ and random string $\vec{s}$, define the robustness of the tester T on $\vec{r}$ and $\vec{s}$, denoted $\rho^T(\vec{r}, \vec{s})$, to be the minimum, over strings $\vec{x}$ that satisfy the circuit C, of relative distance of $\langle r_{i_1}, \ldots, r_{i_q} \rangle$ from $\vec{x}$. Note that this quantity is zero if r is chosen to be a codeword. We will also refer to the expectation $\rho^T(\vec{r}) \stackrel{\text{def}}{=} \mathbb{E}_{\vec{s}}[\rho^T(\vec{r}, \vec{s})]$ as the *robustness* of T on $\vec{r}$. Now we define a robust tester as follows:

Definition 4.7. A tester T is said to be c-robust for a code $\mathcal{C}$ if the tester accepts codeword of $\mathcal{C}$ with probability one (i.e., it has the completeness property) and every string r of length n (where n is the block length of the code) is $c \cdot \rho^T(r)$-close to $\mathcal{C}$. That is, the expected robustness of the tester on every oracle string must be at least proportional to the relative distance of the string from the code.

It turns out that robust testers compose more easily, and robustness makes recursive constructions more convenient. Moreover, robustness implies local testability. Precisely, we have the following:

Proposition 4.8. [BS04] *If a code $\mathcal{C}$ has a c-robust tester T making q queries, then it is locally testable with $O(c \cdot q)$ queries.*

Proof. Let $c' = \lceil c \rceil$. Clearly, $\mathcal{C}$ is also c'-robust. Consider performing the robust test for c' rounds and accepting if and only if they all accept. The completeness of the new tester is clear. Moreover, the robustness property implies that for non-codewords of distance δ from the code, the probability that none of the tests reject is at most $(1 - \frac{\delta}{c})^{c'}$ which turns out to be at most $1 - \frac{\delta}{2}$. Hence the soundness condition is also satisfied. ∎

4.2 Constructions

Now that we have introduced the notion of locally testable codes, a natural question to ask is that what families of such codes are constructible? By constructing a family of locally testable code, we mean an *explicit construction*, i.e., the following steps:

- Give an explicit (and deterministic) description of an efficient algorithm for encoding messages of length k, for infinite possibilities of k.
- Give the description of a codeword testing algorithm, which is by definition a probabilistic polynomial time oracle machine.
- Analyze different parameters of the code and the underlying tester. In particular, query complexity, randomness complexity, the alphabet size, the proximity parameter, and soundness error.

4.2.1 Testing Reed-Muller codes

Many explicit constructions of locally testable codes immediately follow from a polynomial low-degree test construction. One interesting example is the families of Reed-Muller codes. Recall that the codewords of a Reed-Muller code are essentially evaluation vectors of polynomials of certain degree whose coefficients are described by the message vectors. Hence, a codeword test for a Reed-Muller code is nothing but a low-degree test that ensures that the given codeword is indeed the representation of a low-degree polynomial.

In particular, the linearity test (Corollary 3.18) reveals that the Hadamard code that we studied in section 2.5 are locally testable merely with three queries. However, the block length of this code is 2^k, which is exponentially large. Moreover, the randomness complexity of the tester is $3k$, which is large. But for a long time the Hadamard code was the only locally testable binary code known. In fact current constructions of binary locally testable codes that also achieve a good (albeit non-optimal) rate mainly make use of the Hadamard code to reduce the alphabet size to two.

In a similar way, one can construct local testers for Reed-Muller codes of constant degrees (i.e., when the degree of the multivariate polynomial described by the message string is constant). This can be done using the multivariate low-degree described in Algorithm 3.28 which is based on evenly spaced points.

4.2.2 Polynomial-Line codes

In this section, motivated by the proof-assisted low-degree test of Algorithm 3.31 we introduce the family of polynomial-line codes that has also been appeared in [FS95]. The construction of polynomial-line codes is similar to that of Reed-Muller codes. The message vector is interpreted as an m-variate polynomial of degree d over a finite field $\mathbb{F}$, where typically $m \leq d < |\mathbb{F}|$ (more precisely, in a typical configuration, d is picked such that $d = \Theta(m^{c_1})$ for some $c1 > 1$ and the order of the field is chosen to be of order $\Theta(d^{c_2})$ for some $c_2 \geq 1$). In contrast to the Reed-Muller codes, the encoding of a polynomial p (given as the message vector) in the polynomial-line code is the line-representation vector (lines_p) of p, i.e., the restriction of p to all possible lines in $\mathbb{F}^m$, each of which represented by a sequence of $d+1$ coefficients in $\mathbb{F}$.

Note that this code is linear. Now, let us consider the vector space $\Sigma = \mathbb{F}^{d+1}$ as the underlying alphabet of the code. Let k and n denote the message length and the block length of the code, respectively. Then we have

$$n = |\mathbb{F}|^{2m},$$

and,

$$\begin{aligned} k &= \frac{\log \left|\mathbb{F}^{(d)}[X_1, \ldots, X_m]\right|}{\log |\Sigma|} \\ &= \frac{\binom{m+d}{d} \log |\mathbb{F}|}{(d+1)\log |\mathbb{F}|} \\ &= \frac{\binom{m+d}{d}}{d+1} \\ &\overset{m \ll d}{\approx} \left(\frac{d}{m}\right)^m . \end{aligned}$$

If we take a field $\mathbb{F}$ of order polynomial in d and choose m to be small enough with respect to d, we get a code with polynomially large block length.

The following test for the polynomial-line code is inspired by the low-degree test of Algorithm 3.31:

Algorithm 4.9	POLYNOMIAL-LINE-CODE-TEST$(w, m, d, \mathbb{F})$
Input	An alleged codeword $w \in \mathbb{F}^{2m}$, the number of variables m, a degree parameter d, and the underlying field $\mathbb{F}$.
Output	A verdict `ACCEPT/REJECT`.

1. Pick $\mathbf{r} \in_{\mathrm{R}} \mathbb{F}^m$.
2. Pick two lines ℓ, ℓ' in $\mathbb{F}^m$ going through $\mathbf{r}$, uniformly and independently at random. Let t and t' be such that $\ell(t) = \ell(t') = \mathbf{r}$.
3. Let w_ℓ and $w_{\ell'}$ be the coordinates of w corresponding to the lines ℓ and ℓ', respectively.
4. Accept if and only if $w_\ell(t) = w_{\ell'}(t')$. ▮

A similar analysis as in the case of low-degree test reveals that the above test satisfies the required completeness and soundness conditions. That is, if w is a codeword, it passes the test. Moreover, if w is δ-far from the code and the field size is $\Omega(d)$, it is rejected with probability $\Omega(\delta)$.

The problem with the code is that its block length is a quadratic function of $|\mathbb{F}|^m$, which makes the rate small. A sharper analysis of the test in [GS02] using the probabilistic method indicates that a random truncation of the polynomial-line code works as well[1]. In particular, they simply project the code on a randomly chosen

[1] However, the randomized truncation step might be argued to make the construction lose its explicitness.

subset R_m of the coordinates, where $|R_m| = O(|\mathbb{F}|^m \log |\mathbb{F}|)$. In other words, such a truncation is made by considering restrictions of the input polynomial over the lines in R_m, rather than all possible lines. Then the codeword test is modified such that it picks the random lines ℓ and ℓ' from the set R_m.

In this new setting, the message length k is as before, and for $m \ll d$ can be approximated by d^{m-1}/m^m. Moreover, for $|\mathbb{F}| = O(d)$, we have that $n = O(|\mathbb{F}|^m \log |\mathbb{F}|) = O(d)^m$. The following special cases turn out to be particularly interesting:

1. Let $d = m^m$. This yields $k \approx m^{m^2-2m}$ and $n = m^{m^2+o(m)}$ and thus, $n \approx \exp(\sqrt{\log k}) \cdot k$ and $\log |\Sigma| = \log |\mathbb{F}|^{d+1} \approx d \log d \approx \exp(\sqrt{\log k})$. This gives us a locally testable code of "tight nearly linear" block length over a fairly large alphabet.

2. Let $d = m^c$ for a constant $c > 1$. We get $k \approx m^{(c-1)m}$ and $n \approx m^{cm}$ and thus, $n \approx k^{c/(c-1)}$ and $\log |\Sigma| \approx d \log d \approx \log^c k$. The rate of the code becomes worse in this setting; However, the alphabet size is a poly-logarithmic function of the message length.

The following lemma indicates that the truncated code preserves the soundness condition:

Lemma 4.10. [GS02] *Let $|\mathbb{F}| = \Omega(d)$ and $|\mathbb{F}| < \exp(m^m)$. Then, for $1 - o(1)$ fraction of the possible choices of R_m the following holds for every $w \in \Sigma^n$: If w is δ-far from the truncated code defined by R_m, then the codeword test rejects w with probability $\Omega(\delta) - o(1)$.* ▮

In the next step the alphabet size of the truncated code, which is a quite large value $|\mathbb{F}|^{d+1}$, is reduced to $|\mathbb{F}|$, while preserving the performance of the code. This is carried out in [GS02] by concatenating the truncated code with a particular linear inner code.

The inner code $\mathcal{C}'$ maps multilinear forms to their evaluations under all possible assignments. In particular, consider a total degree d' and a positive integer h. Moreover, consider the set of $d'h$ variables $x_i^{(j)}$, where $j \in [d']$ and $i \in [h]$. Then the message vectors are considered as the coefficients of the degree d' multilinear form

$$\sum_{i_1,\ldots,i_{d'} \in [h]} c_{i_1,\ldots,i_{d'}} \cdot x_{i_1}^{(1)} \cdot x_{i_2}^{(2)} \cdots x_{i_{d'}}^{(d')}$$

and their encodings are the evaluation vectors of the above form over $\mathbb{F}^{d'h}$ (i.e., over all possible assignments of the $d'h$ variables in the multilinear form).

Note that the message length (k') and the block length (n') of the above inner code are as follows:

$$k' = h^{d'},$$

and,

$$n' = |\mathbb{F}|^{d'h} = \exp(d' \cdot (k')^{\frac{1}{d'}} \cdot \log |\mathbb{F}|).$$

As the valid codewords of the inner code are multilinear functions of total degree d', a tester for the inner code can be constructed by d' linearity tests (with respect to multivariate sets $\{x_i^{(1)}\}, \ldots, \{x_i^{(d')}\}$, where $i \in [h]$) and an additional total degree test to ensure that the whole representation is a polynomial of total degree d' with respect to all $d'h$ variables. It can be shown that this simple test is complete and sound. In particular, if a vector $w' \in \mathbb{F}^{n'}$ is δ-far from $\mathcal{C}'$, then the codeword test rejects with probability $\Omega(\delta)$. It can also be shown that the codeword test works as well if we replace the conjunction of d' linearity tests with only one of them, selected uniformly at random.

To construct a low-query tester for any concatenated code, one needs to adapt the tester for the outer code to work with the concatenated version. Note that the alphabet of the concatenated code is normally much smaller than the one for the outer code and simply reading the portion of the oracle corresponding to the queries of the outer verifier requires a lot of queries. In particular, queries made by the outer tester should be *emulated* by making a few queries to the portions of the alleged codeword corresponding to the outer queries. This requires the inner code to be self-correctable so that the emulation can be carried out with few queries. Hence, for concatenated code to be locally testable, the outer code is required to be locally testable and the inner code ought to be locally correctable.

For our particular concatenated code, the outer codeword test is emulated as follows. In each query, the outer test only needs the value of a univariate polynomial q of degree d at some point $t \in \mathbb{F}$. Suppose that the parameters are chosen such that $d = h^{d'} - 1$. Note that the coefficient representation of the polynomial q is encoded by the inner code. Now, suppose the value $q(t)$ can be uniquely written as

$$q(t) = \sum_{i_1,\ldots,i_{d'} \in [h]} q_{i_1,\ldots,i_{d'}} t^{(i_1-1)+(i_2-1)h+\cdots+(i_{d'}-1)h^{d'-1}},$$

where $\langle q_{i_1,\ldots,i_{d'}} \rangle$ precisely defines the vector of $d+1$ coefficients of q. Hence, the value $q(t)$ is exactly the entry of the inner code associated with the $d'h$-long sequence

$$\langle t^0, \ldots, t^{h-1}, t^{0 \cdot h}, \ldots, t^{(h-1)\cdot h}, \quad \ldots\ldots \quad , t^{0 \cdot h^{d'-1}}, \ldots, t^{(h-1)\cdot h^{d'-1}} \rangle$$

in the inner-encoding of q. Hence, the value we are looking for can be retrieved by making one query into the oracle. However, this will deteriorate the soundness of the tester (as that particular point might be corrupted). Instead, we shall use a self-corrector for the inner code. As the inner code represents a degree d' polynomial, self correction can be performed by a polynomial interpolation that requires $d'+1$ queries. Thus, the total amount of queries needed for testing the concatenated code is $2(d'+1) = O(d')$.

Now we analyze the parameters of the concatenated code. Let $d = m^c$ and $d' = 2c$, for some constant $c > 1$. Then the concatenated code maps $\mathbb{F}^{kk'}$ to $\mathbb{F}^{nn'}$, where $n \approx k^{c/(c-1)}$ (as analyzed before) and $n' \approx \exp(d^{1/d'}) \approx \exp((\log k)^{c/d'}) = \exp(\sqrt{\log k}) = k^{o(1)}$. Hence, $nn' \approx (kk')^{c/(c-1)}$ and we obtain a locally testable code with a "constant nearly linear" codeword length. However, the alphabet $\mathbb{F}$ is of size $O(d) \approx \log^c k$, which is poly-logarithmically large with respect to the message length.

Finally, we reduce the alphabet size further to obtain a binary locally-testable code. This is done in [GS02] merely by a second concatenation, namely, concatenation of $\mathcal{C}'$ with the Hadamard code. Assume that the elements of $\mathbb{F}$ can be represented by k''-long binary strings. Applying the Hadamard code, each element of the field is encoded into a binary sequence of length $n'' = 2^{k''}$.

To adapt the codeword test with the final code, observe that the tester of $\mathcal{C}'$ merely verifies a constant number of linear constraints in the form $\Sigma_i \alpha_i a_i = 0$, where α_i's are constant coefficients in $\mathbb{F}$ and a_i's are the oracle answers. Hence we just need to emulate such linear tests. This is carried out by picking a k''-bit sequence r uniformly at random and checking whether $\langle r, \sum_i \alpha_i a_i \rangle \equiv 0 \mod 2$ holds (The notion $\langle \cdot, \cdot \rangle$ indicates the inner product modulo 2). Now, the product $\langle r, \sum_i \alpha_i a_i \rangle$ can be written as $\sum_i \langle r, \alpha_i a_i \rangle$, where each product $\langle r, \alpha_i a_i \rangle$ (modulo 2) is a linear combination (determined by r and α_i) of the bits of a_i and its value can be obtained by making a single query to the Hadamard encoding of a_i. Thus, the total number of the queries needed to emulate the outer-code tester is a constant, showing that the final binary code is also locally testable.

The final binary code maps $\mathbb{F}_2^{kk'k''}$ to $\mathbb{F}_2^{nn'n''}$, and we have already seen that $nn' \approx (kk')^{c/(c-1)}$. Now putting $n'' = 2^{k''} = |\mathbb{F}| = \mathrm{poly}(\log k)$, we get that $nn'n'' \approx (kk'k'')^{c/(c-1)}$. Thus, we have obtained a binary locally testable code whose block length is a "constant nearly linear" function of its message length. It is easy to verify that the set of infinite possibilities of the message length of the final code is dense enough. In particular, the successor of each possible message length K is smaller than $(\log K)^{c-1} \cdot K = K^{1+o(1)}$.

4.2.3 Tensor product codes

In this section we review a construction of locally testable codes based the tensor product of codes, proposed by Ben-Sasson and Sudan [BS04]. The construction yield a code with inverse-polynomial rate that is testable with a polylogarithmic amount of queries. However, the construction is generic in the sense that it produces locally testable codes from *any* family of codes satisfying certain (easily achievable) properties. Moreover, the construction is fully combinatorial and does not depend on algebraic objects such as low-degree polynomials.

Recall that a matrix M is a codeword of the tensor product $\mathcal{C} = \mathcal{C}_1 \otimes \mathcal{C}_2$ if and only if every row (resp. column) of M is a codeword of $\mathcal{C}_1$ (resp. $\mathcal{C}_2$). This gives the following natural codeword test for the product code $\mathcal{C}$: Pick a row or column of the matrix M uniformly at random, and verify that it corresponds to a codeword of $\mathcal{C}_1$ (if a row is chosen) or $\mathcal{C}_2$ (if a column is chosen).

The above codeword tester can be inductively generalized for multiple tensor product of the form $\mathcal{C}_1 \otimes \mathcal{C}_2 \otimes \cdots \otimes \mathcal{C}_m$. For simplicity, we restrict out attention to tensor powers of the same code, i.e., products of the form

$$\mathcal{C}^m \stackrel{\text{def}}{=} \underbrace{\mathcal{C} \otimes \cdots \otimes \mathcal{C}}_{m \text{ times}},$$

where $\mathcal{C}$ is an (n, k, d)-code.

For the product code $\mathcal{C}^m$, the product tester generalizes as follows (that we refer to as m-product tester): Pick $b \in_{\mathrm{R}} [m]$ and $i \in_{\mathrm{R}} [n]$ independently, and verify that the alleged codeword with b^{th} coordinate restricted to i is a codeword of $\mathcal{C}^{m-1}$.

The product code $\mathcal{C}^m$ can be thought of as a Tanner product code. Namely, consider the bipartite graph G_m^n with n^m left vertices labeled by the elements of $[n]^m$ and $m \cdot n$ right vertices labeled as (b, i), where $b \in [m]$ and $i \in [n]$. Vertex (b, i) is adjacent to all vertices $(i_1, \ldots, i_m)$ such that $i_b = i$. Then the code $\mathcal{C}^m$ is equivalent to the Tanner product code of the graph G_m^n with the code $\mathcal{C}^{m-1}$. Moreover, the m-product tester corresponds to picking a right vertex in the Tanner graph of $\mathcal{C}^m$ (i.e., G_m^n) and verifying if its neighbors define a codeword of $\mathcal{C}^{m-1}$.

It is known that the tensor product of two codes is not necessarily robust [Val05]. However, the following proposition shows that the m-product tester (for proper values of m) is robust:

Proposition 4.11. [BS04] *For a positive integer m and (n, k, d)-code $\mathcal{C}$ such that $(\frac{d-1}{n})^m \geq \frac{7}{8}$, the m-product tester is 2^{16}-robust for $\mathcal{C}^m$.* ∎

Along with Proposition 4.8, this suggests a natural local tester (with super-constant query complexity) for the code $\mathcal{C}^m$.

Using inductive compositions, it will in fact be possible to test any code of the form $\mathcal{C}^{2^t}$ by verifying that certain projections of the code are codewords of $\mathcal{C}^2$. This along with the above result leads to the following theorem:

Theorem 4.12. [BS04] *Let $\{\mathcal{C}_i\}$ be any infinite family of codes with $\mathcal{C}_i$ being an (n_i, k_i, d_i)-code of inverse polynomial rate. Let t_i be a sequence of integers such that $m_i = 2^{t_i}$ satisfies $\frac{d_i}{n_i} \geq 1 - \frac{1}{7m}$. Then the sequence of codes $\{\mathcal{C}_i^{m_i}\}$ has polynomial block* *length and constant relative distance that is locally testable with polylogarithmic number of queries.* ∎

It is possible to get code families required above using well known constructions such as Reed-Solomon codes or algebraic-geometric codes.

Chapter 5

Codeword Test by Proof Checking

The main motivation for the study of locally testable codes is construction of more efficient proof systems. Assertions and proofs are main ingredients of various concepts in computational complexity theory. In fact, many complexity notions rely on proof systems, and possibly the interaction between two abstract entities known as the *prover* and the *verifier*. We begin by a quick review of the history behind the proof systems.

From classical theory of NP-completeness we know that the complexity class NP is the set of assertions whose proofs can be written as binary strings with a polynomial length in the length of the binary representation of the assertion. Moreover, the validity of such proofs can be verified by a deterministic polynomial-time restricted Turing machine known as the *verifier*. For instance, the problem of finding a Hamiltonian path in a graph is known to be in NP. One can prove that a certain graph with n vertices has a Hamiltonian tour by simply writing down the description of a non-crossing cycle of length n (i.e., a Hamiltonian tour) in the graph. Such a description needs a polynomial length in the size of the graph, and one can easily verify that such a path is indeed a Hamiltonian tour.

In contrast, producing a proof that a certain graph has a Hamiltonian tour (i.e., trying to find such a tour) is suspected to be much more difficult than verifying the proof. This more or less resembles mathematical proofs. Proving a mathematical statement is normally a difficult task; However, once proved, a less skilled person can easily verify the correctness of the proof and get convinced of the truth of the assertion.

On the other hand, the complexity class P is the family of languages whose membership proofs can be produced by a deterministic Turing machine in polynomial time. Hence the famous question $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ is equivalent to saying that whether there exists a family of assertion whose proofs can be verified but not produced deterministically in polynomial time.

The complexity classes EXP and NEXP are "scaled up" versions of the classes P and NP, respectively. The class EXP corresponds to the family of languages decidable with deterministic Turing machines whose running time are at most exponentially large with respect to the binary description of the input. On the other hand, NEXP defines the family of assertions whose proofs can be deterministically verified (or equivalently, non-deterministically produced) in exponential time.

The work of Goldwasser, Micali and Rackoff [GMR85] and independently Babai [Bab85] introduce the notion of *interaction* between the prover and verifier. Namely, in an interactive proof system the *verifier* has the additional power of randomness and is able to interact with a computationally unbounded entity known as the *prover*. The verifier can be modeled as a probabilistic polynomial-time Turing machine that can interact with (i.e., query) the prover for a polynomially-bounded number of rounds. The complexity class IP is then defined as the class of decision problems whose membership property can be verified by interactive verifiers. Namely, the prover is able to convince the verifier of the validity of a true assertion in IP with probability $\frac{2}{3}$, but for the case of false assertions, it is unable to do so with probability more than $\frac{1}{3}$. The notion of *zero-knowledge* proof systems that has impacts in cryptography is also proposed in [GMR85]. In such proof systems the verifier is convinced of the validity of the assertion without obtaining any further information.

The fundamental result of Shamir [Sha92] reveals that the complexity class IP is precisely the family of languages whose membership can be decided by a deterministic Turing machine that consumes a polynomially bounded amount of memory, denoted as the complexity class PSPACE.

The notion of interactive proof systems was generalized by Ben-Or *et al.* [BGKW88] to allow interaction of the verifier with multiple provers that are unable to communicate with each other, hence the complexity class MIP. This would allow construction of zero-knowledge proof systems for all the languages in NP without any cryptographic intractability assumption that was needed to obtain a similar result (cf. [GMW91]) using interactive proof systems.

Followed by the work of Shamir [Sha92], Babai, Fortnow, and Lund [BFL91] proved a similar characterization for multiple interactive proof systems, namely, $\mathsf{MIP} = \mathsf{NEXP}$. Later, Fortnow, Rompel, and Sipser [FRS94] showed that any language in MIP allows exponentially large proofs that can be checked by a probabilistic polynomial verifier. Along with the fact that $\mathsf{MIP} = \mathsf{NEXP}$, this means that the exponentially long proofs in NEXP can be written in a form that can be probabilistically verified in polynomial time, i.e., a poly-logarithmically small time in the size of proof.

Knowing that the complexity class NEXP allows construction of efficiently checkable proofs, it is natural to ask if there is a "scaled down" version of this fact for NP. That is, whether the polynomially long proofs for the languages in NP can be written in a form that is checkable by a probabilistic verifier in poly-logarithmic time. The answer turns out to be positive [BFLS91], provided that both the assertion and proof are properly encoded. Such constructions are known as *holographic proofs*, as the verifier *transparently* verifies the consistency between the encoding of the assertion and the encoding of the proof without entirely reading them. The additional requirement to represent the assertion with an error correcting code seems rather inconvenient. However, such a requirement is necessary as the time restriction of the verifier allows it to read only a poly-logarithmically long portion of the assertion. Thus, this is the best we can hope for if we restrict the time complexity of the verifier.

An alternative approach to the above problem is to allow the verifier to operate in polynomial time so that it will be able to read the assertion in its entirety. On the other hand, to capture the concept of "efficient verification," we provide the verifier with the proof as an oracle string and restrict the number of *queries* it can make into the oracle. That is, we allow the verifier to read only a small portion of the given proof. Hence, instead of restricting the time complexity of the verifier, we restrict its *query complexity*. This is the idea behind the notion of *probabilistically checkable proofs (PCPs)* developed in [FGL$^+$96, AS98, ALM$^+$98]. In this chapter, we will restrict our attention to the notion of PCPs and their relationships with locally testable codes.

5.1 Probabilistically Checkable Proofs

5.1.1 Definitions and facts

As mentioned earlier, in a probabilistically checkable proof (PCP) system, the verifier has the ability to toss random coins, and is provided with the proof as an oracle strings. The queries of the verifier into the proof is counted as its query complexity. More precisely, we have the following:

Definition 5.1. For functions $r, q\colon \mathbb{Z}^+ \to \mathbb{Z}^+$ (for randomness and query complexity, respectively) a probabilistic oracle machine V, known as the verifier, is said to be $(r(n), q(n))$-restricted if for every input (assertion) of size n, its randomness complexity and query complexity are bounded to be at most $r(n)$ and $q(n)$, respectively. Moreover, V must operate in time $\mathrm{poly}(n, r(n), q(n))$. The decision of a verifier upon the input x and oracle π (that we call the *proof*) is denoted by the short form $V^\pi(x)$.

Using the above definition, we can define the family of complexity classes PCP as follows:

Definition 5.2. For functions $r, q\colon \mathbb{Z}^+ \to \mathbb{Z}^+$, a language L is in the complexity class $\mathsf{PCP}(r(n), q(n))$ if there exists an $(r(n), q(n))$-restricted verifier V checking its membership proofs. Namely, the verifier V must satisfy the following conditions:

Completeness. If $x \in L$, then there exists a membership proof π such that $\Pr[V^\pi(x) = 1] = 1$. That is, V accepts the proof π no matter what random bits are fed to it.

Soundness. If $x \notin L$, then for any membership proof π, $\Pr[V^\pi(x) = 1] \leq \frac{1}{2}$. The probability is taken over the choice of random bits used by the verifier.

In other words, in a PCP system, the verifier should be convinced of the validity of true assertions and not easily convinced upon false ones when restricted to use a limited amount of randomness and to have a narrow view of the proof.

Similar to the case of codeword verifiers, we can alter the above definition to allow two-sided errors and also consider the soundness error to be an arbitrary parameter. These are also meaningful; However, the above is the standard definition of PCP verifiers.

It turns out that various restrictions of the PCP system give compact characterization of many classical complexity classes. For instance, the construction of [BFL91] shows that the complexity class NEXP is precisely $\cup_{c>0}\mathsf{PCP}(n^c, n^c)$. However, the main motivation of defining PCPs is to give a characterization of NP. There is actually a chain of work towards such a characterization. In particular, Babai *et al.* [BFLS91] showed that all languages in NP are checkable with poly-logarithmic randomness and query complexity, i.e., $\mathsf{NP} \subseteq \cup_{c>2}\mathsf{PCP}(\log^c n, \log^c n)$. This is in fact the "scaled down" version of the result in [BFL91] regarding exponential time verifications with the difference that it does not give an exact characterization of NP. This result was tightened by Feige *et al.* [FGL+96] whose work implies that $\mathsf{NP} \subseteq \cup_{c>0}\mathsf{PCP}(c \log n \log\log n, c \log n \log\log n)$.

Finally, an exact characterization of NP was discovered by Arora and Safra [AS98] who showed the equality $\mathsf{NP} = \cup_{c>0}\mathsf{PCP}(c \log n, c\sqrt{\log n})$. However, they proposed the question that whether this characterization is optimally tight. This question was finally settled by Arota *et al.* citepcp2 who give an optimally tight characterization of NP (up to constant multipliers) in the celebrated PCP theorem. Namely, they prove that the following:

Theorem 5.3. [ALM+98] *(The PCP Theorem) The complexity class* NP *can be characterized as follows:* $\mathsf{NP} = \mathsf{PCP}(O(\log n), O(1))$. ∎

The major part of the PCP theorem is to show that $\mathsf{NP} \subseteq \mathsf{PCP}(O(\log n), O(1))$ (the reverse inclusion is almost trivial), which implies that any NP proof can be rewritten in a form (with a polynomial blow-up in its length) that is checkable by a probabilistic verifier that reads only a constant number of the bits in the proof. The original proof of the PCP theorem relies on algebraic constructions and in particular, low-degree polynomials. We will not go through the details of the proof here, but the interested reader is referred to Madhu Sudan and Sanjeev Arora's PhD theses [Sud92, Aro94] for almost self-contained write-ups of the proof. Their emphasis is on the analyses of the low-degree tests and the construction of the proof system, respectively.

In fact, the proof idea of the PCP theorem can be generalized to get the following:

Theorem 5.4. [Sud92] *(Extension of the PCP Theorem) For a language L that allows deterministic verification of the membership proofs in time $t(n)$, i.e., $L \in \mathsf{NTIME}(t(n))$, we have the following: $L \in \mathsf{PCP}(\lg(t(n)) + n, O(1))$.* ∎

Hence, for instance, we can obtain the following corollaries:

1. $\mathsf{NEXP} \stackrel{\text{def}}{=} \mathsf{NTIME}(2^{\text{poly}(n)}) = \mathsf{PCP}(\text{poly}(n), O(1))$.
2. $\mathsf{NE} \stackrel{\text{def}}{=} \mathsf{NTIME}(2^n) = \mathsf{PCP}(n, O(1))$.

5.1.2 Relationship with locally testable codes

The similarity between the definitions of locally testable codes and probabilistically checkable proofs brings up the suspicion that they are combinatorial and computational representations of the same entity, and one can transform a locally testable code to a PCP system with similar parameters and vice versa. Even though such an argument is not quite true, there is a close relationship between known constructions of "good" locally testable codes and those of probabilistically checkable proofs. If fact, as we will see later, the best known families of locally testable codes can only be obtained from PCP constructions.

Needless to say, codes are combinatorial objects and more intuitive than computational proofs. Hence, it seems easier to construct locally testable codes and manipulate such an *intermediate* construction to obtain a PCP system rather than constructing the PCP from scratch. In fact, locally testable codes are at least implicit as part of PCP constructions, and this is one of the major motivations to study such codes. One particular example is the constructions of Goldreich and Sudan [GS02]. First, they give a combinatorial and rather simple construction of locally testable codes with "constant nearly linear" block length, namely, the one we studied in section 4.2.2. Using this code, obtaining a PCP with similar proof length is straightforward. However, to improve this result, they modify a PCP construction to obtain a new PCP with "tight nearly linear" proof length and subsequently transform it into a construction of locally testable codes. More efficient locally testable codes of [BGH+04] are also only obtained from a PCP construction. On the other hand, the PCP of [BS05] is obtained by modifying a code construction.

Hence, it is not completely clear whether it is more convenient to start by constructing a locally testable code to obtain a PCP or vice versa. However, transforming a PCP construction to a locally testable code is not as trivial as it may seem to be, as we will just describe.

Recall that a PCP construction provides efficient means of *encoding* standard NP proofs into new (and longer) proofs that can be locally tested by the PCP verifier. In that sense, PCPs can be considered as *locally testable proofs*. In particular, a small change in a valid NP proof might invalidate it. The encoding scheme provided by the PCP guarantees that such a change will be "scattered" in the new proof in a way that its invalidity can be detected by the verifier with high probability. This resembles the properties of error correcting codes. As a particular example, consider a CNF formula over n variables that is tautology (i.e., all possible assignments satisfy it). Hence, every string of length n is a valid NP proof for it. Now consider taking a message string as a (valid) NP proof and its PCP transformation as the corresponding codeword. Does this simple construction give us a locally testable code?

The answer is "not necessarily." Note that the PCP verifier guarantees that codewords are accepted by the PCP verifier with probability 1 and oracles far from codewords are rejected with high probability. The problem is in what considered to be "codeword" by the verifier. The completeness condition implies that for every valid NP proof, there is a valid oracle that makes the verifier accept. However, such an oracle need not be unique. Namely, the verifier accepts valid proofs but not necessarily the ones obtained by applying the encoding scheme of the PCP system on valid NP proofs. In other words, the code that we just obtained is not necessarily *uniquely encodable*, and in fact many PCP oracles may be acceptable for one particular NP proof even though we are only interested in one of them.

Another problem arises when we are only interested in obtaining linear locally testable codes from PCPs. In particular, the composition step[1] in PCP constructions may deteriorate the linearity. Other (less severe) complications will also arise in translation of standard PCPs to locally testable codes. These problems are addressed in [GS02], and they manage to obtain linear locally testable binary codes from their PCP construction by introducing the new notion of *inner verifiers for linear systems* that is well tuned to guarantee unique encoding and linearity. As we will see shortly (section 5.3.4), a slight modification in the definition of PCP verifiers (namely, the notion of PCPs of proximity) allows a smooth and rather "mechanical" conversion of PCPs to locally testable codes with the same parameters.

5.2 Hardness of approximation and efficient PCPs

There is an interesting relationship between probabilistically checkable proofs and inapproximability of NP-optimization problems, that is, showing that certain hard

[1] Composition of verifiers allow combining an "outer" verifier with a proper "inner" verifier to obtain a new verifier with better parameters. This is roughly analogous to code concatenation. See section 5.3.2 for the whole idea and [AS98, ALM$^+$98] for details.

optimization problems are even hard to solve approximately. In this section, we assume familiarity of the reader with the basic concepts in optimization problems and approximation algorithms (refer to, say, [Vaz01, Pap94] for such details). However, we recall four categories of NP-hard optimization problems (from the easiest to the most difficult to approximate):

1. Problems that allow a *fully polynomial-time approximation scheme (FPTAS)*, e.g., Knapsack. For such problems, there exist approximation algorithms that take an approximation ratio ϵ as part of their input and output an approximate solution close to the optimal within the factor ϵ. Moreover, such approximation algorithms run in polynomial time with respect to the input length and the factor ϵ.

2. Problems that allow a *polynomial-time approximation scheme (PTAS)*, e.g., Bin Packing and Euclidean TSP. This category is similar to the previous one with the difference that the approximation algorithm runs in polynomial time with respect to the input length but not necessarily the approximation factor ϵ.

3. Approximable problems, that allow polynomial time approximation for some constant approximation factor. Examples: MAX-2SAT, MAX-3SAT, MAX-CUT, and Vertex Cover.

4. Inapproximable problems, that remain NP-hard for any constant approximation factor. One particular example in this category is the problem of computing the clique number of a graph (i.e., maximum number n such that the input graph has a complete sub-graph with n vertices), for which the best possible polynomial time approximation algorithm is almost as weak as picking a trivial feasible solution (unless $\mathsf{P} = \mathsf{NP}$) [Hås96, FGL$^+$96, FGL$^+$91].

Now we restrict our attention on a class of approximable optimization problems defined by Papadimitriou and Yannakakis [PY91] which is known as (for "maximum strict NP"). Their original definition is based on the possibility to express the objective function in a certain logical form. Here we consider an alternative and more intuitive definition of MAX-SNPbased on the constraint satisfaction problem, which is a universal problem in this class:

Definition 5.5. *(Constraint)* For some finite alphabet Σ, a *constraint* of *arity* c is a function $C\colon \Sigma^c \to \{0, 1\}$. The constraint is said to be *satisfied* by an assignment $a \in \Sigma^c$ if $C(a) = 1$.

In this section, we are only concerned with the constraints over the binary alphabet, i.e., we restrict $\Sigma = \{0, 1\}$.

Definition 5.6. *(CSP: Constraint Satisfaction Problem)* An instance of the CSP problem consists of a set of constraints on a set of variables, where the objective function is defined as the maximum number of satisfiable constraints over all possible assignments of variables. When each constraint depends on at most c variables, for some constant c (i.e., the arity of all constraints is c), the problem is called c-CSP.

Definition 5.7. (MAX-SNP) An optimization problem is said to be in the complexity class MAX-SNP if there exists a constant c such that the problem can be expressed as a c-CSP (over binary variables).

Papadimitriou and Yannakakis [PY91] have also defined a particular approximation ratio preserving reduction, namely, L-reduction (or *linear reduction*) that allows identifying complete problems for MAX-SNP.

Definition 5.8. *(L-reduction)* An optimization problem Π is said to L-reduce to another problem Π' if there exists polynomial time computable functions f, g and nonnegative constants α, β such that

1. If I is an instance of Π, $f(I)$ is an instance of Π' such that $\mathrm{OPT}(I) \leq \alpha \mathrm{OPT}(f(I))$.
2. If s' is feasible solution for $f(I)$, then $g(s')$ is a feasible solution for I such that $|\mathrm{cost}(s') - \mathrm{OPT}(f(I))| \leq \beta\, |\mathrm{cost}(s) - \mathrm{OPT}(I)|$.

Thus, if Π is L-reducible to Π', we can approximate the instances of Π by mapping them to instances of Π' and using an approximation algorithm for Π' to approximate the transformed instance. Subsequently, we can map the obtained approximate solution back to a solution for our original problem. The definition of L-reduction guarantees that if Π' is approximable with a factor ϵ, then using the above reduction we can obtain an approximation algorithm for Π with a factor $\alpha\beta\epsilon$.

Note that any approximability result for a complete language in MAX-SNP translates to a similar result for all languages in MAX-SNP. In particular, existence of an approximation algorithm for a complete (or hard) problem translates to the existence of approximation algorithms for all languages in the family, and, a hardness result for approximating a complete problem translates to hardness results for the whole class.

Many natural optimization problems are known to be in MAX-SNP. In particular, we can mention c-CSP, MAX-3SAT, MAX-2SAT, Independent Set and Vertex Cover in bounded degree graphs, and MAX-CUT. Moreover, the following problems are known to be hard for MAX-SNP: Shortest Superstring, MAX-CLIQUE, Longest Path, Weighted TSP and Minimum Steiner Tree with only weights 1 and 2 allowed (cf. [Sud92]).

Proposition 5.9. *Any problem in* MAX-SNP *is approximable within some constant factor.*

Proof Sketch: It is sufficient to consider the universal problem c-CSP. Let m be the number of constraints that are individually satisfiable. (A constraint is said to be individually satisfiable if there exists an assignment of variables that satisfies it, regardless of the other constraints.) Let OPT denote the optimum solution, i.e., the maximum number of satisfiable constraints. Clearly, OPT $\leq m$. On the other hand, as each constraint depends on only c variables and each of the m individually satisfiable constraints is satisfiable by at least one of the 2^c possible settings of its variables, we can always satisfy at least $\frac{m}{2^c}$ constraints. Thus, $\frac{m}{2^c} \leq$ OPT $\leq m$ and the algorithm that simply outputs m is a 2^c-approximation algorithm. ∎

The following proposition is analogous to the one above but turns out to be much more elaborate to prove:

Proposition 5.10. *Every* MAX-SNP*-hard problem is* NP*-hard to appriximate within some constant factor.*

Proof Sketch: Again, proving the statement for a complete problem is sufficient. Here we discuss the proof given in [ALM$^+$98]. Namely, we give a reduction from any language in NP to an approximate version of MAX-3SAT.

Let L be a language in NP, and V be a $(c \log n, q)$-restricted verifier for it, as guaranteed by the PCP theorem (c and q are some absolute constants). Let x be a string of length n that we wish to decide about its membership in L. Consider poly(n) variables corresponding to individual bits of a potential probabilistically checkable proof π for x. For every choice R of the $c \log n$ random bits used by the verifier, describe the decision of V as a boolean formula $\varphi_{x,R}$ that depends on at most q variables in π. Note that such a formula can be rewritten in a CNF form with at most 2^q clauses, each of length at most q. This can further be transformed to a 3-CNF form by defining up to q auxiliary variables. Now consider the conjunction of all these formulae as a 3-CNF formula $\varphi_x \stackrel{\text{def}}{=} \bigwedge_R \varphi_{x,R}$ with at most $q 2^q 2^{c \log n}$ clauses.

It is clear that if $x \in L$, there is a satisfying assignment for φ_x. On the other hand, if $x \notin L$, the soundness of the verifier implies that it accepts x on at most half choices of the random bits. Thus, the total number of satisfied clauses in φ_x can be at most $(1-\epsilon)m$, for $\epsilon \stackrel{\text{def}}{=} \frac{1}{2q2^q}$, which is an absolute constant.

Thus, an algorithm that approximates MAX-3SAT within a $(1-\epsilon)$ factor will suffice to decided membership of x in L. ∎

The above proposition implies that unless P $=$ NP, there is no hope to find PTAS for problems hard for MAX-SNP, even though all problems in MAX-SNP can be approximated within *some* constant factor. This motivates finding the *approximability threshold* of MAX-SNP-complete problems, i.e., the approximation factor that defines the *boundary* between approximability and inapproximability of the problem. The proof of the Proposition 5.10 suggests that constructing PCPs with smaller query complexity will help getting tight results regarding hardness of approximating MAX-SNP-hard problems.

Therefore, reducing the query complexity of PCPs is a remarkable goal. In fact, Håstad [Hås01] was able to obtain several optimal results regarding hardness of approximation by constructing PCPs with the optimal query complexity three. However, the length of his PCPs were quite high ($\approx n^{10^6}$) but still polynomial. On the other hand, the length of the PCP in the original proof of [ALM+98] is roughly cubic; However, its query complexity is high ($\approx 10^6$ to obtain the soundness error $\frac{1}{2}$).

The trade-off between the proof length and the query complexity of PCPs motivates the effort towards obtaining PCPs that achieve reasonably low values for both parameters. As several constructions of PCPs are implied by construction of locally testable codes, this also motivates the study of locally testable codes with high rate and low query complexity.

Regarding short PCPs, Polishchuk and Spielman [PS94] proposed a construction of PCPs with length $O(n^{1+\epsilon})$ and query complexity $O(\frac{1}{\epsilon})$, for every constant $\epsilon > 0$. Later, Harsha and Sudan [HS00] constructed PCPs with nearly quadratic length $O(n^{2+\epsilon})$ and query complexity 17. Based on this work, Goldreich and Sudan [GS02] constructed PCPs with nearly linear length $n \cdot \exp(\log^c n)$ for every $c > 0.5$ and 19 queries, and also modified their construction to obtain binary locally testable codes with similar parameters. Later, Ben-Sasson *et al.* constructed even shorter PCPs. Namely, they obtain a *spectrum* of PCPs with length $n \cdot 2^{\log^\epsilon n}$ checkable with $O(\frac{1}{\epsilon})$ queries (for every $\epsilon > 0$) and also PCPs with a quasi-polylogarithmic blowup in proof length and super-constant query complexity of $o(\log \log n)$.

Besides efficiency, another importance of constructing short PCPs is the impact of the length of proof on the tightness of inapproximability results with respect to the *running time*, as opposed to the approximation ratio which is related to the query complexity (cf. [Sze99]). For instance, if we assume a time complexity $2^{\Omega(n)}$ for exact solution of SAT, the original PCP theorem implies a lower bound of 2^{n^ϵ} on the time complexity of approximating MAX-SAT, for some $\epsilon > 0$. The work of Polishchuk and Spielman [PS94] allows making ϵ arbitrarily close to 1. However, "tight nearly linear" constructions of [GS02, BGH+04] would imply the time complexity $2^{n^{1-o(1)}}$ to approximate MAX-SAT. This shows another motivation for construction of shorter PCPs and locally testable codes with short block lengths.

5.3 PCPs of proximity

In the standard definition of PCP systems, the verifier is supposed to test the truth of an assertion given as input with the aid of a proof given as an oracle. An alternative definition was given in [BGH+04] under the notion of *PCPs of Proximity (PCPPs)*.

In a PCPP, the assertion itself is also given to the verifier as an oracle. Hence, the verifier has access to a pair of oracles, namely, the assertion and the proof, and the number of queries the it makes into either of the two oracles are counted towards its query complexity. In this sense, PCPPs are similar to holographic proofs introduced in [BFLS91], with the difference that in a PCPP the assertion is not encoded by a proper error-correcting code and is given in the natural "raw" encoding. Moreover, the running time of a PCPP verifier can be polynomially large, as opposed to polylogarithmic in the case of holographic proofs, and the query complexity is considered as a measure of efficiency, as in a standard PCP.

With such a description, it seems impossible to construct PCPPs with low query complexity, as testing the validity of an assertion at least requires reading the entire assertion, because changing merely one bit of an assertion might change its validity. However, PCPPs are not required to do so. A PCP of proximity is only required to distinguish valid assertions from the ones that are *far enough* (in Hamming distance) from being true, hence the term "proximity."

5.3.1 Formal definition

The following is the formal definition of a PCPP system, as appeared in [BGH+04]:

Definition 5.11. For functions $s, \delta\colon \mathbb{Z}^+ \to [0,1]$, a verifier V is a *probabilistically checkable proof of priximity (PCPP)* system for a pair language (i.e., a language consisting of pairs of strings) L with *proximity parameter* δ and *soundness error* s if the following two conditions hold for every pair of strings (x, y) with n being the length of the input string x and y given to the verifier as an oracle:

Completeness. If $(x, y) \in L$, then there exists a proof π such that $V(x)$ accepts oracle $y \circ \pi$ (i.e., the string concatenation of y and π) with probability one.

Soundness. If y is $\delta(n)$-far from $L(x)$ (i.e., the set $\{y' \mid (x, y') \in L\}$), then for every π, the verifier $V(x)$ accepts oracle $y \circ \pi$ with probability strictly less than $s(n)$, where the probability is taken over the choice of random bits used by the verifier.

When s and δ are omitted, they are assumed to be absolute constants in $(0, 1)$.

We remark that it is also reasonable to give a more flexible definition of PCPPs in which the verifier gets the proximity parameter as part of its input and behaves accordingly.

In a PCPP, we require pair languages, even though an alternative definition for arbitrary languages would also make sense. Namely, the first component of a pair language is explicitly given to the verifier as its input, and the complexity of the verifier is measured as function of its input length. On the other hand, the second component of the assertion is considered as a part of the oracle given to the verifier. Along with the proof oracle, the number of probes made into this oracle determines the query complexity of the verifier.

A typical example of the pair languages for which PCPPs are constructed is the P-complete language *circuit value*, denoted by CktVal. The pairs in this language are in the form (C, w), where C is the binary description of a boolean circuit (consisted of various boolean gates, input nodes, and a particular output node) and w is a satisfying assignment for the inputs of the circuit C, i.e., $C(w) = 1$.

Note that the projection of the pair language CktVal on the first component is the language $\{C \mid \exists w$ such that $(C, w) \in$ CktVal$\}$, which is the set of satisfiable boolean circuits. This language, denoted by CktSat is a classical NP-complete language, and is a typical example for construction of standard PCPs. In fact, such a correspondence is not coincidental, as clarified in the following proposition:

Proposition 5.12. [BGH+04] *If* CktVal *has a PCPP, then* CktSat *has a PCP with identical parameters.*

Proof: Let V be a PCPP verifier for CktVal with proximity and soundness parameters δ and s, respectively. Using V we can construct a PCP for CktSat as follows: Let C be a boolean circuit. If $C \in$ CktVal, there is an assignment w and a proof π that makes V accept on (C, w). Thus the proof $w \circ \pi$ can be considered as the PCP proof. On the other hand, if $C \notin$ CktVal, for every assignment and every proof π, the probability that V accepts $(\mathcal{C}, w)$ with the proof π is less than $s(|C|)$, as otherwise the soundness condition of V implies that the assignment w is $\delta(|C|)$-close to a satisfying assignment, which is a contradiction (note that no satisfying assignment exists). Hence, a PCPP verifier for CktVal is also a PCP verifier for CktSat. ∎

The above proof also shows that any PCPP verifier for a pair language is also a PCP verifier for the projection of the language to the first component. In that sense, PCPPs are stronger than PCPs, as constructing a PCPP for an *easy* problem (e.g., CktVal) also implies construction of a standard PCP for a *hard* one (e.g., CktSat).

5.3.2 Robustness and composition

The robustness property of locally testable codes that we discussed in section 4.1 is based on the robustness of proof verifiers. A robust PCP verifier has a strong soundness guarantee, namely, not only false assertions should be rejected with high probability, but they should also be far (in Hamming distance) from being accepted. More formally, we have the following:

Definition 5.13. Think of a non-adaptive PCP verifier V as an algorithm that generates a circuit C and a set of query locations I upon receiving its input and random bits and accepts iff $C(\pi|_I) = 1$, where π is the proof oracle. For functions $s, \rho\colon \mathbb{Z}^+ \to [0,1]$, V is said to have *robust-soundness error* s with *robustness* ρ if the following holds for every $x \in L$ of length n: For every oracle π, the bits of the proof read by the verifier V (i.e., $\pi|_I$) are ρ-close to being accepted with probability less than s, i.e.,

$$\forall \pi\colon \Pr[\exists a \text{ such that } C(a) = 1 \text{ and } \boldsymbol{\delta}(a, \pi|_I) \leq \rho(n)] < s(n).$$

If s and ρ are unspecified, they are assumed to be absolute constants in $(0,1)$.

Note that the above definition is meaningful for both PCPs and PCPPs, but only for non-adaptive oracle verifiers. Moreover, every standard PCP has a trivial robustness $\frac{1}{q}$, where q is its query complexity. However, the "building block" PCPPs of [BGH$^+$04] achieve constant robustness while their query complexity is polynomially large. It is also easy to see that the proximity parameter of a PCPP upper bounds its robustness.

Similar to the idea of code concatenation and also composition of standard PCP verifiers proposed in [AS98], composition of PCPPs is a main tool in recursive construction of proof checking systems and allows us to obtain better verifiers from a pair of existing ones.

The composition theorem of [AS98] roughly implies that an *outer verifier* can be composed with a (properly chosen) *inner verifier* in a way that the resulting verifier *inherits* the query complexity and soundness error of the inner verifier using about the same amount of randomness as the outer verifier (that is typically far less than the randomness of the inner verifier). However, the composition is fairly complicated and introduces a slight increase in the proof length which is not tolerable in construction of short proofs.

The remarkable importance of the robustness property is that it allows a simple and natural composition of PCPs in the following way:

Lemma 5.14. [BGH$^+$04] *Let V_{out} (the outer verifier) be a robust PCP verifier with randomness complexity r_{out}, robustness ρ_{out} and robust soundness error $1 - \epsilon_{\mathrm{out}}$. Assume that V_{out} has* decision complexity d_{out}, *i.e., upon receiving an input of length n and a sequence of random bits, it generates a decision circuit C of size $d_{\mathrm{out}}(n)$ and accepts iff C accepts on the outcome of queries specified by the verifier. Let V_{in} (the inner verifier) be a PCP of proximity for the* CktVal *problem with randomness complexity r_{in}, query complexity q_{in}, decision complexity d_{in}, proximity parameter δ_{in}, and soundness error $1 - \epsilon_{\mathrm{in}}$ such that for every n, $\delta_{\mathrm{in}}(d_{\mathrm{out}}(n)) \leq \rho_{\mathrm{out}}(n)$. Then, there exists a* composed *PCP verifier V_{comp} for the language L that uses V_{out} and V_{in} as* black box *and has the following parameters:*

- *Randomness complexity $r_{\mathrm{out}} + r_{\mathrm{in}}(d_{\mathrm{out}}(n))$,*

- *Query complexity* $q_{\text{in}}(d_{\text{out}}(n))$,
- *Decision complexity* $d_{\text{in}}(d_{\text{out}}(n))$,
- *Soundness error* $1 - \epsilon_{\text{out}}(n) \cdot \epsilon_{\text{in}}(d_{\text{out}}(n))$.

The composed verifier V_{comp} *only makes one call to each of* V_{out} *and* V_{in}*. Moreover, if* V_{out} *is a robust PCPP, then* V_{comp} *would also be a PCPP with the same proximity parameter. Similarly, if* V_{in} *is a robust PCPP with robustness parameter* ρ_{in}*, then* V_{comp} *would also be robust with robustness parameter* $\rho_{\text{in}}(d_{\text{out}}(n))$. ∎

We comment that the above lemma requires the outer verifier to be robust and the inner verifier to be a PCP of proximity with a matching proximity to make the composition work. In fact, the standard composition lemma of [AS98] somehow *simulated* such a requirement by using a proper encoding/decoding scheme.

5.3.3 Nearly linear-sized PCPPs

Recall that in an oracle machine with randomness complexity $r(n)$ and query complexity $q(n)$, the length of the oracle need not be larger than $2^{r(n)}q(n)$. Hence to get a linear-length proof, it is sufficient to construct a verifier with randomness complexity $\lg n + o(\log n)$, provided that the query complexity is constant.

In fact, the construction of [BGH+04] obtains a "tight nearly linear" proof length by considerably saving on randomness complexity, compared to the original construction of [ALM+98]. However, their construction is also algebraic and share many common or similar techniques with the original construction. In particular, their randomness reduction is obtained due to the following steps (cf. [Har04]):

Reduction to algebraic problem: As the first step of the original PCP construction, the computational CktSat problem is reduced to an algebraic problem that allows application of algebraic methods in the following steps (in particular, low-degree tests). This induces a large blow-up in the proof size. The construction of nearly-linear sized PCPs (motivated by [PS94]), reduce the CktSat problem to a coloring problem on certain graphs to obtain a short sized algebraic description.

Randomness-efficient compositions: The original composition introduces extra randomness, and thus, an undesirable blow-up in the proof size. On the other hand, the new construction is based on PCPPs and hence they can benefit from the simple composition procedure we discussed earlier that imposes *no* extra randomness.

Avoiding parallelization: In the original construction, there is a so-called *parallelization* step for reducing the query complexity of the verifier. Roughly, this technique allows reducing a poly-logarithmic query complexity to constant by aggregating all the queries into a constant amount, at the expense of a minor increase in the query answer length[2]. However, parallelization is possible by introducing many auxiliary proofs that causes a large blow-up in randomness complexity and proof size. On the other hand, the new construction completely avoids the parallelization step.

Combinatorial bundling: It is often required in algebraic PCP constructions to *bundle* a super-constant number of low-degree m-variate polynomials in a way that they can all be verified to be low-degree at once. A classical approach, known as *polynomial concatenation*, is to introduce an extra *index* variable i over the underlying field $\mathbb{F}$ and aggregating all the polynomials into one polynomial on $m + 1$ variables. That is, the new polynomial f is defined in a way that $f(x_1, \ldots, x_m, i) \equiv p_i(x_1, \ldots, x_m)$, for $i \in [k]$, where k is the number of initial polynomials. This way, the initial k low-degree tests for p_i's reduces to one low-degree test on the polynomial f. However, the new index variable makes the polynomial f defined over the whole $\mathbb{F}^{m+1}$, and this introduces a blow-up factor of roughly $\frac{|\mathbb{F}|}{k}$. In contrast, the new construction of [BGH+04] uses a combinatorial bundling method that does not increase the proof size at all.

The polynomial-line test: The usage of the polynomial-line test (Algorithm 3.31) in the standard constructions of PCPs (or its similar variants in other constructions, such as using axis-parallel lines or planes instead of lines) has a major impact on the increase on randomness complexity. As we saw in section , a derandomization of the original polynomial-line test will help us considerably save on randomness. However, even such an improvement incurs a blow-up factor of at least $2^{\sqrt{\log n}}$ in the proof size [Har04]. Such a blow-up clearly appears in the PCP construction of [GS02]. However, using PCPs of proximity, it is possible to modify the polynomial-line test in a way to consume less amount of randomness.

Using the above considerations, the construction of [BGH+04] incurs a considerably less amount of randomness compared to the earlier constructions and yields "tight nearly linear" PCPs of even shorter length. Namely, they obtain the following general trade-off:

Proposition 5.15. [BGH+04] *For every $n, t \in \mathbb{N}$ such that $3 \leq t \leq \frac{2 \log \log n}{\log \log \log n}$, there exists a PCP of proximity for* CktVal *(for circuits of size n) with the following*

[2]Note that in the intermediate steps of the construction the proofs are not in binary form and the response obtained from a query can be much longer than one bit

parameters:

- *Randomness complexity:* $\lg n + A_t(n)$, *where*

$$A_t(n) \stackrel{\text{def}}{=} O(t + (\log n)^{\frac{1}{t}}) \log\log n + O((\log n)^{\frac{2}{t}}),$$

- *Constant query complexity,*
- *Perfect completeness,*
- *Soundness error* $1 - \Omega(\frac{1}{t})$ *with respect to proximity parameter* $\Omega(\frac{1}{t})$.

Alternatively, we can have query complexity $O(t)$ *and soundness error* $\frac{1}{2}$ *with the remaining parameters as above.* ▮

Note than a standard PCP with similar parameters immediately follows from the above result using Proposition 5.12. Hence, we get the following corollaries by "instantiating" the above proposition at two extreme cases:

Corollary 5.16. *For every constant* $\epsilon \in (0, 1)$, *there exists a PCPP for* CktVal *(or equivalently, a standard PCP for* CktSat*) with constant query complexity* $O(\frac{1}{\epsilon})$ *and randomness complexity* $\lg n + O(\log^{\epsilon} n)$ *(and hence, a proof length of* $n \cdot \exp(\log^{\epsilon} n)$, *which is in the "tight nearly linear" form).*

Proof: Let $t(n) := \frac{2}{\epsilon}$ in Proposition 5.15. Then we get

$$A_t(n) = O\left(\tfrac{2}{\epsilon} + (\log n)^{\frac{\epsilon}{2}}\right) \log\log n + O(\log^{\epsilon} n) = O(\log^{\epsilon} n).$$

Using the query complexity $O(t) = O(\frac{1}{\epsilon})$ and soundness error $\frac{1}{2}$, the result follows. ▮

Corollary 5.17. *There exists a PCPP for* CktVal *(or equivalently, a standard PCP for* CktSat*) with query complexity* $o(\log\log n)$ *and randomness complexity* $\lg n + O((\log\log n)^2 / \log\log\log n)$ *(and hence, a proof length of* $n \cdot \exp(o(\log\log)^2)$ *that has a quasi poly-logarithmic blowup).*

Proof: This is similarly obtained by plugging $t(n) := \frac{2\log\log n}{\log\log\log n}$ into the statement of Proposition 5.15. ▮

5.3.4 PCPPs and locally testable codes

A remarkable property of PCPPs is that they translate into locally testable codes rather effortlessly. This comes from the fact that their definition is naturally closer to that of locally testable codes, compared to standard PCPs. The following proposition formalizes such a transformation:

Proposition 5.18. [BGH+04] *Let $d\colon \mathbb{Z}^+ \to \mathbb{Z}^+$ and q_{rep} be free parameters, and $\mathcal{C}_0$ be a $(m(k), k, m(k)\rho_0)_2$-code. Let V be a PCPP verifier for membership in $\mathcal{C}_0$. Suppose V expects proofs of length $\ell(k) > m(k)$ and has soundness error $\frac{1}{4}$ for proximity parameter δ_{pcpp} and query complexity q_{pcpp}. Then, there exists an $(n(k), k, n(k)\rho)_2$-code $\mathcal{C}$ that is (q, δ)-locally testable, where the parameters are as follows:*

- *The block length $n(k)$ is $d \cdot \ell(k)$,*
- *The relative distance ρ is at least $\rho_0 - \frac{1}{d}$,*
- *The proximity parameter δ equals $\delta_{\text{pcpp}} + \frac{4}{q_{\text{rep}}} + \frac{1}{d}$,*
- *The query complexity q is $q_{\text{pcpp}} + 2q_{\text{rep}}$.*

Proof Idea: We construct the code $\mathcal{C}$ as follows: Let x be a message of length k, and $m := m(k)$, $\ell := \ell(k)$. Then we define the codeword corresponding to x as $\mathcal{C}(x) \stackrel{\text{def}}{=} (\mathcal{C}_0(x)^t, \pi(x))$, where $\pi(x)$ is the PCPP oracle of length ℓ corresponding to the claim that the input-oracle equals $\mathcal{C}_0(x)$. That is, every codeword of the new code is the repetition of the encoding of x by $\mathcal{C}_0$ concatenated with the proof oracle corresponding to the pair $(x, \mathcal{C}_0(x))$. Note that the repetition of the $\mathcal{C}_0(x)$ is necessary, as the proof oracle is typically much longer than the codeword $\mathcal{C}_0(x)$ and as different proofs of the same statement are not guaranteed to be far, the distance of the new code may be low without the repetition. Using a properly large value t we make the portion of the new codeword corresponding to the proof oracle considerably small. In particular, let $t := (d(k) - 1) \cdot \ell / |\mathcal{C}_0(x)|$ so that the block length of the new code becomes $d \cdot \ell$. Note that the PCPP π constitutes only an $\frac{\ell}{n} = \frac{1}{d}$ fraction of the codeword length, hence the minimum distance of $\mathcal{C}$ cannot be less than $\rho_0 - \frac{1}{d}$. Now we describe the action of a codeword tester T for the new code $\mathcal{C}$. Suppose T is given access to an alleged codeword $w = (w_1, \ldots, w_t, \pi)$. The tester picks $i \in_{\mathrm{R}} [t]$, and emulates the action of V on the input oracle w_i and proof oracle π. Moreover, it randomly verifies the validity of the repetitions. Namely, for q_{rep} times, it picks $i_1, i_2 \in_{\mathrm{R}} [t]$ and $j \in [m]$ and checks whether $(w_{i_1})_j = (w_{i_2})_j$. Hence the query complexity of the tester is precisely $q_{\text{pcpp}} + 2q_{\text{rep}}$. Clearly, the tester has the completeness property. Analyzing the soundness condition is also straightforward and is omitted here. ∎

Using an asymptotically good family of codes in the above proposition, it follows that for every PCPP construction there exists a family of locally testable codes with similar parameters (codeword length and query complexity). Combined with Corollary 5.16 we get the following:

Corollary 5.19. *For every constant $\epsilon \in (0,1)$, there exists a family of (q,δ)-locally testable binary codes $\{\mathcal{C}_k\}$ with block length $n(k) = k \cdot \exp(\log^{\epsilon} k)$ and query complexity $q(\delta) = O(\frac{1}{\delta})$.* ∎

5.4 Combinatorial constructions

Most constructions of PCP systems are based on algebraic techniques and in particular, low degree polynomials. This makes such constructions rather difficult to analyze. In [DR04], the first remarkable attempt has been made to *combinatorialize* such constructions, with the goal of obtaining simpler, more intuitive and possibly more efficient results. Their construction is based on the notion of *assignment testers*, which is essentially equivalent to PCP of proximity for the circuit-value problem. Despite of the equivalence, assignment testers and PCPPs are studied independently. Precisely, we have the following definition:

Definition 5.20. An *assignment tester* with parameters $(R, s, q, \epsilon, \delta)$ (respectively for randomness complexity, output size, query complexity, soundness error, and proximity) is a (deterministic) algorithm whose input is a boolean circuit φ of size n over boolean variables X. The algorithm outputs a system of $R(n)$ boolean circuits $\Psi = \{\psi_1, \ldots, \psi_R\}$ each of size at most $s(n)$ over X (the inputs of φ) and auxiliary variables Y (that represent the proof) such that,

- The running time of the algorithm is polynomial in n and $R(n)$,
- Each ψ_i depends on $q(n)$ variables from $X \cup Y$. The variables in Y take values in an alphabet Σ, and are accessible to ψ_i as a tuple of $w(n) = \lceil \lg |\Sigma| \rceil$ bits (the width parameter $w(n)$ is called the *answer length*),
- For every assignment $a\colon X \to \{0,1\}$, we have the following:

  Completeness. If the assignment a satisfies the input circuit φ, then there exists a proof (i.e., an assignment for the auxiliary variables in Y) $b\colon Y \to \Sigma$ such that $a \cup b$ satisfies all of the output circuits $\psi_1, \ldots, \psi_R$,

  Soundness. If the assignment a is δ-far from every satisfying assignment for the input circuit φ, then for every assignment b of the proof variables, at least $1-\epsilon$ fraction of the output circuits reject $a \cup b$.

Note the correspondence between the above definition and PCPPs. In fact, an assignment tester *emulates* a (non-adaptive) PCPP for CktVal by describing the action of the verifier on all R possible outcomes of the random bits as distinct output circuits. Hence, the two notions are semantically equivalent. This equivalence implies that all properties of PCPPs (such as composition and the relationship with standard PCPs) are also valid for assignment testers. In particular, as in Lemma 5.14, assignment testers also compose in a simple manner. However, recall that the outer tester needs to be robust for the composition to work. The composition lemma of [DR04] uses the obvious robustness of $\frac{1}{q}$ (where q stands for the query complexity) for the outer verifier and *transparently* robustifies the verifier as a part of the proof. This is done by the well known method of using asymptotically good codes with linear-size circuits for encoding and codeword decoding (e.g., Spielman codes [Spi96]).

The composition of assignment testers is a main tool for reducing the output circuit size, i.e., obtaining an assignment tester that generates smaller circuits (it may potentially decrease the query complexity as well, as query complexity is trivially upper bounded by the output size). Dinur and Reingold [DR04] introduce three additional combinatorial operators that generally operate on assignment testers, namely,

1. A proximity reduction operator, which reduces the proximity parameter δ to an arbitrary value at a minor cost in the other parameters. The operator is constructed using a good error-correcting code which is also used in the composition lemma and an efficiently constructible *set hitters*[3] based on expander graphs with proper parameters.

2. A *serial* error reduction operator, which reduces the soundness error parameter ϵ to an arbitrary small value. This operator uses a naive serial repetition of the tester (i.e., taking conjunction of uniformly selected subsets of the output circuits) and comes at the expense of a considerable increase in the query complexity.

3. A *parallel* error reduction operator, which is similar to the previous transformation with the difference that it reduces *both* the soundness error and the query complexity. This is done by aggregating queries by introducing new variables, similar to the *parallelization* step in the original proof of the PCP theorem but in a combinatorial way.

Using the repeated application of the above transformations and an additional assumption that a *weak* assignment tester with polynomial randomness, constant query complexity, soundness error and proximity but a large output size (caused by a large answer width) exists, a strong assignment tester can be inductively constructed. The strong tester has a polynomial randomness and the rest of its parameters are absolute constants. Along with Proposition 5.12, this proves the PCP theorem. However,

[3] A set hitter of size n is roughly a collection of n subsets of $[n]$ such that every subset of $[n]$ of size $\Omega(n)$ intersects a constant fraction of the subsets.

existence of the weak tester is assumed in [DR04] based on the earlier (algebraic) constructions and is not proved by a combinatorial technique.

The work of [DR04] is followed by the result of Dinur [Din05] who gives a fully combinatorial proof of the PCP theorem by *gap amplification* of constraint satisfaction problems.

Recall the constraint satisfaction problem (CSP) from section 5.2. Let $\mathcal{C}$ be a CSP. The *satisfiability gap* of $\mathcal{C}$, denoted by $\overline{\mathtt{SAT}}(\mathcal{C})$, is defined as the smallest fraction of unsatisfied constraints over all possible assignments of the variables. Note that $\mathcal{C}$ is satisfiable if and only if $\overline{\mathtt{SAT}}(\mathcal{C}) = 0$. Moreover, the satisfiability gap of an unsatisfiable CSP with n constraints is trivially lower bounded by $\frac{1}{n}$. Thus, there is a difference of at least $\frac{1}{n}$ between the satisfiability gap of a satisfiable CSP and an unsatisfiable one. The idea behind gap amplification is to increase such a difference to a constant value by repetitive combinatorial transformations.

Observe that the classical SAT problem is a special case of CSP. Hence, the CSP problem as well as c-CSP for $c \geq 3$ is NP-hard. However, the problem 2-CSP can also be NP-hard:

Proposition 5.21. *The problem 2-CSP is NP-hard when the alphabet size is at least 7.*

Proof: Consider the following reduction from 3SAT: Construct a 2-CSP by putting a variable for each clause. There are 7 possible assignments satisfying each individual clause. Consider 7 elements of the alphabet for such assignments, and put consistency constraints between every pair of clauses that share a variable. ∎

The special case of CSP for arity two, namely 2-CSP, is of particular interest, as it gives us the following intuitive graph representation:

Definition 5.22. Let $\mathcal{C}$ be an instance of 2-CSP over the alphabet Σ. The *constraint graph* for $\mathcal{C}$ is defined as an undirected graph with one vertex for each variable. Moreover, for each constraint $c\colon \Sigma \times \Sigma \to \{0, 1\}$ defined on a pair of variables u and v, the constraint graph has an edge labeled with c between the vertices corresponding to u and v.

Now consider the gap-3SAT problem, defined as follows:

Definition 5.23. For some constant $\alpha > 0$ the problem of distinguishing between the two case $\overline{\mathtt{SAT}}(\mathcal{C}) = 0$ (i.e., $\mathcal{C}$ is satisfiable) and $\overline{\mathtt{SAT}}(\mathcal{C}) \geq \alpha$ (i.e., $\mathcal{C}$ is far from being satisfiable), where $\mathcal{C}$ is the instance of CSP corresponding to an instance of 3SAT, is known as the *gap-3SAT* problem.

The following proposition gives us an interesting combinatorial equivalent of the PCP theorem:

Proposition 5.24. *The PCP theorem is equivalent to saying that gap-3SAT is* NP*-hard.*

Proof: The PCP theorem implies hardness of approximating MAX-3SAT and hence hardness of gap-3SAT by Proposition 5.10. Now, suppose that gap-3SAT is NP-hard. We construct a PCP verifier V for SAT. Given a boolean formulae φ, NP-hardness of gap-3SAT implies that we can transform the formula φ in polynomial time into a 3CNF formula φ' of polynomial size such that the following two properties hold:

- If φ is satisfiable, then so it φ',
- If φ is unsatisfiable, then at most a constant fraction $\alpha < 1$ of the clauses in φ' can be satisfied.

Hence, we can consider assignments of the variables in φ' as a PCP for the instance φ of SAT. Then the verification procedure of V goes as follows: Construct the formula φ, pick one of its clauses uniformly at random and check if the given proof satisfies it. This requires only three proofs into the proof oracle. The completeness and soundness of V immediately follows from the two properties of φ mentioned above. To get a soundness error of $\frac{1}{2}$, it suffices to repeat the process for a constant number of times, and accept if and only if all repetitions accept. ∎

The work of [Din05] essentially resembles (and is in fact inspired by) the proof of Reingold [Rei05] for equivalence of the complexity classes L and SL. Namely, Reingold gives a deterministic log-space algorithm for deciding whether there is a path between a pair of vertices in an undirected graph. He gives a space-efficient construction for converting any undirected graph into an expander with constant spectral gap. The construction roughly works in the following lines: The input graph is first pre-processed to get a regular constant degree graph. Then using recursive applications of the graph powering and the zig-zag graph product (using a small and properly chosen auxiliary graph), the spectral gap of the initial graph is improved to a constant while the degree is kept constant.

On the other hand, the construction of [Din05] is quite similar to the one above. Starting with a constraint graph, it is first preprocessed to obtain a regular constant degree constraint graph. Then the notion of graph powering is extended to work on the constraint graphs. A surprising lemma (namely, *gap-amplification* lemma) implies that graph powering increases the satisfiability gap of the graph, which is an analogous to its effect on the spectral gap. However, powering increases the alphabet size of the constraint graph. This is fixed by an additional *composition operation* that transforms a constraint graph and a 2-query assignment tester into a new constraint graph with a constant-sized alphabet. This is roughly analogous to the role of zig-zag graph product in [Rei05]. Using the above techniques, the following main theorem is obtained:

Theorem 5.25. [Din05] *For any alphabet Σ of constant size, there exists constant $C > 0$ and $0 < \alpha < 1$, such that given a constraint graph $G = (V, E)$ with the set of constraints $\mathcal{C}$ over the alphabet Σ, one can construct, in polynomial time, a constraint graph $G' = (V', E')$ with a constant-sized alphabet that represents a set of constraints $\mathcal{C}'$ such that:*

- *The graph sizes satisfy $size(G') \leq C \cdot size(G)$,*
- *(Completeness) If $\overline{\mathtt{SAT}}(\mathcal{C}) = 0$ then $\overline{\mathtt{SAT}}(\mathcal{C}') = 0$,*
- *(Soundness) $\overline{\mathtt{SAT}}(\mathcal{C}') \geq \min\{2 \cdot \overline{\mathtt{SAT}}(\mathcal{C}), \alpha\}$.* ∎

Proposition 5.21 implies NP-hardness of the constraint graph satisfiability problem. Now given a constraint graph, one can repeatedly apply the above theorem until after a logarithmic number of iterations a constant-gap constraint graph is obtained. Afterward, the final graph can be represented as a 3CNF formula with a size of the same order and a constant satisfiability gap. This immediately implies NP-hardness of the gap-3SAT problem, and hence, by Proposition 5.24, the PCP theorem.

Similarly, the above technique can be applied to a constraint graph obtained from the PCP construction of [BS05] that achieves a poly-logarithmic blow-up in the proof size but a query complexity of the same order. This easily implies the existence of assignment testers with randomness $n \cdot \text{poly}(\log n)$ and constant query complexity. Combined with Proposition 5.12, the following theorem is obtained, which is the tightest PCP characterization of NP to date:

Theorem 5.26. *The complexity class* NP *is precisely* $\mathsf{PCP}(\lg(n \cdot \text{poly}(\log n)), O(1))$. ∎

As assignment testers are equivalent to PCPs of proximity, this also implies the existence of locally testable codes with inverse poly-logarithmic rate that are testable with constant queries.

Chapter 6

Conclusions and Related Work

After studying the close relationship between locally testable codes and probabilistically checkable proofs, that are of central role in many aspects of modern complexity theory, the importance of such codes is already clear.

Despite their resemblance, the two notions were introduced in rather different chains of work. The main motivation behind PCPs was the study of proof systems, and obtaining a characterization of NP [AS98, ALM+98]. On the other hand, the notion of locally testable codes mainly appeared in the context of polynomial low-degree testing [FS95, RS96]. However, low-degree testing is itself a major ingredient of most PCP constructions (with remarkable exception of [DR04, Din05]. The application of PCP systems in proving inapproximability results that we sketched in section 5.2 (cf. [ALM+98, Hås01]) came about with a surprise, and is considered as a strong reason for the study of PCPs. The close connection between locally testable codes and PCPs makes the study of these codes as interesting and fundamental.

One of the initial motivations for the study of low degree tests is program self-verification and self-correction (cf. [Sud92]). Namely, we are given a program as a black box that is supposed to compute a certain function. But we are not completely certain of its correctness and would like to develop efficient means of verifying its behavior. The special case that the program computes a low-degree polynomial reduces to the problem of low degree testing.

Low degree test is also regarded as a special case of the more general *property testing* problem. Roughly speaking, in a property testing problem, we are given an object (e.g, a graph) and would like to test (with certain confidence and efficiently) whether a certain property (e.g., bipartiteness) holds for it, or is unlikely to hold. This problem is formally defined in [RS96, GGR98]. Indeed, low degree test can also be formulated as a property testing problem, the object is a polynomial and the property is having a certain degree. Moreover, locally testable codes can also be formulated in the same manner. Namely, given an oracle string, we would like to test the property of being a codeword in it.

In the original notion of property testing no auxiliary proof is allowed for the test. However, PCP verifiers can also be viewed as an extension of property testers, where they are allowed to get auxiliary information.

However, it is natural to wonder if locally testable codes could have other impacts, and in particular, in practical applications. The fact that locally testable codes allow super-efficient mechanism for error detection is a remarkable one and could be of great interest in practice. Is it possible to apply such codes in practical data transmission and storage systems? More interestingly, is it possible to *encode* mathematical proofs in a locally testable format, so that they can be verified by referees with no pain? (refer to [Sud03] for an informal discussion of the latter.) It is also feasible to think of other practical applications such as reliable software development. Unfortunately, current constructions of locally testable codes are practically inefficient, with respect to the codeword length and encoding time, and also difficult to implement.

Another concern with locally testable codes is decoding. Indeed at the end of the day we wish to retrieve the encoded information from the received string! For current constructions, the original message either appears in a systematic form in a codeword or is easily extractable. Thus, one can apply a codeword test first, and after making sure that the received sequence is close enough to a codeword, retrieve the message string with a naive approach. Indeed, this does not guarantee obtaining the message corresponding to the closest codeword. In fact, the decoding procedure is not clear or explicitly mentioned in current constructions, and might be a difficult task (with the exception of well-studied codes with large block lengths that turn out to be locally testable, such as Reed-Muller codes and Hadamard code).

As a solution for the decoding problem, one can define similar objects to locally testable code that allow very efficient decoding. Suppose that we are guaranteed (either by a local tester or not) that the received string is close to a codeword, and we are interested in retrieving one bit of the message string corresponding to the closest codeword. Is it possible to do so with high confidence by probing the received sequence only at a constant number of positions, in a similar fashion to error detection in locally testable codes? In fact, the answer turns out to be positive. The notion of *locally decodable codes* (initiated by Katz and Trevisan [KT00]) precisely captures the above description. Formally, locally decodable codes are defined as follows:

Definition 6.1. For fixed parameters $\delta, p > 0$, a code $\mathcal{C}\colon \Sigma^k \to \Gamma^{n(k)}$ (with a covering radius greater than δ) is said to be $(q(k), \delta, p)$-locally decodable if there exists a probabilistic (typically non-adaptive) oracle machine M of query complexity $q(k)$ such that for every message $x \in \Sigma^k$, index $i \in [k]$, and string y such that $\boldsymbol{\delta}(y, \mathcal{C}(x)) \leq \delta n$, we have that $\Pr[M^y(i) = x_i] \geq p$. That is, A retrieves the correct value of the i^{th} message symbol correctly with probability at least p. The probability is taken over the coin tosses of M.

Note that as in the case of locally testable codes, various modifications to the above definition would be meaningful, in particular, restricting the correctness probability to be at least $\frac{1}{2} + \epsilon$, for some $\epsilon > 0$ (as in the original definition), addition of a completeness property (i.e., requiring the decoder to be correct on uncorrupted codewords with probability 1), allowing the alphabet size to grow with k, etc. We are particularly interested in binary locally decodable codes with constant query complexity.

Besides syntactical and conceptual similarities, there are other connections between locally testable codes and locally decodable codes. For instance, while using a code concatenation for constructing locally testable codes (as we did in section 4.2.2), we need to *emulate* the codeword tester for the outer code by using few queries. This requires the inner code to have a particular structure that allows us to retrieve the information we need in a *local* manner, which is similar to the notion of locally decodable codes. More importantly, many polynomial low-degree tests imply a similar self-correction procedure. For instance, recall the linear self-corrector of Algorithm 3.19 and multivariate polynomial-line corrector of Algorithm 3.33, that are examples of low-degree correctors. In fact, self-correctors naturally imply locally decodable codes the same way low-degree tests imply locally testable codes. Moreover, self-correctors are also used as a part of certain low-degree tests.

Contrary to the case of locally testable codes, the best known locally decodable codes have very poor rates (they are of sub-exponential length), and in fact, short locally decodable codes are believed to be unattainable. Several lower bounds are known for the block length of locally decodable codes. As a few examples, binary codes decodable with two queries are known to be of length $n = 2^{\Omega(k)}$, and this is achieved by the Hadamard code. Moreover, for three queries a lower bound of roughly quadratic and for q queries a lower bound of roughly $\tilde{\Omega}(k^{q/(q-2)})$ is known for binary codes [KdW03], which is far from the sub-exponential upper bound. For arbitrary fields (even infinite alphabets), based on the work of Goldreich *et al.* [GKST02], Dvir and Shpilka [DS05] prove a lower bound of $2^{\frac{\epsilon\delta k}{4}-1}$ for the block length of a linear $(2, \delta, \epsilon)$-locally decodable code[1]. Such lower bounds clearly imply that the set of achievable rates for locally locally decodable codes is a strict subset of that of locally testable codes, and in particular, there is no hope for nearly-linear length locally testable codes to be also locally decodable.

Hence, locally decodable codes seem to be inapplicable in practice. However, they have remarkable theoretical impacts[2]. In particular, we can mention the close relationship between locally decodable codes and *private information retrieval* (PIR)

[1] Their definition requires the oracle machine to be correct with probability at least ϵ for infinite alphabets and $\frac{1}{|\mathbb{F}|} + \epsilon$ for a finite alphabet $\mathbb{F}$.

[2] Refer to [Tre04] For a good survey of such issues as well as other connections of coding theory to computational complexity.

schemes. In informal words, in a PIR, a common *database* is shared among a number of independent *servers*. A *user* who has the power of randomness is intended to retrieve a bit of information from the database by querying each server once. The communication should be carried out in a way that at the end, the user obtains the desired information (with high probability) and in a *private* manner. Namely, none of the servers should be able to tell what particular piece of information the user really needs, or more ideally, the queries made by the user should be independent of the particular location she is interested in. Indeed, in a locally decodable code, the each probe into the codeword oracle corresponds to one of the servers in a PIR system, and the piece of information to be retrieved is a bit of the message string. Moreover, the privacy requirement guarantees the *smoothness* of the decoder.

Inherent lower bounds on the block length of locally decodable codes motivates seeking for a slightly weaker definition that allows us break such barriers. Ben-Sasson *et al.* [BGH$^+$04] proposed the following relaxation of locally decodable codes:

Definition 6.2. A code $\mathcal{C}\colon \{0,1\}^k \to \{0,1\}^{n(k)}$ is *relaxed locally decodable* if for some constants $\delta, \rho > 0$ (where δ is called the *proximity* parameter) there exists an efficient probabilistic oracle machine M that makes a constant number of queries and satisfies the following three conditions with respect to any $w \in \{0,1\}^n$ and $x \in \{0,1\}^k$ such that $n = n(k)$ and $\boldsymbol{\delta}(w, \mathcal{C}(x)) \leq \delta$:

1. If $w = \mathcal{C}(x)$ is a codeword, then the decoder correctly recovers every bit of x with probability at least $\frac{2}{3}$.

2. On input any index $i \in [k]$ and given access to the oracle w, with probability at least $\frac{2}{3}$ machine M outputs either the i^{th} bit of x or a special *failure* symbol, denoted $\perp$.

3. Forat least a ρ fraction of the indices $i \in [k]$, on input i and oracle access to w, with probability at least $\frac{2}{3}$, machine M outputs the i^{th} bit of x.

In other words, we expect the decoder to act correctly with high probability on codewords. For the case of non-codewords with small amount of corruption, we allow the decoder to output a special "don't know" symbol, denoted as $\perp$, rather than err. Additionally, we require that the decoder does not output too much of such symbols.

The above relaxation allows construction of relaxed locally decodable codes with far smaller codeword length than the best known constructions of standard locally decodable codes. In particular, using PCPs of proximity, the following result is shown in [BGH$^+$04]:

Proposition 6.3. [BGH$^+$04] *For every constant $\epsilon > 0$, there exists a code $\mathcal{C}\colon \{0,1\}^k \to \{0,1\}^{n(k)}$, where $n(k) = k^{1+\epsilon}$, that is relaxed locally decodable with query complexity $O(\frac{1}{\epsilon^2})$ and proximity parameter $\frac{\epsilon}{O(1)}$.* ∎

This reveals the existence (and actually construction) of relaxed locally decodable codes with “constant nearly linear” block length. However, the above result does not goes beyond the known lower bounds for standard locally decodable codes, and hence it remains open to give a separation of locally decodable codes and the relaxed ones. This corresponds to the question of whether it is possible to obtain relaxed codes with better rates (a “tight nearly linear” construction would suffice for a separation). In particular, the open problem regarding the existence of asymptotically good locally testable codes also holds for relaxed locally decodable codes.

Bibliography

[ALM$^+$98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45(3):501–555, 1998. Preliminary version in *Proc. of FOCS'92*. (Cited on pages 4, 30, 31, 47, 48, 50, 53, 54, 58 and 67.)

[Aro94] Sanjeev Arora. *Probabilistic checking of proofs and the hardness of approximation problems*. PhD thesis, University of California at Berkeley, 1994. (Cited on pages 32 and 48.)

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM*, 45(1):70–122, 1998. Preliminary version in *Proc. of FOCS'92*. (Cited on pages 4, 30, 47, 48, 50, 57, 58 and 67.)

[Bab85] L. Babai. Trading group theory for randomness. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC'85)*, pages 421–429, 1985. (Cited on page 46.)

[BCH$^+$96] Mihir Bellare, Don Coppersmith, Johan Hastad, Marcos Kiwi, and Madhu Sudan. Linearity testing over characteristic two. *IEEE Transactions on Information Theory*, 42(6):1781–1795, 1996. Preliminary version in *Proc. of FOCS'95*. (Cited on pages 3 and 26.)

[BFL91] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991. Preliminary version in *Proc. of STOC'90*. (Cited on pages 46 and 48.)

[BFLS91] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing (STOC'91)*, pages 21–31, 1991. (Cited on pages 47, 48 and 55.)

[BGH+04] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust PCPs of proximity, shorter PCPs and applications to coding. In *Proceedings of the Thirty Sixth Annual ACM Symposium on Theory of Computing (STOC'04)*, pages 1–10, 2004. (Cited on pages 36, 49, 54, 55, 56, 57, 58, 59, 61 and 70.)

[BGKW88] M. Ben-Or, S. Goldwasser, J. Kilian, and A. Widgerson. Multi-prover interactive proofs: How to remove intractability assumptions. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 113–131, 1988. (Cited on page 46.)

[BGS03] Eli Ben-Sasson, Oded Goldreich, and Madhu Sudan. Bounds on 2-query codeword testing. In *Proceedings of RANDOM 2003*, pages 216–227, 2003. In volume 2764 of *Lecture Notes in Computer Science*, Springer-Verlag. (Cited on page 35.)

[BLR93] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993. (Cited on pages 26 and 27.)

[BS04] Eli Ben-Sasson and Madhu Sudan. Robust locally testable codes and products of codes. In *Proceedings of RANDOM 2004*, pages 286–297, 2004. (Cited on pages 36, 37, 42 and 43.)

[BS05] Eli Ben-Sasson and Madhu Sudan. Simple PCPs with poly-log rate and query complexity. In *Proceedings of the Thirty Seventh Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 266–275, 2005. (Cited on pages 49 and 66.)

[BSHR03] Eli Ben-Sasson, Prahladh Harsha, and Sofya Raskhodnikova. Some 3cnf properties are hard to test. In *Proceedings of the Thirty Fifth Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 345–354, 2003. (Cited on page 35.)

[Din05] Irit Dinur. The PCP theorem by gap amplification. *ECCC* Technical Report TR05-046 (available online at `http://eccc.uni-trier.de/eccc-reports/2005/TR05-046/index.html`), 2005. (Cited on pages 64, 65, 66 and 67.)

[DR04] Irit Dinur and Omer Reingold. Assignment testers: Toward a combinatorial proof of the PCP-theorem. In *Proceedings of the Forty Fifth Annual*

IEEE Symposium on Foundations of Computer Science (FOCS'04), 2004. (Cited on pages 62, 63, 64 and 67.)

[DS05] Zeev Dvir and Amir Shpilka. Locally decodable codes with 2 queries and polynomial identity testing for depth 3 circuits. In *Proceedings of the Thirty Seventh Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 592–601, 2005. (Cited on page 69.)

[FGL+91] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proceedings of the Thirty Second Annual IEEE Symposium on Foundations of Computer Science (FOCS'91)*, pages 2–12, 1991. (Cited on pages 51 and 74.)

[FGL+96] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *Journal of the ACM*, 43(2):268–292, 1996. Preliminary version in *Proc. of FOCS'91*, as [FGL+91]. (Cited on pages 47, 48 and 51.)

[FRS94] L. Fortnow, J. Rompel, and M. Sipser. On the power of multi-prover interactive protocols. *Theoretical Computer Science*, 134(2):545–557, 1994. Preliminary version in *Proc. of IEEE Symp. on Structural Complexity, 1988*. (Cited on page 46.)

[FS95] Katalin Friedl and Madhu Sudan. Some improvements to total degree tests. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems (ISTCS'95)*, pages 190–198, 1995. Corrected version available online at ⟨`http://theory.lcs.mit.edu/~madhu/papers/friedl.ps`⟩. (Cited on pages 29, 32, 38 and 67.)

[Gal63] Robert G. Gallagar. *Low Density Parity Check Codes*. MIT Press, Cambridge, MA, 1963. (Cited on page 12.)

[GGR98] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *Journal of the ACM*, 45(4):653–750, 1998. Preliminary version in *Proc. of FOCS'96*. (Cited on page 67.)

[GKST02] O. Goldreich, H. Karloff, L. Schulman, and L. Trevisan. Lower bounds for linear locally decodable codes and private information retrieval. In *Proceedings of the Seventeenth Annual Computational Complexity Conference (CCC'02)*, pages 175–183, 2002. (Cited on page 69.)

[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 Twenty Third Annual ACM Symposium on Theory of Computing (STOC'91)*, pages 32–42, 1991. (Cited on page 31.)

[GMR85] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal of Computing*, 18(1):186–208, 1985. (Cited on page 46.)

[GMW91] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):691–729, 1991. Preliminary version in *Proc. of FOCS'86*. (Cited on page 46.)

[Gol05] Oded Goldreich. Short locally testable codes and proofs (survey). *ECCC* Technical Report TR05-014 (available online at `http://eccc.uni-trier.de/eccc-reports/2005/TR05-014/index.html`), 2005. (Cited on page 35.)

[GR05] Venkatesan Guruswami and Atri Rudra. Tolerant locally testable codes. *ECCC* Technical Report TR05-019 (available online at `http://eccc.uni-trier.de/eccc-reports/2005/TR05-019/index.html`), 2005. (Cited on page 34.)

[GS02] Oded Goldreich and Madhu Sudan. Locally testable codes and PCPs of almost-linear length. In *Proceedings of the Forty Third Annual IEEE Symposium on Foundations of Computer Science (FOCS'02)*, pages 13–22, 2002. Full revised version is available online at ⟨`http://www.wisdom.weizmann.ac.il/~oded/p_ltc.html`⟩. (Cited on pages 39, 40, 42, 49, 50, 54 and 59.)

[Har04] Prahladh Harsha. *Robust PCPs of proximity and shorter PCPs*. PhD thesis, Massachusetts Institute of Technology, 2004. (Cited on pages 58 and 59.)

[Hås96] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of the Thirty Seventh Annual IEEE Symposium on Foundations of Computer Science (FOCS'96)*, pages 627–636, 1996. (Cited on page 51.)

[Hås01] J. Håstad. Some optimal inapproximability results. *Journal of the ACM*, 48(4):798–859, 2001. Preliminary version in *Proc. of STOC'97*. (Cited on pages 54 and 67.)

[HS00] P. Harsha and Madhu Sudan. Small pcps with low query complexity. *Computational Complexity*, 9:157–201, 2000. Preliminary version in *Proc. of STACS'01*. (Cited on page 54.)

[KdW03] Iordanis Kerenidis and Ronald de Wolf. Exponential lower bound for 2-query locally decodable codes via a quantum argument. In *Proceedings of the Thirty Fifth Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 106–115, 2003. (Cited on page 69.)

[KT00] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *Proceedings of the Thirty Second Annual ACM Symposium on Theory of Computing (STOC'00)*, pages 80–86, 2000. (Cited on page 68.)

[MS81] F. J. MacWilliams and Neil J. A. Sloane. *The Theory of Error-Correcting Codes*. Elsevier, North-Holland, Amsterdam, 1981. (Cited on page 6.)

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. (Cited on pages 5 and 51.)

[PS94] A. Polishchuk and D. Spielman. Nearly-linear size holographic proofs. In *Proceedings of the Twenty Sixth Annual ACM Symposium on Theory of Computing (STOC'94)*, pages 194–203, 1994. (Cited on pages 54 and 58.)

[PY91] C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991. (Cited on pages 51 and 52.)

[Rei05] Omer Reingold. Undirected ST-connectivity in log-space. In *Proceedings of the Thirty Seventh Annual ACM Symposium on Theory of Computing (STOC'05)*, pages 376–385, 2005. (Cited on page 65.)

[RS92] Ronitt Rubinfeld and Madhu Sudan. Some improvements to total degree tests. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'92)*, pages 23–32, 1992. (Cited on page 27.)

[RS96] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM Journal of Computing*, 25(2):252–271, 1996. (Cited on pages 27, 28, 29, 30, 32 and 67.)

[Sch80] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980. (Cited on pages 2 and 17.)

[Sha92] A. Shamir. $\mathsf{IP} = \mathsf{PSPACE}$. *Journal of the ACM*, 39(4):869–877, 1992. Preliminary version in *Proc. of FOCS'90*. (Cited on page 46.)

[Spi95] Daniel A. Spielman. *Computationally Efficient Error-Correcting Codes and Holographic Proofs*. PhD thesis, Massachusetts Institute of Technology, 1995. (Cited on page 32.)

[Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996. (Cited on page 63.)

[Sud92] Madhu Sudan. *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems*. PhD thesis, University of California at Berkeley, 1992. (Cited on pages 21, 28, 30, 48, 49, 52 and 67.)

[Sud03] Madhu Sudan. Quick and dirty refereeing? *Science Magazine*, 301:1191–1192, 2003. (Cited on page 68.)

[Sze99] M. Szegedy. Many-valued logics and holographic proofs. In *Proceedings of the Twenty Sixth International Colloquium of Automata, Languages and Programming (ICALP99)*, pages 676–686, 1999. In volume 1644 of *Lecture Notes in Computer Science*, Springer-Verlag. (Cited on page 54.)

[Tan81] R. Michael Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, 1981. (Cited on page 15.)

[Tre04] Luca Trevisan. Some applications of coding theory in computational complexity. Survey paper, to appear in *Quaderni di Matematica*. Available online at `http://www.cs.berkeley.edu/~luca/pubs/codingsurvey.pdf`., 2004. (Cited on page 69.)

[Val05] Paul Valiant. The tensor product of two codes is not necessarily robustly testable. To appear in *Ninth International Workshop on Randomization and Computation (RANDOM'05)*, 2005. (Cited on page 43.)

[Vaz01] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001. (Cited on pages 5 and 51.)

Index

alphabet, 6
  input, 7
  output, 7
assignment tester, 62
  robustifying, 63

ball, 8
  volume, 8
balls bound, 9
bin packing, 51

check bits, 13
CktSat, 56
CktVal, 56
clique number, 51
codes, 6
  asymptotically good, 9
  block length of, 7
  concatenated, 14
  covering radius of, 8
  dimension of, 11
  dual of, 12
  error correcting, 6
  family of, 7
  Hadamard, 18
  LDPC, 13
  linear, 11
  locally decodable, 68
    relaxed, 70
  locally testable, 33
    tolerant, 34
  long, 19
  maximum distance separable (MDS), 11
  message length of, 7
  polynomial-line, 38
  rate of, 7
  Reed-Muller, 17
  Reed-Solomon, 16
  regular LDPC, 13
  size of, 7
  systematic, 13
  Tanner product, 15
  tensor product of, 14
codeword, 7
codeword tester, 32
  strong, 34
constraint, 51
  arity, 51
constraint graph, 64
CSP (Constraint Satisfaction Problem), 52
curve, 24
  polynomial, 24
  restriction by, 24

decision complexity, 57
decoder, 8
detection probability, 33
division algorithm, 22

evenly spaced points, 28
EXP, 46

FPTAS, 51

gap-3SAT, 64
generator matrix, 11

Gilbert-Varshamov bound, 10

Hamming bound, 9
Hamming distance, 7
  absolute, 7
  minimum, 8
  relative, 7
holographic proof, 47

interactive proof, 29
IP, 46

knapsack, 51

L-Reduction, 52
line, 24
  offset, 24
  slope, 24
linear reduction, 52
linearity test, 26
lines representation, 30
low-degree test, 25
  proof assisted, 29

MAX-CUT, 51
MAX-SAT, 51
MAX-SNP, 52
message bits, 13
MIP, 46

NEXP, 46
NP, 45
NTIME, 49

oracle, 19
  auxiliary, 29
oracle machine, 19
  adaptive, 20

P, 45
parallelization, 59
parity check matrix, 12
PCP, 47
  of proximity, 55
  theorem, 48
PCP, 47
PCPP, 55
  proximity of, 55
  soundness error of, 55
Plotkin bound, 11
polynomial, 21
  degree, 22
  evaluation, 22
  evaluation vector, 23
  interpolation, 23
  monic, 22
  reducible, 22
  ring, 21
  root, 22
  unique factorization, 23
  zero, 22
private information retrieval (PIR), 69
probabilistically checkable proofs, 47
proof composition, 57
property testing, 67
prover, 29, 45
proximity parameter, 33
PSPACE, 46
PTAS, 51

query complexity, 19

robust characterization, 29
robust codeword tester, 36
robustness, 37, 57

satisfiability gap, 64
set hitter, 63
Singleton bound, 11
sphere packing bound, 9
string, 6

Tanner graph, 12
TSP, 51

verifier, 45
  composed, 57
  inner, 57
  outer, 57
vertex cover, 51

weight, 7

zero knowledge, 46