A New Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation

Markus Bläser [*]
Computer Science, Saarland University
mblaeser@cs.uni-saarland.de

Gorav Jindal [†]
Max-Planck Institut für Informatik
gjindal@mpi-inf.mpg.de

ABSTRACT

We present a deterministic algorithm to interpolate an m-sparse n-variate polynomial which uses $\mathsf{poly}(n, m, \log H, \log d)$ bit operations. Our algorithm works over the integers. Here H is a bound on the magnitude of the coefficient values of the given polynomial. The degree of given polynomial is bounded by d and m is upper bound on number of monomials. This running time is polynomial in the output size. Our algorithm only requires modular black box access to the given polynomial, as introduced in [12]. As an easy consequence, we obtain an algorithm to interpolate polynomials represented by arithmetic circuits.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on polynomials*; G.1.1 [**Mathematics of Computing**]: Numerical Analysis—*Interpolation*

General Terms

Algorithms, Theory, Interpolation

Keywords

Polynomial interpolation; Black box interpolation; Arithmetic circuits

1. INTRODUCTION

Polynomial interpolation has always been an important problem in mathematics and computer science. Interpolation techniques by Lagrange and Newton have been very useful for interpolating univariate polynomials over fields of characteristic zero. There also has been lot of work on interpolating multivariate polynomials over fields of any characteristic. Zippel [26] presented a randomized algorithm for polynomial interpolation. This inspired a lot of further research for finding deterministic algorithms for polynomial interpolation. An important technique for devising deterministic algorithms for polynomial interpolation was provided by Grigoriev and Karpinski [14], in their work on finding matchings for bipartite graphs having bounded permanent. Ben-Or and Tiwari [6] developed a deterministic algorithm for interpolating m-sparse multivariate polynomial in the black-box model using the ideas of Grigoriev and Karpinski [14]. This algorithm cleverly chooses the following points for evaluation

$$(p_1^i, p_2^i, \ldots, p_n^i)$$

where i is in range from 0 to $2m-1$ and p_1, p_2, $\ldots$, p_n are the first n primes. The algorithm by Ben-Or and Tiwari uses $m^2(\log^2 m + \log nd)$ ring operations and $2m$ evaluations of the polynomial. But this operation count is in the algebraic RAM model and not in the traditional Turing model. Kaltofen, Lakshman and Wiley [18] presented a modified version of Ben-Or and Tiwari algorithm [6]. This algorithm uses modular arithmetic to counter coefficient growth in Ben-Or and Tiwari algorithm [6]. Interpolation over finite fields have proved to be more difficult because $x^q \equiv x$ in $\mathbb{F}_q$. Polynomial interpolation has been extensively studied over finite fields in [24, 15, 9]. Grigoriev, Karpinski and Singer [15] devised the first NC algorithm for interpolating m-sparse polynomials over finite fields. Their algorithm can be used to interpolate any m-sparse and n-variate polynomial in $O(\log^3(nm))$ Boolean parallel time, using $O(n^2m^6\log^2(nm))$ processors. Clausen, Dress, Grabmeier and Karpinski [9] showed that the number of queries to the black-box depends upon the degree of the extension field in which we make queries. For finite fields, if the degree of extension is equal to number of variables then we can interpolate any m-sparse polynomial with $m+1$ queries. Klivans and Spielman [19] discovered an algorithm for polynomial interpolation over fields of large characteristic. Reader should note that sparsity of the polynomial is very important when considering interpolation. If number of monomials m is large, then the description of the polynomial is also large. Representing an m-sparse n-variate polynomial as sum of monomials consumes $O(m \cdot (\log H + n \log d))$ space, H being the bound on the magnitude of coefficient values and d being the bound on the degree of every variable. So our algorithm running in time $\mathsf{poly}(n, m, \log H, \log d)$, is really optimal in the sense that the running time is polynomial in the output size.

[*]Work supported by DFG grant Bl 511/10-1

[†]Saarbrücken Graduate School of Computer Science

ISSAC '14, July 23 - 25, 2014, Kobe, Japan

ACM 978-1-4503-2501-1/14/07 ...$15.00.
http://dx.doi.org/10.1145/2608628.2608648.

Kaltofen and Lee [17] gave an interpolation algorithm which adapts to the degree or the number of monomials when these are not given in the input. Avendaño, Krick and Pacetti [4] studied sparse polynomial interpolation over $\mathbb{Z}[x]$ and $\mathbb{C}[x]$. They developed a new version of Newton-Hensel lifting specifically for interpolation questions. For t-sparse polynomial $f \in \mathbb{C}[x]$ represented by straight line programs of length L, Avendaño et al. [4] found an algorithm which interpolates such f in time $O(t^4L)$. Also, they studied interpolation over $(\mathbb{Z}/p\mathbb{Z})[x]$. Avendaño et al. [4] found an algorithm which interpolates t-sparse polynomials in $(\mathbb{Z}/p\mathbb{Z})[x]$ in time $O((t^2+p)\log p)$. But this requires values of the polynomial at special points $\{\rho^i\}_{0\leq i<2t}$, here $\rho \in \mathbb{Z}$ is a primitive root modulo p. Their new version of Newton-Hensel lifting allows the interpolation over $\mathbb{Z}[x]$ from interpolation over $(\mathbb{Z}/p\mathbb{Z})[x]$.

For the case of polynomials represented by straight line programs, Garg and Schost [10] gave an algorithm which uses $\mathsf{poly}(m, n, \log d, l)$ ring operations, here l is the size of given straight line program. It was also shown in [10] that when underlying ring is of integers, algorithm in [10] can be modified to to obtain an interpolation algorithm which uses $\mathsf{poly}(m, n, \log d, l, \log H)$ number of bit operations. Basic idea in [10] is to evaluate the given straight line program polynomial modulo $x^{p_i} - 1$ for sufficiently many primes p_i. This can be done easily if every operation of given straight line program is performed modulo $x^{p_i} - 1$. This gives all the exponents (modulo p_i) of required polynomial. By applying Chinese remaindering on the polynomials whose roots are exponents (modulo p_i), we can compute the polynomial whose roots are exponents of the given straight line program polynomial. Factorization of this polynomial gives us all the exponents. After that it is not hard to compute corresponding coefficients. Note that this technique cannot be applied for black-box polynomial interpolation.

Javadi and Monagan [16] modified Ben-Or and Tiwari algorithm [6] for interpolating polynomials over rings with characteristic p for any prime p.

Giesbrecht and Roche [12] studied polynomial interpolation over a new black box model. We shall describe this new *modular black box* in the next section. For this black box model, a polynomial time interpolation algorithm was described in [12]. We also achieve polynomial running time. The algorithm in [12] worked in 2 stages. In the first stage, polynomial is interpolated modulo a lot of suitable primes. In the second stage, these interpolated polynomials modulo many suitable primes are used to construct a polynomial whose roots are exponents of the polynomial given by the black box. Then the factorization of this polynomial gives the exponents of the required polynomial. Finding coefficients after finding the exponents is quite straightforward. Giesbrecht and Roche [13] studied polynomial interpolation over *remainder black box* which is somewhat similar to the *modular black box* introduced in [12]. A Las Vegas randomized algorithm that uses fewer black box evaluations was presented in [13] for this *remainder black box*. Arnold, Giesbrecht and Roche [2] devised a Monte Carlo algorithm for interpolating polynomials represented by straight-line programs. The algorithm in [2] is more efficient than the ones presented in [10] and[13] . Arnold, Giesbrecht and Roche [3] devised a randomized algorithm for interpolating sparse polynomials represented by straight-line programs over finite fields, which is faster than previous known algorithms. As in case of [12], our algorithm also works in 2 stages. First stage is quite similar to that of [12]. Our second stage uses a novel idea different from [12].

1.1 New Black-box Model

Our algorithm uses access to the black-box in a new way. This was introduced in [12] by the name of *modular black box*. In the traditional black-box model we are given a black-box which represents a polynomial $P(x_1, x_2, \ldots, x_n) \in \mathcal{R}[x_1, x_2, \ldots, x_n]$, here $\mathcal{R}$ is the underlying ring. The black-box takes a point $(a_1, a_2, \ldots, a_n) \in \mathcal{R}^n$ as input and returns $P(a_1, a_2, \ldots, a_n) \in \mathcal{R}$ as output. Any interpolation algorithm in this model asks for value of P at some set of points and after that it has to output P as a list of coefficients along with corresponding monomials.

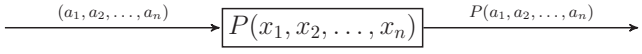

Figure 1.1: Traditional Black-Box Model

As mentioned earlier, in our case $\mathcal{R} = \mathbb{Z}$. In our new black-box model, there are two inputs instead of one. The first input is same as the traditional model, i.e., a point $(a_1, a_2, \ldots, a_n) \in \mathbb{Z}^n$. The second input is a positive number $N \in \mathbb{N}^+$. As an output, we get $P(a_1, a_2, \ldots, a_n) \bmod N$ instead of $P(a_1, a_2, \ldots, a_n)$.

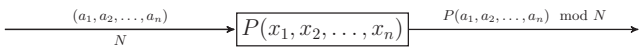

Figure 1.2: Our Black-Box Model

This new black-box model is a very natural extension of the traditional black-box model. The first reason for this is that we do not get any extra information in this new black-box model. The traditional black-box model can easily be simulated in this new black-box model, we just need to choose N (in Figure 1.2) large enough. Or one can also use Chinese remaindering to compute $P(a_1, a_2, \ldots, a_n)$ by computing $P(a_1, a_2, \ldots, a_n) \bmod N$ for many relatively prime N. It is true even the other way round. We can easily simulate this new black-box model by using the traditional black-box model. Secondly, if we want to have an interpolation algorithm which works in time sub-linear in the degree d, then we cannot use the traditional black-box model. This is because $P(a_1, a_2, \ldots, a_n)$ will have bit size of $\Omega(d)$ almost always. Lastly, this new black-box model is a generalized version of arithmetic circuits. It is easy to simulate the new black-box model when the polynomial is given as an arithmetic circuit. We shall make this last point more precise in Section 5.

1.2 Our approach

It is not at all trivial to see why this new black-box model should help us to get faster algorithms for interpolation, since the traditional black-box model has all the information as the new black-box model. One thing that this new black-box model has in its favor is the fact that by choosing N (in Figure 1.2), we can control the size of output returned by the black-box. Still, it is not clear how to use this to devise faster interpolation algorithms. Apart from

the fact that we can control bit length of output returned by black-box, this new black-box model does not help at all. We precisely use this feature of the new black-box model in a novel way. Our approach has two main components. The first component finds some relatively small prime number q_0 such that projection of the polynomial (which we are trying to interpolate) in $\mathbb{F}_{q_0}[x]/(x^{q_0}-x)$ has the same number of monomials as the polynomial itself. For this, we cannot just consider any prime. We shall use primes generated by arithmetic progressions for this purpose. The second component uses this prime q_0 to find sufficiently many primes p of a very special form. Primes p in this phase are also generated by arithmetic progressions. But we choose these arithmetic progressions in a subtle way so that computing the polynomial from its projections in $\mathbb{F}_p[x]/(x^p-x)$ (for many primes p) is efficient. If we chose these primes p by some arbitrary strategy then it would not be clear how to compute the polynomial from its projections in $\mathbb{F}_p[x]/(x^p-x)$. Our novel way of choosing these primes p in the second phase of our algorithm, makes sure that it is easy to compute the polynomial from its projections.

1.3 Organization of the paper

In the next section, we describe all the background needed to understand the main algorithm of this paper. In section 3, we describe our algorithm for interpolating univariate polynomials. This algorithm forms the crux of this paper. Section 4 describes how the algorithm for interpolation of univariate polynomials can be used to interpolate multivariate polynomials. Section 5 adapts the black-box algorithm to arithmetic circuits.

2. PRELIMINARIES

In this paper, $\mathbb{Z}$ denotes the set of integers. $\mathbb{N}$ denotes the set of natural numbers. $\mathbb{N}^+$ denotes the set of positive integers. $[n]$ denotes the set $\{1,2,\ldots,n\}$. We also use the notation $\tilde{O}(t)$ to denote the complexity $O(t\cdot(\log t)^{O(1)})$. $\mathbb{Z}_p$ denotes the quotient group $\mathbb{Z}/p\mathbb{Z}$. The degree of a polynomial is defined as the maximum degree of any variable. $F^p(x)$ denotes the polynomial $F(x) \bmod p$. Note that $F^p(x)$ is the projection of $F(x)$ into $\mathbb{F}_p[x]/(x^p-x)$. More precisely, if $F(x)=\sum_{i=1}^m c_i x^{\alpha_i}$ then $F^p(x)=\sum_{i=1}^m (c_i \bmod p)x^{\alpha_i \bmod_1 (p-1)}$.

Here $c_i \bmod p$ is the unique integer $r\in\{0,1,2,\ldots,p-1\}$ such that $c_i = kp+r$ for some integer k. Similarly for $\alpha_i \bmod (p-1)$. More specifically, whenever the operator " $\bmod N$" appears in this paper for some positive number N, we just think of the result of the operator as the remainder (a number in $\{0,1,2,\ldots,N-1\}$) when the operand is divided by N. Operator $\bmod_1$ can be defined as follows.

$$a \bmod_1 b = \begin{cases} b & \text{if } a\neq 0 \text{ and } a \bmod b = 0 \\ a \bmod p & \text{otherwise} \end{cases}$$

It is easy to see that if we interpolate $F(x)$ modulo a prime p then we obtain $F^p(x)$ (due to Fermat's little theorem). Define $F_p(x)=\sum_{i=1}^m (c_i \bmod p)x^{\alpha_i \bmod (p-1)}$. In the rest of paper, we will always work with $F_p(x)$ for many primes p. Whenever we say "time" in this paper, we mean the number of bit operations.

Some of the techniques used in this paper can be attributed to [7], hence some of the results in this section can also be found in [7]. See also [21] for more details. We reproduce all the main results below to make this paper self contained.

Fact 2.1. *Any integer $n\geq 1$ has at most $\log n$ distinct prime divisors.*

Fact 2.2. *The k-th prime number is of order $\Theta(k\log k)$.*

Using the same notation as in [7], let $P(k)$ denote the smallest prime number in the arithmetic progression $\{jk+1 \mid j\geq 1\}$. Linnik's Theorem gives an *unconditional* upper bound on $P(k)$.

Theorem 2.3 (Linnik's Theorem [20])**.** *There is a constant $L>1$ (called Linnik's constant) such that $P(k)<k^L$ for every sufficiently large $k\geq k_0$.*

The current best upper bound for L is known to be 5 due to [25]. For a discussion on Linnik's Theorem, reader is referred to [22].

In the ensuing discussion, we would want that for different primes $q_1\neq q_2$, we get that $P(q_1)\neq P(q_2)$. However this may not be always true. But the following lemma makes sure that $P(q)$ cannot be the same for too many distinct primes q.

Theorem 2.4 (Lemma 2 in [7])**.** *Let k_0 be the constant mentioned in Linnik's Theorem above. Let $q_1,q_2\ldots,q_v$ be distinct primes of magnitude at least k_0 and p be the prime such that $\forall i\in[v]: P(q_i)=p$. Then, $v\leq 5$.*

We would need the following result about modular computation. The reader is referred to [11] for details.

Fact 2.5 (COROLLARY 4.7 in [11])**.** *One arithmetic operation, that is, addition, multiplication, or division by an invertible element in $\mathbb{Z}_p$ can be performed using $O(\log^2 p)$ bit operations.*

We shall be using the Chinese remainder theorem and gcd computations in the next section. We shall use the following Generalized Chinese remainder theorem and we shall also need an algorithm to find the solution in the case of generalized Chinese remainder theorem. We refer the reader to [5] for the following facts.

Fact 2.6 (COROLLARY 4.2.4 in [5])**.** *Let u,v be positive integers. We can compute the greatest common divisor of u and v using $O(\log u\cdot\log v)$ bit operations.*

Lemma 2.7 (THEOREM 5.4.1 in [5])**.** *For positive integers a,e and n, there is an algorithm to compute $a^e \bmod n$ in $O((\log e)(\log n)^2)$ bit operations, here $0\leq a<n$.*

Theorem 2.8 (Generalized Chinese Remainder Theorem, THEOREM 5.5.5 in [5])**.** *Let $m_1,m_2,\ldots,m_k$ be positive integers. Then the system of congruences*

$$x\equiv x_i \bmod m_i, 1\leq i\leq k$$

has a solution iff $x_i\equiv x_j \pmod{\gcd(m_i,m_j)}$ for all $i\neq j$. If the solution exists, it is unique $\pmod{\operatorname{lcm}(m_1,m_2,\ldots,m_k)}$.

Lemma 2.9 (COROLLARY 5.5.6 in [5])**.** *Let $m_1,m_2,\ldots,m_k$ be positive integers, each ≥ 2, define $m=m_1m_2\ldots m_k$, and $m'=\operatorname{lcm}(m_1,m_2,\ldots,m_k)$. Given the system S of congruences*

$$x\equiv x_i \bmod m_i, 1\leq i\leq k$$

we can determine if S has a solution, using $O((\log m)^2)$ bit operations, and if so, we can find the unique solution $\pmod{m'}$, using $O((\log m)^2)$ bit operations.

3. UNIVARIATE INTERPOLATION

With all the machinery introduced in the last section, we are ready to describe our algorithm for interpolating univariate polynomials. For describing the algorithm, we need some definitions. In the following discussion, we always assume that we are trying to interpolate the following kind of univariate polynomial:

$$F(x) = \sum_{i=1}^{m} c_i x^{\alpha_i}.$$

Here $|c_i| \leq H$. And, $\forall i \in [m] : d \geq \alpha_i$. Also, the α_i's are pairwise distinct. From now on, all the primes we encounter will be at least k_0, the constant mentioned in Linnik's Theorem.

3.1 Finding a Good prime

As we noted earlier, if we interpolate a polynomial $F(x)$ modulo a prime p then we get all the coefficients modulo p. This means that all the coefficients which are 0 modulo p vanish while interpolating modulo p and hence do not appear in $F_p(x)$. Since we do not want to miss any coefficients for complete interpolation, we would want to avoid interpolating modulo any prime p such that some coefficient is 0 modulo p. Also, we saw that we get all exponents modulo $(p-1)$. If the exponents of two different monomials are the same modulo $(p-1)$ then these monomials get merged into a single monomial while interpolating modulo p. We would also like to avoid this situation. The following definitions formalize this notion.

Definition 3.1 (Coefficientbad prime)**.** A prime q is called *Coefficientbad* for a polynomial $F(x)$ if there exists some coefficient c_i such that $c_i \equiv 0 \bmod q$.

Lemma 3.2. *There are at most $m \log H$ Coefficientbad primes.*

Proof. This lemma is a direct implication of the Fact 2.1 and the fact that $|c_i| \leq H$. □

Definition 3.3 (LinnikCoefficientbad number (or prime))**.** A number (or prime) q is called *LinnikCoefficientbad* for a polynomial $F(x)$ if $P(q)$ is *Coefficientbad* prime for $F(x)$.

Lemma 3.4. *There are at most $5m \log H$ LinnikCoefficientbad primes.*

Proof. This lemma is a direct implication of Theorem 2.4 and Lemma 3.2. □

Definition 3.5 (Powerbad prime)**.** A prime q is called *Powerbad* for a polynomial $F(x)$ if there exist $1 \leq i \neq j \leq m$ such that $\alpha_i \equiv \alpha_j \bmod (q-1)$.

Definition 3.6 (LinnikPowerbad number (or prime))**.** A number (or prime) q is called *LinnikPowerbad* for a polynomial $F(x)$ if $P(q)$ is *Powerbad* prime for $F(x)$.

Lemma 3.7. *There are at most $\binom{m}{2} \log d$ LinnikPowerbad primes.*

Proof. By way of contradiction, assume that there are more than $\binom{m}{2} \log d$ *LinnikPowerbad* primes. Let these primes be $q_1, q_2, \ldots, q_k$, here $k > \binom{m}{2} \log d$. Then we have that for all $l \in [k]$ there are $i \neq j$ such that $\alpha_i \equiv \alpha_j \bmod (P(q_l) - 1)$. Since $k > \binom{m}{2} \log d$, by pigeonhole principle there exist $t \geq \log d$ primes $r_1, r_2, \ldots, r_t$ in $q_1, q_2, \ldots, q_k$ such that we have two different indices $i \in [m]$ and $j \in [m]$ satisfying $\forall l \in [t] : \alpha_i \equiv \alpha_j \bmod (P(r_l) - 1)$. Hence $\forall l \in [t] : (P(r_l) - 1) \mid (\alpha_i - \alpha_j)$. This implies that $\forall l \in [t] : r_l \mid (|\alpha_i - \alpha_j|)$. We have that $|\alpha_i - \alpha_j| \leq d$. Using the Fact 2.1, $|\alpha_i - \alpha_j|$ can have at most $\log d$ distinct prime divisors. But we have t (more than $\log d$) distinct prime divisors of $|\alpha_i - \alpha_j|$. Hence our assumption that there are more than $\binom{m}{2} \log d$ *LinnikPowerbad* primes, is wrong. Thus there are at most $\binom{m}{2} \log d$ *LinnikPowerbad* primes. □

Definition 3.8 (Bad number (or prime))**.** A number (or prime) q is called *Bad* for a polynomial $F(x)$ if it is *LinnikCoefficientbad* or *LinnikPowerbad* or both.

Lemma 3.9. *There are at most $\binom{m}{2} \log d + 5m \log H$ Bad primes.*

Proof. Follows from Lemma 3.4 and Lemma 3.7. □

Definition 3.10 (Good number (or prime))**.** A number (or prime) q is called *Good* for a polynomial $F(x)$ if it is not a *Bad* number (or prime).

Note that if we interpolate $F(x)$ modulo a prime p then we obtain the polynomial $F^p(x)$. But we can easily obtain $F_p(x)$ from $F^p(x)$.

Suppose q is some *Good* prime. If we interpolate $F(x)$ modulo $P(q)$ then we shall get a list of all m coefficients modulo $P(q)$ and corresponding powers modulo $(P(q) - 1)$ in $F_{P(q)}(x)$. On the other hand, if q was a *Bad* prime then by definition of *Bad* primes, we would obtain less than m coefficients while interpolating modulo $P(q)$. This happens because for a bad prime q, some coefficient will be 0 modulo $P(q)$ or two monomials will merge into one or both. This argument gives us a test to find *Good* primes. Since we know that there are at most $\binom{m}{2} \log d + 5m \log H$ *Bad* primes, if we try more than $\binom{m}{2} \log d + 5m \log H$ primes then we shall surely find a *Good* prime. And out of these primes, primes which give maximum number of coefficients are surely *Good* primes. But we need to make sure that interpolation alone does not take too much time. The following lemma makes sure that interpolation modulo a prime can be performed efficiently. From now on we use $b = \binom{m}{2} \log d + 5m \log H$ to denote the maximum number of *Bad* primes.

Theorem 3.11 (Theorem 5.1 in [11])**.** *Let $\mathbb{F}$ be a field. Also, let $f(x) \in \mathbb{F}[x]$ be a polynomial of degree less than n. Given the value of $f(x)$ at n distinct points $u_0, u_1, \ldots, u_{n-1} \in \mathbb{F}$, interpolation of $f(x)$ can be performed with $O(n^2)$ operations in $\mathbb{F}$.*

Lemma 3.12. *Interpolation $F_p(x)$ of $F(x)$ modulo a prime p can be computed in time $\tilde{O}(p^2)$ from the values $F(i) \bmod p$, here i ranges from 0 to $p-1$.*

Proof. $F^p(x) \in \mathbb{F}_p[x]$ is a polynomial of degree $p-1$, hence can be interpolated using $O(p^2)$ operations in $\mathbb{F}_p$ using Theorem 3.11. For this we need to know the value of $F^p(x)$ at p distinct points in $\mathbb{F}_p$. The value of $F(x) \bmod p$ is equal to $F^p(x)$ at all points of $\mathbb{F}_p$. Since each operation in $\mathbb{F}_p$ can be performed in $O(\log^2 p)$ bit operations using Fact 2.5, interpolation $F^p(x)$ of $F(x)$ modulo a prime p can be computed in time $\tilde{O}(p^2)$. It is easy to recover $F_p(x)$ from $F^p(x)$. More specifically, if $F^p(x) = \sum_{j=1}^{m} c_j x^{\alpha_j}$ then $F_p(x) = \sum_{j=1}^{m} c_j x^{\beta_j}$. Here

$$\beta_j = \begin{cases} \alpha_j & \text{if } \alpha_j \neq (p-1) \\ 0 & \text{otherwise} \end{cases}$$

□

Now we describe the algorithm to find a *Good* prime.

Algorithm 1 Algorithm to find a *Good* prime and the exact value of m

Input: Black-box for polynomial $F(x)$. Here m is upper bound on the number of monomials in $F(x)$. d is upper bound on the degree of $F(x)$. H is upper bound on magnitude of coefficients of $F(x)$.

Output: A *Good* prime and the exact value of number of monomials in $F(x)$.

1. Let $b = \binom{m}{2}\log d + 5m\log H$. Compute $b+1$ primes $p_1 < p_2 < \ldots < p_{b+1}$ and also $P(p_1), P(p_2), \ldots, P(p_{b+1})$.
2. Interpolate $F(x)$ modulo $P(p_1), P(p_2), \ldots, P(p_{b+1})$ to obtain $F_{P(p_1)}(x), F_{P(p_2)}(x), \ldots, F_{P(p_{b+1})}(x)$.
3. Find any prime p_i such that $F_{P(p_i)}(x)$ has maximum number of monomials among $F_{P(p_1)}(x), F_{P(p_2)}(x), \ldots, F_{P(p_{b+1})}(x)$.
4. Output p_i as a *Good* prime and the number of monomials in $F_{P(p_i)}(x)$ as m.

Claim 3.13. Algorithm 1 finds a *Good* prime and m in time $\mathsf{poly}(m, \log d, \log H)$.

Proof. Correctness follows from the earlier discussion. Since $b = \binom{m}{2}\log d + 5m\log H$, Step 1 finds all the required primes in time $\mathsf{poly}(m, \log d, \log H)$. Here we use the Fact 2.2 to make sure that p_{b+1} (and hence $P(p_{b+1})$, due to the Theorem 2.3) is of absolute value $\mathsf{poly}(m, \log d, \log H)$. To check for primality of a number, we can use AKS algorithm [1]. The total time of Step 1 is still $\mathsf{poly}(m, \log d, \log H)$. Since all primes in Step 1 are of magnitude $\mathsf{poly}(m, \log d, \log H)$, computing $F_{P(p_i)}(x)$ takes $\mathsf{poly}(m, \log d, \log H)$ due to Lemma 3.12. Hence Step 2 takes $(b+1)\mathsf{poly}(m, \log d, \log H)$ time. This time is still $\mathsf{poly}(m, \log d, \log H)$. Step 3 and 4 trivially take $\mathsf{poly}(m, \log d, \log H)$ time. Hence the total time taken by Algorithm 1 is $\mathsf{poly}(m, \log d, \log H)$. □

Now we have seen how to find a *Good* prime. Note that Algorithm 1 can also be extended to find as many *Good* primes as we need. But can we find $F(x)$ from many interpolations $F_{P(p)}(x)$ for many *Good* primes p? Suppose we have found $F_{P(p_i)}(x)$ for many *Good* primes p_i. Suppose $F_{P(p_i)}(x) = \sum_{j=1}^{m} c_{ij}x^{\alpha_{ij}}$. We know that $c_{ij} = c_k \bmod P(p_i)$ for some k. Note that we do not know the index k. We need to know the index k for each interpolation $F_{P(p_i)}(x)$ to correctly use Chinese remaindering to find c_k. There is similar problem with using Chinese remaindering on α_{ij}'s. That is why we would need notion of *Goodg* primes, which will be defined in the next section. Lemma 3.18 justifies the usefulness of this notion of *Goodg* primes.

From now on, let q_0 be the *Good* prime found by Algorithm 1 and also let $g = P(q_0) - 1$. Note that the absolute value of g is $\mathsf{poly}(m, \log d, \log H)$. Let us fix this g for the rest of paper. Now we shall not find *Good* primes but we shall try to find *Good* numbers of the form gp, where p is some prime.

3.2 Finding many Goodg primes

Definition 3.14 (Badg prime)**.** A prime q is called *Badg* for the polynomial $F(x)$ if gq is a *Bad* number. (Note that the term "*Badg*" depends on the number g, which has been fixed above.)

Definition 3.15 (Goodg prime)**.** A prime q is called *Goodg* for the polynomial $F(x)$ if it is not *Badg*.

Lemma 3.16. *There are at most* $b = \binom{m}{2}\log d + 5m\log H$ *Badg primes.*

Proof. It is easy to extend the proof of Lemma 3.4 to show that there are at most $5m\log H$ primes p such that gp is a *LinnikCoefficientbad* number. And similarly the proof of Lemma 3.7 can be extended to show that gp is *LinnikPowerbad* number for at most $\binom{m}{2}\log d$ many primes p. Thus there are at most $\binom{m}{2}\log d + 5m\log H$ *Badg* primes. □

For the following discussion, t will be used to denote the number $\max\{\lceil \log H \rceil + 1, \lceil \log d \rceil\}$. The below written algorithm finds t *Goodg* primes $q_1, q_2, \ldots, q_t$ in time $\mathsf{poly}(m, \log d, \log H)$ and also performs the interpolation of $F(x)$ modulo $P(gq_1), P(gq_2), \ldots, P(gq_t)$, i.e., it computes $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$.

Algorithm 2 Algorithm to find t *Goodg* primes and corresponding interpolation

Input: Black-box for polynomial $F(x)$, $g = p(q_0) - 1$ and m. Here q_0 is the *Good* prime obtained by Algorithm 1 and m is the exact number of monomials in $F(x)$ given by Algorithm 1. Let $b = \binom{m}{2}\log d + 5m\log H$ and $t = \max\{\lceil \log H \rceil + 1, \lceil \log d \rceil\}$.

Output: t *Goodg* primes $q_1, q_2, \ldots, q_t$ and $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$.

1. Compute $b+t$ primes $p_1 < p_2 < \ldots < p_{b+t}$ and also $P(gp_1), P(gp_2), \ldots, P(gp_{b+t})$.
2. Interpolate $F(x)$ modulo $P(gp_1), P(gp_2), \ldots, P(gp_{b+t})$ to obtain $F_{P(gp_1)}(x), F_{P(gp_2)}(x), \ldots, F_{P(gp_{b+t})}(x)$.
3. Find any t *Goodg* primes $q_1, q_2, \ldots, q_t$ from $p_1 < p_2 < \ldots < p_{b+t}$ such that $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$ all have exactly m monomials.

Claim 3.17. Algorithm 2 finds t *Goodg* primes and performs the corresponding interpolation in time $\mathsf{poly}(m, \log d, \log H)$.

Proof. Correctness follows from Lemma 3.16. We just need to show the desired time bound. Step 1 clearly runs in time $\mathsf{poly}(m, \log d, \log H)$, since $b+t$ and g are of absolute value $\mathsf{poly}(m, \log d, \log H)$. In Step 2, since $P(gp_i)$ is also of absolute value $\mathsf{poly}(m, \log d, \log H)$, we can compute $F_{P(gp_1)}(x)$, $F_{P(gp_2)}(x), \ldots F_{P(gp_{b+t})}(x)$ in time $\mathsf{poly}(m, \log d, \log H)$ due to Lemma 3.12. Hence Step 2 also takes $\mathsf{poly}(m, \log d, \log H)$ time. Step 3 clearly takes $\mathsf{poly}(m, \log d, \log H)$ time. Hence total time taken by Algorithm 2 is $\mathsf{poly}(m, \log d, \log H)$. □

3.3 Main Algorithm for Interpolation

After applying Algorithm 2, we have found $F_{P(gq_1)}(x)$, $F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$ for t *Goodg* primes $q_1, q_2, \ldots, q_t$. We need to use $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$ to compute $F(x)$.

Note that each $F_{P(gq_i)}(x)$ is nothing but a list of size m, each member of the list is a pair of some coefficient mod $P(gq_i)$ and a corresponding power mod $(P(gq_i)-1)$. From now on, assume that each $F_{P(gq_i)}(x)$ is of the following form:

$$F_{P(gq_i)}(x) = \sum_{j=1}^{m} c_{ij} x^{\alpha_{ij}}$$

We can construct the c_i's and the α_i's from the c_{ij}'s and the α_{ij}'s using Chinese remaindering but we do not know which are the correct indexes to join. For example, we know that c_1 appears in $F_{P(gq_i)}(x)$ as some $c_{ij} = c_1 \bmod P(gq_i)$ but we do not know the index j. Similarly for the powers α_i's. We know α_1 appears in $F_{P(gq_i)}(x)$ as some $\alpha_{ij} = \alpha_1 \bmod (P(gq_i)-1)$ but we do not know the index j. But we shall show how to use the Chinese remaindering to compute c_i's and α_i's from c_{ij}'s and α_{ij}'s. We shall need the following lemma for this purpose.

Lemma 3.18. *Let u, v be distinct positive integers $\leq t$, and $s = \gcd(P(gq_u)-1, P(gq_v)-1)$. Then for any $j \in [m]$, there exists a unique $j' \in [m]$ such that $\alpha_{uj} \equiv \alpha_{vj'} \bmod s$.*

Proof. First we show the existence of j'. We know that $\alpha_{uj} \equiv \alpha_i \pmod{P(gq_u)-1}$ for some $i \in [m]$. Let j' be such that $\alpha_{vj'} \equiv \alpha_i \pmod{P(gq_v)-1}$. Since α_i is a solution to

$$x \equiv \alpha_{uj} \bmod (P(gq_u)-1)$$
$$x \equiv \alpha_{vj'} \bmod (P(gq_v)-1),$$

by using Theorem 2.8, we need to have

$$\alpha_{uj} \equiv \alpha_{vj'} \pmod{\gcd(P(gq_u)-1, P(gq_v)-1)}.$$

Therefore $\alpha_{uj} \equiv \alpha_{vj'} \bmod s$. Now for uniqueness, by way of contradiction assume that there exists $j'' \neq j'$ such that $\alpha_{uj} \equiv \alpha_{vj''} \bmod s$. Then we get that $\alpha_{vj'} \equiv \alpha_{vj''} \bmod s$. Let k be such that $\alpha_{vj''} \equiv \alpha_k \pmod{P(gq_v)-1}$. Thus $\alpha_{vj''} \equiv \alpha_k \pmod s$. We also have $\alpha_{vj'} \equiv \alpha_i \pmod s$. Altogether, we have the following congruences

$$\alpha_{vj'} \equiv \alpha_{vj''} \bmod s$$
$$\alpha_{vj'} \equiv \alpha_i \bmod s$$
$$\alpha_{vj''} \equiv \alpha_k \bmod s.$$

Hence $\alpha_i \equiv \alpha_k \pmod s$. Note that $g \mid s$. It implies that $\alpha_i \equiv \alpha_k \pmod g$. But this cannot happen because $g = (P(q_0)-1)$ and q_0 was chosen to be a *Good* prime. Therefore there does not exist such $j'' \neq j'$. □

Lemma 3.18 gives us a method to find the correct α_{ij}'s, from which we can recover all α_u's. The algorithm described below completes the interpolation algorithm. It takes $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$ and primes $P(gq_1), P(gq_2), \ldots, P(gq_t)$ as input and outputs $F(x)$. We shall also show that it runs in time $\mathsf{poly}(m, \log d, \log H)$.

Algorithm 3 Algorithm to find $F(x)$

Input: t *Goodg* primes $q_1, q_2, \ldots, q_t$ and $F_{P(gq_1)}(x), F_{P(gq_2)}(x), \ldots, F_{P(gq_t)}(x)$.
Output: $F(x)$.

1. for $j = 1$ to m
   (a) for $i = 2$ to t
      i. $s_i = \gcd(P(gq_1)-1, P(gq_i)-1)$.
      ii. Find $k_{ij} \in [m]$ such that $\alpha_{ik_{ij}} \equiv \alpha_{1j} \bmod s_i$.
   (b) Solve the following equations by Chinese remaindering to obtain c_j:

$$\begin{aligned} x &\equiv c_{1j} \bmod P(gq_1) \\ x &\equiv c_{2k_{2j}} \bmod P(gq_2) \\ &\vdots \\ x &\equiv c_{tk_{tj}} \bmod P(gq_t) \end{aligned}$$

   (c) Solve the following equations by Chinese remaindering to obtain α_j:

$$\begin{aligned} x &\equiv \alpha_{1j} \bmod (P(gq_1)-1) \\ x &\equiv \alpha_{2k_{2j}} \bmod (P(gq_2)-1) \\ &\vdots \\ x &\equiv \alpha_{tk_{tj}} \bmod (P(gq_t)-1) \end{aligned}$$

   (d) If $c_j > H$ then $c_j = c_j - \prod_{i=1}^{t} P(gq_i)$.
2. end for loop
3. Output $F(x)$ as $F(x) = \sum_{i=1}^{m} c_i x^{\alpha_i}$.

Claim 3.19. Algorithm 3 computes $F(x)$ in time $\mathsf{poly}(m, \log d, \log H)$.

Proof. Algorithm 3 uses the result of Lemma 3.18 repeatedly. Step 1a finds the corresponding alignment of monomials. Here, u of Lemma 3.18 is fixed to be 1 whereas v runs from 2 to t. Step 1b computes the coefficient of the monomial whose alignment is found in Step 1a. Step 1c does the similar computation for the power of the corresponding monomial. Step 1d takes care of the fact that c_i can be negative also. And since $t \geq \max\{\lceil \log H \rceil + 1, \lceil \log d \rceil\}$, we have $\prod_{i=1}^{t} P(gq_i) > 2H$ and $\mathrm{lcm}\{P(gq_1)-1, P(gq_2)-1, \ldots, P(gq_t)-1\} > d$. Hence correctness follows. We just have to show a running time upper bound of $\mathsf{poly}(m, \log d, \log H)$. The main loop runs m times. We show that one iteration of the loop takes $\mathsf{poly}(m, \log d, \log H)$ time. Step 1a runs a loop $t-1$ times. Time taken by Step 1(a)i is also $\mathsf{poly}(m, \log d, \log H)$ due to Fact 2.6 and the fact that $P(gq_i)$ is of size $\mathsf{poly}(m, \log d, \log H)$. Step 1(a)ii clearly works in time $\mathsf{poly}(m, \log d, \log H)$ as we just need to iterate over m monomials of $F_{P(gq_i)}(x)$ and find the correct k_{ij}. Step 1b works in time $\mathsf{poly}(m, \log d, \log H)$ because of the fact that $\log^2(\prod_{i=1}^{t} P(gq_i)) = \mathsf{poly}(t, m, \log d, \log H) = \mathsf{poly}(m, \log d, \log H)$ and Lemma 2.9. Similarly, Step 1c works in time $\mathsf{poly}(m, \log d, \log H)$. The rest of the steps trivially works in time $\mathsf{poly}(m, \log d, \log H)$. □

Remark 3.20. If we analyze the running time of our algo-

rithm more precisely, we can see that it runs in the time $\tilde{O}(b^{2L^2+2L+1})$. Here $b = \binom{m}{2} \log d + 5m \log H$ and L is the constant we encountered in Theorem 2.3.

4. MULTIVARIATE INTERPOLATION

In the last section, we presented the algorithm to interpolate univariate polynomials which used $\mathsf{poly}(m, \log d, \log H)$ time. Now we show how to use that algorithm to interpolate multivariate polynomials. We need the following lemma for that, which goes back to Kronecker.

Lemma 4.1 (See e.g. Lemma 1 in [7]). *Let $\mathcal{R}$ be a ring. Let $f \in \mathcal{R}[x_1, \ldots, x_n]$ be a polynomial of the degree at most d. Then the substitution $x_i \mapsto X^{(d+1)^{i-1}}$ maps f to a univariate polynomial $g \in \mathcal{R}[X]$ of degree at most $(d+1)^n - 1$ such that any two distinct monomials in f map to distinct monomials in g. In particular, if f is not identically zero in $\mathcal{R}[x_1, \ldots, x_n]$, then g is not identically zero in $\mathcal{R}[X]$.*

Substitution in Lemma 4.1 converts f into a univariate of the degree at most $(d+1)^n - 1$. And given f in univariate, getting back multivariate f is also easy. This gives us the following algorithm to interpolate multivariate polynomials.

Algorithm 4 Algorithm to interpolate Multivariate polynomial

Input: Black-box for polynomial $P(x_1, x_2, \ldots, x_n)$.
Output: Polynomial $P(x_1, x_2, \ldots, x_n)$.

1. Given the n-variate polynomial $P(x_1, x_2, \ldots, x_i, \ldots, x_n)$ of degree at most d, interpolate the univariate polynomial $P'(X) = P(X, X^{(d+1)}, \ldots, X^{(d+1)^{i-1}}, \ldots, X^{(d+1)^{n-1}})$ of degree at most $(d+1)^n - 1$.
2. Convert P' to P by computing the corresponding unique n-variate monomial for each univariate monomial in P.

Claim 4.2. Algorithm 4 interpolates any n-variate polynomial of degree at most d in $\mathsf{poly}(m, n, \log d, \log H)$ time .

Proof. Correctness follows from Lemma 4.1. The running time for interpolating P' is $\mathsf{poly}(m, \log((d+1)^n - 1), \log H) = \mathsf{poly}(m, n, \log d, \log H)$ due to Lemma 3.19. We just need to make sure that the substitution $x_i \mapsto X^{(d+1)^{i-1}}$ can be computed efficiently. Note that all the numbers encountered in the algorithm for univariate polynomial interpolation are of absolute value $\mathsf{poly}(m, n, \log d, \log H)$. And for the purpose of asking the black-box, we need to compute $X^{(d+1)^{i-1}} \bmod N$ for some number N of absolute value $\mathsf{poly}(m, n, \log d, \log H)$. Hence $X^{(d+1)^{i-1}} \bmod N$ can be computed in time
$\mathsf{poly}(m, n, \log d, \log H)$ using Lemma 2.7. Step 2 is just a trivial conversion to base $(d+1)$. □

5. CONCLUSION AND APPLICATIONS

Note that all the numbers used in our algorithm have magnitude $\mathsf{poly}(m, n, \log d, \log H)$. This makes our algorithm perfectly suitable for interpolating sparse polynomials represented by arithmetic circuits. For an extensive discussion on arithmetic circuits, we refer reader to a recent survey on arithmetic circuits by Shpilka and Yehudayoff [23]. Let C be an arithmetic circuit of size s and description length l which computes the polynomial $P(x_1, x_2, \ldots, x_n)$. Here the size s is the number of gates in the circuit and the description length l is the length of a reasonable encoding of the circuit as a binary string. In particular, 2^s is a bound on the degree of P and every constant appearing in the circuit is bounded by 2^l. In our black-box model, we assumed that we can obtain the value of P at any point modulo some integer in unit time. But it is also easy to see that the value of P at points we encountered can be obtained in time $\mathsf{poly}(m, n, s, l, \log d, \log H)$. The following algorithm exactly performs this task.

Algorithm 5 Algorithm to Evaluate circuit

Input: A description of length l of a division free circuit C (over $\mathbb{Z}$) of size s, $(a_1, a_2, \ldots, a_n) \in \mathbb{Z}^n$ and $N \in \mathbb{N}^+$.
Output: $C(a_1, a_2, \ldots, a_n) \bmod N$.

1. Evaluate every gate of C modulo N.
2. Return result of output gate.

Claim 5.1. Algorithm 5 computes the result of output gate in time $\mathsf{poly}(s, l, \log N, \max_i \log(|a_i| + 1))$.

Proof. Using Fact 2.5, evaluation at each gate of C can be performed in time $O(s \cdot \log^2 N)$. We may need to perform more bit operations when we have to deal with constants in C and a_i's. Let c be the constant used in C of maximum magnitude. We have that $\log(|c| + 1) \leq l$. While evaluating gates which have a constant c as input, we perform $O(s \cdot (\log(|c| + 1) \cdot \log N)^2)$ bit operations. Similarly while evaluating input gates where we have to deal with a_i's, we perform $O(s \cdot (\log(|a_i| + 1) \cdot \log N)^2)$ bit operations. Since we have to do at most s evaluations, Algorithm 5 runs in time $O(s^2.(\max_i \log(|a_i| + 1) \cdot \log(|c| + 1) \cdot \log N)^2)$ time. Since $\log c \leq l$, the whole algorithm takes $O(s^2.(\max_i \log(|a_i| + 1) \cdot l \cdot \log N)^2)$. Hence Algorithm 5 runs in time $\mathsf{poly}(s, l, \log N, \max_i \log(|a_i| + 1))$. □

Using Claim 5.1 it is easy to see that our algorithm for black-box interpolation can be used to interpolate sparse polynomial represented by circuits.

Corollary 5.2. *A polynomial P with at most m monomials represented by an arithmetic circuit C of size s and description length l can be interpolated in time $\mathsf{poly}(m, \log H, l)$, here H is bound on magnitude of coefficients of P.*

Proof. Combining Algorithm 5 with Algorithm 4 gives us a method to interpolate P in time $\mathsf{poly}(s, l, m, n, \log H, \log d)$. Since we have that $d \leq 2^s$ and $n \leq s$, we get that whole algorithm runs in the time $\mathsf{poly}(m, \log H, s, l)$. Also, we have $s \leq l$. Hence whole algorithm runs in the time $\mathsf{poly}(m, \log H, l)$. □

Remark 5.3. If we use faster algorithms for computing $F_p(x)$, such as $\tilde{O}(p)$ algorithm in [8], then we can reduce the running time of our algorithm to $\tilde{O}(b^{L^2+L+1})$. Here $b = \binom{m}{2} \log d + 5m \log H$ and L is the constant we encountered in Theorem 2.3.

6. REFERENCES

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. Primes is in p. *Ann. of Math*, 2:781–793, 2002.

[2] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. Faster sparse interpolation of straight-line programs. In *CASC*, pages 61–74, 2013.

[3] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. Faster sparse polynomial interpolation of straight-line programs over finite fields. *CoRR*, abs/1401.4744, 2014.

[4] Martín Avendaño, Teresa Krick, and Ariel Pacetti. Newton-hensel interpolation lifting. *Foundations of Computational Mathematics*, 6:2006.

[5] E. Bach and J.O. Shallit. *Algorithmic Number Theory: Efficient Algorithms*. Number v. 1 in Algorithmic Number Theory. The Mit Press, 1996.

[6] Michael Ben-Or. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 301–309, New York, NY, USA, 1988. ACM.

[7] Markus Bläser, Moritz Hardt, Richard J. Lipton, and Nisheeth K. Vishnoi. Deterministically testing sparse polynomial identities of unbounded degree. *Information Processing Letters*, 109(3):187 – 192, 2009.

[8] P. Bürgisser, M. Clausen, and M.A. Shokrollahi. *Algebraic Complexity Theory*. A series of comprehensive studies in mathematics. Springer, 1997.

[9] Michael Clausen, Andreas Dress, Johannes Grabmeier, and Marek Karpinski. On zero-testing and interpolation of k-sparse multivariate polynomials over finite fields. *Theoretical Computer Science*, 84(2):151 – 164, 1991.

[10] Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.*, 410(27-29):2659–2662, June 2009.

[11] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[12] Mark Giesbrecht and Daniel S. Roche. Interpolation of shifted-lacunary polynomials. *computational complexity*, 19(3):333–354, 2010.

[13] Mark Giesbrecht and Daniel S. Roche. Diversification improves interpolation. In *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation*, ISSAC '11, pages 123–130, New York, NY, USA, 2011. ACM.

[14] Dima Grigoriev and Marek Karpinski. The matching problem for bipartite graphs with polynomially bounded permanents is in NC (extended abstract). In *FOCS*, pages 166–172, 1987.

[15] Dima Yu. Grigoriev, Marek Karpinski, and Michael F. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. COMPUT*, 19(6):1059–1063, 1990.

[16] Seyed Mohammad Mahdi Javadi and Michael Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 160–168, New York, NY, USA, 2010. ACM.

[17] Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *J. Symb. Comput.*, 36(3-4):365–400, September 2003.

[18] Erich Kaltofen, Lakshman Y. N, and John michael Wiley. Modular rational sparse multivariate polynomial interpolation. In *In ISSAC Š90: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 135–139. ACM Press, 1990.

[19] Adam R. Klivans and Daniel Spielman. Randomness efficient identity testing of multivariate polynomials. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 216–223, New York, NY, USA, 2001. ACM.

[20] U. V. Linnik. On the least prime in an arithmetic progression. I. The basic theorem. *Mat. Sbornik N.S.*, 15(57):139–178, 1944.

[21] Richard J. Lipton and Nisheeth K. Vishnoi. Deterministic identity testing for multivariate polynomials. In *SODA*, pages 756–760. ACM/SIAM, 2003.

[22] P. Ribenboim. *The New Book of Prime Number Records*. Springer-Verlag, 1996.

[23] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010.

[24] Kai Werther. The complexity of sparse polynomial interpolation over finite fields. *Applicable Algebra in Engineering, Communication and Computing*, 5(2):91–103, 1994.

[25] Triantafyllos Xylouris. *Über die Nullstellen der Dirichletschen L-Funktionen und die kleinste Primzahl in einer arithmetischen Progression*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät der Universität Bonn, 2011.

[26] Richard Zippel. Probabilistic algorithms for sparse polynomials. In EdwardW. Ng, editor, *Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer Berlin Heidelberg, 1979.