On Lacunary Polynomial Perfect Powers

Mark Giesbrecht*
Symbolic Computation Group
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
mwg@uwaterloo.ca
www.cs.uwaterloo.ca/~mwg/

Daniel S. Roche
Symbolic Computation Group
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
droche@cs.uwaterloo.ca
www.cs.uwaterloo.ca/~droche/

ABSTRACT

We consider the problem of determining whether a t-sparse or lacunary polynomial f is a perfect power, that is, $f = h^r$ for some other polynomial h and $r \in \mathbb{N}$, and of finding h and r should they exist. We show how to determine if f is a perfect power in time polynomial in t and $\log \deg f$, i.e., polynomial in the size of the lacunary representation. The algorithm works over $\mathbb{F}_q[x]$ (at least for large characteristic) and over $\mathbb{Z}[x]$, where the cost is also polynomial in $\log \|f\|_\infty$. Subject to a conjecture, we show how to find h if it exists via a kind of sparse Newton iteration, again in time polynomial in the size of the sparse representation. Finally, we demonstrate an implementation using the C++ library NTL.

Categories and Subject Descriptors

I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*Algebraic algorithms, Analysis of algorithms*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on polynomials, Number-theoretic computations*

General Terms

Algorithms

Keywords

Lacunary polynomial, black box polynomial, sparse polynomial, perfect power

1. INTRODUCTION

Computational work on *lacunary* polynomials has proceeded apace for the past three decades. From the dramatic initial intractability results of [16, 17], through progress in algorithms (e.g., [3, 22, 12]) and complexity (e.g., [13, 18, 9]), to recent breakthroughs in root finding and factorization [7, 11, 14], these works have important and practical consequences.

By a lacunary or *supersparse* polynomial f, we mean with

$$f = \sum_{1 \le i \le t} c_i \overline{x}^{\overline{e}_i} \in \mathsf{F}[x_1, \ldots, x_\ell], \tag{1.1}$$

where F is a field, $c_0, \ldots, c_t \in \mathsf{F}\backslash\{0\}$, $e_1, \ldots, e_t \in \mathbb{N}^\ell$ are distinct exponent tuples with $0 \le \|e_1\|_1 \le \cdots \le \|e_t\|_1 = \deg f$, and by $\overline{x}_i^{\overline{e}_i}$ we mean the monomial $x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_\ell^{e_{i\ell}}$ of degree $\|\overline{e}_i\|_1 = \sum_{1 \le j \le \ell} e_{ij}$. We say f is *t-sparse* and write $\tau(f) = t$. We will largely consider the univariate case

$$f = \sum_{1 \le i \le t} c_i x^{e_i} \in \mathsf{F}[x], \tag{1.2}$$

where $0 \le e_1 < e_2 < \cdots < e_t = \deg f$.

In this paper, we examine an important operation: detecting whether a lacunary polynomial f is a nontrivial perfect power of another (not necessarily lacunary) polynomial h, and if so producing the power r and possibly the h such that $f = h^r$.

We will always assume that $\tau(f) \ge 2$; otherwise, $f = x^n$, and determining whether f is a perfect power is equivalent to determining if n is not prime, for which there are well-established methods.

The defining methodology of our and previous work in this area is the sensitivity of the cost to the sparse representation. That is, we want algorithms which require a number of bit operations that is polynomial in t and $\log \deg f$. When $f \in \mathbb{Z}[x]$, we furthermore want algorithms for which the number of bit operations is polynomial in $\log \|f\|_\infty$, where $\|f\|_\infty = \max_{1 \le i \le t} |c_i|$ (for $f \in \mathbb{Q}[x]$, we simply work with $\overline{f} = cf \in \mathbb{Z}[x]$, for the smallest $c \in \mathbb{Z}\backslash\{0\}$). This size reflects that of the typical linked representation of polynomials in modern computer algebra systems like Maple and Mathematica.

1.1 Related work and methods

Two well-known techniques can be applied to the problem of testing for perfect powers, and both are very efficient when $f = h^r$ is dense. We can compute the squarefree decomposition of f as in [24], and determine whether f is a perfect power by checking whether the GCD of the exponents of all nontrivial factors in the squarefree decomposition is at least 2. An even faster method (in theory and practice) to find h given $f = h^r$ is by a Newton iteration. This technique has also proven to be efficient in computing perfect roots of (dense) multi-precision integers [2, 4]. In summary however,

*Supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

ISSAC'08, July 20–23, 2008, Hagenberg, Austria.

we note that both these methods require approximately linear time in the *degree* of f, which may be exponential in the lacunary size.

Newton iteration has also been applied to finding perfect polynomial roots of lacunary (or other) polynomials given by straight-line programs. Kaltofen [10] shows how to compute a straight-line program for h, given a straight-line program for $f = h^r$ and the value of r. This method has complexity polynomial in the size of the straight-line program for f, and in the degree of h, and in particular is effective for large r. We do not address the powerful generality of straight-line programs, but do avoid the dependence on the degree of h.

Closest to this current work, Shparlinski [22] shows how to recognize whether $f = h^2$ for a lacunary polynomial f. Shparlinski uses random evaluations and tests for quadratic residues. How to determine whether a lacunary polynomial is *any* perfect power is posed as an open question.

1.2 Our contributions

Given a lacunary polynomial $f \in \mathbb{Z}[x]$, we present an algorithm to compute an $r \in \mathbb{Z}_{>1}$ such that $f = h^r$ for some $h \in \mathbb{Z}[x]$, or determine that no such r exists. Our algorithm requires polynomial time in the sparse input size and in fact is quite efficient, requiring about $O\tilde{}(t \log^2 \|f\|_\infty \log^2 n)$ machine operations (for convenience here we use soft-Oh notation: for functions σ and φ we say $\sigma \in O\tilde{}(\varphi)$ if $\sigma \in O(\varphi \log^c \varphi)$ for some constant $c > 0$). Our algorithms are probabilistic of the Monte Carlo type. That is, they have the ability to generate random bits at unit cost and, for any input, on any execution have a probability of getting an incorrect answer of less than $1/2$ (this possibility of error can be made arbitrarily small with a few repeated executions).

We also answer Shparlinski's open question on perfect powers of lacunary polynomials over finite fields, at least for the case of large characteristic. That is, when the characteristic q of the finite field is greater than $\deg f$, we provide a Monte Carlo algorithm that determines if there exists an $h \in \mathbb{F}_q[x]$ and r such that $f = h^r$, and finds r if it exists.

An implementation of our algorithm in NTL indicates excellent performance on sparse inputs when compared to a fast implementation based on previous technology (a variable-precision Newton iteration to find a power-series rth root of f, followed by a Monte Carlo correctness check).

Actually computing h such that $f = h^r$ is a somewhat trickier problem. Conjectures of Schinzel [20] suggest that again provided F has zero or sufficiently large characteristic, h may well be lacunary as well. In fact, we show that if $f \in \mathbb{Z}[x]$, then the number of terms in h is bounded by $\|f\|_\infty$. Conditional on the truth of a (we believe) reasonable conjecture, and with the knowledge that f is a perfect rth power, we can explicitly compute $h \in \mathsf{F}[x]$, again in time polynomial in $\log n$, t, and $\log \|f\|_\infty$.

The remainder of the paper is arranged as follows. In Section 2 we present the main theoretical tool for our algorithms and then show how to employ it for polynomials over finite fields and the integers. We also show that if a lacunary polynomial is a perfect power of some h, then it cannot be a high power, and show how to identify it. We also show how to reduce the multivariate problem to the univariate one. In Section 3 we show how to compute h such that $f = h^r$ (given that such h and r exist), subject to a conjecture we posit quite reasonable. Finally, in Section 4, we present an experimental implementation of our algorithm in NTL.

2. TESTING FOR PERFECT POWERS

In this section we describe a method to determine if a lacunary polynomial $f \in \mathbb{Z}[x]$ is a perfect power. That is, do there exist $h \in \mathbb{Z}[x]$ and $r > 1$ such that $f = h^r$? The polynomial h need not be lacunary, though some conjectures suggest it may well have to be.

We first describe algorithms to test if f is an rth power of some polynomial h, where f and r *are both given* and r is assumed to be prime. We present and analyze variants that work over finite fields $\mathbb{F}_q$ and over $\mathbb{Z}$. In fact, these algorithms for fixed r are for *black-box* polynomials: they only need to evaluate f at a small number of points. That this evaluation can be done quickly is a property of lacunary and other classes of polynomials.

For lacunary f we then show that, in fact, if h exists at all then r must be small unless $f = x^n$. And if f is a perfect power, then there certainly exists a prime r such that f is an rth power. So in fact the restrictions that r is small and prime are sufficient to cover all nontrivial cases, and our method is complete.

2.1 Detecting given rth powers

Our main tool in this work is the following theorem which says that, with reasonable probability, a polynomial is an rth power if and only if the modular image of an evaluation in a specially constructed finite field is an rth power.

THEOREM 2.1. *Let $\varrho \in \mathbb{Z}$ be a prime power and $r \in \mathbb{N}$ a prime dividing $\varrho - 1$. Suppose that $f \in \mathbb{F}_\varrho[x]$ has degree $n \le 1 + \sqrt{\varrho}/2$ and is* not *a perfect rth power in $\mathbb{F}_\varrho[x]$. Then*

$$R_f^{(r)} = \#\{c \in \mathbb{F}_\varrho : f(c) \in \mathbb{F}_\varrho \text{ is an } r\text{th power}\} \le \frac{3\varrho}{4}.$$

PROOF. The rth powers in $\mathbb{F}_\varrho$ form a subgroup H of $\mathbb{F}_\varrho^*$ of index r and size $(\varrho-1)/r$ in $\mathbb{F}_\varrho^*$. Also, $a \in \mathbb{F}_\varrho^*$ is an rth power if and only if $a^{(\varrho-1)/r} = 1$. We use the method of "completing the sum" from the theory of character sums. We refer to [15], Chapter 5, for an excellent discussion of character sums. By a multiplicative character we mean a homomorphism $\chi : \mathbb{F}_\varrho^* \to \mathbb{C}$ which necessarily maps $\mathbb{F}_\varrho$ onto the unit circle. As usual we extend our multiplicative characters χ so that $\chi(0) = 0$, and define the trivial character $\chi_0(a)$ to be 0 when $a = 0$ and 1 otherwise.

For any $a \in \mathbb{F}_\varrho^*$,

$$\frac{1}{r} \sum_{\chi^r = \chi_0} \chi(a) = \begin{cases} 1 & \text{if } a \in H, \\ 0 & \text{if } a \notin H, \end{cases}$$

where χ ranges over all the multiplicative characters of order r on $\mathbb{F}_\varrho^*$ — that is, all characters that are isomorphic to the trivial character on the subgroup H. Thus

$$R_f^{(r)} = \sum_{a \in \mathbb{F}_\varrho^*} \left(\frac{1}{r} \sum_{\chi^r = \chi_0} \chi(f(a)) \right) = \frac{1}{r} \sum_{\chi^r = \chi_0} \sum_{a \in \mathbb{F}_\varrho^*} \chi(f(a))$$

$$\le \frac{\varrho}{r} + \frac{1}{r} \sum_{\substack{\chi^r = \chi_0 \\ \chi \ne \chi_0}} \left| \sum_{a \in \mathbb{F}_\varrho} \chi(f(a)) \right|.$$

Here we use the obvious fact that

$$\sum_{a \in \mathbb{F}_\varrho^*} \chi_0(f(a)) \le \sum_{a \in \mathbb{F}_\varrho} \chi_0(f(a)) = \varrho - d \le \varrho,$$

where d is the number of distinct roots of f in $\mathbb{F}_\varrho$. We next employ the powerful theorem of Weil [23] on character sums with polynomial arguments (see Theorem 5.41 of [15]), which shows that if f is *not* a perfect rth power of another polynomial, and χ has order $r > 1$, then

$$\left| \sum_{a \in \mathbb{F}_\varrho} \chi(f(a)) \right| \leq (n-1)\varrho^{1/2} \leq \frac{\varrho}{2},$$

using the fact that we insisted $n \leq 1 + \sqrt{\varrho}/2$. Summing over the $r-1$ non-trivial characters of order r, we deduce that

$$R_f^{(r)} \leq \frac{\varrho}{r} + \frac{r-1}{r} \cdot \frac{\varrho}{2} \leq \frac{3\varrho}{4}. \quad \square$$

2.2 Certifying specified powers over $\mathbb{F}_q[x]$

Theorem 2.1 allows us to detect when a polynomial $f \in \mathbb{F}_\varrho[x]$ is a perfect rth power, for known r dividing $\varrho - 1$: choose random $\alpha \in \mathbb{F}_\varrho$ and evaluate $\xi = f(\alpha)^{(\varrho-1)/r} \in \mathbb{F}_\varrho$. Recall that $\xi = 1$ if and only if $f(\alpha)$ is an rth power.

- If f is an rth power, then clearly $f(\alpha)$ is an rth power and we always have $\xi = 1$.
- If f is not an rth power, Theorem 2.1 demonstrates that for at least $1/4$ of the elements of $\mathbb{F}_\varrho$, $f(\alpha)$ is not an rth power. Thus, for α chosen randomly from $\mathbb{F}_\varrho$ we would expect $\xi \neq 1$ with probability at least $1/4$.

For a polynomial $f \in \mathbb{F}_q[x]$ over an arbitrary finite field $\mathbb{F}_q$, where q is a prime power such that $q-1$ is not divisible by r, we proceed by constructing an extension field $\mathbb{F}_{q^{r-1}}$ over $\mathbb{F}_q$. From Fermat's Little Theorem and the fact that $r \nmid q$, we know $r \mid (q^{r-1} - 1)$, and we can proceed as above. We now present and analyze this more formally.

Algorithm `IsPerfectRthPowerGF`

Input: A prime power q, $f \in \mathbb{F}_q[x]$ of degree $n \leq 1 + \sqrt{q}/2$, $r \in \mathbb{N}$ a prime dividing n, and $\epsilon \in \mathbb{R}_{>0}$

Output: True if f is the rth power of a polynomial in $\mathbb{F}_\varrho[x]$; False otherwise.

1: Find an irreducible $\Gamma \in \mathbb{F}_q[z]$ of degree $r-1$, successful with probability at least $\epsilon/2$
2: $\varrho \leftarrow q^{r-1}$
3: Define $\mathbb{F}_\varrho = \mathbb{F}_q[z]/(\Gamma)$
4: $m \leftarrow 2.5(1 + \lceil \log_2(1/\epsilon) \rceil)$
5: **for** i from 1 to m **do**
6: Choose random $\alpha \in \mathbb{F}_\varrho$
7: $\xi \leftarrow f(\alpha)^{(\varrho-1)/r} \in \mathbb{F}_\varrho$
8: **if** $\xi \neq 1$ **then**
9: **return** False
10: **return** True

Notes on `IsPerfectRthPowerGF`.

To accomplish Step 1, a number of fast probabilistic methods are available to find irreducible polynomials. We employ the algorithm of Shoup [21]. This algorithm requires $O((r^2 \log r + r \log q) \log r \log\log r)$ operations in $\mathbb{F}_q$. It is probabilistic of the Las Vegas type, and we assume that it always stops within the number of operations specified, and returns the correct answer with probability at least $1/2$ and "Fail" otherwise (it never returns an incorrect answer). The algorithm is actually presented in [21] as *always* finding an irreducible polynomial, but requiring *expected* time as above; by not iterating indefinitely our restatement allows for a Monte Carlo analysis in what follows. To obtain an irreducible Γ with failure probability at most $\epsilon/2$ we run (our modified) Shoup's algorithm $1 + \lceil \log_2(1/\epsilon) \rceil$ times.

The restriction that $n \leq 1 + \sqrt{2}$ (or alternatively $q \geq 4(n-1)^2$) is not problematic. If this condition is not met, simply extend $\mathbb{F}_q$ with an extension of degree $\nu = \lceil \log_q(4(n-1)^2) \rceil$ and perform the algorithm over $\mathbb{F}_{q^\nu}$. At worst, each operation in $\mathbb{F}_{q^\nu}$ requires $O(\mathrm{M}(\log n))$ operations in $\mathbb{F}_q$.

Here we define $\mathrm{M}(r)$ as a number of operations in F to multiply two polynomials of degree $\leq r$ over F, for any field F, or the number of bit operations to multiply two integers with at most r bits. Using classical arithmetic $\mathrm{M}(r)$ is $O(r^2)$, while using the fast algorithm of [5] we may assume $\mathrm{M}(r)$ is $O(r \log r \log\log r)$.

THEOREM 2.2. *Let q be a prime power, $f \in \mathbb{F}_q[x]$, $r \in \mathbb{N}$ a prime dividing* $\deg f$ *and $\epsilon > 0$. If f is a perfect rth power the algorithm* `IsPerfectRthPowerGF` *always reports this. If f is not a perfect rth power then, on any invocation, this is reported correctly with probability at least $1 - \epsilon$.*

PROOF. That the algorithm always works when f is perfect power is clear from the above discussion. When f is not a perfect power, each iteration of the loop will obtain $\xi \neq 1$ (and hence a correct output) with probability at least $1/4$. By iterating the loop m times we ensure that the probability of failure is at most $\epsilon/2$. Adding this to the probability that Shoup's algorithm for Step 1 will fail yields a total probability of failure of at most ϵ. $\square$

THEOREM 2.3. *On inputs as specified, the algorithm* `IsPerfectRthPowerGF` *requires $O((rM(r) \log r \log q) \cdot \log(1/\epsilon))$ operations in $\mathbb{F}_q$ plus the cost to evaluate $\alpha \mapsto f(\alpha)$ at $O(\log(1/\epsilon))$ points $\alpha \in \mathbb{F}_{q^{r-1}}$.*

PROOF. As noted above, Shoup's [21] algorithm requires $O((r^2 \log r + r \log q) \log r \log\log r)$ field operations per iteration, which is within the time specified. The main cost of the loop in Steps 4–8 is computing $f(\alpha)^{(\varrho-1)/r}$, which requires $O(\log \varrho)$ or $O(r \log q)$ operations in $\mathbb{F}_\varrho$ using repeated squaring, plus one evaluation of f at a point in $\mathbb{F}_\varrho$. Each operation in $\mathbb{F}_\varrho$ requires $O(\mathrm{M}(r))$ operations in $\mathbb{F}_q$, and we repeat the loop $O(\log(1/\epsilon))$ times. $\square$

COROLLARY 2.4. *Given $f \in \mathbb{F}_q[x]$ of degree n with $\tau(f) = t$, and $r \in \mathbb{N}$ a prime dividing n, we can determine if f is an rth power with*

$$O\left((rM(r) \log r \log q + tM(r) \log n) \cdot \log(1/\epsilon)\right)$$

operations in $\mathbb{F}_q$. When f is an rth power, the output is always correct, while if f is not an rth power, the output is correct with probability at least $1 - \epsilon$.

2.3 Certifying specified powers over $\mathbb{Z}[x]$

For an integer polynomial $f \in \mathbb{Z}[x]$, we proceed by working in the homomorphic image of $\mathbb{Z}$ in $\mathbb{F}_p$ (and then in an extension of that field). We must ensure that the homomorphism preserves the perfect power property we are interested in with high probability. Let $\mathrm{disc}(f) = \mathrm{res}(f, f') \in \mathbb{Z}$ be the discriminant of f. The proof of the following is left to the reader.

LEMMA 2.5. *Let $f \in \mathbb{Z}[x]$ and p a prime such that $p \nmid \mathrm{disc}(f)$. Then f is a perfect power in $\mathbb{Z}[x]$ if and only if $\tilde{f} = f \bmod p$ is a perfect power in $\mathbb{F}_p[x]$.*

Using the Hadamard Inequality, it is easily shown that $|\mathrm{disc}(f)| \le n^n \|f\|_2^{2n-1}$, which has at most

$$\mu = \lceil \lceil \log_2(n^n \|f\|_2^{2n-1}) \rceil / \lfloor \log_2(4(n-1)^2) \rfloor \rceil$$

prime factors greater than $4(n-1)^2$ (we require the lower bound $4(n-1)^2$ to employ Theorem 2.1 without resorting to field extensions). Here $\|f\|_2$ is the coefficient 2-norm of f: if f is as in (1.2) then $\|f\|_2 = (\sum_{1\le i\le t} |c_i|^2)^{1/2}$. Choose a $\gamma \ge 4(n-1)^2$ such that the number of primes $\pi(2\gamma) - \pi(\gamma)$ between γ and 2γ is at least $4\mu+1$. By [19], $\pi(2\gamma) - \pi(\gamma) \ge 2\gamma/(5\ln\gamma)$ for $\gamma \ge 59$.Thus if $\gamma \ge \max\{14\mu\ln(14\mu), 100\}$, then a random prime not equal to r in the range $\gamma \ldots 2\gamma$ divides $\mathrm{disc}(f)$ with probability at most 1/4. Primes p of this size have only $\log_2 p \in O(\log n + \log\log\|f\|_\infty)$ bits.

Algorithm `IsPerfectRthPowerZ`

Input: $f \in \mathbb{Z}[x]$ of degree n; $r \in \mathbb{N}$ a prime dividing n; $\epsilon \in \mathbb{R}_{>0}$;
Output: True if f is the rth power of a polynomial in $\mathbb{Z}[x]$; False otherwise

1: $\mu \leftarrow \lceil \lceil \log_2(n^n \|f\|_2^{2n-1}) \rceil / \lfloor \log_2(4(n-1)^2) \rfloor \rceil$
2: $\gamma \leftarrow \max\{14\mu\ln(14\mu), 4(n-1)^2, 100\}$
3: **for** i from 1 to $\ldots \lceil \log_2(1/\epsilon) \rceil$ **do**
4: $\quad p \leftarrow$ random prime in the range $\gamma \ldots 2\gamma$
5: $\quad$ **if** NOT `IsPerfectRthPowerGF`$(p,\ f \bmod p,\ r,\ 1/4\ )$ **then**
6: $\quad\quad$ **return** False
7: **return** True

Theorem 2.6. *Let $f \in \mathbb{Z}[x]$ of degree n, $r \in \mathbb{N}$ dividing n and $\epsilon \in \mathbb{R}_{>0}$. If f is a perfect rth power, the algorithm* `IsPerfectRthPowerZ` *always reports this. If f is not a perfect rth power, on any invocation of the algorithm, this is reported correctly with probability at least $1-\epsilon$.*

Proof. If f is an rth power then so is $f \bmod p$ for any prime p, and so is any $f(\alpha) \in \mathbb{F}_p$. Thus, the algorithm always reports that f is an rth power. Now suppose f is not an rth power. If $p \mid \mathrm{disc}(f)$ it may happen that $f \bmod p$ *is* an rth power. This happens with probability at most 1/4 and we will assume that the worst happens in this case. When $p \nmid \mathrm{disc}(f)$, the probability that `IsPerfectRthPowerGF` incorrectly reports that f is an rth power is also at most 1/4, by our choice of parameter ϵ. Thus, on any iteration of steps 4–6, the probability of finding that f is an rth power is at most 1/2. The probability of this happening $\lceil \log_2(1/\epsilon) \rceil$ times is clearly at most ϵ. □

Theorem 2.7. *On inputs as specified, the algorithm* `IsPerfectRthPowerZ` *requires*

$$O\Big(rM(r)\log r \cdot M(\log n + \log\log\|f\|_\infty) \cdot (\log n + \log\log\|f\|_\infty)\cdot\log(1/\epsilon)\Big),$$

or $O^\sim(r^2(\log n + \log\log\|f\|_\infty)^2 \cdot \log(1/\epsilon))$ bit operations, plus the cost to evaluate $(\alpha, p) \mapsto f(\alpha) \bmod p$ at $O(\log(1/\epsilon))$ points $\alpha \in \mathbb{F}_p$ for primes p with $\log p \in O(\log n + \log\log\|f\|_\infty)$.

Proof. The number of operations required by each iteration is dominated by Step 5, for which $O(r\mathrm{M}(r)\log r \log p)$ operations in $\mathbb{F}_p$ is sufficient by Theorem 2.3. Since $\log p \in O(\log n + \log\log\|f\|_\infty)$ we obtain the final complexity as stated. □

Again, we obtain the following corollary for t-sparse polynomials in $\mathbb{Z}[x]$. This follows since the cost of evaluating a t-sparse polynomial $f \in \mathbb{Z}[x]$ modulo a prime p is $O(t\log\|f\|_\infty \log p + t\log n \mathrm{M}(\log p))$ bit operations.

Corollary 2.8. *Given $f \in \mathbb{Z}[x]$ of degree n, with $\tau(f) = t$, and $r \in \mathbb{N}$ a prime dividing n, we can determine if f is an rth power with*

$$O^\sim\big((r^2\log^2 n + t\log^2 n + t\log\|f\|_\infty \log n)\cdot\log(1/\epsilon)\big)$$

bit operations. When f is an rth power, the output is always correct, while if f is not an rth power, the output is correct with probability at least $1-\epsilon$.

2.4 An upper bound on r.

In this subsection we show that if $f = h^r$ and $f \ne x^n$ then r must be small. Over $\mathbb{Z}[x]$ we show that $\|h\|_2$ is small as well. A sufficiently strong result over many fields is demonstrated in [20], Theorem 1, where it is shown that if f has sparsity $t \ge 2$ then $t \ge r+1$ (in fact a stronger result is shown involving the sparsity of h as well). This holds when either the characteristic of the ground field of f is zero or greater than $\deg f$.

Here we give a (much) simpler result for polynomials in $\mathbb{Z}[x]$, which bounds $\|h\|_2$ and is stronger at least in its dependency on t though it also depends upon the coefficients of f.

Theorem 2.9. *Suppose $f \in \mathbb{Z}[x]$ with $\deg f = n$ and $\tau(f) = t$, and $f = h^r$ for some $h \in \mathbb{Z}[x]$ of degree s and $r \ge 2$. Then $\|h\|_2 \le \|f\|_1^{1/r}$.*

Proof. Let $p > n$ be prime and $\zeta \in \mathbb{C}$ a pth primitive root of unity. Then

$$\|h\|_2^2 = \sum_{0\le i\le s} |h_i|^2 = \frac{1}{p}\sum_{0\le i<p} |h(\zeta^i)|^2.$$

(this follows from the fact that the Discrete Fourier Transform (DFT) matrix is orthogonal). In other words, the average value of $|h(\zeta^i)|^2$ for $i = 0\ldots p-1$ is $\|h\|_2^2$, and so there exists a $k \in \{0,\ldots,p-1\}$ with $|h(\zeta^k)|^2 \ge \|h\|_2^2$. Let $\theta = \zeta^k$. Then clearly $|h(\theta)| \ge \|h\|_2$. We also note that $f(\theta) = h(\theta)^r$ and $|f(\theta)| \le \|f\|_1$, since $|\theta| = 1$. Thus,

$$\|h\|_2 \le |h(\theta)| = |f(\theta)|^{1/r} \le \|f\|_1^{1/r}. \quad \square$$

The following corollary is particularly useful.

Corollary 2.10. *If $f \in \mathbb{Z}[x]$ is not of the form x^n, and $f = h^r$ for some $h \in \mathbb{Z}[x]$, then*

(i) $r \le 2\log_2\|f\|_1$.

(ii) $\tau(h) \le \|f\|_1^{2/r}$

Proof. Part (i) follows since $\|h\|_2 \ge \sqrt{2}$. Part (ii) follows because $\|h\|_2 \ge \sqrt{\tau(h)}$. □

These bounds relate to the sparsity of f since $\|f\|_1 \le \tau(f)\|f\|_\infty$.

2.5 Perfect Power Detection Algorithm

We can now complete the perfect power detection algorithm, when we are given only the t-sparse polynomial f (and not r).

Algorithm `IsPerfectPowerZ`

Input: $f \in \mathbb{Z}[x]$ of degree n and sparsity $t \geq 2$, $\epsilon \in \mathbb{R}_{>0}$
Output: True and r if $f = h^r$ for some $h \in \mathbb{Z}[x]$
False otherwise.

1: $\mathcal{P} \leftarrow \{\text{primes } r \mid n \text{ and } r \leq 2\log_2(t\|f\|_\infty)\}$
2: **for** $r \in \mathcal{P}$ **do**
3: **if** `IsPerfectRthPowerZ`$(f, r, \epsilon/\#\mathcal{P})$ **then**
4: **return** True and r
5: **return** False

THEOREM 2.11. *If $f \in \mathbb{Z}[x] = h^r$ for some $h \in \mathbb{Z}[x]$, the algorithm* `IsPerfectPowerZ` *always returns "True" and returns r correctly with probability at least $1-\epsilon$. Otherwise, it returns "False" with probability at least $1-\epsilon$. The algorithm requires $\tilde{O}(t\log^2\|f\|_\infty \log^2 n \log(1/\epsilon))$ bit operations.*

PROOF. From the preceding discussions, we can see that if f is a perfect power, then it must be a perfect rth power for some $r \in \mathcal{P}$. So the algorithm must return true on some iteration of the loop. However, it may incorrectly return true *too early* for an r such that f is not actually an rth power; the probability of this occurring is the probability of error when f is not a perfect power, and is less than $\epsilon/\#\mathcal{P}$ at each iteration. So the probability of error on any iteration is at most ϵ, which is what we wanted.

The complexity result follows from the fact that each $r \in O(\log t + \log\|f\|_\infty)$ and using Corollary 2.8. □

For polynomials in $\mathbb{F}_q[x]$ we use Schinzel's bound that $r \leq t-1$ and obtain the following algorithm.

Algorithm `IsPerfectPowerGF`

Input: $f \in \mathbb{F}_q[x]$ of degree n and sparsity t, where the characteristic of $\mathbb{F}_q$ is greater than n, and $\epsilon \in \mathbb{R}_{>0}$
Output: True and r if $f = h^r$ for some $h \in \mathbb{F}_q[x]$;
False otherwise.

1: $\mathcal{P} \leftarrow \{\text{primes } r \mid n \text{ and } r \leq t\}$
2: **for** $p \in \mathcal{P}$ **do**
3: **if** `IsPerfectRthPowerGF`$(f, r, \epsilon/\#\mathcal{P})$ **then**
4: **return** True and r;

THEOREM 2.12. *If $f = h^r$ for $h \in \mathbb{F}_q[x]$, the algorithm* `IsPerfectPowerGF` *always returns "True" and returns r correctly with probability at least $1-\epsilon$. Otherwise, it returns "False" with probability at least $1-\epsilon$. The algorithm requires $\tilde{O}(t^3(\log q + \log n))$ operations in $\mathbb{F}_q$.*

PROOF. The proof is equivalent to that of Theorem 2.11, using the complexity bounds in Corollary 2.4. □

2.6 Detecting multivariate perfect powers

In this subsection we examine the problem of detecting multivariate perfect powers. That is, given a lacunary $f \in \mathsf{F}[x_1, \ldots, x_\ell]$ as in (1.1), how do we determine if $f = h^r$ for some $h \in \mathsf{F}[x_1, \ldots, x_\ell]$ and $r \in \mathbb{N}$. This is done simply as a reduction to the univariate case.

The proof of the following is left to the reader.

LEMMA 2.13. *Given $f \in \mathsf{F}[x_1, \ldots, x_\ell]$ of total degree $n > 0$ and such that $\deg_{x_1} f > 0$. Let*

$$\Delta = \operatorname{disc}_{x_1}(f) = \operatorname{res}_{x_1}(f, \partial f/\partial x_1) \in \mathsf{F}[x_2, \ldots, x_\ell].$$

Assume that $a_2, \ldots, a_\ell \in \mathsf{F}$ with $\Delta(a_2, \ldots, a_\ell) \neq 0$. Then $f(x_1, \ldots, x_\ell)$ is a perfect power if and only if $f(x_1, a_2, \ldots, a_\ell) \in \mathsf{F}[x_1]$ is a perfect power.

It is easy to see that the total degree of Δ is less than $2n^2$. Thus, for randomly chosen $a_2, \ldots, a_\ell$ from a set $\mathcal{S} \subseteq \mathsf{F}$ of size at least $8n^2$ we have $\Delta(a_2, \ldots, a_\ell) = 0$ with probability less than $1/4$. This can be made arbitrarily small by increasing the set size and/or repetition. We then run the appropriate univariate algorithm over $\mathsf{F}[x_1]$ (depending upon the field) to identify whether or not f is a perfect power, and if so, to find r.

3. COMPUTING PERFECT ROOTS

Once we have determined that $f \in \mathsf{F}[x]$ is equal to h^r for some $h \in \mathsf{F}[x]$, an obvious question to ask is how to actually compute h. Here we give an algorithm to accomplish this task, subject to a conjecture.

3.1 Sparsity bounds

The conjecture we rely on relates to some questions first raised by Erdös almost 60 years ago [8] on the number of terms of a square of a polynomial. Schinzel later answered these questions and in fact generalized to the case of perfect powers. For any polynomial $h \in \mathsf{F}[x]$, Schinzel proved that $\tau(h^r)$ tends to infinity as $\tau(h)$ tends to infinity [20]. He also gave an explicit upper bound on $\tau(h)$ in terms of $\tau(h^r)$, which unfortunately is an exponential function and therefore not useful for us to prove polynomial-time complexity.

However, our own (limited) investigations, along with more extensive ones by Coppersmith & Davenport [6], and later Abbott [1], suggest that, for any $h \in \mathsf{F}[x]$, where the characteristic of F is not too small, $\tau(h) \in O(\tau(h^r) + r)$.

We make use of the following slightly stronger conjecture, which suffices to prove our algorithm runs in polynomial time.

CONJECTURE 3.1. *For $r, s \in \mathbb{N}$, if the characteristic of F is zero or greater than rs, and $h \in \mathsf{F}[x]$ with $\deg h = s$, then*

$$\tau(h^i \bmod x^{2s}) < \tau(h^r \bmod x^{2s}) + r, \qquad i = 1, 2, \ldots, r-1.$$

This corresponds to intuition and experience, as the system is still overly contrained with only s degrees of freedom. A weaker conjecture would suffice to prove polynomial time, but we use the stated bounds as we believe these give more accurate complexity measures.

3.2 Perfect root computation algorithm

Our algorithm is essentially a Newton iteration, with special care taken to preserve sparsity. We start with the image of h modulo x, using the fact that $f(0) = h(0)^r$, and at Step $i = 1, 2, \ldots, \lceil\log_2(\deg h + 1)\rceil$, we compute the image of h modulo x^i.

Here, and for the remainder of this section, we will assume that $f, h \in \mathsf{F}[x]$ with degrees n and s respectively such that $f = h^r$ for $r \in \mathbb{N}$ at least 2, and that the characteristic of F is either zero or greater than n. As usual, we define $t = \tau(f)$. We require the following simple lemma.

LEMMA 3.2.[1] *Let $k, \ell \in \mathbb{N}$ such that $\ell \leq k$ and $k + \ell \leq s$, and suppose $h_1 \in \mathsf{F}[x]$ is the unique polynomial with degree less than k satisfying $h_1^r \equiv f \bmod x^k$. Then*

$$\tau(h_l^{r+1} \bmod x^{k+\ell}) \leq 2t(t+r).$$

[1]Lemma subject to the validity of Conjecture 3.1.

PROOF. Let $h_2 \in \mathsf{F}[x]$ be the unique polynomial of degree less than ℓ satisfying $h_1 + h_2 x^k \equiv h \bmod x^{k+\ell}$. Since $h^r = f$,

$$f \equiv h_1^r + r h_1^{r-1} h_2 x^k \mod x^{k+\ell}.$$

Multiplying by h_1 and rearranging gives

$$h_1^{r+1} \equiv h_1 f - r f h_2 x^k \mod x^{k+\ell}.$$

Because $h_1 \bmod x^k$ and $h_2 \bmod x^\ell$ each have at most $\tau(h)$ terms, which by Conjecture 3.1 is less than $t-r$, the total number of terms in $h_1^{r-1} \bmod x^{k+\ell}$ is less than $2t(t-r)$. □

This essentially tells us that the "error" introduced by examining higher-order terms of h_1^r is not too dense. It leads to the following algorithm for computing h.

Algorithm `ComputePolyRoot`

Input: $f \in \mathsf{F}[x]$, $r \in \mathbb{N}$ such that f is a perfect rth power
Output: $h \in \mathsf{F}[x]$ such that $f = h^r$

1: $u \leftarrow$ highest power of x dividing f
2: $f_u \leftarrow$ coefficient of x^u in f
3: $g \leftarrow f/(f_u x^u)$
4: $h \leftarrow 1, \quad k \leftarrow 1$
5: **while** $kr \le \deg g$ **do**
6: $\quad \ell \leftarrow \min\{k, (\deg g)/r + 1 - k\}$
7: $\quad a \leftarrow \dfrac{hg - h^{r+1} \bmod x^{k+\ell}}{r x^k}$
8: $\quad h \leftarrow h + (a/g \bmod x^\ell) x^k$
9: $\quad k \leftarrow k + \ell$
10: $b \leftarrow$ *any* rth root of f_u in F
11: **return** $bhx^{u/r}$

THEOREM 3.3. *If $f \in \mathsf{F}[x]$ is a perfect rth power, then* `ComputePolyRoot` *returns an $h \in \mathsf{F}[x]$ such that $h^r = f$.*

PROOF. Let u, f_u, g be as defined in Steps 1–4. Thus $f = f_u g x^u$. Now let $\hat{h}$ be some rth root of f, which we assume exists. If we similarly write $\hat{h} = \hat{h}_v \hat{g} x^v$, with $\hat{g}(0) = 1$, then $\hat{h}^r = \hat{h}_v{}^r \hat{g}^r x^{vr}$. Therefore f_u must be a perfect rth power in F, $r | u$, and g is a perfect rth power in $\mathsf{F}[x]$ of some polynomial with constant coefficient equal to 1.

Denote by h_i the value of h at the beginning of the ith iteration of the while loop. So $h_1 = 1$. We claim that at each iteration through Step 6, $h_i^r \equiv g \bmod x^k$. From the discussion above, this holds for $i = 1$. Assuming the claim holds for all $i = 1, 2, \ldots, j$, we prove it also holds for $i = j+1$.

From Step 8, $h_{j+1} = h_j + (a/g \bmod x^l) x^k$, where a is as defined on the jth iteration of Step 7. We observe that

$$h_j h_j^r \equiv h_j^{r+1} + r h_j^r (a/g \bmod x^l) x^k \mod x^{k+\ell}.$$

From our assumption, $h_j^r \equiv f \bmod x^k$, and $l \le k$, so we have

$$h_j h_{j+1}^r \equiv h_j^{r+1} + r a x^k \equiv h_j^{r+1} + h_j f - h_j^{r+1} \equiv h_j f \mod x^{k+\ell}$$

Therefore $h_{j+1}^r \equiv f \bmod x^{k+\ell}$, and so by induction the claim holds at each step. Since the algorithm terminates when $kr > \deg g$, we can see that the final value of h is an rth root of g. Finally, $\left(bhx^{u/r}\right)^r = f_u g x^u = f$, so the theorem holds. □

THEOREM 3.4.[2] *If $f \in \mathsf{F}[x]$ has degree n and t nonzero terms, then* `ComputePolyRoot` *uses $O\left((t+r)^4 \log r \log n\right)$ operations in F and an additional $O\left((t+r)^4 \log r \log^2 n\right)$ bit operations, not counting the cost of Step 10.*

PROOF. First consider the cost of computing h^{r+1} in Step 7. This will be accomplished by repeatedly squaring and multiplying by h, for a total of at most $2\lfloor \log_2(r+1) \rfloor$ multiplications. As well, each intermediate product will have at most $\tau(f) + r < (t+r)^2$ terms, by Conjecture 3.1. The number of field operations required, at each iteration, is $O\left((t+r)^4 \log r\right)$, for a total cost of $O\left((t+r)^4 \log r \log n\right)$.

Furthermore, since $k + \ell \le 2^i$ at the i'th step, for $1 \le i < \log_2 n$, the total cost in bit operations is less than

$$\sum_{1 \le i < \log_2 n} (t+r)^4 \log_2 r i \in O\left((t+r)^4 \log r \log^2 n\right).$$

In fact, this is the most costly step. The initialization in Steps 1–4 uses only $O(t)$ operations in F and on integers at most n. And the cost of computing the quotient on Step 8 is proportional to the cost of multiplying the quotient and dividend, which is at most $O(t(t+r))$. □

The method used for Step 10 depends on the field F. For $\mathsf{F} = \mathbb{Q}$, we just need to find two integer perfect roots, which can be done in "nearly linear" time [4]. Otherwise, we can compute a root of $x^r - f_u$ using $O(r^{O(1)})$ operations in F.

When $\mathsf{F} = \mathbb{Q}$, we must account for coefficient growth. We use the normal notion of the size of a rational number: For $\alpha \in \mathbb{Q}$, write $\alpha = a/b$ for a, b relatively prime integers. Then define $\mathcal{H}(\alpha) = \max\{|a|, |b|\}$. And for $f \in \mathbb{Q}[x]$ with coefficients $c_1, \ldots, c_t \in \mathbb{Q}$, write $\mathcal{H}(f) = \max \mathcal{H}(c_i)$.

Thus, the size of the lacunary representation of $f \in \mathbb{Q}[x]$ is proportional to $\tau(f), \deg f$, and $\log \mathcal{H}(f)$. Now we prove the bit complexity of our algorithm is polynomial in these values, when $\mathsf{F} = \mathbb{Q}$.

THEOREM 3.5.[2] *Suppose $f \in \mathbb{Q}[x]$ has degree n and t nonzero terms, and is a perfect rth power.* `ComputePolyRoot` *computes an rth root of f using $O^\sim\left(t(t+r)^4 \cdot \log n \cdot \log \mathcal{H}(f)\right)$ bit operations.*

PROOF. Let $h \in \mathbb{Q}[x]$ such that $h^r = f$, and let $c \in \mathbb{Z}_{>0}$ be minimal such that $ch \in \mathbb{Z}[x]$. Gauß's Lemma tells us that c^r must be the least positive integer such that $c^r f \in \mathbb{Z}[x]$ as well. Then, using Theorem 2.9, we have:

$$\mathcal{H}(h) \le \|ch\|_\infty \le \|ch\|_2 \le (t \|c^r f\|_\infty)^{1/r} \le t^{1/r} \mathcal{H}(f)^{(t+1)/r}.$$

(The last inequality comes from the fact that the lcm of the denominators of f is at most $\mathcal{H}(f)^t$.)

Hence $\log \mathcal{H}(h) \in O\left((t \log \mathcal{H}(f))/r\right)$. Clearly the most costly step in the algorithm will still be the computation of h_i^{r+1} at each iteration through Step 7. For simplicity in our analysis, we can just treat h_i (the value of h at the ith iteration of the while loop in our algorithm) as equal to h (the *actual* root of f), since we know $\tau(h_i) \le \tau(h)$ and $\mathcal{H}(h_i) \le \mathcal{H}(h)$.

Lemma 3.2 and Conjecture 3.1 tell us that $\tau(h^i) \le 2(t+r)^2$ for $i = 1, 2, \ldots, r$. To compute h^{r+1}, we will actually compute $(ch)^{r+1} \in \mathbb{Z}[x]$ by repeatedly squaring and multiplying by ch, and then divide out c^{r+1}. This requires at most $\lfloor \log_2 r + 1 \rfloor$ squares and products.

Note that $\|(ch)^{2i}\|_\infty \le (t+r)^2 \|(ch)^i\|_\infty^2$ and $\|(ch)^{i+1}\|_\infty \le (t+r)^2 \|(ch)^i\|_\infty \|ch\|_\infty$. Therefore

$$\|(ch)^i\|_\infty \le (t+r)^{2r} \|ch\|_\infty^r, \quad i = 1, 2, \ldots, r,$$

[2]Theorem subject to the validity of Conjecture 3.1.

and thus $\log \|(ch)^i\|_\infty \in O\left(r(t+r) + t\log \mathcal{H}(f)\right)$, for each intermediate power $(ch)^i$.

Thus each of the $O\left((t+r)^4 \log r\right)$ field operations at each iteration costs $O(\mathrm{M}(t\log \mathcal{H}(f) + \log r(t+r)))$ bit operations, which then gives the stated result. □

3.3 Further comments on the algorithm

We have shown that, subject to the truth of Conjecture 3.1, and using the bounds on r from Corollary 2.10 and [20], our algorithm runs in polynomial time in lacunary representation size of the input. We hope to understand more about the size of a perfect rth root so that our algorithm can be made unconditional, as it seems to perform well in practice.

Note that `ComputePolyRoot` and the theorems that follow require that f is actually a perfect rth power. However, with the concrete bounds on the size of the rth root we have proven and conjectured, it would be easy to terminate the algorithm immediately whenever the partial computation of the result h is "too big" (either in sparsity or height). Finally, using the ideas of sparse interpolation from [12], we can certify that the h computed is actually a perfect rth root with $O(t)$ evaluations in f and in h. These modifications will not affect the asymptotic complexity of the algorithm.

Our algorithm could also be used to compute a right decomposition factor of f of degree n/r (that is, $h \in \mathsf{F}[x]$ such that $f = g \circ h$ for some $g \in \mathsf{F}[x]$) when one exists. This is because the high-order terms of f are the same as those from h^r. However, even less is known about the size of h in this case, so proving a polynomial-time complexity would be more difficult (or rely on more shaky conjectures).

Another approach might be to construct a black box for evaluating h from f and r (and hence avoid difficult conjectures on the sparsity of h). We could then choose to reconstruct h via sparse interpolation. The techniques used in [10] to compute polynomial roots of straight-line programs might be useful here, though it is unclear to us how to avoid the dependence on the degree of h.

4. IMPLEMENTATION

To investigate the practicality of our algorithms, we implemented `IsPerfectPowerZ` using Victor Shoup's NTL. This is a high-performance C++ for fast dense univariate polynomial computations over $\mathbb{Z}[x]$ or $\mathbb{F}_q[x]$.

NTL does not natively support a lacunary polynomial representation, so we wrote our own using vectors of coefficients and of exponents. In fact, since `IsPerfectPowerZ` is a black-box algorithm, the only sparse polynomial arithmetic we needed to implement was for evaluation at a given point.

The only significant diversion between our implementation and the algorithm specified in Section 2 is our choice of the ground field. Rather than working in a degree-$(r-1)$ extension of $\mathbb{F}_p$, we simply find a random p in the same range such that $(r-1) \mid p$. It is more difficult to prove that we can find such a p quickly (using e.g. the best known bounds on Linnik's Constant), but in practice this approach is very fast because it avoids computing in field extensions.

As a point of comparison, we also implemented the Newton iteration approach to computing perfect polynomial roots, which appears to be the fastest known method for dense polynomials. This is not too dissimilar from the techniques from the previous section on computing a lacunary rth root, but without paying special attention to sparsity. We work modulo a randomly chosen prime p to compute an rth perfect root h, and then use random evaluations of h and the original input polynomial f to certify correctness. This yields a Monte Carlo algorithm with the same success probability as ours, and so provides a suitable and fair comparison.

We ran two sets of tests comparing these algorithms. The first set, depicted in Figure 1, does not take advantage of sparsity at all; that is, the polynomials are dense and have close to the maximal number of terms. It appears that the worst-case running time of our algorithm is actually a bit better than the Newton iteration method on dense input, but on the average they perform roughly the same. The lower triangular shape comes from the fact that both algorithms can (and often do) terminate early. The visual gap in the timings for the sparse algorithm comes from the fact that exactly half of the input polynomials were perfect powers. It appears our algorithm terminates more quickly when the polynomial is not a perfect power, but usually takes close to the full amount of time otherwise.

The second set of tests, depicted in Figure 2, held the number of terms of the perfect power, $\tau(f)$, roughly fixed, letting the degree n grow linearly. Here we can see that, for sufficiently sparse f, our algorithm performs significantly and consistently better than the Newton iteration. In fact, we can see that, with some notable but rare exceptions, it appears that the running time of our algorithm is largely independent of the degree when the number of terms remains fixed. The outliers we see probably come from inputs that were unluckily dense (it is not trivial to produce examples of h^r with a given fixed number of nonzero terms, so the sparsity did vary to some extent).

Perhaps most surprisingly, although the choices of parameters for these two algorithms only guaranteed a probability of success of at least $1/2$, in fact over literally millions of tests performed with both algorithms and a wide range of input polynomials, not a single failure was recorded. This is of course due to the loose bounds employed in our analysis, indicating a lack of understanding at some level, but it also hints at the possibility of a deterministic algorithm, or at least one which is probabilistic of the Las Vegas type.

Both implementations are available as C++ code downloadable from the second author's website.

5. REFERENCES

[1] J. Abbott. Sparse squares of polynomials. *Math. Comp.*, 71(237):407–413 (electronic), 2002.

[2] E. Bach and J. Sorenson. Sieve algorithms for perfect power testing. *Algorithmica*, 9(4):313–328, 1993.

[3] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. STOC 1988*, pages 301–309, New York, N.Y., 1988. ACM Press.

[4] D. J. Bernstein. Detecting perfect powers in essentially linear time. *Mathematics of Computation*, 67(223):1253–1283, 1998.

[5] D. Cantor and E. Kaltofen. Fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[6] D. Coppersmith and J. Davenport. Polynomials whose powers are sparse. *Acta Arith.*, 58(1):79–87, 1991.

[7] F. Cucker, P. Koiran, and S. Smale. A polynomial time algorithm for Diophantine equations in one variable. *J. Symbolic Comput.*, 27(1):21–29, 1999.

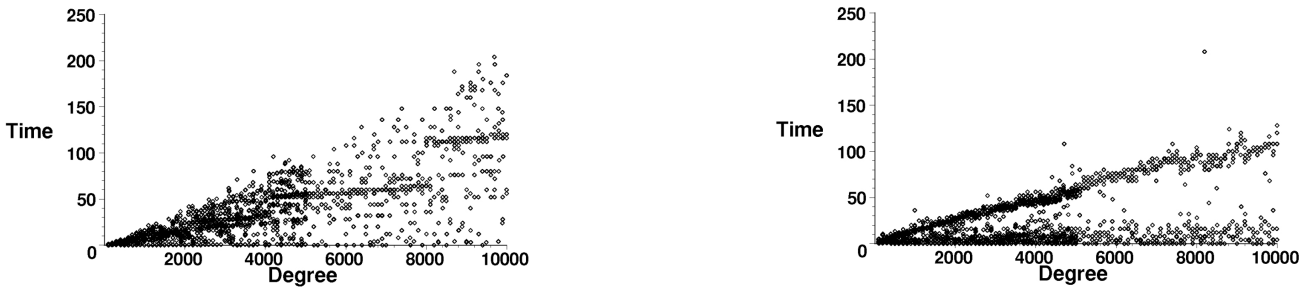

Figure 1: Comparison of Newton Iteration (left) vs. our `IsPerfectPowerZ` (right). Inputs are dense.

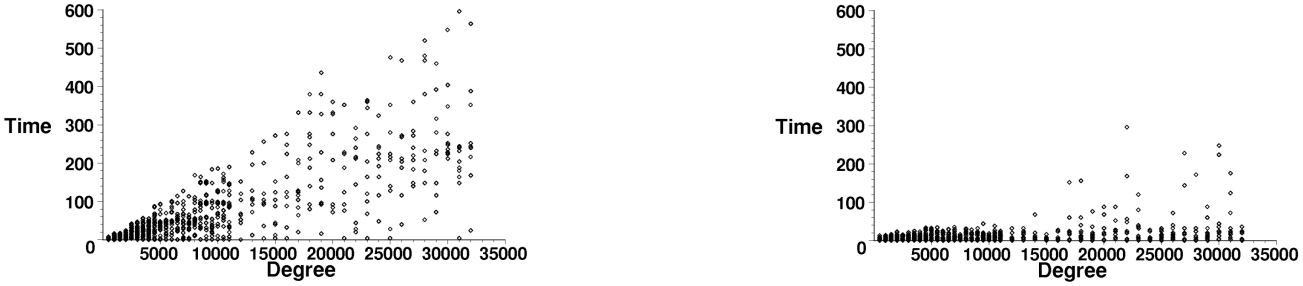

Figure 2: Comparison of Newton Iteration (left) vs our `IsPerfectPowerZ` (right). Inputs are sparse, with sparsity fixed around 500.

[8] P. Erdös. On the number of terms of the square of a polynomial. *Nieuw Arch. Wiskunde (2)*, 23:63–65, 1949.

[9] J. von zur Gathen, M. Karpinski, and I. Shparlinski. Counting curves and their projections. In *ACM Symposium on Theory of Computing*, pages 805–812, 1993.

[10] E. Kaltofen. Single-factor hensel lifting and its application to the straight-line complexity of certain polynomials. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 443–452, New York, NY, USA, 1987. ACM.

[11] E. Kaltofen and P. Koiran. Finding small degree factors of multivariate supersparse (lacunary) polynomials over algebraic number fields. In *ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 162–168. ACM Press, New York, NY, USA, 2006.

[12] E. Kaltofen and W s. Lee. Early termination in sparse interpolation algorithms. *J. Symbolic Comput.*, 36(3-4):365–400, 2003. International Symposium on Symbolic and Algebraic Computation (ISSAC'2002) (Lille).

[13] M. Karpinski and I. Shparlinski. On the computational hardness of testing square-freeness of sparse polynomials. *Electronic Colloquium on Computational Complexity (ECCC)*, 6(027), 1999.

[14] H. W. Lenstra, Jr. Finding small degree factors of lacunary polynomials. In *Number theory in progress, Vol. 1 (Zakopane-Kościelisko, 1997)*, pages 267–276. de Gruyter, Berlin, 1999.

[15] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading MA, 1983.

[16] D. A. Plaisted. Sparse complex polynomials and polynomial reducibility. *J. Comp. and System Sciences*, 14:210–221, 1977.

[17] D. A. Plaisted. New NP-hard and NP-complete polynomial and integer divisibility problems. *Theor. Computer Science*, 31:125–138, 1984.

[18] A. Quick. Some gcd and divisibility problems for sparse polynomials. Technical Report 191/86, University of Toronto, 1986.

[19] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Ill. J. Math.*, 6:64–94, 1962.

[20] A. Schinzel. On the number of terms of a power of a polynomial. *Acta Arith.*, 49(1):55–70, 1987.

[21] V. Shoup. Fast construction of irreducible polynomials over finite fields. *J. Symbolic Comput.*, 17(5):371–391, 1994.

[22] I. Shparlinski. Computing Jacobi symbols modulo sparse integers and polynomials and some applications. *J. Algorithms*, 36(2):241–252, 2000.

[23] A. Weil. On some exponential sums. *Proc Nat. Acad. Sci. U.S.A.*, 34:204–207, 1948.

[24] David Y.Y. Yun. On square-free decomposition algorithms. In *SYMSAC '76: Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 26–35, New York, NY, USA, 1976. ACM.