Fast interpolation and multiplication of unbalanced polynomials

Pascal Giorgi
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
pascal.giorgi@lirmm.fr

Bruno Grenet
LJK, Univ. Grenoble Alpes, CNRS
Grenoble, France
bruno.grenet@univ-grenoble-alpes.fr

Armelle Perret du Cray
University of Waterloo
Waterloo, ON, Canada
aperretducray@uwaterloo.ca

Daniel S. Roche
United States Naval Academy
Annapolis, Maryland, U.S.A
roche@usna.edu

February 20, 2024

Abstract

We consider the classical problems of interpolating a polynomial given a black box for evaluation, and of multiplying two polynomials, in the setting where the bit-lengths of the coefficients may vary widely, so-called *unbalanced* polynomials. Writing s for the total bit-length and D for the degree, our new algorithms have expected running time $\tilde{O}(s \log D)$, whereas previous methods for (resp.) dense or sparse arithmetic have at least $\tilde{O}(sD)$ or $\tilde{O}(s^2)$ bit complexity.

1 Introduction

Consider a univariate polynomial with integer coefficients, written

$$f = c_1 x^{e_1} + c_2 x^{e_2} + \cdots + c_t x^{e_t}, \tag{1}$$

and write $H = \mathsf{height}(f) = \max_i |c_i|$ and $D = \deg f = \max_i e_i$.

Traditionally, fast algorithms have either used the *dense representation* as a list of coefficients with total size $O(D \log H)$, or in the *sparse representation* (a.k.a. *supersparse* or *lacunary*) as a list of t nonzero coefficient-exponent pairs with total size $O(t \log(DH))$. Considerable effort has been made since the 1970s to develop quasi-linear algorithms for many problems with dense and sparse polynomials, But notice that in either case, there is an implicit assumption that all coefficients have the same bit-size $\log_2 H$.

Instead, we focus on the *total bit-length* of f, which we write as $s = \sum_i (\log_2 |c_i| + \log_2 e_i)$. Note that both sizes above — and therefore also any quasi-linear time algorithms in those models — can be as large as quadratic in s. Our goal of quasi-linear complexity with respect to s can be viewed as a natural progression: dense algorithms use $\log_2 H$ bits to represent every possible coefficient; sparse algorithms are more refined by avoiding storage of zero coefficients; and *unbalanced* algorithms (as we are proposing) refine further by using the exact number of bits for every term.

1.1 Our contributions

We provide new algorithms for interpolation and multiplication of polynomials with unbalanced coefficients:

- (Theorem 5.2) Algorithm 5 UInterpolate is a randomized algorithm of the Monte Carlo type, that takes a *modular black box* (see Definition 2.3) for evaluating an unknown polynomial $f \in \mathbb{Z}[x]$, and bounds D, s on its degree and total bit-size, and produces the sparse, unbalanced representation of f with high probability. It uses $O(s \log D \log s)$ black box evaluations and $\tilde{O}(s \log D)$ additional bit operations.
- (Theorem 6.3) Algorithm 6 UnbalancedProd is a randomized algorithm of the Atlantic City type, that takes two polynomials $f, g \in \mathbb{Z}[x]$ and with high probability produces their product fg with expected running time $\tilde{O}(s \log D)$, where s, D are upper bounds on the total bit-sizes and degree of the inputs and outputs. Note that s needs not be known in advance.

Unbalanced interpolation works by recovering terms in *slices* according to coefficient size, starting with the largest ones (Section 5). It relies on a new technique (Section 4) that first recovers a superset of the exponents of terms of interest, in order to avoid carry propagation from smaller terms which have not yet been identified.

Unbalanced multiplication relies on unbalanced interpolation, along with a bound on the total bit-length of the product and a probabilistic product verification method (Section 6).

Our ultimate goal is to achieve quasi-optimal $\tilde{O}(s)$ running time in terms of total bit-size of all coefficients and exponents; Section 7 gives some insights on the barriers to achieving this so far.

Along the way, we also carefully define a new black box model which acts as a precise specification for the interface between our various subroutines (Section 2), and give some separation bounds on carry propagation between terms of different coefficient sizes (Section 3).

1.2 Related work

Dense polynomial multiplication The product of two polynomials $f, g \in \mathsf{R}[z]$ over a generic ring can be computed using the schoolbook algorithm, Karatsuba's algorithm [49], Toom-Cook algorithm [67, 15] or FFT-based algorithms *à la* Schönhage-Strassen [66, 58, 12]. The best ring complexity is $O(D \log D \operatorname{loglog} D)$ via the Cantor-Kaltofen algorithm [12]. For finite fields one can get a better bit complexity using more specialized algorithms [27, 26].

When $R = \mathbb{Z}$, one must take into account coefficient growth during the computation. In particular, FFT-based algorithms require a height bound on the output. If H is the height (i.e. largest coefficient magnitude) of the input polynomials, then the height of the product is at most $H^2 D$, which is $O(\log(DH))$ in terms of bit-length.

Then the fastest method for integer polynomial multiplication is Kronecker substitution [50, 65, 17, 13], which translates the problem into the multiplication of two integers of bit-length $O(D \log(DH))$, by evaluating the polynomials at sufficiently high power of 2. The complexity is therefore $O(D \log(DH) \log(D \log H))$ using the fast integer multiplication [25].

None of these can efficiently handle unbalanced coefficients. To the best of our knowledge, only the Toom-Cook algorithm has been adapted to the case of unbalanced coefficients, by Bodrato and Zanoni [11]. They reduce the problem to bivariate Toom-Cook multiplication, but do not provide a formal complexity analysis.

Sparse polynomial multiplication When f, g are t-sparse polynomials as in (1), the schoolbook algorithm for computing fg requires $O(t^2)$ ring operations plus $O(t^2 \log D)$ bit operations on the exponents. In the worst case this is the best complexity possible, though many practical improvements have been proposed [42, 54, 55]. Output-sensitive algorithms have been designed to provide a better complexity when fewer terms are expected [63, 29, 28, 14, 6]. More recent algorithms, based on sparse interpolation, managed to reach quasi-linear dependency in the output sparsity [57, 20, 22].

Sparse interpolation Modern algorithms to interpolate a sparse integer polynomial given a black box function for its evaluation started with Zippel [69], Ben-Or and Tiwari [7], the latter of which is based on Prony's method from exponential analysis [62]. Numerous extensions have been proposed [70, 45, 38] to handle finite field coefficients [24, 34, 41, 19, 36], discover the sparsity adaptively via early termination [44], or extend to (sparse) rational function recovery [43, 46, 48, 16, 32]. Some algorithms require slight extensions of the normal black box model [24, 53, 1, 56, 48, 19, 9, 30].

Algorithms whose complexity is polynomial in $\log D$ rather than D are termed *supersparse*, and date from [47, 53, 1]. Garg and Schost [18] gave a supersparse algotithm for *white-box* interpolation of a straight-line program; that technique has now been extended to (some kinds of) black boxes over finite fields, or modular rings [3, 4, 39, 37, 35]. More details on these algorithms and techniques can be found in [2, 31, 59]. The first quasi-linear algorithm in terms of $\tilde{O}(t \log(DH))$ was recently given by the authors [22].

Despite this considerable progress, no prior work provides a complexity estimate in terms of the total (possibly unbalanced) bit-length of the output.

Multivariate polynomials A common approach to handling multivariate polynomials is to reduce to the univariate case via variable substitution [47, 5, 38, 22, 33], where such polynomials naturally become (super)sparse. The univariate algorithms we propose will work well when the bit-length of exponents all have bit-lengths close to the maximum $\log D$, where the normal Kronecker substitution preserves the total exponent bit-length. But effectively handling unbalanced exponents remains an open problem.

Linear algebra Tangentially related to our work is the study of integer and polynomial matrices with unbalanced entry sizes, where recent work has focused on complexity in terms of average degree or total bit-length rather than maximum. A cornerstone result is for order basis computation [68], which led to algorithms with similar complexity for Hermite normal form, determinant, and rank, and recently Smith form of an integer matrix [52, 51, 8].

2 Preliminary

2.1 Bit lengths

We use the usual notion of the bit-length of an integer, that is $\mathsf{bitlen}_+(a \in \mathbb{N}) = \lceil \log_2(a+1) \rceil$ and $\mathsf{bitlen}(a \in \mathbb{Z}) = 1 + \mathsf{bitlen}_+(|a|)$.

The bit-length of a sparse polynomial f as in (1) is defined as

$$\mathsf{bitlen}_x(f) = \sum_{i=1}^{t} \left(\mathsf{bitlen}(c_i) + \mathsf{bitlen}_+(e_i)\right),$$

and as usual we define $\mathsf{height}(f) = \max_i |c_i|$.

Useful for us will be this simple lower bound on the sparsity of f in terms of its bit-length.

Lemma 2.1. *Let $f \in \mathbb{Z}[x]$ be a nonzero polynomial with bit-length $s = \mathsf{bitlen}_x(f)$. The number of nonzero terms in f is bounded by $\#f < 2s/\log_2 s$.*

Proof. For any $t \geq 0$, the smallest (in terms of bit-length) polynomial with t terms is $1+x+\cdots+x^{t-1}$, which has total bit-length

$$\sum_{i=0}^{t-1}(\mathsf{bitlen}(1) + \mathsf{bitlen}_+(i)) = 2t + \sum_{i=1}^{t}\lceil \log_2 i \rceil > 2t + \log_2 t!$$

Taking this value as s and t as $\#f$, the lemma is easily confirmed for all $s \leq 16$. So assume $s \geq 17$. We first apply Stirling's approximation of factorial to get

$$s = 2t + \log_2 t! > 2t + t\log_2 \tfrac{t}{e} = t\log_2 \tfrac{4t}{e} > t\log_2 t.$$

Note also that because $s \geq 17$, we have $\log_2 \log_2 s < \frac{1}{2}\log_2 s$.

Finally, by way of contradiction, if $t \geq 2s/\log_2 s$, then

$$t\log_2 t > \tfrac{2s}{\log_2 s}(\log_2 s - \log_2 \log_2 s) > s,$$

which is a contradiction to the inequality shown above. ☐

2.2 Reducing bit length of a polynomial

We will often need to reduce a sparse polynomial $f \in \mathbb{Z}[x]$ modulo $x^p - 1$ over $\mathbb{Z}/m\mathbb{Z}$, for some integers p, m with $p \leq m$, which we denote as $f \bmod \langle x^p - 1, m\rangle$.

Lemma 2.2. *Given $f \in \mathbb{Z}[x]$ and $p, m \in \mathbb{Z}$ with $p \leq m$, computing $f \bmod \langle x^p - 1, m\rangle$ requires $O(\mathsf{bitlen}_x(f) \operatorname{loglog} m)$ bit operations.*

Proof. Consider a single coefficient c_i. Reducing c_i modulo m requires $O\left(\frac{\log|c_i|}{\log m}\right)$ multiplications of $\log_2(m)$-bit integers using the standard method, which has bit complexity $O(\log|c_i| \operatorname{loglog} m)$. Summing over all coefficients, and doing the same for the exponents modulo p, gives the stated total bit complexity. ☐

2.3 Black Boxes: MBBs and MDBBs

We begin with the standard definition of a modular black box for evaluating a sparse integer polynomial, with additional parameters B, L for the later complexity analysis.

Definition 2.3. *A* modular black box *(MBB) π for unknown $f \in \mathbb{Z}[x]$ parameterized by B, L is a procedure which, given $a, m \in \mathbb{N}$ where $a < m$, computes and returns $\pi(a, m)$ that equals $f(a) \bmod m$, using $O(B + L\log m \operatorname{loglog} m)$ bit operations.*

Intuitively, this corresponds to the notion that the MBB works by performing $O(L)$ operations in $\mathbb{Z}/m\mathbb{Z}$ plus $O(B)$ other operations. Moreover, in the final complexity measure, B tracks the total number of calls to the MBB, whereas L (ignoring sub-logarithmic factors) tracks the total size of the outputs produced by the MBB.

Our algorithms for interpolation and multiplication are composed of multiple subroutines and interconnected procedures, both in this paper and from prior work. Of particular importance is defining carefully the interface of these procedures as it concerns the unknown polynomial black box.

While the MBB model above is an appropriate starting point (and indeed can serve as the input for our algorithms), in subroutines we frequently need to update the unknown f with an explicitly-constructed partial result f^*. This update creates a problem for the MBB model, as we do not know any quasi-linear time algorithm for MBB evaluation of a known sparse polynomial where the degree and evaluation modulus may both be large. For instance, computing $a^b \bmod p$ when $\log b \approx \log p$ costs $\tilde{O}(\log^2 p)$ bit operations.

Instead, we will rely on a more specific black box for evaluation.

Definition 2.4. *A* multi-point modular derivative black box *(MDBB) π for an unknown $f \in \mathbb{Z}[x]$ is a procedure which, given $p, \omega, m, k \in \mathbb{N}$ where ω is a pth primitive root of unity (PRU) in $\mathbb{Z}/m\mathbb{Z}$, produces two length-k sequences containing the evaluations of both f and xf' at $1, \omega, \omega^2, \ldots, \omega^{k-1}$, where f' is the formal derivative of f.*

We now show four efficient MDBB constructions: from a MBB, an given polynomial, or the sum or product of two MDBB.

Producing MDBB evaluations from a MBB black box follows the same technique already present in prior work, which exploits the fact that $(1 + m)^e \equiv 1 + em \bmod m^2$, see e.g. [6, 22].

Algorithm 1 MBBtoMDBB(π, p, ω, m, k)

Input: MBB π for unknown $f \in \mathbb{Z}[x]$ and MDBB inputs p, ω, m, k
Output: Evaluations of f and f' at $1, \omega, \ldots, \omega^{k-1}$ modulo m

for $i = 0, 1, \ldots, k-1$ **do**
  $\alpha_i \leftarrow \pi(\omega^i, m^2)$ ▷ $\alpha_i = f(\omega^i) \bmod m^2$
  $\beta_i \leftarrow \pi((1+m)\omega^i, m^2)$ ▷ $\beta_i = f((1+m)\omega^i) \bmod m^2$
  $\gamma_i \leftarrow (\beta_i - \alpha_i)/m$ using exact integer division
return $(\alpha_i \bmod m)_{0 \le i < k}$ and $(\gamma_i)_{0 \le i < k}$

Lemma 2.5. *If π is an MBB for $f \in \mathbb{Z}[x]$ with cost parameters B, L as in Definition 2.3, then Algorithm 1* MBBtoMDBB *correctly produces a single set of MDBB evaluations of f and f' and has bit complexity $O(Bk + (L+1)k \log m \operatorname{loglog} m)$.*

Proof. Any single term cx^e in f is mapped to $(c + cem)x^e$ in $f((1+m)x)$ modulo m^2. Therefore $f((1+m)x) - f(x) \equiv xmf'(x)$, and the γ_i's are indeed evaluations of $xf'(x)$ as required. The bit complexity comes from the cost of performing $O(k)$ MBB evaluations and ring operations in $\mathbb{Z}/m\mathbb{Z}$. ☐

To perform MDBB evaluations of a given polynomial f, we compute the derivative f' explicitly, reduce both f, f' modulo $\langle x^p - 1, m\rangle$, and then perform two multi-point evaluations. If $p < \#f$, then the reduced polynomial is dense and we use a DFT for the multi-point evaluations. Otherwise, we treat it as a sparse multi-point evaluation on points in geometric progression, which is a matrix-vector product with a transposed Vandermonde matrix (see e.g., [22, Fact 3.2]). The bit complexity is $\tilde{O}(\min(p \log m + s \operatorname{loglog} m, (s+k) \log p \log m)$, or more precisely:

Lemma 2.6. *There exists a procedure* ExplicitMDBB *that, given a polynomial f with $\mathsf{bitlen}_x(f) = s$, and MDBB inputs $p, \omega, m, k \in \mathbb{N}$, correctly computes the MDBB evaluations of f and f' in time*

$$O\Big(\min\big(p \log p \log m \operatorname{loglog} m + s \operatorname{loglog} m,\\ (s + k \log s) \log(sp) \operatorname{loglog} s \log m \operatorname{loglog} m\big)\Big).$$

Proof. First we reduce f modulo $\langle x^p - 1, m\rangle$ in $O(s \operatorname{loglog} m)$ according to Lemma 2.2, and in the same time separately reduce exponents modulo m and compute each coefficient-exponent product $(ce \bmod m)$ to derive the coefficients of xf'.

Treating the reduced polynomials as dense with size p, we can use Bluestein's algorithm for the DFT [10] in $O(p \log p)$ operations modulo m to get the $k \le p$ evaluations, i.e. $\omega \in \mathbb{Z}/m\mathbb{Z}$ is a p-PRU.

Alternatively, we may use a sparse approach: compute ω^{e_i} for each exponent in the reduced f (which are the same as in xf') in $O(\#f \log p)$ and then perform two $k \times \#f$ transposed Vandermonde matrix-vector products, for a total of $O\big((\#f + k) \log^2 \#f \operatorname{loglog} \#f\big)$ operations modulo m. Applying the bound on $\#f$ from Lemma 2.1 gives the stated complexity. ☐

Finally we consider computing an MDBB for the sum or product of two MDBB polynomials. The sum is straightforward; one can simply sum the corresponding evaluations. For the evaluations of the derivative of the product $h = fg$, we can use the product rule from elementary calculus:

$$xh'(x) = xf'(x) \cdot g(x) + xg'(x) \cdot f(x).$$

Lemma 2.7. *There exist procedures* SumMDBB *and* ProdMDBB *which, given two MDBBs π_1, π_2 for unknown $f_1, f_2 \in \mathbb{Z}[x]$, and any MDBB input tuple (p, ω, m, k), compute respectively MDBB evaluations of the sum $f_1 + f_2$ or product $f_1 f_2$. The cost is a single MDBB evaluation each of π_1 and π_2 plus (resp.) $O(k \log m)$ or $O(k \log m \operatorname{loglog} m)$ bit operations.*

2.4 Images of MDBBs

Our main subroutines for unbalanced interpolation need at each step to recover f and xf' modulo $\langle x^p - 1, m\rangle$, and to update such images with respect to a partially-recovered explicit polynomial f^*. The next (almost trivial) algorithm shows how to compute such modular images from MDBB evaluations.

Algorithm 2 MDBBImage(π, p, ω, m)

Input: MDBB π for unknown $f \in \mathbb{Z}[x]$; and $p, \omega, m \in \mathbb{N}$ s.t. ω is a p-PRU modulo m
Output: f and xf' modulo $\langle x^p - 1, m\rangle$

$(\alpha_i)_{0 \le i < k}, (\gamma_i)_{0 \le i < k} \leftarrow \pi(p, \omega, m, p)$
$\hat{\alpha}_0, \ldots, \hat{\alpha}_{p-1} \leftarrow \mathrm{IDFT}_\omega(\alpha_0, \ldots, \alpha_{p-1})$ over $\mathbb{Z}/m\mathbb{Z}$
$\hat{\gamma}_0, \ldots, \hat{\gamma}_{p-1} \leftarrow \mathrm{IDFT}_\omega(\gamma_0, \ldots, \gamma_{p-1})$ over $\mathbb{Z}/m\mathbb{Z}$
return $\sum_{i=0}^{t-1} \hat{\alpha}_i x^i$ and $\sum_{i=0}^{t-1} \hat{\gamma}_i x^i$

The running time is straightforward using Bluestein's algorithm [10] for the IDFTs.

Lemma 2.8. *Algorithm 2* MDBBImage *always produces the correct output using 2 calls to the MDBB with $k = p$ and an additional $O(p \log m \log p \operatorname{loglog} m)$ bit operations.*

3 Coefficient collisions and carries

Our general approach to find the terms of f with largest coefficients will be to exploit the fact that there cannot be too many of them. So we will take images of $f \bmod (x^p - 1)$, for small values of p, and extract the largest terms. The goal of this section is to define certain coefficient size boundaries, bound the probability of larger terms colliding modulo $x^p - 1$, and prove that the carries from smaller terms colliding will not affect the larger terms too much.

For the remainder, assume s, D, H are bounds (resp.) on the bit-length, degree, and height of an unknown nonzero $f \in \mathbb{Z}[x]$.

We define 4 categories of coefficients according to size:

Definition 3.1. *For any nonzero term cx^e of f:*

- *If $|c| < H^{1/6}$, we say c is* small
- *If $|c| \geq H^{1/6}$, we say c is* medium
- *If $|c| \geq \frac{1}{2}H^{13/30}$, we say c is* large
- *If $|c| \geq H^{1/2}$, we say c is* huge

Note that every huge term is also considered large, and every huge or large term is also considered medium. The number of terms in each category is also limited by s; for example, the number of huge terms is at most $2s/\log_2 H$.

We now provide conditions on a randomly chosen prime p to avoid collisions modulo $(x^p - 1)$. Formally, we say two nonzero terms $c_1 x^{e_1}$ and $c_2 x^{e_2}$ *collide* modulo p iff $e_1 \equiv e_2 \bmod p$. The following lemma follows the standard pattern from numerous previous results (see for example [21, Proposition 2.5]) relying on the density of primes in an interval [64].

Lemma 3.2. *Let $D, n > 0$ be given and $e_0, e_1, \ldots, e_n$ be $n+1$ distinct exponents with each $0 \leq e_i \leq D$. If $\lambda \geq \max(21, 3n\log_2 D)$, and if p a random prime from $(\lambda, 2\lambda)$, then with probability at least $\frac{1}{2}$, exponent e_0 does not collide with any of $e_1, \ldots, e_n$ modulo p.*

By repeatedly choosing primes, we can ensure that every term in f avoids collisions at least once. Note here that the set $a_1, \ldots, a_n$ need not be actual exponents in f.

Corollary 3.3. *Let $D, n > 0$ be given and $a_1, \ldots, a_n$ be distinct integers all satisfying $0 \leq a_i \leq D$, and let P be a list of primes in $(\lambda, 2\lambda)$, each chosen independently and uniformly at random. If $\lambda \geq \max(21, 3n\log_2 D)$ and $\#P \geq 2\log_2 s + 3$, then with probability at least $1 - 1/(4s\log_2 s)$, for every term $c_i x^{e_i}$ of f, there exists at least one $p \in P$ such that e_i does not collide with any $a_j \neq e_i$ modulo p.*

Proof. For any single term, by Lemma 3.2 the probability it is in collision with at least one of the a_j's for *every* $p \in P$ is at most $1/2^{2\log_2 s+3} < 1/(8s^2)$. Applying the bound on $\#f$ from Lemma 2.1, and taking the union bound, gives the stated result. □

Similar to prior works on sparse interpolation, our algorithm will recover the exponent e_0 of a term $c_0 x^{e_0}$ by division of coefficients between f and f' as produced by the CDBB. But this division is not exactly equal to $e_0 c_0 / c_0$ because we cannot avoid collisions with *small* terms in f. The following crucial lemma shows that such collisions still allow exact recovery of the exponent e_0 whenever the coefficient c_0 is large.

Lemma 3.4. *Suppose $\log_2 H \geq \max(61,\ 15\log_2 s,\ 6\log_2 D)$ and $p, m \in \mathbb{N}$ with $m \geq 4H^{7/6}$, and let $c_0 x^{e_0}$ be any large term of f as in Definition 3.1. If this term does not collide with any other medium term modulo p, then e_0 can be accurately recovered by approximate division of coefficients in f and f' modulo $\langle x^p - 1, m\rangle$.*

Proof. Let $\mathcal{S}$ be the set of indices of terms $c_i x^{e_i}$ in f that collide with e_0, so that $e_i \equiv e_0 \bmod p$ for all $i \in \mathcal{S}$. By assumption, all corresponding coefficients are "small" as in Definition 3.1, so each each $|c_i| < H^{1/6}$ for $i \in \mathcal{S}$.

First we establish a bound on the magnitude of the sum of any number of small coefficients. Using the fact that $\#\mathcal{S} \leq \#f \leq s$ and the assumption $\log_2 H \geq 15\log_2 s$, we have, for any set of small term indices $\mathcal{S}$,

$$\left|\textstyle\sum_{i\in\mathcal{S}} c_i\right| \leq \textstyle\sum_{i\in\mathcal{S}} |c_i| < \#\mathcal{S}H^{1/6} \leq sH^{1/6} \leq H^{7/30}. \tag{2}$$

Then the terms corresponding to $c_0 x^{e_0}$ in xf' and in $f \bmod \langle x^p - 1, m\rangle$ are respectively

$$\begin{aligned} c_0 e_0 + \textstyle\sum_{i\in\mathcal{S}} c_i e_i &\qquad \bmod m \\ c_0 + \textstyle\sum_{i\in\mathcal{S}} c_i &\qquad \bmod m \end{aligned}$$

Note that, from the bound (2) above, and the facts that $|c_0| \leq H$ and $e_i \leq D \leq H^{1/6}$, both of these are less than $2H^{7/6}$ in absolute value, and therefore modulo $m \geq 4H^{7/6}$ they can be recovered exactly as signed integers.

Our aim is to show that the quotient of these integers, rounded to the nearest integer, equals e_0 exactly, or equivalently that

$$\left| \frac{\sum_{i\in\mathcal{S}} c_i e_i - e_0 \sum_{i\in\mathcal{S}} c_i}{c_0 + \sum_{i\in\mathcal{S}} c_i} \right| < \frac{1}{2} \tag{3}$$

Using (2), along with the bound $D \le H^{1/6}$ from the statement of the lemma, the numerator magnitude from (3) is at most

$$\left|\sum_{i\in\mathcal{S}} c_i(e_i - e_0)\right| \le D \sum_{i\in\mathcal{S}} |c_i| < DH^{7/30} \le H^{2/5}.$$

A lower bound on the denominator from (3) can be obtained similarly, since $|c_0| \ge \frac{1}{2}H^{13/30}$ by the definition of a large term:

$$\left|c_0 + \sum_{i\in\mathcal{S}} c_i\right| \ge |c_0| - \sum_{i\in\mathcal{S}} |c_i| \ge H^{2/5}\left(\tfrac{1}{2}H^{1/30} - H^{-5/30}\right)$$

Because $\log_2 H \ge 61$, one can easily confirm that $\frac{1}{2}H^{1/30} - H^{-5/30} > 2$. Therefore the denominator is at least $2H^{2/5}$ and the inequality from (3) is satisfied. □

4 Recovering huge terms

This section presents the heart of our new algorithm for unbalanced interpolation, Algorithm 4 UInterpolateSlice, which recovers an explicit partial interpolant $f^* \in \mathbb{Z}[x]$ such that, with high probability, $\mathsf{height}(f - f^*) \le \sqrt{H}$.

Recall the formal notion of small/medium/large/huge terms from Definition 3.1. Slice interpolation works in two phases, first calling subroutine Algorithm 3 SupportSuperset to find (a superset of) the support of all large terms, and secondly using this set to reliably recover the huge terms' coefficients.

The first subroutine, Algorithm 3 SupportSuperset, works by sampling from MDBBImage multiple times, for sufficiently large exponent moduli p so that most of the medium (or larger) terms do not collide. From Lemma 3.4 in the previous section, even with some small-term collisions, we will be able to accurately recover the exponents of any large terms. Erroneous entries in $\mathcal{T}$ will likely result from medium/large/huge collisions, but that is acceptable as long as we don't "miss" any true exponents of large terms.

Algorithm 3 SupportSuperset(π, s, D, H)

Input: MDBB π for unknown $f \in \mathbb{Z}[x]$, bounds s, D, H on (resp.) the bit-length, degree, and height of f, satisfying the conditions of Lemma 3.4

Output: Set $\mathcal{T} \subset \mathbb{N}$ which contains the exponents of all terms in f with large (or huge) coefficients w.h.p.

$\lambda \leftarrow \max(21, 18s\log_2 D/\log_2 H)$; $\quad \mathcal{T} \leftarrow \{\}$
for $i = 1, 2, \ldots, \lceil 2\log_2 s\rceil + 3$ **do**
  Choose random prime $p \in (\lambda, 2\lambda)$
  Construct m, ω with $m \ge 4H^{7/6}$, and ω a p-PRU mod m
  $g, h \leftarrow$ MDBBImage(π, p, ω, m)
  for each corresponding terms ax^e, bx^e in g and h **do**
    if $|a| \ge \frac{1}{2}H^{13/30}$ and $0 \le \lfloor b/a\rceil \le D$ **then**
      if $\#\mathcal{T} < 60s(\log_2 s + 2)/(13\log_2 H)$ **then**
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{\lfloor b/a\rceil\}$
      else return $\{\}$
return $\mathcal{T}$

Lemma 4.1. *Algorithm 3* SupportSuperset *always produces a set $\mathcal{T} \subseteq \{0, \ldots, D\}$ with $\#\mathcal{T} \in O(s\log s/\log H)$; makes $O(\log s)$ MDBB calls with $p, k \in O(s\log D/\log H)$ and $\log m \in O(\log H)$; and uses $O\big(s\log D\log^2 s\,\mathrm{loglog}\,H\big)$ additional bit operations. With probability at least $1 - 1/(4s\log_2 s)$, $\mathcal{T}$ includes the exponents of every large term of f.*

Proof. Let cx^e be an arbitrary large term of f. By Definition 3.1, the number of medium terms in f is at most $6s/\log_2 H$. Applying Corollary 3.3, we see that with at least the stated probability, for each large term of f there exists an iteration i where that term does not collide with any other medium (or larger) term. And then by Lemma 3.4, the actual exponent of that term is accurately recovered from rounding division of coefficients on step i and added to $\mathcal{T}$. This proves the stated probabilistic correctness.

For the size of $\mathcal{T}$, observe that the bit-length of $f \bmod (x^p - 1)$ is at most s, so there are at most $\frac{30}{13}s/\log_2 H$ new integers added to $\mathcal{T}$ in each of the $O(\log s)$ iterations.

The bit complexity is dominated by the cost of the $O(\log s)$ calls to MDBBImage, which by the choices of p, m and from Lemma 2.8 gives the stated cost. □

We now use the set $\mathcal{T}$ to recover the coefficients of the huge terms (only). The same number of MDBBImage evaluations are performed, but with slightly larger p values to avoid collisions with all exponents (whether correct or not) in T. Because $\mathcal{T}$ contains the exponents of all large terms, and any errors from medium and small terms cannot propagate up to the "huge" level, every huge term obtained on this step is accurately added to the result.

Algorithm 4 UINTERPOLATESLICE(π, s, D, H)

Input: MDBB π for unknown $f \in \mathbb{Z}[x]$, and bounds s, D, H on (resp.) the bit-length, degree, and height of f, satisfying the conditions of Lemma 3.4

Output: $f^* \in \mathbb{Z}[x]$ such that w.h.p. $\mathsf{height}(f - f^*) \le \sqrt{H}$

$\mathcal{T} \leftarrow$ SUPPORTSUPERSET(π, s, D, H)
$\lambda \leftarrow \max(21, 3 \cdot \#\mathcal{T} \cdot \log_2 D)$; $\quad f^* \leftarrow 0$
for $i = 1, 2, \ldots, \lceil 2 \log_2 s \rceil + 3$ **do**
    Choose random prime $p \in (\lambda, 2\lambda)$
    Construct m, ω with $m \ge 2H$ and ω a p-PRU mod m
    $\pi^* \leftarrow$ SUMMDBB$(\pi,$ EXPLICITMDBB$(-f^*))$
    $g.h \leftarrow$ MDBBIMAGE(π^*, p, ω, m)
    $E_j \leftarrow \{\}$ for $0 \le j < p$
    for each $a \in \mathcal{T}$ **do**
        $E_{a \operatorname{rem} p} \leftarrow E_{a \operatorname{rem} p} \cup \{a\}$
    for each term cx^i in g **do**
        if $|c| \ge \frac{1}{2} H^{1/2}$ and $\#E_i = 1$ **then**
            $e \leftarrow$ the unique element of E_i
            if f^* does not contain a term with x^e **then**
                $f^* \leftarrow f^* + cx^e$
                if $\mathsf{bitlen}_x(f^*) > s$ **then return** 0
return f^*

Lemma 4.2. *Algorithm 4* UINTERPOLATESLICE *always produces a polynomial $f^* \in \mathbb{Z}[x]$ with bit-length at most s, makes $O(\log s)$ MDBB calls with $p, k \in O(s \log D \log s / \log H)$ and $\log m \in O(\log H)$, and uses $O\big(s \log D \log^3 s \operatorname{loglog} H\big)$ additional bit operations. It returns $f^* \in \mathbb{Z}[x]$ such that, with probability at least $1 - 1/(2s \log_2 s)$, the height of $(f - f^*)$ is less than $\sqrt{H}$.*

Proof. Consider any set of non-large terms. By the upper bound on the total number of terms from Lemma 2.1, the condition $\log_2 H \ge 15 \log_2 s$ from Lemma 3.4, and from Definition 3.1, the sum of these non-large terms is at most

$$\sum |c_i| < \tfrac{1}{2} s H^{13/30} \le \tfrac{1}{2}\sqrt{H}. \tag{4}$$

For the probabilistic correctness, suppose that (i) the call to Algorithm 3 SUPPORTSUPERSET correctly returns a set $\mathcal{T}$ which contains the exponents of all large terms, and (ii) each large or huge term does not collide with any of the other exponents in $\mathcal{T}$ for at least one of the chosen p's in the outer for loop. Taking a union bound with Lemma 4.1 and Corollary 3.3, both (i) and (ii) are true with probability at least $1 - 1/(2s \log_2 s)$, as required. We proceed to prove the algorithm returns f^* correctly under these two assumptions.

Consider any huge term $c_0 x^{e_0}$ in f, i.e., with $|c_0| \ge \sqrt{H}$. From (4), the resulting collision with any number of non-large terms will still result in a coefficient larger than $\frac{1}{2}\sqrt{H}$ in absolute value, so this term will be added to f^*.

Conversely, consider any term cx^e which is added to f^* in the algorithm. By assumption that $\mathcal{T}$ is correct, c must be the sum of exactly one large (or huge) term plus some number of non-large coefficients. By (4) again, the coefficient of this term in $f - f^*$ is at most the size of the colliding non-large coefficients' sum, which is less than $\frac{1}{2}\sqrt{H}$.

Because f^* contains some term corresponding to each huge term in f, and every term added to f^* reduces the height of that coefficient in $f - f^*$ below $\sqrt{H}$, we conclude that $\mathsf{height}(f - f^*) < \sqrt{H}$ as required.

For the complexity analysis, we have $\#\mathcal{T} \in O(s \log s / \log H)$ from Lemma 4.1, which means $p \in O(s \log D \log s / \log H)$. As before, the cost of the evaluations MDBBIMAGE dominates the complexity, which comes from Lemma 2.8. $\square$

In most cases, UINTERPOLATESLICE will be called with a MDBB π which is a sum of an actual unknown black-box polynomial minus an explicit partial interpolant $f^* \in \mathbb{Z}[x]$ recovered so far, via EXPLICITMDBB and SUMMDBB. The next corollary shows that this does not affect the bit complexity as long as f^* has bit-length also bounded by s.

Corollary 4.3. *Let π be any MDBB and $f^* \in \mathbb{Z}[x]$ such that $\mathsf{bitlen}_x(f^*) \le s$. Then calling* UINTERPOLATESLICE(π^*, s, D, H) *with $\pi^* =$* SUMMDBB$(\pi,$ EXPLICITMDBB$(-f^*))$ *has the same asymptotic cost as* UINTERPOLATESLICE(π, s, D, H).

Proof. From Lemma 2.6 and Lemma 2.7, the cost of a single MDBB evaluation of π^* with p, ω, m, k is a single call to π plus

$$O(s \operatorname{loglog} m + p \log p \log m \operatorname{loglog} m)$$

bit operations. When $k \in O(p)$, $\log m \in O(\log H)$, and $p \log m \in O(s \log D \log s)$, this simplifies to $O\big(s \log D \log^2 s \operatorname{loglog} H\big)$.

Then from Lemma 4.2, the total cost of calls to π^* in UInterpolateSlice is the same calls to π, plus an additional bit cost which is always bounded by the bit-cost of UInterpolateSlice itself. ☐

5 Unbalanced interpolation algorithm

Our overall algorithm will call Algorithm 4 UInterpolateSlice repeatedly starting with $H = s$, until the height decreases such that the conditions of Lemma 3.4 are no longer satisfied, and at this point switches to a *balanced* sparse interpolation algorithm for the base case.

For that, we use Algorithm 2 Interpolate_mbb from [23], which solves the problem in $\tilde{O}(T \log D + T \log H)$ time. In the base case when Lemma 3.4 no longer applies, we have $\log H \in O(\log D + \log s)$, so this complexity becomes simply $\tilde{O}(s \log D)$ as required.

The only slight change from [23] is that Interpolate_mbb should take a MDBB for us rather than a usual MBB as the black-box evaluation routine for unknown $f \in \mathbb{Z}[x]$. But the MDBB model already fits the case perfectly; the algorithm as written in [23] is using the MBB only to evaluate f at consecutive powers of a p-PRU, which is precisely what a MDBB does already.

This discussion is summarized in the following claim, where the precise complexity bound comes from Theorem 3.18 in [59].

Fact 5.1. *Let π be a MDBB for unknown polynomial $f \in \mathbb{Z}[x]$, and bounds D, T, H on (resp.) the degree, sparsity, and height of f. Write $n = T(\log_2 D + \log_2 H)$. Then there exists a Monte Carlo algorithm* Interpolate_mbb(π, D, T, H) *that always produces a polynomial f^* within the given bounds using $O(\log T)$ evaluations of the MDBB and $O\big(n \log^3 n \log^2 T (\operatorname{loglog} n)^2\big)$ bit operations. In each MDBB call, we have $p \in O(n \log T)$ and $\log m \in O(\log(DH))$, and the sum of k values over all $O(\log T)$ calls is $O(T)$. If the unknown f actually satisfies the given bounds D, T, H, then with probability at least $\frac{2}{3}$, the returned polynomial f^* equals f.*

As mentioned already in [23] at the end of the proof of Theorem 3.4, to decrease the failure probability arbitrarily low to some $\epsilon > 0$, it just requires iterating the call at least $48 \ln \frac{1}{\epsilon}$ times and returning the majority result.

We are now ready to present the main algorithm, UInterpolate, which recovers an unknown polynomial with possibly unbalanced coefficients from a given MDBB and bounds only on the degree and total bit-length.

Algorithm 5 UInterpolate(π, s, D)

Input: MDBB π for unknown $f \in \mathbb{Z}[x]$, and bounds s, D on (resp.) the bit-length and degree of f
Output: $f^* \in \mathbb{Z}[x]$ such that $f = f^*$ w.h.p.

```
H ← 2^s;   f* ← 0;   π* ← π
while H ≥ max(61, 15 log_2 s, 6 log_2 D) do
    f* ← f* + UInterpolateSlice(π*, s, D, H)
    if bitlen_x(f*) > s then return 0
    π* ← SumMDBB(π, ExplicitMDBB(−f*))
    H ← √H
R ← empty list
for i = 1, 2, ..., ⌈48 ln(2s)⌉ do
    f** ← Interpolate_mbb(π*, 2s/ log_2 s, D, H)
    Append f** to R
if R has a majority element f** and bitlen_x(f**) ≤ s then
    return f**
else return 0
```

The precise cost estimate is given in the following theorem.

Theorem 5.2. *Algorithm 5* UInterpolate *always returns a polynomial with bit-length at most s and uses $O\big(\log^2 s\big)$ MDBB calls plus an additional*

$$O\big(s \log D \log^5 s (\operatorname{loglog} s)^2\big)$$

bit operations. Each MDBB call has $p \in O(s \log D)$, $\log m \in O(s)$, but $(p+k) \log m \in O(s \log D \log s)$; furthermore, the sum of k over all MDBB calls is $O(s \log D \log s)$. If the unknown f actually satisfies bit-length and degree bounds s, D, then with probability at least $1 - 1/s$, the returned f^{**} equals f.

Proof. First observe that due to the checks in the algorithm, f^* and the returned f^{**} always have bit-length at most s and degree at most D.

For the bit complexity, first notice that at the end of the first loop we have $\log H \in O(\log D + \log s)$. And then because the bit-length s could never be more than $D \log_2 H$, we see that in fact $\log s + \log H \in O(\log D)$ at this point in the algorithm. Applying Fact 5.1, the total bit-cost of the $O(\log s)$

calls to INTERPOLATE_MBB dominates the (non-black box) bit complexity and gives the stated bound.

The black box calls are dominated instead by the first loop, whose cost is verified by applying Lemma 4.2 and observing that, since $\log H$ is exponentially decreasing from s down to at least $\log s$, the sum of k values in all calls to *Lemma* 4.2 is as claimed in the theorem statement.

For probabilistic correctness, the union bound over all calls to UINTERPOLATESLICE, Lemma 4.2 gives at most a $\frac{1}{2s}$ chance that *any* of those steps does not return the correct result. In the subsequent calls to INTERPOLATE_MBB, setting $T = 2s/\log_2 s$ is valid by Lemma 2.1. From the preceding discussion, we have the same probability of failure of the majority vote approach at the end. Therefore the total failure probability is at most $1/s$ as claimed. ☐

Similarly to Corollary 4.3, we first clarify that adding an explicit polynomial to the black box π does not affect the bit complexity. This is clearly the case as the MDBB π^* in the algorithm in fact already includes a sum with an explicit polynomial whose bit-length is at most s.

Corollary 5.3. *Let π be any MDBB and $f^* \in \mathbb{Z}[x]$ such that $\mathsf{bitlen}_x(f^*) \le s$. Then calling* UINTERPOLATE(π^*, s, D) *with* $\pi^* =$ SUMMDBB$(\pi,$ EXPLICITMDBB$(-f^*))$ *has the same asymptotic cost as* UINTERPOLATE(π, s, D).

The following corollary gives the concrete cost to run UINTERPOLATE when the input is a modular black box (MBB). The stated bit complexity, which provides the clearest comparison to prior works in the literature, can be even more simply stated as $\tilde{O}((B + L)s \log D)$.

Corollary 5.4. *If π is a MBB for unknown $f \in \mathbb{Z}[x]$ with cost parameters B, L as in Definition 2.3, and if f has bit-length at most s and degree at most D, then* UINTERPOLATE(MBBTOMDBB$(\pi), s, D)$ *will correctly return f with probability at least $1 - \frac{1}{s}$ and with total bit cost*

$$O\big(\big(B + L\log^3 s + \log^4 s(\mathrm{loglog}\, s)^2\big)\, s \log D \log s\big)$$

Proof. Follows directly from Theorem 5.2 and Lemma 2.5. ☐

6 Unbalanced multiplication

In this section we will show how to multiply two polynomials $f, g \in \mathbb{Z}[x]$ with degree less than D total input bit-length ℓ, in time $\tilde{O}(s \log D)$, where $s = \ell + \mathsf{bitlen}_x(fg)$ is the total input and output bit-length.

As motivated in the introduction, this is the first algorithm which is sub-quadratic in s for unbalanced coefficients, *even for dense polynomials*. For sparse integer polynomial multiplication, the state of the art comes from [59, Algorithm 20], based on the (balanced) sparse interpolation of [22], with bit complexity $\tilde{O}(t \log H + t \log D)$, which is $\tilde{O}(s^2 + s \log D)$ in terms of the total bit-length s. For the remainder we address only the case of sparse polynomials as the dense case works in exactly the same way.

Our general approach is to construct a MDBB for the product fg and then use Algorithm 5 UINTERPOLATE to interpolate it with complexity dependent on (unbalanced) total bit-length of the inputs and output. But we will show that the pessimistic bound on $\mathsf{bitlen}_x(fg)$ is quadratic in the input sizes, which means that a naïve application of UINTERPOLATE would still not be quasi-linear time in the actual size of fg.

Instead, we develop an efficient (probabilistic) verification method, which allows us to try interpolating fg with smaller bounds on the output bit-length, then repeatedly doubling the *optimistic* bound until the verification passes.

6.1 Unbalanced product bit-length

Suppose $f, g \in \mathbb{Z}[x]$ have t nonzero terms and total bit-length ℓ. The number of terms in $h = fg$ is at most t^2 but the total bit-length of h remains $O(ts)$, instead of $O\big(t^2(s + \log t)\big)$. Indeed, when no collision occurs, every entry of f and g will contribute exactly t times to the bit length of h. Collision among non-zero terms can only decrease the sum of the bit-length of the t^2 coefficient products. Hence the bit-length of h is bounded by $2t$ times the bit-length of the input.

Lemma 6.1. *If* $\mathsf{bitlen}_x(f) + \mathsf{bitlen}_x(g) \le \ell$, *then* $\mathsf{bitlen}_x(fg) \le 4s^2/\log s$ *and* $\mathsf{height}(fg) \le 4^s \cdot s$

6.2 Unbalanced product verification

Our unbalanced interpolation algorithm UINTERPOLATE takes a bit-length bound and always returns a polynomial with that size, but gives (obviously) no correctness guarantee if that bound was too small. To avoid relying on the pessimistic upper bound on $\mathsf{bitlen}_x(fg)$, we provide an efficient verification method. While deterministic verification seems to be a difficult task, a straightforward analysis of the probabilistic algorithm proposed in [21, Section 5.2] is already satisfactory to reach the needed complexity:

Lemma 6.2. *There is a one-sided error randomized algorithm, called* VerifProd (f, g, h, ϵ)*, that, given* f*,* g*,* $h \in \mathbb{Z}[x]$ *with degrees and bit-lengths at most* D *and* s*, and failure probability* $\epsilon \in (0, 1)$*, tests whether* $h = f \times g$ *with false-positive probability at most* ϵ*, using* $O(s \log \frac{s}{\epsilon} \operatorname{loglog} \frac{s}{\epsilon} + s \log \frac{D}{\epsilon} \operatorname{loglog} \frac{D}{\epsilon})$ *bit operations.*

Proof. The approach is exactly the same as in [21, Section 5.2]. We choose random primes p, q, evaluate $\delta = (h - fg) \bmod \langle x^p - 1, q\rangle$ at a random point $\alpha \in \mathbb{F}_q$, and check whether the result is zero. If $h = fg$, then $\delta = 0$ and the check always succeeds. Otherwise, from Lemma 6.1 and Lemma 2.2 one may show that taking $p = \Omega(\frac{1}{\epsilon} \max(s, D))$ and $q = O(\frac{1}{\epsilon} s)$ is enough to guarantee that δ is zero with a probability less than $\frac{2}{3}\epsilon$. Using [21, Theorem 3.1] one can show that evaluating $\delta(\alpha) = 0$ costs the claimed complexity and that α is a root of δ with probability less than $\frac{1}{3}\epsilon$. □

6.3 Adaptative unbalanced polynomial multiplication

One can easily construct an MDBB π for the product of two polynomials $f, g \in \mathbb{Z}[x]$ by composing the procedures ProdMDBB and ExplicitMDBB given in Section 2.3. Feeding this MDBB into our unbalanced interpolation algorithm UInterpolate with a correct bit-length bound yields a Monte-Carlo algorithm for computing the product fg.

We now provide an Atlantic-city algorithm for that computation where the correct bit-length is discovered through our probabilistic verification of the computed result.

Algorithm 6 UnbalancedProd

Input: $f, g \in \mathbb{Z}[x]$ of degree at most D and bit-lengths ℓ;
Output: $h \in \mathbb{Z}[x]$ such that $h = f \times g$ w.h.p.

```
h ← 0; s ← ℓ
s_max ← 2ℓ + 4ℓ²/log₂(ℓ); ε ← (s_max(8ℓ + 4))⁻¹
π ← ProdMDBB(ExplicitMDBB(f), ExplicitMDBB(g))
while s < 2s_max and not VerifProd (f, g, h, ε) do
    h ← UInterpolate (π, s, 2D)
    s ← 2s
return h
```

Theorem 6.3. *Let two polynomials* $f, g \in \mathbb{Z}[x]$ *of degree at most* D *and bit-length* ℓ*. Algorithm* UnbalancedProd *is an Atlantic-City algorithm that returns the polynomial* $h = f \times g$ *with probability at least* $1 - \frac{1}{s}$*, using an expected total of* $O(s \log D \log^5 s (\log \log s)^2)$ *bit operations where* $s = O(\ell + \mathsf{bitlen}_x(h))$*. This is the actual complexity with probability at least* $1 - \frac{1}{s}$*. In the worst-case it requires* $O(s^2 \log D \log^4 s (\log \log s)^2)$ *bit operations.*

Proof. First one should note that according to Lemma 6.1, s_{max} bounds $\mathsf{bitlen}_x(fg) + \mathsf{bitlen}_x(f) + \mathsf{bitlen}_x(g)$. So taking s at most $2s_{max} - 1$ is enough to guarantee at least one iteration is done with a correct bit-length. This iteration will fail with the same probability as UInterpolate, which is $< 1/s$. Let us assume that UInterpolate never produces a correct answer even when s correctly bounds the bit-length of fg. Since the algorithm requires at most $\lceil \log_2(2s_{max}/l)\rceil < 8\ell + 4$ iterations before terminating, the probability that at least one verification test fails is $\leq (8\ell + 2)\epsilon = 1/s_{max}$. Hence, the probability of a wrong answer is $< 1/s_{max}$.

For the running time we note that all polynomials returned by UInterpolate have a bit-length smaller than s even for wrong results. Therefore, each iteration of UInterpolate has a cost of $O(s \log D \log^5 s (\log \log s)^2)$ bit operations, plus the MDBB evaluations.

Evaluating the MDBB π entails one ProdMDBB and two ExplicitMDBBs on polynomials whose bit-length is at most ℓ. From Lemma 2.6 and Lemma 2.7 this is dominated by the cost of ExplicitMDBB, and then by Corollary 5.3 the cost of all MDBB evaluations does not dominate the bit-cost of UInterpolate itself.

The bit cost from UInterpolate dominates the complexity of the corresponding call to VerifProd because $\log \frac{1}{\epsilon} = O(\log \ell) = O(\log s)$. Since the value of s is doubling at each iteration, the global running time is dominated by the cost of the last one, which is done with $s = O(\mathsf{bitlen}_x(fg))$.

It remains to see if there are *extra* iterations after reaching a bound $s_h \geq \mathsf{bitlen}_x(fg)$. There are $\gamma < 8\ell + 4$ extra iterations if both UInterpolate and VerifProd fail γ times in a row. Even ignoring the probability of failure of VerifProd, this cannot happen with probability more than $(\frac{1}{s_h})^\gamma$. Let $C(s)$ be the cost of one iteration. Since it is quasi-linear in s, the i-th extra iteration costs $O(2^i C(s_h))$. Hence the expected cost is $O\left(\sum_{i=1}^{\gamma} 2^i C(s_h)(\frac{1}{s_h})^i\right) = O(C(s_h))$. □

We remark that one might hope to simplify this approach by using early termination instead of (probabilistic) verification, stopping the loop as soon as several interpolations in a row lead to the same polynomial. But without any guarantees on the outputs of UInterpolate when given a too-small bit-length bound s, there is no way to analyze the early termination strategy.

7 Open question: Unbalanced exponents

A natural question, which we leave open, is whether a soft-optimal algorithm for integer polynomial interpolation with unbalanced coefficients *and large, unbalanced exponents* is possible. That is, we have shown an algorithm that runs in $\tilde{O}(s \log D)$ time; is even better $\tilde{O}(s)$ possible?

We will outline two possible approaches towards this improvement, and briefly explain why both do not seem to work with our current techniques.

At a high level, our algorithm as presented in UINTERPOLATE and its subroutine UINTERPOLATESLICE is *top-down* in nature: it first retrieves only the largest terms by making relatively few evaluations at very high precision. Then these are added to the result, and we proceed to find more terms at the next level, using a greater number of lower-precision evaluations.

The complexity challenge comes at the point of performing evaluations; in particular we need some $p = \tilde{O}(k \log D)$ evaluations at each step in order to retrieve the k largest terms. In prior work such as [23], this extra $\log D$ factor in evaluations is avoided by using Prony's method as in Ben-Or and Tiwari [7], but that doesn't work in this context because *we do not actually have a k-sparse polynomial*! Instead we have a polynomial with many more than k terms, from which we only want to extract the k largest coefficients. So the Prony method (in exact arithmetic) cannot be used, and we have to resort to a dense "over-sampling" approach and incur the extra $\log D$ factor in cost.

(Note that the numerical community has developed the sparse FFT and related methods [40, 61, 60] which do tackle this problem of retrieving only the largest coefficients via evaluation/interpolation. But numerical evaluation unfortunately does not fit with our update step to cancel out the large terms once recovered.)

A completely opposite approach to ours would be to instead start by recovering all of the *small* terms of the unknown f, then cancel these and iterate, at each step retrieving fewer terms, at higher size and evaluation precision.

This *bottom-up* approach seems at first glance to work very well; in particular it solves the aforementioned issue of sparsity because, at the point of attempting to recover some k large terms, the difference polynomial $f - f^*$ really is k-sparse, and only $O(k)$ evaluations would be needed at each step.

But unfortunately there is a subtle and seemingly devastating obstacle to this approach, which is the cost of evaluating the already-recovered terms f^* at each step. For example, consider an extreme case where f has roughly $s/\log s$ very small terms with $O(\log s)$ bits each, and then a constant number of very large terms with $O(s)$ bits each. After recovering all the small terms, the explicit polynomial f^* has sparsity $O(s)$ (but small height), and now to find the remaining few large terms, we need to evaluate this polynomial to very high precision, that is, with a s-bit modulus.

But to our knowledge there is no known technique to evaluate a large low-height polynomial at just a few points to very high precision. Instead, the standard approach to evaluate $f^*(\omega) \bmod m$ with an s-bit modulus m would require first computing $\omega^{e_i} \bmod m$ for each exponent e_i in f^*, which already results in a bit-cost of $\tilde{O}(s^2)$, obviously not quasi-linear time.

Of course there could be an entirely different approach to achieving $\tilde{O}(s)$ runtime, but we thought it would be prudent to share these two (failed) attempts of ours so far, in the hopes of provoking new ideas and future work.

References

[1] Noga Alon and Yishay Mansour. 1995. ϵ-Discrepancy Sets and Their Application for Interpolation of Sparse Polynomials. *Inform. Process. Lett.* 54, 6 (1995), 337–342. https://doi.org/10.1016/0020-0190(95)00032-8

[2] Andrew Arnold. 2016. *Sparse polynomial interpolation and testing.* Ph. D. Dissertation. University of Waterloo.

[3] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. 2014. Sparse interpolation over finite fields via low-order roots of unity. In *ISSAC'14*. 27–34. https://doi.org/10.1145/2608628.2608671

[4] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. 2016. Faster sparse multivariate polynomial interpolation of straight-line programs. *J. Symb. Comput.* 75 (2016), 4–24. https://doi.org/10.1016/j.jsc.2015.11.005

[5] Arnold Arnold and Daniel S. Roche. 2014. Multivariate sparse interpolation using randomized Kronecker substitutions. In *ISSAC'14*. 35–42. https://doi.org/10.1145/2608628.2608674

[6] Andrew Arnold and Daniel S. Roche. 2015. Output-sensitive algorithms for sumset and sparse polynomial multiplication. In *ISSAC'15*. 29–36. https://doi.org/10.1145/2755996.2756653

[7] Michael Ben-Or and Prasoon Tiwari. 1988. A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. In *STOC'88*. 301–309. https://doi.org/10.1145/62212.62241

[8] Stavros Birmpilis, George Labahn, and Arne Storjohann. 2023. A fast algorithm for computing the Smith normal form with multipliers for a nonsingular integer matrix. *Journal of Symbolic Computation* 116 (2023), 146–182. https://doi.org/10.1016/j.jsc.2022.09.002

[9] Markus Bläser and Gorav Jindal. 2014. A New Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC '14)*. Association for Computing Machinery, New York, NY, USA, 51–58. https://doi.org/10.1145/2608628.2608648

[10] Leo Bluestein. 1970. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics* 18, 4 (1970), 451–455. https://doi.org/10.1109/TAU.1970.1162132

[11] Marco Bodrato and Alberto Zanoni. 2020. Univariate Polynomials with Long Unbalanced Coefficients as Bivariate Balanced Ones: A Toom-Cook Multiplication Approach. In *Computer Algebra in Scientific Computing*, François Boulier, Matthew England, Timur M. Sadykov, and Evgenii V. Vorozhtsov (Eds.). 91–107. https://doi.org/10.1007/978-3-030-60026-6_6

[12] David G. Cantor and Erich Kaltofen. 1991. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica* 28 (1991), 693–701.

[13] Changbo Chen, Svyatoslav Covanov, Farnam Mansouri, Marc Moreno Maza, Ning Xie, and Yuzhen Xie. 2016. Parallel Integer Polynomial Multiplication. In *18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 72–80. https://doi.org/10.1109/SYNASC.2016.024

[14] Richard Cole and Ramesh Hariharan. 2002. Verifying candidate matches in sparse and wildcard matching. In *STOC'02*. 592–601. https://doi.org/10.1145/509907.509992

[15] Stephen A. Cook. 1966. *On the Minimum Computation Time of Functions*. Ph. D. Dissertation. Harvard.

[16] Annie Cuyt and Wen-shin Lee. 2011. Sparse Interpolation of Multivariate Rational Functions. *Theoretical Computer Science* 412, 16 (April 2011), 1445–1456. https://doi.org/10.1016/j.tcs.2010.11.050

[17] Richard J Fateman. 2010. Can You Save Time in Multiplying Polynomials by Encoding Them as Integers? (2010). https://people.eecs.berkeley.edu/~fateman/papers/polysbyGMP.pdf.

[18] Sanchit Garg and Éric Schost. 2009. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.* 410, 27 (2009), 2659–2662. https://doi.org/10.1016/j.tcs.2009.03.030

[19] Mark Giesbrecht and Daniel S. Roche. 2011. Diversification Improves Interpolation. In *ISSAC'11*. 123–130. https://doi.org/10.1145/1993886.1993909

[20] Pascal Giorgi, Bruno Grenet, and Armelle Perret du Cray. 2020. Essentially optimal sparse polynomial multiplication. In *ISSAC'20*. 202–209. https://doi.org/10.1145/3373207.3404026

[21] Pascal Giorgi, Bruno Grenet, and Armelle Perret du Cray. 2023. Polynomial Modular Product Verification and Its Implications. *Journal of Symbolic Computation* 116 (2023), 98–129. https://doi.org/10.1016/j.jsc.2022.08.011

[22] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. 2022. Sparse polynomial interpolation and division in soft-linear time. In *Proceedings of the 2022 international symposium on symbolic and algebraic computation (ISSAC'22)*. ACM, 459–468. https://doi.org/10.1145/3476446.3536173 arXiv:2202.08106

[23] Pascal Giorgi, Bruno Grenet, Armelle Perret du Cray, and Daniel S. Roche. 2022. Sparse Polynomial Interpolation and Division in Soft-linear Time. In *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation* (Villeneuve-d'Ascq, France) *(ISSAC '22)*. Association for Computing Machinery, New York, NY, USA, 459–468. https://doi.org/10.1145/3476446.3536173

[24] Dima Yu. Grigoriev, Marek Karpinski, and Michael F. Singer. 1990. Fast Parallel Algorithms for Sparse Multivariate Polynomial Interpolation over Finite Fields. *SIAM J. Comput.* 19, 6 (Dec. 1990), 1059–1063. https://doi.org/10.1137/0219073

[25] David Harvey and Joris van der Hoeven. 2021. Integer Multiplication in Time $O(n \log n)$. *Annals of Mathematics* 193, 2 (2021). https://doi.org/10.4007/annals.2021.193.2.4

[26] David Harvey and Joris van der Hoeven. 2022. Polynomial Multiplication over Finite Fields in Time $O(n \log n)$. *J. ACM* 69, 2 (2022), 12:1–12:40. https://doi.org/10.1145/3505584

[27] David Harvey, Joris van der Hoeven, and Grégoire Lecerf. 2017. Faster Polynomial Multiplication over Finite Fields. *J. ACM* 63, 6 (jan 2017). https://doi.org/10.1145/3005344

[28] Joris van der Hoeven, Romain Lebreton, and Éric Schost. 2013. Structured FFT and TFT: Symmetric and Lattice Polynomials. In *ISSAC'13*. ACM, 355–362. https://doi.org/10.1145/2465506.2465526

[29] Joris van der Hoeven and Grégoire Lecerf. 2012. On the Complexity of Multivariate Blockwise Polynomial Multiplication. In *ISSAC'12*. 211–218. https://doi.org/10.1145/2442829.2442861

[30] Joris van der Hoeven and Grégoire Lecerf. 2014. Sparse Polynomial Interpolation in Practice. *ACM Commun. Comput. Algebra* 48, 3/4 (2014), 187–191. https://doi.org/10.1145/2733693.2733721

[31] Joris van der Hoeven and Grégoire Lecerf. 2019. Sparse polynomial interpolation. Exploring fast heuristic algorithms over finite fields. (2019). https://hal.archives-ouvertes.fr/hal-02382117 preprint.

[32] Joris van der Hoeven and Grégoire Lecerf. 2021. On Sparse Interpolation of Rational Functions and Gcds. *ACM Communications in Computer Algebra* 55, 1 (May 2021), 1–12. https://doi.org/10.1145/3466895.3466896

[33] Joris van der Hoeven and Grégoire Lecerf. 2023. Fast interpolation of sparse multivariate polynomials. arXiv:2312.17664.

[34] Ming-Deh A Huang and Ashwin J Rao. 1999. Interpolation of Sparse Multivariate Polynomials over Large Finite Fields with Applications. *Journal of Algorithms* 33, 2 (Nov. 1999), 204–228. https://doi.org/10.1006/jagm.1999.1045

[35] Qiao-Long Huang. 2019. Sparse Polynomial Interpolation over Fields with Large or Zero Characteristic. In *ISSAC'19*. 219–226. https://doi.org/10.1145/3326229.3326250

[36] Qiao-Long Huang. 2021. Sparse Polynomial Interpolation Based on Diversification. *Science China Mathematics* (April 2021). https://doi.org/10.1007/s11425-020-1791-5

[37] Qiao-Long Huang. 2023. Sparse Polynomial Interpolation Based on Derivatives. *Journal of Symbolic Computation* 114 (Jan. 2023), 359–375. https://doi.org/10.1016/j.jsc.2022.06.002

[38] Qiao-Long Huang and Xiao-Shan Gao. 2019. Revisit Sparse Polynomial Interpolation Based on Randomized Kronecker Substitution. In *CASC'19*. Springer, 215–235. https://doi.org/10.1007/978-3-030-26831-2_15

[39] Qiao-Long Huang and Xiao-Shan Gao. 2020. Faster interpolation algorithms for sparse multivariate polynomials given by straight-line programs. *J. Symb. Comput.* 101 (2020), 367–386. https://doi.org/10.1016/j.jsc.2019.10.005

[40] Piotr Indyk, Michael Kapralov, and Eric Price. [n. d.]. *(Nearly) Sample-Optimal Sparse Fourier Transform*. 480–499. https://doi.org/10.1137/1.9781611973402.36

[41] Seyed Mohammad Mahdi Javadi and Michael Monagan. 2010. Parallel Sparse Polynomial Interpolation over Finite Fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation (PASCO '10)*. Association for Computing Machinery, New York, NY, USA, 160–168. https://doi.org/10.1145/1837210.1837233

[42] Stephen C. Johnson. 1974. Sparse polynomial arithmetic. *ACM SIGSAM Bulletin* 8, 3 (1974), 63–71. https://doi.org/10.1145/1086837.1086847

[43] Erich Kaltofen, Yagati N. Lakshman, and J.-M. Wiley. 1990. Modular Rational Sparse Multivariate Polynomial Interpolation. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC '90)*. Association for Computing Machinery, New York, NY, USA, 135–139. https://doi.org/10.1145/96877.96912

[44] Erich Kaltofen and Wen-shin Lee. 2003. Early termination in sparse interpolation algorithms. *J. Symb. Comput.* 36, 3 (2003), 365–400. https://doi.org/10.1016/S0747-7171(03)00088-9

[45] Erich Kaltofen and Lakshman Yagati. 1989. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC'88*, P. Gianni (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 467–474.

[46] Erich Kaltofen and Zhengfeng Yang. 2007. On Exact and Approximate Interpolation of Sparse Rational Functions. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*. Association for Computing Machinery, New York, NY, USA, 203–210. https://doi.org/10.1145/1277548.1277577

[47] Erich L. Kaltofen. 2010. Fifteen years after DSC and WLSS2: What parallel computations I do today [invited lecture at PASCO 2010]. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation* (Grenoble, France) *(PASCO '10)*. ACM, 10–17. https://doi.org/10.1145/1837210.1837213

[48] Erich L. Kaltofen and Michael Nehring. 2011. Supersparse Black Box Rational Function Interpolation. In *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation (ISSAC '11)*. Association for Computing Machinery, New York, NY, USA, 177–186. https://doi.org/10.1145/1993886.1993916

[49] Anatolii Karatsuba and Yu Ofman. 1962. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady* 7 (12 1962), 595.

[50] Leopold Kronecker. 1882. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal für die reine und angewandte Mathematik* 92 (1882), 1–122.

[51] George Labahn, Vincent Neiger, Thi Xuan Vu, and Wei Zhou. 2022. Rank-Sensitive Computation of the Rank Profile of a Polynomial Matrix. In *Proceedings of the 2022 International Symposium on Symbolic and Algebraic Computation* (Villeneuve-d'Ascq, France) *(ISSAC '22)*. Association for Computing Machinery, New York, NY, USA, 351–360. https://doi.org/10.1145/3476446.3535495

[52] George Labahn, Vincent Neiger, and Wei Zhou. 2017. Fast, deterministic computation of the Hermite normal form and determinant of a polynomial matrix. *Journal of Complexity* 42 (2017), 44–71. https://doi.org/10.1016/j.jco.2017.03.003

[53] Yishay Mansour. 1995. Randomized Interpolation and Approximation of Sparse Polynomials. *SIAM J. Comput.* 24, 2 (1995), 357–368. https://doi.org/10.1137/S0097539792239291

[54] Michael Monagan and Roman Pearce. 2009. Parallel sparse polynomial multiplication using heaps. In *ISSAC'09*. 263–270. https://doi.org/10.1145/1576702.1576739

[55] Michael Monagan and Roman Pearce. 2011. Sparse polynomial division using a heap. *J. Symb. Comput.* 46, 7 (2011), 807–822. https://doi.org/10.1016/j.jsc.2010.08.014

[56] Hirokazu Murao and Tetsuro Fujise. 1996. Modular Algorithm for Sparse Multivariate Polynomial Interpolation and Its Parallel Implementation. *Journal of Symbolic Computation* 21, 4 (1996), 377–396. https://doi.org/10.1006/jsco.1996.0020

[57] Vasileios Nakos. 2020. Nearly Optimal Sparse Polynomial Multiplication. *IEEE T. Inform. Theory* 66, 11 (2020), 7231–7236. https://doi.org/10.1109/TIT.2020.2989385

[58] Henri Nussbaumer. 1980. Fast Polynomial Transform Algorithms for Digital Convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28, 2 (1980), 205–215. https://doi.org/10.1109/TASSP.1980.1163372

[59] Armelle Perret du Cray. 2023. *Algorithmes pour les polynômes creux: interpolation, arithmétique, test d'identité*. Ph. D. Dissertation. L'Université de Montpellier.

[60] Gerlind Plonka, Katrin Wannenwetsch, Annie Cuyt, and Wen-shin Lee. 2018. Deterministic sparse FFT for M-sparse vectors. *Numerical Algorithms* 78, 1 (01 May 2018), 133–159. https://doi.org/10.1007/s11075-017-0370-5

[61] Daniel Potts, Manfred Tasche, and Toni Volkmer. 2016. Efficient Spectral Estimation by MUSIC and ESPRIT with Application to Sparse FFT. *Frontiers in Applied Mathematics and Statistics* 2 (2016). https://doi.org/10.3389/fams.2016.00001

[62] Gaspard Clair François Marie Riche de Prony. 1795. Essai expérimental et analytique sur les lois de la Dilatabilité de fluides élastique et sur celles de la Force expansive de la vapeur de l'eau et de la vapeur de l'alkool, à différentes températures. *J. École Polytechnique* 1, Floréal et Prairial III (1795), 24–76. https://gallica.bnf.fr/ark:/12148/bpt6k433661n/f32.item

[63] Daniel S. Roche. 2011. Chunky and equal-spaced polynomial multiplication. *J. Symb. Comput.* 46, 7 (2011), 791–806. https://doi.org/10.1016/j.jsc.2010.08.013

[64] J. Barkley Rosser and Lowell Schoenfeld. 1962. Approximate formulas for some functions of prime numbers. *Illinois J. Math.* 6, 1 (1962), 64–94. https://doi.org/10.1215/ijm/1255631807

[65] Arnold Schönhage. 1982. Asymptotically Fast Algorithms for the Numerical Muitiplication and Division of Polynomials with Complex Coefficients. In *EUROCAM*. Springer, 3–15. https://doi.org/10.1007/3-540-11607-9_1

[66] Arnold Schönhage and Volker Strassen. 1971. Schnelle Multiplikation großer Zahlen. *Computing* 7, 3 (1971), 281–292. https://doi.org/10.1007/BF02242355

[67] Andrei L. Toom. 1963. The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers. *Soviet Mathematics Doklady* 3 (1963), 714–716.

[68] Wei Zhou and George Labahn. 2012. Efficient Algorithms for Order Basis Computation. 47, 7 (2012), 793–819.

[69] Richard Zippel. 1979. Probabilistic algorithms for sparse polynomials. In *Symbolic and Algebraic Computation (EUROSAM'79, Vol. 72)*, G. Goos, J. Hartmanis, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and Edward W. Ng (Eds.). Springer, 216–226. https://doi.org/10.1007/3-540-09519-5_73

[70] Richard Zippel. 1990. Interpolating Polynomials from Their Values. *Journal of Symbolic Computation* 9, 3 (March 1990), 375–403. https://doi.org/10.1016/S0747-7171(08)80018-1