Random primes without primality testing

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

Bruno Grenet
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
bruno.grenet@lirmm.fr

Armelle Perret du Cray
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
armelle.perret-du-cray@lirmm.fr

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

March 5, 2022

Abstract

Numerous algorithms call for computation over the integers modulo a randomly-chosen large prime. In some cases, the quasi-cubic complexity of selecting a random prime can dominate the total running time. We propose a new variant of the classic D5 algorithm for "dynamic evaluation", applied to a randomly-chosen (composite) integer. Unlike the D5 principle which has been used in the past to compute over a direct product of fields, our method is simpler as it only requires following a single path after any modulus splits. The transformation we propose can apply to any algorithm in the algebraic RAM model, even allowing randomization. The resulting transformed algorithm avoids any primality tests and will, with constant positive probability, have the same result as the original computation modulo a randomly-chosen prime. As an application, we demonstrate how to compute the exact number of nonzero terms in an unknown integer polynomial in quasi-linear time. We also show how the same algorithmic transformation technique can be used for computing modulo random irreducible polynomials over a finite field.

1 Introduction

Consider the following situation which arises commonly in exact computational problems: We have a problem over the integers $\mathbb{Z}$ to solve, but perhaps due to intermediate expression swell or the need for exact divisions, solving direcly in $\mathbb{Z}$ or $\mathbb{Q}$ is infeasible. There may be fast algorithms for this problem over a finite field, say modulo a prime p, but then some (unknown) conditions on the prime p must be met in order for the solution over $\mathbb{F}_p$ to coincide with the actual integer solution. Typically, p must not be a divisor of some unknown (but bounded) large value.

Then a classical approach, called the *big-prime technique*, is to randomly choose a large prime p, with sufficient bit-length to overcome any non-divisibility conditions with high probability, solve the problem over $\mathbb{F}_p$, and return the result; see [12, Chapter 5].

When the computation itself is expensive, we can ignore the cost of prime number generation for practical purposes; the running time to compute p will be dwarfed by the computations within $\mathbb{F}_p$ that follow. But as the computation becomes more efficient, and particularly for algorithms that have quasi-linear complexity, the cost of prime number generation is more significant, and may be a performance bottleneck in theory and/or in practice.

The most efficient Monte Carlo method to generate a random prime number is to sample random integers of the required size and then test them for primality. To find a b-bit prime, using fast arithmetic, each primality test costs $\widetilde{O}\left(b^2\right)$ time, and because the density of primes among all b-bit numbers is proportional to $1/b$, the total cost of prime generation in this way is $\widetilde{O}\left(b^3\right)$. Faster practical techniques by [28, 19] incorporate many clever ideas but do not improve on this cubic complexity bound; see [36, Chapter 10].

In summary: the cubic cost of generating a large probable-prime may dominate the total cost of the big-prime method when the needed primes are quite large and the mod-p algorithm is quite fast. The aim of this paper is to tackle this issue with a general technique that uses random moduli, *not necessarily prime*, with various extra checks along the way, to provably get the same result in many cases as would be achieved by explicitly generating a random prime.

An analogous situation occurs for polynomials over finite fields: if the original problem is in a fixed finite field $\mathbb{F}_q$, then in many cases one needs to compute over an extension field $\mathbb{F}_{q^k}$ for k sufficiently large. This then poses the challenge of computing a random irreducible degree-k polynomial in $\mathbb{F}_q[x]$, which again takes at least quasi-cubic time using the best current methods. We will show that our same basic algorithm transformation technique for integers applies in this case as well, with similar probability bounds.

1.1 Algorithmic transformation technique

Our method builds on the long line of techniques known as *dynamic evaluation*, or the D^5 principle [6]. This is a very general technique which has since been employed for a wide range of computational problems [9, 27, 10, 29, 5, 32, 7, 16, 33]. For our purposes the idea is to start computing modulo a possibly-composite m, and "split" the evaluation with an efficient GCD computation whenever we need to test for zero or perform a division.

Like the recent *directed* evaluation technique proposed by van der Hoeven and Lecerf [16], we opt to first take the larger-size branch in any GCD splitting. But unlike their method (and prior work) which is ultimately focused on recovering the correct result over the original product of fields, here the goal is only to have an answer which is consistent with what it *would have been* in some randomly-chosen finite field. For that reason, we can ignore the smaller branch of any split, avoiding the reconstruction process altogether.

In terms of running time, the key observation is that GCDs can be computed in quasi-linear bit complexity using the half-GCD algorithm of [30, 4], and therefore this transformation has the same soft-oh bit complexity as it would to compute over an actual prime of the desired size. We save because there is no longer a need to actually generate (and test for) the random prime.

Proving the probabilistic correctness of this approach is the main challenge and contribution of our paper. First we need good estimates on the probability that a random integer has a large prime factor. In our technique, having a single prime factor $p \mid m$ where $p^2 \geq m$ is necessary and sufficient.

Second, and more challengingly, we need to allow for a general model of computation where the algorithm may sample uniformly from the random field it is computing over, and probabilistic correctness over a randomly-chosen field $\mathbb{F}_p$ should carry over to probabilistic correctness of our method with random not-necessarily-prime moduli.

To our knowledge, previous applications of the D^5 principle have considered only computational models which are deterministic and do not allow random sampling of field elements. van der Hoeven and Lecerf [16] suggest that this can be overcome by providing the deterministic computation tree with a "pool" of pre-selected random field elements, and the recent paper of Neiger, Salvy, Schost, and Villard [31] mentions this limitation and also skirts around it by providing pre-selected "random" choices as additional algorithm inputs.

But this pre-selection does not really make sense in our setting, where the initial modulus m itself is randomly chosen as well, as we must do in order to avoid "unlucky" choices of the underlying finite field $\mathbb{F}_p$. Which is to say, for any *fixed*, pre-selected value of m and "random" elements modulo m, there is no way to argue that the result will be correct with high probability; but it is also impossible to choose uniform-random elements modulo m or p without knowing the modulus in advance. Instead, we take care to actually allow randomization within the transformed algorithm, and prove that probabilistic correctness modulo most sufficiently large primes p does indeed imply probabilistic correctness modulo a large-enough random integer m (and equivalently with random extension fields over a fixed finite field).

Our optimization to compute on only one branch in fact adds new wrinkles to the challenge of proving correctness. As a small illustration, consider a very simple algebraic algorithm which simply chooses a random field element and tests whether it is zero. Over $\mathbb{F}_2$ there should clearly be a $\frac{1}{2}$ probability of each outcome. But if we instead compute with initial modulus $m = 30$ and use our "largest branch" splitting technique, whenever 2 divides the final modulus, there is only a $\frac{1}{3}$ chance of getting zero over $\mathbb{F}_2$. The reason this can occur is that the branches, and hence the choice of which modulus to use at the end, *may themselves depend on previous random choices*. In fact it is not hard to construct pathological algorithmic examples which are usually correct modulo some certain-sized primes p, but usually *incorrect* modulo m when m is a multiple of p. Overcoming such issues in a proven and generic way is a major challenge of the present investigation.

1.2 Application to sparsity determination

As an important application of our technique to avoid primality testing in randomized computation, and indeed our original motivation for this work, we develop a new algorithm to compute the sparsity of an unknown black-box polynomial.

This Monte Carlo randomized algorithm uses samples of a *modular black box*, via which an unknown sparse integer polynomial $f \in \mathbb{Z}[x_1, \dots, x_n]$ can be evaluated for any chosen modulus $m \in \mathbb{N}$ and point $(\theta_1, \dots, \theta_n) \in [0, m)^n$, as well as bounds H and D on the height and max degree, respectively, to determine the number of nonzero terms $\#f$ in the unknown polynomial, correct with high probability. The bit complexity (accounting for black box evaluations) is softly-linear in the size f; see Theorem 5.2 for a precise statement.

This problem is closely related to the more general problem of sparse interpolation [3, 23, 22, 11, 18, 20, 2, 15, 1, 17], where all coefficients and exponents of f are to be recovered. Oftentimes such algorithms assume they are given an upper bound $T \geq \#f$ and have running time proportional to this T, so having a fast way to determine $\#f$ exactly can be valuable in practice.

Our method mostly follows the *early termination* strategy by Kaltofen and Lee [21], which was the first efficient sparse interpolation algorithm not to require an *a priori* upper bound $T \geq \#f$. We save on the running time by explicitly stopping the algorithm early as soon as $\#f$ is learned, and avoiding costly later steps which require special field structure.

Moreover, while the algorithm of [21] works over a more general domain, its bit complexity over $\mathbb{Z}$ is exponentially large; thus, a "big prime" technique is commonly employed; see [24, 20, 18]. But now that we have reduced the arithmetic complexity to quasi-linear, the cubic bit-cost of large prime generation becomes significant. This is where our new algorithm transformation technique comes into play: we use our new methods to obtain the same results as if working modulo a random prime, while actually computing modulo a random composite number with at most twice the bit length.

1.3 Summary of contributions

The main results of this paper are:

- A new variant of the D^5 principle which focuses on computing modulo random primes rather than in a given product of fields (Section 4.1);
- A careful analysis which shows that any algorithm which is *probably* correct for *most* random, sufficiently-large primes, can be solved without the cost of prime number generation using our new technique (Section 4.3);
- A new algorithm with nearly-optimal bit complexity to determine the number of nonzero terms of an unknown sparse integer polynomial (Section 5); and
- An adaptation of the same techniques to computing modulo random irreducible polynomials over finite fields without irreducibility testing (Section 6).

2 Prime density and counting bounds

In this section we review some mostly-known results on the number of primes in intervals with certain properties, that will be needed for the probabilistic analysis of our main results.

Throughout, we use the notation $[a, b)$ to denote the set of integers n satisfying $a \le n < b$, and we use the term *b-bit integer* to mean an integer in the range $[2^{b-1}, 2^b)$. Note that there are 2^{b-1} integers with bit-length b.

Definition 2.1. *For any positive integers m and b, we say m is b-fat if m has a prime divisor $p \ge 2^b$.*

This follows the definition of M-fat in the classic paper of Karp and Rabin [25], except that we focus only on power-of-two bounds.

We first prove that at least half of all $2b$-bit integers have a prime factor with at least b bits, largely following [25, Lemma 8], which in turn is based on bounds from Rosser and Schoenfeld [34].

Lemma 2.2. *For any $b \ge 1$, the number of $(2b)$-bit integers which are b-fat is at least 2^{2b-2}.*

Proof. The claim is verified numerically for $1 \le b \le 6$.

If $b \ge 7$, then Lemma 8 of [25] tells us that the number of b-fat integers in the range $[1, 2^{2b}]$ is at least 2^{2b-1}.

We need to show that at least half of the b-fat integers are in top half of this range.

For any prime p with $p \ge 2^b$, consider the multiples of p in the range $[1, 2^{2b}]$. By definition, all such multiples are b-fat. Let $k \in \mathbb{N}$ so that kp is the largest multiple of p less than 2^{2b-1}. Thus exactly k multiples of p have bit-length strictly less than $2b$. And because $2kp < 2^{2b}$, there are at least k multiples of p with exactly $2b$ bits. Incorporating the fact that 2^{2b} is never a multiple of p, we see that at least half of the multiples of p in the range $[1, 2^{2b}]$ have bit-length exactly $2b$.

Because any number less than 2^{2b} can only be divisible by at most one prime $p \ge 2^b$, the sets of multiples for $2^b \le p < 2^{2b}$ in fact form a partition of the b-fat numbers with at most $2b$ bits. Therefore, summing over all sets in this partition, we see that the total number of b-fat integers with $2b$ bits is at least 2^{2b-2}. □

Many algorithms which perform computations modulo a random prime in fact need to avoid a certain number of unlucky or "bad" primes. Here we give an upper bound on the chance that a b-fat number is divisible by a large bad prime. The proof is trivially just dividing the range by p.

Lemma 2.3. *For any $b \ge 1$ and prime $p \ge 2^b$, at most 2^{b-1} integers with bit-length $2b$ are multiples of p.*

3 Computational model

Our main result is a transformation which, roughly speaking, takes any algorithm in the algebraic RAM model for any chosen prime p, and converts it into a randomized algorithm in the (non-algebraic) RAM model that produces the same output most of the time while avoiding prime number generation.

Here we must take care to define the requirements on the initial algebraic algorithm. Prior work such as [6, 5, 16] considered more general settings beyond computing in random finite fields, but in fairly restricted models of deterministic algebraic computation such as straight-line programs or computational trees. Here we have a more restricted algebraic setting but a more general computational one, allowing for loops, memory, and pseudorandom integer or field element generation.

3.1 Modular PRNGs

We define a *modular pseudo-random number generator*, or modular PRNG, as a pair of deterministic algorithms:

- $\textsc{RandMod}(s, m) \longrightarrow x$ takes a fixed-length state s and any positive integer m and produces a pseudorandom value x uniformly distributed from the range $[0, m)$.
- $\textsc{RandUpdate}(s) \longrightarrow s'$ takes the current state and produces a value (with the same bit-length) for the next state.

(In practice there would also be an initialization procedure which takes a smaller seed value to produce the initial state, but this detail is unimportant for our discussion. That is, we treat the seed as synonymous with the initial state s.)

Conceptually, both of these functions should be indistinguishable from random. More precisely, define $r(s, i, m)$ as the i'th output modulo m from initial state s, that is,

$\textsc{RandMod}(\underbrace{\textsc{RandUpdate}(\cdots(\textsc{RandUpdate}}_{i \text{ times}}(s))), m).$

Then, even with an oracle to compute $r(s, i', m')$ for any tuple (i',m') not both equal to (i, m), it should be infeasible to distinguish $r(s, i, m)$ from a truly random output modulo m.

3.2 Algebraic RAM with integer I/O

Our model of computation is a classical *random access machine* (RAM), with an "integer side" and an "arithmetic side". The integer side is a normal RAM machine with instructions involving integer inputs and outputs, and the arithmetic side involves *only* the following computations with elements of an arbitrary field:

- Ring arithmetic: $+, -, \times$
- Multiplicative inverse (which in turn allows exact division)
- Pseudo-random generation of a field element

The multiplicative inverse computation may fail, e.g., on division by zero, in which case the algorithm returns the special symbol $\perp$.

There are also two special instructions which take input from one side and output to the other side:

- Conversion of an integer to a field element
- Zero-testing of any field element

Note that zero-testing and subtraction obviously allow equality comparisons between field elements, but not ordering them.

We restrict the inputs and outputs to be integers (or output of $\perp$ on error). This is necessary for our use case where the inputs are actually integers which are then reduced modulo p, but in any case is not a restriction due to the integer-to-field-element conversion instruction.

For computations over prime fields, we can formally define an *algorithm* in our algebraic RAM model as having:

- Parameters: prime p and initial Modular PRNG state s
- Input: $\mathbf{x} \in \mathbb{Z}^n$
- Output: $y \in \mathbb{Z} \bigcup \{\perp\}$
- Program: A series of integer instructions as in a normal RAM model and arithmetic instructions as defined above, along with labels, conditional branches, and memory load/store operations

We write $\mathcal{A}_{p,s}(\mathbf{x})$ for the output y of algorithm $\mathcal{A}$ with field $\mathbb{F}_p$ and initial PRNG state s.

Note that some abuses of the RAM model are possible, particularly for algebraic algorithms, by using arbitrarily large integers. Rather than avoid such abuses by, e.g., using a word RAM model, we merely note that our transformations do not affect the integer side operations at all, and thus would in principle work the same under any such model restrictions. Indeed, our work applies over most algebraic computational models we know of, such as straight-line programs, branching programs, multi-tape Turing machines, or the closely-related BSS model.

4 Algorithm transformation for random prime fields

Recall the general idea of our approach, building on [6, 16]: Given an algorithm $\mathcal{A}$ which works over any prime field, we transform it into a transformed version $\overline{\mathcal{A}}$ that instead samples a possibly-composite modulus m which *probably* contains a large prime factor $p \geq \sqrt{m}$. Any time a comparison or multiplicative inverse occurs, $\overline{\mathcal{A}}$ first performs a GCD of the operand with the current modulus m. If a nontrivial factor k of m is found, then the (unknown) large prime factor of m must divide $\max(k, m/k)$. Update the modulus accordingly and continue.

We first fully present our algorithmic transformation, then examine some thorny issues related to the use of pseudorandom numbers in the algorithms, and finally prove the correctness and performance bounds which are the main results of this paper.

4.1 Transformation procedure

We begin with the crucial subroutine Algorithm 1 (NewModulus), which shows how to update the modulus while carefully ensuring no significant blow-up in bit or arithmetic complexity.

Algorithm 1: NewModulus(a, m)

```
Input: Integer $a \in \mathbb{Z}$ and modulus $m \in \mathbb{N}$
Output: New modulus $m' \in \mathbb{N}$

$g_1 \longleftarrow \gcd(a, m)$
if $g_1^2 > m$ then
    return $g_1$
else
    $b \longleftarrow g_1^{\lfloor \log_2 m \rfloor} \bmod m$
    $g_2 \longleftarrow \gcd(b, m)$
    return $m/g_2$
```

Lemma 4.1. *Given any $a, m \in \mathbb{Z}$, the integer $m' \in \mathbb{N}$ returned by NewModulus(a, m) has the following properties:*

- *$m' \mid m$*
- *a is either zero or invertible modulo m'*
- *If p is a prime with $p^2 > m$ and $p \mid m$, then $p \mid m'$ also*

Proof. Let $a, m \in \mathbb{Z}$ be arbitrary and g_1, b, g_2 as in the algorithm.

For the first property, we see that the integer returned is either g_1 or m/g_2, both of which are always divisors of m.

For the second property, consider two cases. First, if $g_1^2 > m$ and thus $m' = g_1$ is returned, then $g_1 \mid a$, so a is zero modulo m'.

Otherwise, let $q \geq 2$ be any common factor of a and m (if one exists). By the definition of gcd, $q \mid g_1$. Now let $k \geq 1$ such that q^k is the largest power of q that divides m. Because $q \geq 2$, $k \leq \log_2 m$, and therefore $q^k \mid g_1^{\lfloor \log_2 m \rfloor}$. From the gcd algorithm, this means that $q^k \mid g_2$, and thus q does *not* divide m/g_2. Hence a and m/g_2 do not share any common factor, i.e., $\gcd(a, m') = 1$ as required.

For the third property, assume m has a prime factor p with $p^2 > m$. If $p \mid g_1$, then $g_1^2 > m$ and the new modulus $m' = g_1$ returned in the first case is be divisible by p. Otherwise, if p does not divide g_1, then p does not divide g_2 either, and hence $p \mid m/g_2$. □

Lemma 4.2. *For any $a \in \mathbb{Z}$ and $m \in \mathbb{N}$, the worst-case bit complexity of Algorithm NewModulus(a, m) is $\widetilde{O}(\log a + \log m)$.*

Proof. The bit complexity is dominated by the two GCD computations and the modular exponentiation to compute b. Using the asymptotically fast Half-GCD algorithm [35, 37], and binary powering for the modular exponentiation, all three steps have bit cost within the stated bound. □

We now proceed to incorporate the NewModulus subroutine into the algorithm transformation procedure. Algorithm $\mathcal{A}$ will be in the model of an algebraic RAM from Section 3.2. We also require a companion procedure $\mathcal{B}$ which takes any input $\mathbf{x} \in \mathbb{Z}^n$ for $\mathcal{A}$ and *deterministically* produces a positive integer b, which should be a minimal bit-length of primes to ensure that $\mathcal{A}$ produces correct results on input $\mathbf{x}$ with high probability. (In most applications we can imagine, $\mathcal{B}$ is a simple function of the input sizes and bit-lengths.)

Algorithm 2 details the construction of the transformed algorithm $\overline{\mathcal{A}}$ based on $\mathcal{A}$ and $\mathcal{B}$.

Algorithm 2: $\overline{\mathcal{A}}$ produced from $\mathcal{A}$ and $\mathcal{B}$

```
  Input: Input $\mathbf{x} \in \mathbb{Z}^n$ and PRNG initial state $s$
  Output: $y \in \mathbb{Z} \bigcup \{\perp\}$
  $b \longleftarrow \mathcal{B}(\mathbf{x})$
  $m \longleftarrow$ pseudorandom $(2b)$-bit integer stored in memory
  Proceed with identical instructions of $\mathcal{A}$, except:
      • All conversions from integers to algebraic values are replaced by explicit
        reduction modulo $m$.
      • All additions, subtractions, and multiplications on the "algebraic side" are
        replaced by integer arithmetic followed by explicit reduction modulo $m$.
      • Any random field element generation instructions are replaced by sampling
        a number in the range $[0, m)$ using the modular PRNG.
      • All zero tests and multiplicative inverses on the "algebraic side" for some
        value $a$ first call NewModulus($a$, $m$), update the modulus accordingly, and
        then perform either a divisible-by-$m$ test or modular inverse computation,
        respectively.
```

Observe that the transformed algorithm $\overline{\mathcal{A}}$ no longer works in the algebraic RAM model, but handles explicit integers only. More precisely, each algebraic operation in $\mathcal{A}$ is replaced with a constant number of arithmetic or gcd computations on integers with bit-length at most $2b$.

4.2 Correlated PRNGs

A technical detail of our analysis requires a tight relationship between the random choices made by $\mathcal{A}$ and $\overline{\mathcal{A}}$ for the same input and seed value. Even using the same PRNG for both algorithms, there is no reason to think that a random sample in $\mathcal{A}$ modulo a prime p would have any relationship to a corresponding random sample in $\overline{\mathcal{A}}$ modulo a multiple m of p.

We achieve this by defining a pair of modular PRNGs — one for $\mathcal{A}$ and one for $\overline{\mathcal{A}}$ — both based on the same underlying high-quality PRNG, and having the desired correlation property. The constructions are based on three key insights. First, in defining the transformed algorithm $\overline{\mathcal{A}}$ above, we are careful to sample the initial modulus m *inside* the algorithm, rather than taking it as input. Second, the PRNG for the algebraic algorithm $\mathcal{A}$ also samples a random modulus m even though this is not directly used in the computations, in order to match random outputs between $\mathcal{A}$ and $\overline{\mathcal{A}}$. Third, our correlated PRNG construction doubles the state size so that this modulus m is chosen completely independently from the future sampled random values; this allows us later to translate probabilistic correctness modulo p in $\mathcal{A}$ to probabilistic correctness modulo m in $\overline{\mathcal{A}}$.

Before describing the PRNGs in detail, it is important to stress that *this is purely a proof technique.* While these PRNG constructions are efficient and realizable, their need is motivated only by the probabilistic analysis that follows; in reality, we could just use any normal high-quality PRNG and achieve the same results except for some pathological algorithms would not arise in practice.

For what follows, we assume the existence of a high-quality modular PRNG G as defined in Section 3.1.

We begin by describing the PRNG for the transformed algorithm $\overline{\mathcal{A}}$ of Algorithm 2, which we call $\overline{G'}$.

We will use two simultaneous instances of the underlying modular PRNG G, and therefore double the state size so that the seed for $\overline{G'}$ can be written $s = (s_0, s_1)$. The first instance with initial state s_0 is used only to generate the initial modulus m on the first step.

Afterwards, a random sample modulo some integer $t \in \mathbb{N}$ is generated as follows. If $t \mid m$, then $\overline{G'}$ first generates a random integer modulo m using G with the second part of the current state s_1, and then reduces the result again modulo t before returning it. Because $t \mid m$, this is indistinguishable from (though less efficient than) randomly choosing an integer modulo t directly.

Otherwise, if t is not a divisor of the original modulus m, then $\overline{G'}$ simply calls G directly with the current state s_1 and the requested modulus.

The PRNG G' for the algebraic algorithm $\mathcal{A}$ is almost identical to $\overline{G'}$, except that the value of m is only used *inside the PRNG.* Indeed, for a random seed and run of the algorithm modulo some prime p, it is unlikely that $p \mid m$, and then this construction doesn't change the results at all compared to using G with the second part of the state s_1 alone.

The need for this strange PRNG construction is the correlation that exists between $\mathcal{A}$ and $\overline{\mathcal{A}}$ whenever the seed $s = (s_0, s_1)$ is the same for both, as captured in the following lemma:

Lemma 4.3. *Let $s = (s_0, s_1)$ be a PRNG state for G' or $\overline{G'}$, m be the initial $2b$-bit random value chosen using s_0, and $p, m' \in \mathbb{N}$ such that $p \mid m'$ and $m' \mid m$. Writing r as the pseudorandom result from G' with current state s modulo p, and $\overline{r}$ as the result from $\overline{G'}$ with the same state s and larger modulus m', then we have $r \equiv \overline{r} \bmod p$.*

The proof follows directly from the PRNG definitions above and the divisibility conditions given.

Besides this correlation property, we also need to know that using this constructed PRNG in the algebraic algorithm $\mathcal{A}$ does not affect the probabilistic correctness. Note that the assumption on the underlying PRNG G is an idealistic one, as any PRNG with fixed seed length cannot actually produce uniformly random values in an arbitrary range.

Lemma 4.4. *If the underlying PRNG G produces uniformly random values for any sequence of moduli, then for* any *fixed value of s_0, the constructed modular PRNG G' also produces uniformly random values for any sequence of moduli.*

Proof. Let s_0 be a fixed first half of the seed. The value of s_0 purely determines what m is chosen, so let $m \in \mathbb{N}$ be that arbitrary value.

Now consider any modulus t given to G'. If t does not divide m, then G' returns a value from the underlying PRNG G, which by assumption is uniformly random.

Otherwise, if $t \mid m$, then G is first sampled for some random value r in $[0, m)$. By assumption r takes on any value in this range with probability $\frac{1}{m}$. Then because t is a divisor of m, reducing r mod t produces each value in $[0, t)$ with probability $\frac{1}{t}$. □

From this, we can draw a crucial conclusion on the probabilistic correctness of $\mathcal{A}$ when only s_1 is varied.

Corollary 4.5. *For some prime p and input $\mathbf{x}$, suppose $\mathcal{A}$ produces a given output y with probability $1 - \epsilon$ when provided an ideal perfectly-random number generator. Then if the underlying PRNG G is uniformly random and for any fixed value of s_0, $\overline{\mathcal{A}}$ returns y with the same probability $1 - \epsilon$, where the probability is over all choices of s_1 only.*

4.3 Analysis

We now proceed to the main result of our paper: for any algebraic algorithm $\mathcal{A}$ that produces correct results with high probability for primes of bit-length at least that given by $\mathcal{B}$, Algorithm 2 produces a randomized algorithm $\overline{\mathcal{A}}$ which, with constant probability, produces the same correct results with the same number of steps. That is, we can achieve (probabilistically) the same results as computing modulo random primes, without actually needing to ever conduct a primality test.

To prove this probabilistic near-equivalence, first define a *run* of an algorithm as the sequence of internal memory states for given inputs and seed value (and in the case of an algebraic algorithm, choice of field).

We first show that any run of the transformed algorithm $\overline{\mathcal{A}}$ is equivalent to a run of the original $\mathcal{A}$ with the same input and seed for some choice of p. Here and for the remainder of this section, we assume that $\mathcal{A}$ and $\overline{\mathcal{A}}$ use the constructed PRNGs G' and $\overline{G'}$ as defined in Section 4.2.

Lemma 4.6. *For any input $\mathbf{x}$ and seed s, consider the resulting run of $\overline{\mathcal{A}}$. If m' is the final stored value of m in this run, then for any prime factor p of m', an identical run of $\mathcal{A}$ is produced over $\mathbb{F}_p$ with the same input $\mathbf{x}$ and seed s, where all algebraic values from the run of $\overline{\mathcal{A}}$ are reduced modulo p.*

Proof. Consider the memory states at some point in the program where they are equivalent between the two runs. We show that, no matter the next instruction in $\mathcal{A}$, the memory states after that instruction (and the corresponding instruction(s) in $\overline{\mathcal{A}}$ according to Algorithm 2) are the still equivalent.

Any arithmetic-side operations are unchanged in Algorithm 2.

Because reduction modulo p is a homomorphism, any algebraic additions, subtractions, or multiplications also maintain equivalence.

Let m be the original modulus chosen at the beginning of $\overline{\mathcal{A}}$. From the first point of Lemma 4.1, we know that $m' \mid m$, and therefore $p \mid m$ also. The same is true for any intermediate value of m in the run.

This means that any conversion operation in $\mathcal{A}$, reducing an integer modulo p, will be mod-p equivalent to the corresponding operation in $\overline{\mathcal{A}}$, reducing modulo the current value of m.

Considering zero-test instructions, let a (resp. $\overline{a}$) be the value of some algebraic value in the run of $\mathcal{A}$ (resp. $\overline{\mathcal{A}}$). If $a = 0$ in $\mathbb{F}_p$, then $p \mid \overline{a}$. Then, because p divides every modulus value m in the run, $\overline{a}$ is not invertible modulo m. Then according to the second point in Lemma 4.1, $\overline{a}$ is zero mod m, and the zero test will have the same result.

By the same reasoning, any multiplicative inverse instruction will also be equivalent between the runs of $\mathcal{A}$ and $\overline{\mathcal{A}}$, or in the case the denominator is zero, both runs will result in $\perp$.

Finally, by using the correlated PRNGs defined in Section 4.2, we can apply Lemma 4.3 to conclude that any random field element generation instruction also results in equivalent outputs at the same step of both runs.

This covers all possible types of instructions and completes the proof. □

If the original algorithm $\mathcal{A}$ is deterministic, this equivalence of runs is enough to prove correctness of $\overline{\mathcal{A}}$. Indeed, this has been the assumption in most prior works on the D^5 principle, which also often employ simpler models of computation such as straight-line programs or deterministic computation trees.

By contrast, we want to allow $\mathcal{A}$ to be randomized. First, we affirm that any *deterministic property* of the output is preserved after our transformation to $\overline{\mathcal{A}}$ in the following lemma, which follows directly from Lemma 4.6.

Lemma 4.7. *Suppose $\mathbf{x} \in \mathbb{Z}^n$ is an input for $\mathcal{A}$, and $Y \subseteq \mathbb{Z} \bigcup \{\perp\}$ is a family of outputs, such that for* any *prime p and random seed s, the output of $\mathcal{A}$ on input $\mathbf{x}$ is always a member of Y. Then the output of $\overline{\mathcal{A}}$ on input $\mathbf{x}$ is always a member of Y as well.*

The more difficult case is when $\mathcal{A}$ may return incorrect values. There are two types of causes for an incorrect result: when the prime p is one of a small set of "unlucky" values, or when randomly-sampled field elements in the algorithm are unlucky and produce and incorrect result. Although many actual algorithms only have one of these two types of failure, we account for both types in order to have the most general result.

For convenience of exposition, we will capture these failure modes in the following definition:

Definition 4.8. *Let $\mathbf{x} \in \mathbb{Z}^n$ be any input for $\mathcal{A}$, $k \in \mathbb{N}$ and $\epsilon \in \mathbb{R}$ with $0 \leq \epsilon < 1$. We say that $\mathcal{A}$ is (k, ϵ)-correct for input $\mathbf{x}$ if, for all but at most k primes p with $p \geq 2^{\mathcal{B}(\mathbf{x})}$, running $\mathcal{A}$ on $\mathbf{x}$ produces a correct output with probability at least $1 - \epsilon$.*

The following theorem, which is the main result of our paper, combines the prevalence of b-fat integers with the run-equivalence of $\overline{\mathcal{A}}$ to prove probabilistic correctness.

Theorem 4.9. *Let* $\mathbf{x} \in \mathbb{Z}^n$, $k \in \mathbb{N}$, *and* $\epsilon \in \mathbb{R}$ *with* $0 \le \epsilon < \frac{1}{2}$, *and write* $b = \mathcal{B}(\mathbf{x})$. *If* $\mathcal{A}$ *is* (k, ϵ)*-correct for input* $\mathbf{x}$, *then the probability that* $\overline{\mathcal{A}}$ *produces the correct output for input* $\mathbf{x}$ *is at least*

$$\frac{1}{2} - \frac{k}{2^{b-2}} - \frac{\epsilon}{2}.$$

Proof. Let m be the initial prime modulus chosen uniformly in the range $[2^{2b-1}, 2^{2b})$ by $\overline{\mathcal{A}}$. Two things can make m an unlucky choice: if it has no large prime divisor (i.e., if it is not b-fat), or if its largest prime divisor is one of the k unlucky primes that cause $\mathcal{A}$ to fail.

Lemma 2.2 tells us that the probability of the former is at most $\frac{1}{2}$. And, disjointly, the probability that m *does* have a large prime factor but it is one of the k unlucky choices for $\mathcal{A}$ is at most $k/2^{b-1}$.

Therefore, over all choices of the first part of the PRNG initial state s_0, at least $1/2 - k/2^{b-2}$ of them lead to an m with a prime factor $p \ge 2^b$ which is not one of the k "unlucky" primes for $\mathcal{A}$ on this input.

For such "lucky" choices of s_0 and thereby m, because $p^2 > m$, from Lemma 4.1 we know that p will always divide the updated modulus m after any call to NewModulus, and in particular, p will divide the final modulus m'. We can therefore apply Lemma 4.6 with the same p for *all* possible runs of $\overline{\mathcal{A}}$ with the same s_0.

Finally, considering the remaining part of the PRNG initial state s_1, Corollary 4.5 tells us that in these cases $\overline{\mathcal{A}}$ produces the correct result with probability at least $1 - \epsilon$, conditional on the previously-derived chance that m is "lucky". □

When combined with an efficient verification algorithm, Theorem 4.9 immediately yields a Las Vegas randomized algorithm.

But without an efficient verifier, the result seems to be not very useful: it proves that $\overline{\mathcal{A}}$ is a Monte Carlo randomized algorithm with success probability strictly less than one-half.

Still, we can combine Theorem 4.9 with the preceding Lemma 4.7 in the case of algorithms $\mathcal{A}$ which have *one-sided error*, meaning that, for any prime p and initial PRNG state s, the output y from $\mathcal{A}$ is never larger (or, equivalently, never smaller) than the correct answer.

Corollary 4.10. *Let* $\mu > 0$ *such that, for any input* $\mathbf{x}$, *algorithm* $\mathcal{A}$ *is* (k, ϵ)*-correct where* $1 - \epsilon - k/2^{\mathcal{B}(\mathbf{x})-1} < \mu$. *If* $\mathcal{A}$ *furthermore has only one-sided error, then the correct output can be determined with high probability after* $O(\frac{1}{\mu})$ *runs of* $\overline{\mathcal{A}}$.

Proof. Because of one-sided error, you can repeatedly run $\overline{\mathcal{A}}$ and take the maximum (resp. minimum) result if the output of $\mathcal{A}$ never larger (resp. smaller) than the correct output. □

5 Computing the sparsity of integer polynomials

Sparse polynomial interpolation is an important and well-studied problem in computer algebra: Given an unknown polynomial $f \in \mathsf{R}[x_1, \ldots, x_n]$ through a blackbox or a *Straight Line Program* (SLP), one wants to recover the non-zero coefficients of f and their corresponding exponents. Of course, the main goal is to have an algorithm with a complexity that is quasi-linear in the bit-length of the output, that is $\widetilde{O}(nt(\log d + \log h))$, where $t = \#f$ is the number of non-zero terms, d the maximal degree and h the height, i.e., largest absolute value of any coefficient.

The fastest algorithm for this task are of two kinds, depending of the model in which the polynomial is given. For Straight Line Programs, following the deterministic polynomial time algorithm of Garg and Schost [11], many improvements have been made through randomization to reach a quasi-linear complexity in every parameter of f, i.e., t, $\log d$ and $\log h$ [17]. For a polynomial given instead by a *black box* for its evaluation, following the seminal papers of Ben-Or and Tiwari [3] and Kaltofen and Yagati [23], we have now reached a quasi-linear complexity in the sparsity of f, see Arnold's PhD thesis [1]. Note that an algorithm with quasi-linear complexity in *all* the parameters of f in this model is still not available.

One of the main ingredients of these algorithms is that they require bounds for every parameter (t, d, h) of f. A few works considered to replace these bounds with explicit randomized algorithms. As outlined in [22, 1], one can calculate such a degree bound in polynomial time but this might dominate the cost of the interpolation. The situation is clearly different for the sparsity parameter as interpolation with early termination exists [21]. This is only the case with Prony-style interpolation, popularized by Ben-Or and Tiwari [3] for polynomials given as a blackbox.

In this work we will consider the model of *Modular Blackbox* (MBB) that allows to control the size of the evaluation of the polynomial. In particular, this is of great interest to efficiently deal with sparse polynomials over the integers as one evaluation might be exponentially large than the polynomial itself.

Kronecker substitution [26] is a fairly classical tool to reduce multivariate problems to univariate ones. It is easy to see that this transformation does not change the size of the polynomial and its sparsity. Therefore, we will only focus on the univariate case here for simplicity of presentation.

5.1 Sparsity over a sufficiently large field

First we develop a Monte Carlo algorithm to compute $\#f$ over a sufficiently-large finite field $\mathbb{F}_q$. For this, we can use Ben-Or and Tiwari [3] and the extension of Kaltofen and Lee [21] that study the probability that some early zeros appear during the course of the Berlekamp-Massey algorithm.

Taking a random $\alpha \in \mathbb{F}_q$ and $a_i = f(\alpha^i) \in \mathbb{F}_q$, it is shown that for $s = 1, \dots, t$ all Hankel matrices $H_s = [a_{i+j}]_{i,j=0}^{s-1}$ are non-singular with a probability greater than $1 - \frac{Dt(t-1)(t+1)}{3q}$; see [21, 14, 1]. The so-called early termination strategy for sparse interpolation is then to run the Berlekamp-Massey algorithm on the infinite sequence $(a_0, a_1, a_2, \dots)$ and to stop the algorithm whenever a zero discrepancy occurs. It is showed in [21] that the zero discrepancy corresponds exactly to hitting a singular Hankel matrix H_s. Since the sequence $(a_1, a_2, \dots)$ corresponds to the evaluation of a t-sparse polynomial at a geometric sequence, the minimal generator Λ of the recurrence sequence $(a_0, a_1, a_2, \dots)$ has degree exactly $\#f$ [3].

Fact 5.1. *Given a blackbox for $f \in \mathbb{F}_q[X]$ with $q \geq 16D^4$ where $D > \deg f$, there exists a Monte Carlo algorithm that computes an integer t such that $t \leq \#f$. With probability at least $1 - \frac{1}{48}$, we have $t = \#f$ exactly. The computation requires $2t$ probes to the modular blackbox and $\widetilde{O}(t)$ arithmetic operations in $\mathbb{F}_q$.*

We note that there is nothing special about the constant 16; this just arises from what we need later, and it is convenient to have a very low probability of error.

If f were given by a straight-line program instead of a blackbox, the same algorithm may be employed with bit complexity $\widetilde{O}(L\#f \log q)$, where L is the length of the SLP.

Correctness comes from the previous discussion on the probability that the Hankel matrices H_s are non-singular up to $s = t$; see [21, Theorem 9] for a complete proof. For the complexity, we can use the fast iterative order basis algorithm of [13] since the Berlekamp-Massey algorithm is related to Padé approximant involving the series $\sum_{i>0} a_i x^i$ [8]. The algorithm iPM-basis from [13] provides a fast iterative variant for Padé approximation that can incorporate the early termination strategy (looking for a zero constant term in the residual, denoted F^v).

One may remark that the algorithm of Fact 5.1 is a *one-sided* randomized algorithm; the returned value t never exceeds the true sparsity $\#f$.

5.2 Sparsity over the integers

Now suppose $f \in \mathbb{Z}[x]$ is an integer polynomial given via a modular blackbox, along with bounds D, H such that $\deg f < D$ and each coefficient of f is bounded by H in absolute value. We want to use the techniques of Section 4 to adapt the algorithm of Fact 5.1 to find the sparsity of f.

The first question is how to incorporate the modular blackbox into the algebraic RAM model of Section 3.2. We will say that the algebraic RAM is endowed with an additional instruction to probe the MBB: given any *algebraic* value α, the MBB instruction returns a new algebraic value for $f(\alpha)$. In the original algorithm $\mathcal{A}$, each MBB evaluation will be modulo p, and in the transformed algorithm $\overline{\mathcal{A}}$, evaluations will be modulo the current value of m.

Observe that this functionality is exactly what is already specified in the definition of a modular black box. Importantly, we do *not* require the MBB to be given in any particular computational model (such as algebraic RAM), and the instruction transformations described in Algorithm 2 will *not* apply inside the blackbox itself.

The next question is how large the prime p should be to ensure correctness with high probability. Fact 5.1 gives a lower bound for p so that the mod-p algorithm succeeds, but we also need to ensure that the sparsity of f modulo p is the same as the actual value of $\#f$ over the integers. This will be true as long as none of the coefficients of f vanish modulo p. Then we simply observe that, with bounds D, H on the degree and height of f respectively, the number of "bad primes" which cause the sparsity to drop modulo p is at most $D \log_2 H$.

Set $b = \lceil 4 + 4\log_2 D + \log_2 \log_2 H \rceil$. Then any $p \geq 2^b$ satisfies the condition of Fact 5.1, so we can say the algorithm is $(D \log_2 H, \frac{1}{48})$-correct by Definition 4.8. The following theorem, our main result for this section, follows immediately after observing that $D \log_2 H / 2^{b-2} < \frac{1}{4}$.

Theorem 5.2. *Given a MBB for $f \in \mathbb{Z}[x]$ and bounds D, H with $\deg f < D$ and each coefficient of f is at most H in absolute value, there exists a Monte Carlo randomized algorithm that computes an integer t such that $t \leq \#f$. With probability at least 0.239, we have $t = \#f$ exactly. The computation requires $2t$ probes to the MBB and $\widetilde{O}(t)$ arithmetic operations, all with moduli that have bit-length $O(\log D + \log\log H)$.*

Because the error is again one-sided, Corollary 4.10 applies and can be used to make the success probability arbitrarily high.

If the MBB for f is in fact a straight-line program of length L, the total bit complexity becomes $\widetilde{O}(Lt(\log D + \log\log H))$. While this is technically *sub*-linear in the bit-length of f itself, we note that this is somewhat "hiding" some computation in the evaluation model itself, since to actually produce a polynomial with degree D and height H with bounded constants, the length L of the SLP would need to be at least $\Omega(\log D + \log H)$.

6 Random irreducible polynomials without irreducibility testing

In this section, we adapt our approach to computing in a field extension of a fixed finite field $\mathbb{F}_q$. This need arises frequently in settings where the base field $\mathbb{F}_q$ is too small, and one needs to find more than q distinct elements in it. In that case, the standard approach is to compute a random irreducible polynomial φ of degree s and to work within $\mathbb{F}_{q^s} = \mathbb{F}_q[x]/\langle\varphi\rangle$. If the algorithm that is run in $\mathbb{F}_{q^s}$ is fairly fast, the cost of producing an irreducible polynomial of degree s, $\widetilde{O}\big(s^3 \log q\big)$, may become predominant.

We show how to adapt our techniques to compute modulo an arbitrary random polynomial. Many aspects are very similar to the integer case, so we highlight only the main differences. We begin with the polynomial counterpart of the notion of b-fat integers given in Section 2.

Definition 6.1. *For any positive integer d and finite field $\mathbb{F}_q$, a polynomial $f \in \mathbb{F}_q[x]$ is said to be d-fat if it has an irreducible factor of degree $> d$.*

Lemma 6.2. *For any d, the number of degree-$2d$ monic polynomials over $\mathbb{F}_q$ that are d-fat is at least $\frac{1}{4}q^{2d}$.*

Proof. A monic degree-$2d$ polynomial can have at most one monic irreducible factor of degree $> d$. A given monic irreducible polynomial g of degree $\ell > d$ divides exactly $q^{2d-\ell}$ monic polynomials of degree $2d$. Moreover, the number of irreducible monic polynomials of degree ℓ over $\mathbb{F}_q$ is at least $q^\ell/2\ell$ for any ℓ [36, Lemma 19.12].

Therefore, there are at least $q^{2d}/2\ell$ monic degree-$2d$ polynomials of $\mathbb{F}_q[x]$ that have an irreducible factor of degree ℓ, for $\ell > d$. Summing from $\ell = d+1$ to $2d$, there are at least $\frac{1}{2}q^{2d}(H_{2d} - H_d)$ where $H_n = \sum_{i=1}^{n} 1/i$ denotes the ith harmonic number. Since $1/2(n+1) < H_n - \ln(n) - \gamma < 1/2n$ [38], $H_{2d} - H_d \geq \ln(2) + 1/2(2d+1) - 1/2d \geq \frac{1}{2}$ for $d \geq 2$, and $H_2 - H_1 = \frac{1}{2}$. The result follows. □

Note that the bound of that lemma is smaller than in the integer version, since we only proved that at least one fourth of the degree-$2d$ polynomials over $\mathbb{F}_q$ are d-fat. Actually, the number of monic irreducible polynomials of degree ℓ approaches q^ℓ/ℓ for large values of q or ℓ [36]. With the same proof, this shows that the fraction of degree-$2d$ polynomials that are d-fat approaches $\ln(2) \geq 0.693$ for large values of d or q.

We now turn to the algorithm transformation. We work in the same algebraic RAM model. We adapt the definition of an *algorithm* of Section 3. The parameter p is replaced by a parameter φ which is an irreducible degree-s polynomial over $\mathbb{F}_q$. This requires to fix a correspondence between integers and polynomials over $\mathbb{F}_q$. This is easily done by using the q-adic expansion of integers.

The NewModulus algorithm works similarly *mutatis mutandis*. As input, the algorithm takes a polynomial over $\mathbb{F}_q$ (or equivalently an integer that represents this polynomial) and a modulus $m \in \mathbb{F}_q[x]$, and returns a new modulus $m' \in \mathbb{F}_q[x]$. The test "$g_1^2 > m$" is replaced by a test "$2\deg(g_1) > \deg(m)$", and "$g_1^{\lfloor \log_2 m \rfloor} \bmod m$" is replaced by "$g_1^{\deg m} \bmod m$". The proof of Lemma 4.1 is easily adapted. The worst-case bit complexity of the adapted algorithm is $\widetilde{O}((\deg a + \deg m) \log q)$.

Finally, the algorithm transformation itself is easily adapted. The bound $\mathcal{B}(\mathbf{x})$ returns the minimal degree s of an extension for the algorithm to produce correct results with high probability. The transformed algorithm $\overline{\mathcal{A}}$ computes $2s$ pseudorandom integers modulo q using the modular PRNG, and interprets them as a monic degree-$2s$ polynomial m over $\mathbb{F}_q$. Also, conversions from integers to algebraic values are done by interpreting the integer as a polynomial over $\mathbb{F}_q$ and reducing it modulo m.

The rest of the arguments are similar as in the integer case. Using the bound of Lemma 6.2 on the density of d-fat polynomials, we obtain a similar theorem as Theorem 4.9: If $\mathcal{A}$ is (k, ϵ)-correct of input $\mathbf{x}$ and $\mathcal{B}(\mathbf{x})$ returns s, the transformed algorithm $\overline{\mathcal{A}}$ produces the correct output for input $\mathbf{x}$ with probability at least

$$\frac{1}{4} - \frac{k}{q^{s+1}} - \frac{\epsilon}{4}.$$

Again, this error probability means the technique is only useful for one-sided Monte Carlo algorithms, or combined with an efficient verification algorithm.

References

[1] Andrew Arnold. *Sparse Polynomial Interpolation and Testing*. PhD thesis, University of Waterloo, 2016. URL `http://hdl.handle.net/10012/10307`. Referenced on pages 4 and 14.

[2] Andrew Arnold, Mark Giesbrecht, and Daniel S. Roche. Faster sparse multivariate polynomial interpolation of straight-line programs. *Journal of Symbolic Computation*, 2015. ISSN 0747-7171. doi: 10.1016/j.jsc.2015.11.005. Referenced on page 4.

[3] Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the Twentieth Annual ACM*

Symposium on Theory of Computing, STOC '88, page 301–309, New York, NY, USA, 1988. Association for Computing Machinery. DOI: 10.1145/62212.62241. Referenced on pages 4 and 14.

[4] Richard P. Brent, Fred G. Gustavson, and David Y. Y. Yun. Fast solution of toeplitz systems of equations and computation of padé approximants. *J. Algorithms*, 1(3): 259–295, 1980. DOI: 10.1016/0196-6774(80)90013-9. Referenced on page 3.

[5] Xavier Dahan, Marc Moreno Maza, Éric Schost, and Yuzhen Xie. On the complexity of the d5 principle. In *Transgressive Computing 2006: A conference in honor of Jean Della Dora*, pages 149–168, 2006. Referenced on pages 2 and 6.

[6] Jean Della Dora, Claire Dicrescenzo, and Dominique Duval. About a new method for computing in algebraic number fields. In Bob F. Caviness, editor, *EUROCAL '85: European Conference on Computer Algebra Linz, Austria, April 1–3 1985 Proceedings Vol. 2: Research Contributions*, pages 289–290, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-540-39685-7. DOI: 10.1007/3-540-15984-3_279. Referenced on pages 2, 6 and 7.

[7] Gemma Maria Diaz-Toca and Henri Lombardi. Dynamic galois theory. *Journal of Symbolic Computation*, 45(12):1316–1329, 2010. DOI: 10.1016/j.jsc.2010.06.012. MEGA'2009. Referenced on page 2.

[8] Jean Louis Dornstetter. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Transactions on Information Theory*, 33(3):428–431, 1987. DOI: 10.1109/TIT.1987.1057299. Referenced on page 15.

[9] Dominique Duval. Rational Puiseux expansions. *Compositio Mathematica*, 70(2): 119–154, 1989. Referenced on page 2.

[10] Dominique Duval and Jean-Claude Reynaud. Sketches and computation–ii: dynamic evaluation and applications. *Mathematical Structures in Computer Science*, 4(2):239–271, 1994. Referenced on page 2.

[11] Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science*, 410(27-29):2659–2662, 2009. ISSN 0304-3975. DOI: 10.1016/j.tcs.2009.03.030. Referenced on pages 4 and 14.

[12] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra (third edition)*. Cambridge University Press, 2013. ISBN 9781107039032. Referenced on page 2.

[13] Pascal Giorgi and Romain Lebreton. Online order basis and its impact on block Wiedemann algorithm. In *Proceedings of the 2014 international symposium on symbolic and algebraic computation*, ISSAC'14, pages 202–209. ACM, 2014. DOI: 10.1145/2608628.2608647. Referenced on page 15.

[14] Elena Grigorescu, Kyomin Jung, and Ronitt Rubinfeld. A local decision test for sparse polynomials. *Information Processing Letters*, 110(20):898–901, 2010. ISSN 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2010.07.012. Referenced on page 14.

[15] Joris van der Hoeven and Grégoire Lecerf. Sparse polynomial interpolation in practice. *ACM Commun. Comput. Algebra*, 48(3/4):187–191, February 2015. DOI: 10.1145/2733693.2733721. Referenced on page 4.

[16] Joris van der Hoeven and Grégoire Lecerf. Directed evaluation. *Journal of Complexity*, 60, 2020. DOI: 10.1016/j.jco.2020.101498. Referenced on pages 2, 3, 6 and 7.

[17] Qiao-Long Huang. Sparse Polynomial Interpolation over Fields with Large or Zero Characteristic. In *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation - ISSAC '19*, pages 219–226, Beijing, China, 2019. ACM Press. DOI: 10.1145/3326229.3326250. Referenced on pages 4 and 14.

[18] Seyed Mohammad Mahdi Javadi and Michael Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, page 160–168, New York, NY, USA, 2010. Association for Computing Machinery. DOI: 10.1145/1837210.1837233. Referenced on page 4.

[19] Marc Joye, Pascal Paillier, and Serge Vaudenay. Efficient generation of prime numbers. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2000*, pages 340–354, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. Referenced on page 2.

[20] Erich Kaltofen. 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*, PASCO '10, pages 10–17, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0067-4. DOI: 10.1145/1837210.1837213. Referenced on page 4.

[21] Erich Kaltofen and Wen-shin Lee. Early termination in sparse interpolation algorithms. *Journal of Symbolic Computation*, 36(3-4):365–400, 2003. ISSN 0747-7171. DOI: 10.1016/S0747-7171(03)00088-9. ISSAC 2002. Referenced on pages 4, 14 and 15.

[22] Erich Kaltofen and Barry M. Trager. Computing with Polynomials Given By Black Boxes for Their Evaluations: Greatest Common Divisors, Factorization, Separation of Numerators and Denominators. *Journal of Symbolic Computation*, 9(3):301–320, 1990. DOI: 10.1016/S0747-7171(08)80015-6. Referenced on pages 4 and 14.

[23] Erich Kaltofen and Lakshman Yagati. Improved sparse multivariate polynomial interpolation algorithms. In P. Gianni, editor, *Proc. ISSAC*, pages 467–474, 1988. DOI: 10.1007/3-540-51084-2_44. Referenced on pages 4 and 14.

[24] Erich Kaltofen, Yagati N. Lakshman, and John-Michael Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '90, pages 135–139, Tokyo, Japan, 1990. ACM Press. DOI: 10.1145/96877.96912. Referenced on page 4.

[25] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987. DOI: 10.1147/rd.312.0249. Referenced on page 5.

[26] Leopold Kronecker. Grundzüge einer arithmetischen theorie der algebraischen grössen. *Journal für die reine und angewandte Mathematik*, 92:1–122, 1882. Referenced on page 14.

[27] Daniel Lazard. A new method for solving algebraic systems of positive dimension. *Discrete Applied Mathematics*, 33(1):147–160, 1991. DOI: 10.1016/0166-218X(91)90113-B. Referenced on page 2.

[28] Ueli M. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, 8(3):123–155, 1995. DOI: 10.1007/BF00202269. Referenced on page 2.

[29] Marc Moreno Maza and Renaud Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 365–382, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. DOI: 10.1007/3-540-60114-7_28. Referenced on page 2.

[30] Robert T. Moenck. Fast computation of gcds. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 142–151. ACM, 1973. DOI: 10.1145/800125.804045. Referenced on page 3.

[31] Vincent Neiger, Bruno Salvy, Éric Schost, and Gilles Villard. Faster modular composition. *CoRR*, abs/2110.08354, 2021. URL `https://arxiv.org/abs/2110.08354`. Referenced on page 3.

[32] Masayuki Noro. Modular dynamic evaluation. In *Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, ISSAC '06, page 262–268, New York, NY, USA, 2006. Association for Computing Machinery. DOI: 10.1145/1145768.1145812. Referenced on page 2.

[33] Adrien Poteaux and Martin Weimann. Computing puiseux series: a fast divide and conquer algorithm. *Annales Henri Lebesgue*, 2021. Referenced on page 2.

[34] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Ill. J. Math.*, 6:64–94, 1962. URL `http://projecteuclid.org/euclid.ijm/1255631807`. Referenced on page 5.

[35] Arnold Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica*, 1(2):139–144, Jun 1971. DOI: 10.1007/BF00289520. Referenced on page 8.

[36] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2008. ISBN 978-0-521-51644-0. Referenced on pages 2, 16 and 17.

[37] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In Duncan Buell, editor, *Algorithmic Number Theory: 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings*, pages 411–425, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. DOI: 10.1007/978-3-540-24847-7_31. Referenced on page 8.

[38] Robert M. Young. 75.9 Euler's Constant. *The Mathematical Gazette*, 75(472): 187–190, 1991. DOI: 10.2307/3620251. Referenced on page 16.