Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator

Alin Bostan, Pierrick Gaudry, Eric Schost

SIAM J. COMPUT.
Vol. 36, No. 6, pp. 1777–1806

LINEAR RECURRENCES WITH POLYNOMIAL COEFFICIENTS AND APPLICATION TO INTEGER FACTORIZATION AND CARTIER–MANIN OPERATOR*

ALIN BOSTAN†, PIERRICK GAUDRY‡, AND ÉRIC SCHOST‡

Abstract. We study the complexity of computing one or several terms (not necessarily consecutive) in a recurrence with polynomial coefficients. As applications, we improve the best currently known upper bounds for factoring integers deterministically and for computing the Cartier–Manin operator of hyperelliptic curves.

Key words. linear recurrences, factorization, Cartier–Manin operator

AMS subject classifications. 11Y16, 68Q25, 11Y05

DOI. 10.1137/S0097539704443793

1. Introduction. We investigate complexity questions for linear recurrent sequences. Our main focus is on the computation of one term, or several terms not necessarily consecutive, in a recurrence with polynomial coefficients. As applications, we improve the deterministic complexity of factoring integers and of computing the Cartier–Manin operator of hyperelliptic curves.

A well-known particular case is that of linear recurrences with constant coefficients. In this case, the Nth term can be computed with a complexity logarithmic in N, using binary powering. In the general case, there is a significant gap, as no algorithm with a complexity polynomial in $(\log N)$ is known. However, Chudnovsky and Chudnovsky showed in [11] how to compute one term in such a sequence without computing all intermediate ones. This algorithm is closely related to Strassen's algorithm [48] for integer factorization; using baby steps/giant steps (BSGS) techniques, it requires a number of operations which are roughly linear in $\sqrt{N}$ to compute the Nth term.

Precisely, let R be a commutative ring with unity and let M (resp., MM) be a function $\mathbb{N} \to \mathbb{N}$ such that polynomials of degree less than d (resp., matrices of size $n \times n$) can be multiplied in $\mathsf{M}(d)$ (resp., $\mathsf{MM}(n)$) operations $(+, -, \times)$; for $x \in \mathbb{R} - \mathbb{N}$, we write $\mathsf{M}(x) = \mathsf{M}(\lceil x \rceil)$. Next let $M(X)$ be an $n \times n$ matrix with entries in $R[X]$ of degree at most 1. Given a vector of initial conditions $U_0 \in R^n$, define the sequence (U_i) of vectors in R^n by the vector recurrence

$$U_{i+1} = M(i+1)U_i \text{ for all } i \geq 0.$$

Then, assuming that $2, \ldots, \lceil \sqrt{N} \rceil$ are units in R, Chudnovsky and Chudnovsky showed that U_N can be computed using

$$O\big(\mathsf{MM}(n)\mathsf{M}(\sqrt{N}) + n^2\mathsf{M}(\sqrt{N})\log N\big)$$

*Received by the editors May 17, 2004; accepted for publication (in revised form) September 14, 2006; published electronically March 22, 2007. A preliminary version of this paper appears in [6]; with the exception of Theorem 5, all results here are new.
http://www.siam.org/journals/sicomp/36-6/44379.html

†Domaine de Voluceau, B.P. 105, E-78153 Le Chesnay Cedex, France (alin.bostan@inria.fr).

‡Laboratoire d'Informatique LIX, École Polytechnique, 91128 Palaiseau Cedex, France (gaudry@lix.polytechnique.fr, schost@lix.polytechnique.fr).

operations. Both terms in this estimate describe basic operations on polynomial matrices of degree $\sqrt{N}$ (resp., multiplication and multipoint evaluation); using FFT-based multiplication, the cost becomes linear in $\sqrt{N}$, up to polylogarithmic factors. Our goal in this paper is to improve, generalize, and give applications of this algorithm.

- We prove that the Nth term in the sequence (U_i) above can be computed in

$$O\big(\mathsf{MM}(n)\sqrt{N} + n^2\,\mathsf{M}\big(\sqrt{N}\big)\big)$$

  operations. Compared to [11], the dependence in n stays the same; however, for fixed n, we save polylogarithmic factors in N. Chudnovsky and Chudnovsky suggested a lower bound of about $\sqrt{N}$ base ring operations for this problem; thus, our improvement gets us closer to the possible optimal bound. Furthermore, in practice, saving such polylogarithmic factors is far from negligible, since in some instances of an application, as detailed below (Cartier–Manin operator computation), N may be of order 2^{32}.
- We give a generalization to the computation of *several* selected terms, which are of indices $N_1 < \cdots < N_r = N$. When the number r of terms to be computed does not exceed $\sqrt{N}$, we show that all of them can be obtained in a time complexity which is the same as above, that is, essentially linear in $\sqrt{N}$, so we are close to the optimal.
- Along the way, we consider a question of basic polynomial arithmetic: Given the values taken by a univariate polynomial P on a large enough set of points, how fast can we compute the values of P on a shift of this set of points? An obvious solution is to use fast interpolation and evaluation techniques, but we show that this can be done faster when the evaluation points form an arithmetic progression.

In all these algorithms, we will consider polynomial matrices with coefficients of degree at most 1, which is quite frequent in applications, e.g., in the two applications presented below. However, this is not a real restriction: the case of coefficients of larger degree can be handled mutatis mutandis at the cost of a more involved presentation.

A first application is the deterministic factorization of integers. To find the prime factors of an integer N, we note that Strassen's algorithm [48] has a complexity of

$$O\big(\mathsf{M}_{\mathrm{int}}(\sqrt[4]{N}\log N)\log N\big)$$

bit operations, where we denote by $\mathsf{M}_{\mathrm{int}}$ a function such that integers of bit-size d can be multiplied in $\mathsf{M}_{\mathrm{int}}(d)$ bit operations (as above, we extend this function to take arguments in $\mathbb{R}$). Chudnovsky and Chudnovsky's algorithm generalizes Strassen's; thus, our modifications apply here as well. We prove that there exists a deterministic algorithm that outputs the complete factorization of an integer N with a bit complexity in

$$O\big(\mathsf{M}_{\mathrm{int}}(\sqrt[4]{N}\log N)\big).$$

To our knowledge, this gives the fastest deterministic integer factorization algorithm. Prior to Strassen's work, the record was held by Pollard's algorithm [35]; for any $\delta > 0$, its bit complexity is in $O(\mathsf{M}_{\mathrm{int}}(\sqrt[4]{N}\log N)N^{\delta})$. Other deterministic factorization algorithms exist [36, 30]; some have a better conjectured complexity, whose validity relies on unproved number-theoretic conjectures. The fastest probabilistic algorithm for integer factorization, with a fully established complexity bound, is due to Lenstra, Jr. and Pomerance [27], with a bit complexity polynomial in

$\exp(\sqrt{\log N \log\log N})$. The number field sieve [26] has a better conjectured complexity, expected to be polynomial in $\exp(\sqrt[3]{\log N(\log\log N)^2})$.

In accordance with these estimates, the latter algorithms are better suited for practical computations than our deterministic variant, and all recent record-sized computations rely on the number field sieve. However, as already pointed out by Pollard [35], proving unconditional, deterministic upper bounds remains an important challenge.

Our second application is point-counting in cryptography, related to the computation of the Cartier–Manin operator [10, 28] of hyperelliptic curves over finite fields.

The basic ideas already appear for elliptic curves [44, Chapter V]. Suppose we are to count the number n of solutions of the equation $y^2 = f(x)$ over $\mathbb{F}_p$, where $p > 2$ is prime and f has degree 3. Let $\chi : \mathbb{F}_p \to \mathbb{F}_p$ be the map $x \mapsto x^{(p-1)/2}$. For $x \neq 0$, $\chi(x) = 1$ when x is a square, and $\chi(x) = -1$ otherwise. Hence, n equals $\sum_{x\in\mathbb{F}_p} \chi(f(x))$ modulo p. For $i \neq 0$, $\sum_{x\in\mathbb{F}_p} x^i$ equals -1 if $p-1$ divides i, and 0 otherwise; one deduces that n modulo p is the opposite of the coefficient of x^{p-1} in $f(x)^{(p-1)/2}$.

Generalizing these ideas to hyperelliptic curves leads to the notions of the Hasse–Witt matrix and Cartier–Manin operator. Using a result of Manin [28], the Hasse–Witt matrix can be used as part of a point-counting procedure. As above, for hyperelliptic curves given by an equation $y^2 = f(x)$, the entries of this matrix are coefficients of $h = f^{(p-1)/2}$.

The coefficients of h satisfy a linear recurrence with rational function coefficients. Using our results on linear recurrences, we deduce an algorithm to compute the Hasse–Witt matrix whose complexity is essentially linear in $\sqrt{p}$. For instance, in a fixed genus, for a curve defined over the finite field $\mathbb{F}_p$, the complexity of our algorithm is

$$O\big(\mathsf{M}_{\mathrm{int}}(\sqrt{p}\log p)\big)$$

bit operations. This improves the methods of [18] and [29] which have a complexity essentially linear in p. Note that when p is small enough, other methods, such as the p-adic methods used in Kedlaya's algorithm [24], also provide very efficient point-counting procedures, but their complexity is at least linear in p; see [17].

Main algorithmic ideas. We briefly recall Strassen's factorization algorithm and Chudnovsky and Chudnovsky's generalization, and describe our modifications.

To factor an integer N, trial division with all integers smaller than $\sqrt{N}$ has a cost linear in $\sqrt{N}$. To do better, Strassen proposed to group all integers smaller than $\sqrt{N}$ into c blocks of c consecutive integers, where $c \in \mathbb{N}$ is of order $\sqrt[4]{N}$. Write $f_0 = 1\cdots c \mod N$, $f_1 = (c+1)\cdots(2c) \mod N, \ldots, f_{c-1} = (c^2-c+1)\cdots(c^2) \mod N$. If the values $f_0, \ldots, f_{c-1}$ can be computed efficiently, then finding a prime factor of N becomes easy, using the gcd's of $f_0, \ldots, f_{c-1}$ with N. Thus, the main difficulty lies in computing the values f_i, whose cost will actually dominate the whole complexity.

To perform this computation, let $R = \mathbb{Z}/N\mathbb{Z}$ and let F be the polynomial $(X+1)\cdots(X+c) \in R[X]$. The "baby steps" part of the algorithm consists of computing F: using the subproduct tree algorithm [15, Chapter 10], this is done in $O(\mathsf{M}(c)\log c)$ operations in R. Then, the "giant steps" consist of evaluating F at $0, c, \ldots, (c-1)c$, since $F(ic) = f_i$. Using fast evaluation, these values can be computed in $O(\mathsf{M}(c)\log c)$ operations. Since c has order $\sqrt[4]{N}$, the whole process has a complexity of $O(\mathsf{M}(\sqrt[4]{N})\log N)$ operations in R. This is the core of Strassen's factorization algorithm; working out the complexity estimates in a boolean complexity model yields the bounds given before.

Independently of the factorization question, one sees that multiplying the values $f_0, f_1, \ldots, f_{c-1}$ yields the product $1 \cdots c^2$ modulo N. Thus, this algorithm can be used to compute factorials: the analysis above shows that in any ring R, for any $N \geq 0$, the product $1 \cdots N$ can be computed in $O(\mathsf{M}(\sqrt{N}) \log N)$ operations in R. Note the improvement obtained over the naive iterative algorithm, whose complexity is linear in N.

Now, the sequence $U_N = N!$ is a basic example of a solution of a linear recurrence with polynomial coefficients, namely $U_N = NU_{N-1}$. Chudnovsky and Chudnovsky thus generalized the above BSGS process to compute the Nth term of an $n \times n$ matrix recurrence with polynomial coefficients of degree at most 1. The main tasks are the same as above. Computing the matrix equivalent of the polynomial F can be done using $O(\mathsf{MM}(n)\mathsf{M}(\sqrt{N}))$ operations if $2, \ldots, \lceil \sqrt{N} \rceil$ are units in R, and $O(\mathsf{MM}(n)\mathsf{M}(\sqrt{N}) \log N)$ otherwise. Then, the subsequent evaluation can be done using $O(n^2 \mathsf{M}(\sqrt{N}) \log N)$ operations. This gives the complexity estimate mentioned before.

Let us now describe our approach for the factorial (the matrix case is similar). We are not interested in the coefficients of the polynomial F, but in its values on suitable points. Now, both F and the evaluation points have special structures: F is the product of $(X+1)$, $(X+2)$, $\ldots$, $(X+c)$, whereas the evaluation points form the arithmetic progression $0, c, \ldots, (c-1)c$. This enables us to reduce the cost of evaluating F from $O(\mathsf{M}(c) \log c)$ to $O(\mathsf{M}(c))$. We use a divide-and-conquer approach; the recursive step consists of evaluating a polynomial akin to F, with degree halved, on an arithmetic progression of halved size. Putting this idea into practice involves the following operation, which is central to all our algorithms: Given the values of a polynomial P on an arithmetic progression, compute the values of P on a shift of this arithmetic progression.

In the general case of an $n \times n$ matrix recurrence, our fast solution to this problem will enable us to dispense completely with polynomial matrix multiplications, and to reduce by a logarithmic factor all costs related to multipoint evaluation. However, it will impose suitable invertibility conditions in R; we will pay special attention to such conditions, since in our two applications the base ring contains zero-divisors.

Organization of the paper. Section 2 introduces notation and previous results. Section 3 gives our algorithm for shifting a polynomial given by its values on an arithmetic progression; it is used in section 4 to evaluate some polynomial matrices, with an application in section 5 to integer factorization. In section 6, we give our modification on Chudnovsky and Chudnovsky's algorithm for computing one term in a recurrence with polynomial coefficients; a generalization to several terms is given in section 7. In section 8, we apply these results to the computation of the Cartier–Manin operator.

2. Notation and basic results. We use two computational models. Our algorithms for linear recurrent sequences apply over arbitrary rings, so their complexity is expressed in an algebraic model, counting at unit cost the base ring operations. Algorithms for integer factorization and Cartier–Manin operator computation require us to count bit operations: for this purpose, we will use the multitape Turing machine model.

Our algorithms use BSGS techniques. As usual with such algorithms, the memory requirements essentially follow the time complexities (whereas naive iterative algorithms for linear recurrences run in constant memory). We will thus give memory estimates for all our algorithms.

In what follows, $\log x$ is the base-2 logarithm of x; $\lfloor x \rfloor$ and $\lceil x \rceil$ denote, respectively, the largest integer less than or equal to x, and the smallest integer larger than or equal to x.

To make some expressions below well defined, if f is defined as a map $\mathbb{N} \to \mathbb{N}$ we may implicitly extend it to a map $\mathbb{R} \to \mathbb{N}$ by setting $f(x) = f(\lceil x \rceil)$ for $x \in \mathbb{R} - \mathbb{N}$.

All our rings will be commutative and unitary. If R is such a ring, the map $\mathbb{N} \to R$ sending n to $1 + \cdots + 1$ (n times) extends to a map $\varphi : \mathbb{Z} \to R$; we will still denote by $n \in R$ the image $\varphi(n)$.

2.1. Algebraic complexity model. The algorithms of sections 3, 4, 6, and 7 apply over arbitrary rings. To give complexity estimates, we use the straight-line program model, counting at unit cost the operations $(+, -, \times)$ in the base ring; see [8]. Hence, the time complexity of an algorithm is the size of the underlying straight-line program; we will simply speak of "ring operations." Branching and divisions are not used; thus, if we need the inverses of some elements, they will be given as inputs to the algorithm.

To assign a notion of space complexity to a straight-line program, we play a pebble game on the underlying directed acyclic graph; see [3] for a description. However, we will not use such a detailed presentation: we will simply speak of the number of ring elements that have to be stored, or of "space requirements"; such quantities correspond to the number of pebbles in the underlying pebble game.

Basic operations. Let R be a ring. The following lemma (see [32] and [33, p. 66]) shows how to trade inversions (when they are possible) for multiplications.

LEMMA 1. *Let $r_0, \ldots, r_d$ be units in R. Given $(r_0 \cdots r_d)^{-1}$, one can compute $r_0^{-1}, \ldots, r_d^{-1}$ in $O(d)$ operations and space $O(d)$.*

Proof. We first compute $R_0 = r_0$, $R_1 = r_0 r_1, \ldots, R_d = r_0 r_1 \cdots r_d$ in d multiplications. The inverse of R_d is known; by d more multiplications we deduce $S_d = R_d^{-1}$, $S_{d-1} = r_d S_d, \ldots, S_0 = r_1 S_1$, so that S_i equals $(r_0 \cdots r_i)^{-1}$. We obtain the inverse s_i of r_i by computing $s_0 = S_0$, $s_1 = R_0 S_1, \ldots, s_d = R_{d-1} S_d$ for d additional operations. ∎

In what follows, we need to compute some constants in R. For $i, d \in \mathbb{N}$, and $a \in R$, set

$$\delta(i,d) = \prod_{j=0,\ j \neq i}^{d} (i-j) \quad \text{and} \quad \Delta(a,i,d) = \prod_{j=0}^{d} (a+i-j). \tag{1}$$

Then, we have the following results.

LEMMA 2. *Suppose that $2, \ldots, d$ are units in R. Given their inverses, one can compute the inverses of $\delta(0,d), \ldots, \delta(d,d)$ in $O(d)$ operations and space $O(d)$.*

Suppose that $a-d, \ldots, a-1$ are units in R. Given their inverses, one can compute $\Delta(a,0,d), \ldots, \Delta(a,d,d)$ in $O(d)$ operations and space $O(d)$.

Proof. We use the following formulas, where i ranges from 1 to d:

$$\frac{1}{\delta(0,d)} = \frac{1}{\prod_{j=1}^{d}(-j)}, \qquad \frac{1}{\delta(i,d)} = \frac{i-d-1}{i} \frac{1}{\delta(i-1,d)},$$

$$\Delta(a,0,d) = \prod_{j=0}^{d} (a-j), \qquad \Delta(a,i,d) = \frac{a+i}{a+i-d-1} \Delta(a,i-1,d). \qquad ∎$$

Algorithms for polynomials. We denote by $\mathsf{M} : \mathbb{N} - \{0\} \to \mathbb{N}$ a function such that over any ring, the product of polynomials of degree less than d can be computed in $\mathsf{M}(d)$ ring operations. Using the algorithms of [41, 39, 9], $\mathsf{M}(d)$ can be taken in $O(d \log d \log\log d)$. Following [15, Chapter 8], we suppose that for all d and d', M satisfies the inequalities

$$\frac{\mathsf{M}(d)}{d} \leq \frac{\mathsf{M}(d')}{d'} \quad \text{if } d \leq d' \quad \text{and} \quad \mathsf{M}(dd') \leq d^2\mathsf{M}(d'), \tag{2}$$

and that the product in degree less than d can be computed in space $O(d)$. These assumptions are satisfied for naive, Karatsuba, and Schönhage–Strassen multiplications. The first inequality implies that $\mathsf{M}(d) + \mathsf{M}(d') \leq \mathsf{M}(d + d')$ and that $d \leq \mathsf{M}(d)$; the second one is used to derive the inclusion $\mathsf{M}(O(d)) \subset O(\mathsf{M}(d))$.

We use the following results for arithmetic over a ring R. The earliest references we know of are [22, 31, 47, 4], and [7] gives more recent algorithms.

Evaluation. If P is in $R[X]$, of degree at most d, and $r_0, \ldots, r_d$ are in R, then $P(r_0), \ldots, P(r_d)$ can be computed in time $O(\mathsf{M}(d) \log d)$ and space $O(d \log d)$. Using the algorithm of [16, Lemma 2.1], space can be reduced to $O(d)$, but this will not be used here.

Interpolation. For simplicity, we consider only interpolation at $0, \ldots, d$. Suppose that $2, \ldots, d$ are units in R; given their inverses, from the values $P(0), \ldots, P(d)$, one can recover the coefficients of P using $O(\mathsf{M}(d) \log d)$ operations, in space $O(d \log d)$.

See the appendix for a description of the underlying algorithmic ideas.

Matrix multiplication. We denote by $\mathsf{MM} : \mathbb{N} - \{0\} \to \mathbb{N}$ a function such that the product of $n \times n$ matrices over any ring can be computed in $\mathsf{MM}(n)$ base ring operations, in space $O(n^2)$. Thus, one can take $\mathsf{MM}(n) \in O(n^3)$ using classical multiplication, and $\mathsf{MM}(n) \in O(n^{\log 7}) \subset O(n^{2.81})$ using Strassen's algorithm [46]. We do not know whether the current record estimate [13] of $O(n^{2.38})$ satisfies our requirements. Note that $n^2 \leq \mathsf{MM}(n)$; see [8, Chapter 15].

2.2. Boolean complexity model. In sections 5 and 8, we discuss the complexity of factoring integers and of computing the Cartier–Manin operator on curves over finite fields. For these applications, the proper complexity measure is bit complexity. For this purpose, our model will be the multitape Turing machine; see, for instance, [40]. We will speak of *bit operations* to estimate time complexities in this model. Storage requirements will be expressed in bits as well, taking into account input, output, and intermediate data size.

Boolean algorithms will be given through high-level descriptions, and we shall not give the details of their multitape Turing implementations. We just mention the following relevant fact: for each algorithm, there is a corresponding multitape Turing machine. Using previously designed algorithms as subroutines is then possible; each of the corresponding machines is attributed a special band that plays the role of a stack to handle subroutine calls. We refer to [40] for examples of detailed descriptions along these lines.

Integer operations. Integers are represented in base 2. The function $\mathsf{M}_{\text{int}} : \mathbb{N} \to \mathbb{N}$ is such that the product of two integers of bit-size d can be computed within $\mathsf{M}_{\text{int}}(d)$ bit operations. Hence, multiplying integers bounded by N takes at most $\mathsf{M}_{\text{int}}(\log N)$ bit operations.

We suppose that M_{int} satisfies inequalities (2), and that product in bit-size d can be done in space $O(d)$. Using the algorithm of [41], $\mathsf{M}_{\text{int}}(d)$ can be taken in

$O(d \log d \log\log d)$. Euclidean division in bit-size d can be done in time $O(\mathsf{M}_{\mathrm{int}}(d))$ and space $O(d)$; see [12]. The extended gcd of two bit-size d integers can be computed in time $O(\mathsf{M}_{\mathrm{int}}(d)\log d)$ and space $O(d)$; see [25, 38].

Effective rings. We next introduce effective rings as a way to obtain results of a general nature in the Turing model.

Let R be a finite ring, let ℓ be in $\mathbb{N}$, and consider an injection $\sigma : R \hookrightarrow \{0,1\}^\ell$. We use σ to represent the elements of R. Polynomials in $R[X]$ are represented by the sequence of the σ-values of their coefficients. Matrices over R are represented in *row-major ordering*: an $m \times n$ matrix $A = (a_{i,j})$ is represented as $\sigma(a_{1,1}), \ldots, \sigma(a_{1,n}), \ldots, \sigma(a_{m,1}), \ldots, \sigma(a_{m,n})$.

An *effective ring* is the data of such R, ℓ, and σ, together with constants $\mathsf{m}_R, \mathsf{s}_R \in \mathbb{N}$, and maps $\mathsf{M}_R, \mathsf{S}_R$ and $\mathsf{MM}_R, \mathsf{SM}_R : \mathbb{N} - \{0\} \to \mathbb{N}$ meeting the following criteria. First, through the σ representation, we ask that

- the sum and product of elements in R can be computed in time $\mathsf{m}_R \geq \ell$ and space $\mathsf{s}_R \geq \ell$;
- the product of polynomials of degree less than d in $R[X]$ can be computed in time $\mathsf{M}_R(d)$ and space $\mathsf{S}_R(d)$;
- the product of size n matrices over R can be computed in time $\mathsf{MM}_R(n)$ and space $\mathsf{SM}_R(n)$.

We ask that for all d and d', M_R satisfies the inequalities (2),

$$\frac{\mathsf{M}_R(d)}{d} \leq \frac{\mathsf{M}_R(d')}{d'} \quad \text{if } d \leq d' \quad \text{and} \quad \mathsf{M}_R(dd') \leq d^2 \mathsf{M}_R(d'),$$

as well as $d\mathsf{m}_R \leq \mathsf{M}_R(d)$. We also ask that, for all d and d', S_R satisfies

$$\mathsf{s}_R \leq \mathsf{S}_R(d) \quad \text{and} \quad \mathsf{S}_R(dd') \leq d^2 \mathsf{S}_R(d').$$

Finally, as to matrix multiplication, we require that for all n, MM_R and SM_R satisfy

$$n^2 \mathsf{m}_R \leq \mathsf{MM}_R(n) \quad \text{and} \quad \mathsf{s}_R \leq \mathsf{SM}_R(n).$$

In what follows, our results will be first given in the algebraic model, and then on an effective ring, with a bit complexity estimate; note that for both algebraic and Turing models, all constants hidden in the $O(\ )$ estimates will be independent of the base ring.

Effective rings will enable us to state bit complexity results similar to algebraic complexity ones. We have, however, no general transfer theorem from algebraic to bit complexity. First, nonarithmetic operations (loop handling, stack managing for recursive calls) are not taken into account in the former model. In most cases however, the corresponding cost is easily seen to be negligible, so we will not spend time discussing this. A more important difference is that the algebraic model does not count time to access data, that is, the number of tape movements done in the Turing model. This point will be checked for the algorithms we will discuss on Turing machines.

For concrete applications, the following lemma, proved in the appendix, gives the basic examples of effective rings. The results for matrix multiplication are not the sharpest possible, since this would take us too far afield.

LEMMA 3. *Let N be in $\mathbb{N}$, let $R_0 = \mathbb{Z}/N\mathbb{Z}$, and let P be monic of degree m in $R_0[T]$. Then $R = R_0[T]/P$ can be made an effective ring, with*

- $\ell = m\lceil \log N \rceil$,
- $\mathsf{m}_R \in O(\mathsf{M}_{\mathrm{int}}(m \log(mN)))$ *and* $\mathsf{s}_R \in O(m \log(mN))$,

- $\mathsf{M}_R(d) \in O(\mathsf{M}_{\mathrm{int}}(dm\log(dmN)))$ *and* $\mathsf{S}_R(d) \in O(dm\log(dmN))$,
- $\mathsf{MM}_R(n) \in O(n^{\log 7}\mathsf{m}_R)$ *and* $\mathsf{SM}_R(n) \in O(n^2\ell + \mathsf{s}_R)$.

Finally, the results given before in the algebraic model have the following counterpart in the Turing model, using again the notation $\delta(i,d)$ and $\Delta(a,i,d)$ introduced in (1). The proofs are given in the appendix.

LEMMA 4. *Let R be an effective ring. Then the following hold:*

1. *Suppose that $r_0,\dots,r_d$ are units in R. Given $r_0,\dots,r_d,(r_0\cdots r_d)^{-1}$, one can compute $r_0^{-1},\dots,r_d^{-1}$ in time $O(d\mathsf{m}_R)$ and space $O(d\ell+\mathsf{s}_R)$.*
2. *Suppose that $2,\dots,d$ are units in R. Given their inverses, one can compute the inverses of $\delta(0,d),\dots,\delta(d,d)$ in time $O(d\mathsf{m}_R)$ and space $O(d\ell+\mathsf{s}_R)$.*
3. *Suppose that $a-d,\dots,a-1$ are units in R. Given their inverses, one can compute $\Delta(a,0,d),\dots,\Delta(a,d,d)$ in time $O(d\mathsf{m}_R)$ and space $O(d\ell+\mathsf{s}_R)$.*
4. *Let $P=\sum_{i=0}^d p_iX^i$ be in $R[X]$. Given $p_0,\dots,p_d$ and elements $r_0,\dots,r_d$ in R, $P(r_0),\dots,P(r_d)$ can be computed in time $O(\mathsf{M}_R(d)\log d)$ and space $O(\ell d\log d+\mathsf{S}_R(d))$.*
5. *Suppose that $2,\dots,d$ are units in R. If $P\in R[X]$ has degree at most d, then given $P(0),\dots,P(d)$ and the inverses of $2,\dots,d$, one can compute the coefficients of P in time $O(\mathsf{M}_R(d)\log d)$ and space $O(\ell d\log d+\mathsf{S}_R(d))$.*

3. Shifting evaluation values. We now address a special case of the question of *shifting evaluation values* of a polynomial. Let R be a ring, let P be of degree d in $R[X]$, and let a and $r_0,\dots,r_d$ be in R. Given $P(r_0),\dots,P(r_d)$, how fast can we compute $P(r_0+a),\dots,P(r_d+a)$? We stress the fact that the coefficients of P are *not* part of the input.

Suppose that all differences r_i-r_j, $i\neq j$ are units in R. Then using fast interpolation and evaluation, the problem can be solved using $O(\mathsf{M}(d)\log d)$ operations in R. We propose an improved solution, in the special case when $r_0,\dots,r_d$ form an arithmetic progression; its complexity is in $O(\mathsf{M}(d))$, so we gain a logarithmic factor.

Our algorithm imposes invertibility conditions on the sample points slightly more general than those above. Given α,β in R and d in $\mathbb{N}$, we define the following property: $\mathsf{h}(\alpha,\beta,d)$**:** β, $2,\dots,d$ and $\alpha-d\beta$, $\alpha-(d-1)\beta,\dots,\alpha+(d-1)\beta$, $\alpha+d\beta$ are units. We then define $\mathsf{d}(\alpha,\beta,d)=\beta\cdot 2\cdots d\cdot(\alpha-d\beta)\cdots(\alpha+d\beta)\in R$. Assumption $\mathsf{h}(\alpha,\beta,d)$ holds if and only if $\mathsf{d}(\alpha,\beta,d)$ is a unit.

THEOREM 5. *Let α,β be in R and d be in $\mathbb{N}$ such that $\mathsf{h}(\alpha,\beta,d)$ holds, and suppose that the inverse of $\mathsf{d}(\alpha,\beta,d)$ is known. Let F be in $R[X]$ of degree at most d and $r\in R$. Given*

$$F(r),F(r+\beta),\dots,F(r+d\beta),$$

one can compute

$$F(r+\alpha),F(r+\alpha+\beta),\dots,F(r+\alpha+d\beta),$$

in time $2\mathsf{M}(d)+O(d)$ and space $O(d)$, in the algebraic model. If R is effective, then the bit complexity is $2\mathsf{M}_R(d)+O(d\mathsf{m}_R)$ and the space complexity is $O(\mathsf{S}_R(d))$ bits.

Proof. Our algorithm reduces to the multiplication of two suitable polynomials of degrees at most d and $2d$; $O(d)$ additional operations come from pre- and post-processing operations. All operations below on integer values take place in R.

First, we perform a change of variables. Define $P(X)=F(\beta X+r)$; then our assumption is that the values $P(0),P(1),\dots,P(d)$ are known. Let us write $a=\alpha/\beta$;

our objective is then to determine the values $P(a), \dots, P(a+d)$. To this effect, assumption $\mathsf{h}(\alpha, \beta, d)$ enables us to write the Lagrange interpolation formula:

$$P = \sum_{i=0}^{d} P(i) \frac{\prod_{j=0, j\neq i}^{d} (X-j)}{\prod_{j=0, j\neq i}^{d} (i-j)} = \sum_{i=0}^{d} \widetilde{P}_i \prod_{j=0, j\neq i}^{d} (X-j),$$

with $\widetilde{P}_i = P(i)/\delta(i,d)$, where $\delta(i,d)$ is defined in (1). For k in $0, \dots, d$, let us evaluate P at $a+k$:

$$P(a+k) = \sum_{i=0}^{d} \widetilde{P}_i \prod_{j=0, j\neq i}^{d} (a+k-j).$$

Assumption $\mathsf{h}(\alpha, \beta, d)$ implies that $a-d, \dots, a+d$ are units. We can thus complete each product by the missing factor $a+k-i$:

$$P(a+k) = \sum_{i=0}^{d} \widetilde{P}_i \frac{\prod_{j=0}^{d} (a+k-j)}{a+k-i} = \left(\prod_{j=0}^{d} (a+k-j) \right) \cdot \left(\sum_{i=0}^{d} \widetilde{P}_i \frac{1}{a+k-i} \right). \tag{3}$$

We now use the sequence $\Delta(a,k,d)$ introduced in (1) and define $Q_k = P(a+k)/\Delta(a,k,d)$. Using these values, (3) reads

$$Q_k = \sum_{i=0}^{d} \widetilde{P}_i \frac{1}{a+k-i}. \tag{4}$$

Let $\widetilde{P}$ and S be the polynomials

$$\widetilde{P} = \sum_{i=0}^{d} \widetilde{P}_i X^i, \quad S = \sum_{i=0}^{2d} \frac{1}{a+i-d} X^i;$$

then by (4), for $k = 0, \dots, d$, Q_k is the coefficient of degree $k+d$ in the product $\widetilde{P}S$. From the knowledge of Q_k, we easily deduce $P(a+k)$.

Let us analyze the complexity of this algorithm, first in the algebraic model. Using Lemma 1, from the inverse of $\mathsf{d}(\alpha, \beta, d)$, we obtain those of β, $2, \dots, d$ and $\alpha - d\beta, \dots, \alpha + d\beta$ in $O(d)$ operations. Using the equality $(a+id)^{-1} = \beta(\alpha + id\beta)^{-1}$, we obtain the inverses of $a-d, \dots, a+d$ in $O(d)$ further operations. Lemma 2 then gives all $\Delta(a,i,d)$ and the inverses of all $\delta(i,d)$ for $O(d)$ operations as well. The sequence $\widetilde{P}_i$ is deduced for $O(d)$ operations.

The coefficients Q_i are then obtained by a polynomial multiplication in degrees d and $2d$; this can be reduced to two polynomial multiplications in degrees less than d, and $O(d)$ additional operations, for a complexity of $2\mathsf{M}(d) + O(d)$. Given $Q_0, \dots, Q_d$, we deduce $P(a), \dots, P(a+d)$ by multiplications with the coefficients $\Delta(a,i,d)$; this requires $O(d)$ ring operations. This concludes the algebraic complexity estimates, since space requirements are easily seen to be in $O(d)$.

When R is effective, we have to implement this algorithm on a multitape Turing machine. For this simple algorithm, there is no difficulty; we give details to show the manipulations that need to be made, making no effort to minimize the number of tapes. For the next algorithms, we will be more sketchy and concentrate on difficult points.

Initially, $P(0), \ldots, P(d)$ and the inverse of $\mathsf{d}(\alpha, \beta, d)$ are contiguous blocks of ℓ bits on the input tape. First, we produce on an auxiliary tape T_1 all elements whose inverses will be used, *in a suitable order*, namely β, $2, \ldots, d, \alpha - d\beta, \ldots, \alpha + d\beta$. Then the inverse of $\mathsf{d}(\alpha, \beta, d)$ is appended to these elements; using Lemma 4, we obtain β^{-1}, $2^{-1}, \ldots, d^{-1}$ and $(\alpha - d\beta)^{-1}, \ldots, (\alpha + d\beta)^{-1}$ on a tape T_2. As before, we deduce $(a-d)^{-1}, \ldots, (a+d)^{-1}$; this is done is a single sweep of T_2, and the results are stored on a tape T_3. Then, using Lemma 4, all $\delta(i, d)^{-1}$ are computed and stored on a tape T_4, and all $\Delta(a, i, d)$ on a tape T_5. The whole cost up to now is $O(d\mathsf{m}_R)$, the cost of the tape movements being $O(d\ell)$. The space complexity is in $O(d\ell + \mathsf{s}_R)$.

The coefficients of S are copied from T_3 to a tape T_6, and those of $\widetilde{P}$ are computed and stored on a tape T_7; the cost is $O(d\mathsf{m}_R)$, since the data are well organized on tapes. We then compute the product of S and $\widetilde{P}$. The result is the list of coefficients Q_k, stored on a tape T_8 after a time $2\mathsf{M}_R(d) + O(d\mathsf{m}_R)$. Finally the target values $P(a+k)$ are computed at an additional cost of $O(d\mathsf{m}_R)$, since again everything is well organized. This concludes the time analysis. The space complexity is easily seen to fit the required bound. □

Remark 1. In [20], the operation called *middle product* is defined: Given a ring R, and A, B in $R[X]$ of respective degrees at most d and $2d$, write $AB = C_0 + C_1 X^{d+1} + C_2 X^{2d+2}$, with all C_i of degree at most d; then the middle product of A and B is the polynomial C_1. This is precisely what is needed in the algorithm above.

Up to considering the reciprocal polynomial of A, the middle product by A can be seen as the transpose of the map of multiplication by A. General program transformation techniques then show that it can be computed in time $\mathsf{M}(d) + O(d)$ (but with a possible loss in space complexity): this is the *transposition principle* for linear algorithms, which is an analogue of results initiated by Tellegen [49] and Bordewijk [2] in circuit theory. Thus, the time complexity of the algorithm above can be reduced to $\mathsf{M}(d) + O(d)$ ring operations, but possibly with an increased space complexity. We refer to [23, Problem 6] for a longer discussion and [8, Theorem 13.20] for a proof; see also [20] for the independent discovery of the middle product, and [7] for additional applications.

Remark 2. Using the notation of the proof above, we mention an alternative $O(\mathsf{M}(d))$ algorithm which does *not* require any invertibility assumption, in the special case when $a = d + 1$. The key fact is that for any polynomial P of degree d, the sequence $P(0), P(1), \ldots$ is linearly recurrent, of characteristic polynomial $Q(X) = (1 - X)^{d+1}$. Thus, if the first terms $P(0), \ldots, P(d)$ are known, the next $d + 1$ terms $P(d + 1), \ldots, P(2d + 1)$ can be recovered in $O(\mathsf{M}(d))$ using the algorithm in [43, Theorem 3.1].

4. Algorithms for polynomial matrix evaluation. In Strassen's algorithm sketched in the introduction, an important part of the effort lies in evaluating polynomials on points that form an arithmetic progression. A generalization of this question appears in Chudnovsky and Chudnovsky's algorithm for matrix recurrences, where one has to evaluate a polynomial matrix at points in an arithmetic progression. We now present such an evaluation algorithm, in the special case when the polynomial matrix has the form

$$M_k(X) = M(X + \alpha k) \cdots M(X + \alpha),$$

where $M(X)$ is a given $n \times n$ polynomial matrix with entries of degree at most 1: this is enough to handle both Strassen's and Chudnovsky and Chudnovsky's algorithms. Using the result of the previous section, we propose a divide-and-conquer approach,

which, for fixed n, saves a logarithmic factor in k compared to classical multipoint evaluation techniques.

Let R be a ring. We will need several invertibility assumptions in R, in order to apply Theorem 5 along all recursive steps of the algorithm; we discuss this first. With a positive integer k, we associate the sequence $k_0, \dots, k_{\lfloor \log k \rfloor}$ defined by $k_0 = k$ and $k_{i+1} = \lfloor k_i/2 \rfloor$, so that $k_{\lfloor \log k \rfloor} = 1$.

Then, given α, β in R and k in $\mathbb{N}$, we say that assumption $\mathsf{H}(\alpha, \beta, k)$ holds if assumptions $\mathsf{h}(\beta(k_i+1), \beta, k_i)$ and $\mathsf{h}(\alpha k_i, \beta, k_i)$ of the previous section hold for $i = 1, \dots, \lfloor \log k \rfloor$: this is what we need for the algorithm below. We write $\mathsf{D}(\alpha, \beta, k)$ for the product

$$\prod_{i=1}^{\lfloor \log k \rfloor} \mathsf{d}(\beta(k_i+1), \beta, k_i)\, \mathsf{d}(\alpha k_i, \beta, k_i);$$

note that $\mathsf{H}(\alpha, \beta, k)$ holds if and only if $\mathsf{D}(\alpha, \beta, k)$ is a unit in R. We mention a few basic results related to this definition; the straightforward proofs are left to the reader.

LEMMA 6. *Given α, β, and k, $\mathsf{D}(\alpha, \beta, k)$ can be computed in time and space $O(k)$, in the algebraic model. If R is effective, this can be done in time $O(k\mathsf{m}_R)$ and space $O(k\ell + \mathsf{s}_R)$.*

Condition $\mathsf{H}(\alpha, \beta, k)$ asserts that $O(k)$ elements are units in R. It is easy, but cumbersome, to give the list of these elements. It will be enough to note the following particular cases.

LEMMA 7.

- $\mathsf{H}(k, 1, k)$ *holds if and only if* $2, \dots, 2k_i+1$ *and* $kk_i - k_i, \dots, kk_i + k_i$ *are units in R, for* $i = 1, \dots, \lfloor \log k \rfloor$.
- $\mathsf{H}(1, 2^s, 2^s)$ *holds if and only if* $2, \dots, 2^s + 1$ *are units in R.*

We can now state the main result of this section.

THEOREM 8. *Suppose that $\mathsf{H}(\alpha, \beta, k)$ holds and that the inverse of $\mathsf{D}(\alpha, \beta, k)$ is known. Then the scalar matrices $M_k(0), M_k(\beta), \dots, M_k(k\beta)$ can be computed in*

$$O(\mathsf{MM}(n)k + n^2\mathsf{M}(k))$$

ring operations, in space $O(n^2k)$. If R is effective, then the bit complexity is

$$O\big(\mathsf{MM}_R(n)k + n^2\mathsf{M}_R(k) + n^2\ell k \min(\log k, \log n)\big),$$

and the space complexity is $O(n^2k\ell + \mathsf{S}_R(k) + \mathsf{SM}_R(n))$ bits.

Proof. We first deal with inverses. Let $k_0, \dots, k_{\lfloor \log k \rfloor}$ be defined as above. In the following we need the inverses of all

$$\mathsf{d}(\alpha k_i, \beta, k_i) \quad \text{and} \quad \mathsf{d}(\beta(k_i+1), \beta, k_i) \quad \text{for } i = 1, \dots, \lfloor \log k \rfloor.$$

For any i, both $\mathsf{d}(\alpha k_i, \beta, k_i)$ and $\mathsf{d}(\beta(k_i+1), \beta, k_i)$ can be computed in $O(k_i)$ ring operations; hence, all of them can be computed in $O(k)$ operations. Using Lemma 1, their inverses can be deduced from that of $\mathsf{D}(\alpha, \beta, k)$ for $O(\log k)$ products.

We will then give an estimate on the complexity of computing the values of $M_{k_i}(X)$ on $0, \beta, \dots, k_i\beta$, for decreasing values of i. The case $i = \lfloor \log k \rfloor$ is obvious, since then $k_i = 1$ and $M_{k_i}(X) = M(X + \alpha)$, which can be evaluated at 0 and β in $O(n^2)$ operations.

Then, for some $i = \lfloor \log k \rfloor, \dots, 1$, suppose that the values of $M_{k_i}(X)$ are known on $0, \beta, \dots, k_i\beta$. We now show how to deduce the values of $M_{k_{i-1}}(X)$ on $0, \beta, \dots, k_{i-1}\beta$.

To this effect, we will use Theorem 5, using the fact that all entries of $M_{k_i}(X)$ have degree at most k_i. To keep control on the $O(\ )$ constants, we let C be a constant such that the complexity estimate in Theorem 5 is upper-bounded by $2\mathsf{M}(d) + \mathsf{C}d$.

- Applying Theorem 5 to each entry of $M_{k_i}(X)$ to perform a shift by $(k_i+1)\beta$, we see that the values of $M_{k_i}(X)$ on $(k_i+1)\beta, \dots, (2k_i+1)\beta$ can be computed for $n^2(2\mathsf{M}(k_i)+\mathsf{C}k_i)$ ring operations, since $\mathsf{d}(\beta(k_i+1), \beta, k_i)^{-1}$ is known. We then have at our disposal the values of $M_{k_i}(X)$ at $0, \beta, \dots, (2k_i+1)\beta$.
- Applying Theorem 5 to each entry of $M_{k_i}(X)$ to perform shifts by $k_i\alpha$ and then by $(k_i+1)\beta$, we see that the values of $M_{k_i}(X+k_i\alpha)$ on $0, \beta, \dots, (2k_i+1)\beta$ can be computed for $2n^2(2\mathsf{M}(k_i) + \mathsf{C}k_i)$ ring operations. For the first shift we need $\mathsf{d}(\alpha k_i, \beta, k_i)^{-1}$ and for the second we need $\mathsf{d}(\beta(k_i+1), \beta, k_i)^{-1}$; they have both been precomputed in the preamble.

From these values, it is easy to deduce the values of $M_{k_{i-1}}(X)$ at $0, \beta, \dots, k_{i-1}\beta$. We distinguish two cases, according to the parity of k_{i-1}.

- Suppose first that k_{i-1} is even, so that $k_{i-1} = 2k_i$. Using the equality
$$M_{k_{i-1}}(X) = M_{k_i}(X + \alpha k_i) \cdot M_{k_i}(X),$$
we obtain the values of $M_{k_{i-1}}(X)$ at $0, \beta, \dots, k_{i-1}\beta$ by multiplying the known values of $M_{k_i}(X + \alpha k_i)$ and $M_{k_i}(X)$ at $0, \beta, \dots, k_{i-1}\beta$. This takes $(k_{i-1} + 1)\mathsf{MM}(n)$ operations.
- Suppose now that k_{i-1} is odd, so that $k_{i-1} = 2k_i + 1$. Using the equality
$$M_{k_{i-1}}(X) = M(X + \alpha k_{i-1}) \cdot M_{k_i}(X + \alpha k_i) \cdot M_{k_i}(X),$$
we obtain the values of $M_{k_{i-1}}(X)$ at $0, \beta, \dots, k_{i-1}\beta$ as follows. We first multiply the values of $M_{k_i}(X + \alpha k_i)$ and $M_{k_i}(X)$ at $0, \beta, \dots, k_{i-1}\beta$, for $(k_{i-1}+1)\mathsf{MM}(n)$ operations. Then we evaluate $M(X+\alpha k_{i-1})$ at these points for $2n^2(k_{i-1} + 1)$ operations, from which we deduce the requested values of $M_{2k_i+1}(X)$ for $(k_{i-1} + 1)\mathsf{MM}(n)$ operations.

Let us denote by $T(k_{i-1})$ the cost of these operations. From the analysis above, we deduce the inequality

$$T(k_{i-1}) \le T(k_i) + 2(k_{i-1} + 1)\mathsf{MM}(n) + 6n^2\mathsf{M}(k_i) + n^2\mathsf{C}'k_i$$

for a constant C' that can be taken to be $\mathsf{C}' = 3\mathsf{C} + 4$. Using the definition of the sequence k_i and our assumptions on the function M, we deduce the estimate $T(k) = T(k_0) \in O(\mathsf{MM}(n)k + n^2\mathsf{M}(k))$. The space complexity estimate is easily dealt with, since all computations at step i can be done in space $O(n^2k_i)$.

We now come to the last statement, where R is effective. In the Turing context, we have to pay attention to the way matrices and vectors are represented. The preamble, where inverses are precomputed, poses no problem; it suffices to organize the elements to invert in the correct order. However, at the heart of the procedure, we have to switch between two representations of matrices of vectors over R. This is free in an algebraic model but has to be justified to have negligible cost in a Turing model.

At the input of the inner loop on i, we have on a tape the values of $M_{k_i}(X)$ at $0, \beta, \dots, k_i\beta$. They are stored as a sequence of matrices over R, each in row-major representation. In order to apply Theorem 5, we need its input to be contiguous on a tape. Hence, we must first reorganize the data, switching to a representation as a matrix of vectors, with vectors of size $k_i + 1$: Corollary 19 in the appendix shows how to do this in cost $O(\ell n^2 k_i \min(\log k_i, \log n))$. After the applications of Theorem 5, we

have at our disposal the values of $M_{k_i}(X)$ and of $M_{k_i}(X+k_i\alpha)$ at $0, \beta, \dots, (2k_i+1)\beta$, organized as matrices of vectors. We switch back to their representation as a sequence of matrices over R to perform the matrix multiplications. This is the converse problem as before, and it admits the same $O(\ell n^2 k_i \min(\log k_i, \log n))$ solution. The matrix products can then be done at the expected cost, since the input data is contiguous, and finally we are ready to enter again the loop with the next value of i.

Putting together the costs of these operations, the additional cost compared to the algebraic model is $O(\ell n^2 k \min(\log k, \log n))$. The memory requirements consist of $O(n^2 k\ell)$ for storing all intermediate matrices and polynomials, plus a term in $O(\mathsf{S}_R(k))$ coming from Theorem 5, and the term $\mathsf{SM}_R(n)$ for matrix multiplication. □

The case $\alpha = \beta = 1$ is not covered in the last theorem. However, this case is easier to handle (note also that there are no invertibility conditions).

PROPOSITION 9. *Suppose that $\alpha = 1$, that is, $M_k(X) = M(X+k)\cdots M(X+1)$. Then the scalar matrices $M_k(0), M_k(1), \dots, M_k(k)$ can be computed using $O(\mathsf{MM}(n)k)$ ring operations, in space $O(n^2k)$. If R is effective, the bit complexity is $O(\mathsf{MM}_R(n)k)$ and the space requirement is $O(n^2k\ell + \mathsf{SM}_R(n))$ bits.*

Proof. We first evaluate all matrices $M(1), \dots, M(2k)$; this requires $O(n^2k)$ operations and takes space $O(n^2k)$. The conclusion is now similar to the proof of Lemma 1. Denoting by I the $n \times n$ identity matrix, we first compute the products

$$R_k = I, \quad R_{k-1} = R_k M(k), \quad R_{k-2} = R_{k-1}M(k-1), \dots, R_0 = R_1 M(1)$$

and

$$L_0 = I, \quad L_1 = M(k+1)L_0, \quad L_2 = M(k+2)L_1, \dots, L_k = M(2k)L_{k-1}.$$

This takes $O(\mathsf{MM}(n)k)$ ring operations, and $O(n^2k)$ space. We conclude by computing the matrices $M_k(i) = L_i R_i$ for $0 \le i \le k$. This also takes $O(\mathsf{MM}(n)k)$ ring operations and $O(n^2k)$ space. The estimates in the Turing model come similarly. □

In section 7, we have to deal with a similar evaluation problem, but with arbitrary sample points. The following corollary of Proposition 9 will then be useful; compared to the particular case of Theorem 8, we lose a logarithmic factor in the evaluation process.

COROLLARY 10. *Let notation be as in Proposition 9 and assume that $2, \dots, k$ are units in R, and that their inverses are known. Then for any set of $k+1$ elements β_i of R, the matrices $M_k(\beta_0), M_k(\beta_1), \dots, M_k(\beta_k)$ can be computed in*

$$O(\mathsf{MM}(n)k + n^2\mathsf{M}(k)\log k)$$

operations, in space $O(kn^2 + k\log k)$. If R is effective, the bit complexity is

$$O(\mathsf{MM}_R(n)k + n^2\mathsf{M}_R(k)\log k),$$

and the space requirement is $O(k\ell n^2 + \mathsf{SM}_R(n) + \ell k \log k + \mathsf{S}_R(k))$ bits.

Proof. We start by evaluating $M_k(X)$ on $0, \dots, k$; by Proposition 9, this takes $O(\mathsf{MM}(n)k)$ operations and space $O(n^2k)$. Using the inverses of $2, \dots, k$, it is then possible to interpolate all entries of $M_k(X)$ in $O(n^2\mathsf{M}(k)\log k)$ operations, in space $O(k\log k)$. Then, we can evaluate all entries of this matrix at the points $\beta_0, \beta_1, \dots, \beta_k$, with the same cost.

In the Turing model, we face the same problem as in Theorem 8. The output of Proposition 9 is given as a sequence of scalar matrices, so we switch to a matrix of

vectors to apply interpolation and evaluation, and convert it back to a sequence of matrices. Proposition 9 gives the cost for evaluation at $0, \dots, k$; Lemma 4 gives that for interpolation at these points, and for evaluation at $\beta_0, \dots, \beta_k$; Corollary 19 gives the cost for changing the data's organization. The conclusion follows. ∎

5. Application to integer factorization. In this section, we apply the algorithm of Theorem 8 to reduce by a logarithmic factor the best upper bound for deterministic integer factorization, due to Strassen [48]. Strassen's result is that a positive integer N can be completely factored using at most $O\big(\mathsf{M}_{\mathrm{int}}(\sqrt[4]{N}\log N)\log N\big)$ bit operations. The following theorem improves this result.

THEOREM 11. *There exists a deterministic algorithm that outputs the complete factorization of any positive integer N using at most $O(\mathsf{M}_{\mathrm{int}}(\sqrt[4]{N}\log N))$ bit operations. The space complexity is $O(\sqrt[4]{N}\log N)$ bits.*

Our proof closely follows that of Strassen, in the presentation of [15], the main ingredient being now Theorem 8; some additional complications arise due to the nontrivial invertibility conditions required by that theorem. First, applying Lemma 3 (with $m = 1$) gives the data describing $\mathbb{Z}/N\mathbb{Z}$ as an effective ring:

- $\ell = \lceil \log N \rceil$,
- $\mathsf{m}_R \in O(\mathsf{M}_{\mathrm{int}}(\log N))$ and $\mathsf{s}_R \in O(\log N)$,
- $\mathsf{M}_R(d) \in O(\mathsf{M}_{\mathrm{int}}(d\log(dN)))$ and $\mathsf{S}_R(d) \in O(d\log(dN))$.

The data for matrix multiplication is not required here, since all matrices have size 1.

LEMMA 12. *Let $f_0, \dots, f_{k-1}$ be in $\mathbb{Z}/N\mathbb{Z}$. Then one can decide whether all f_i are invertible modulo N and, if not, find a noninvertible f_i in time*

$$O\big(k\,\mathsf{M}_{\mathrm{int}}(\log N) + \log k\,\mathsf{M}_{\mathrm{int}}(\log N)\log\log N\big)$$

and space $O(k\log N)$ bits.

Proof. For $i \le k-1$, we will denote by F_i the canonical preimage of f_i in $[0, \dots, N-1]$. Hence, our goal is to find one F_i such that $\gcd(F_i, N) > 1$.

We first form the "subproduct tree" associated with the values F_i. By completing with enough 1's, we can assume that k is a power of 2, i.e., that $k = 2^m$. First, we define $F_{i,m} = F_i$ for $i = 0, \dots, k-1$; then iteratively we let $F_{i,j-1} = F_{2i,j}F_{2i+1,j} \bmod N$, for $j = m, \dots, 1$ and $i = 0, \dots, 2^{j-1}-1$. These numbers are organized in a tree similar to the one described in the appendix for evaluation and interpolation; see also [15, Chapter 10]. The number of multiplications to perform in $\mathbb{Z}/N\mathbb{Z}$ is at most k; hence, their total cost is $O(k\,\mathsf{M}_{\mathrm{int}}(\log N))$, the number of tape movements being a negligible $O(k\log N)$. The space complexity is $O(k\log N)$ bits as well.

By computing $\gcd(F_{0,0}, N)$, we can decide whether all of $F_0, \dots, F_{k-1}$ are invertible modulo N. If this is not the case, finding one i for which $\gcd(F_i, N) > 1$ amounts to going down the tree from the root to one leaf. Suppose indeed that we have determined that $\gcd(F_{i,j}, N) > 1$ for some $j < m$. Computing $\gcd(F_{2i,j+1}, N)$ enables us to determine one of $F_{2i,j+1}$ or $F_{2i+1,j+1}$ which has a nontrivial gcd with N.

The cost of a gcd is $O(\mathsf{M}_{\mathrm{int}}(\log N)\log\log N)$. Since the tree has depth $\log k$, we compute only $\log k$ gcd's. Again, since the tree is well organized on the tape, going down the tree is done in a one pass process; so our claim on the time complexity is proved. The space complexity does not increase, concluding the proof. ∎

Our second intermediate result is the cornerstone of the algorithm; we improve the estimate of [15, Theorem 19.3] using Theorem 8.

LEMMA 13. *Let b and N be positive integers with $2 \le b < N$. Then one can compute a prime divisor of N bounded by b, or prove that no such divisor exists, in*

$$O\big(\mathsf{M}_{\mathrm{int}}(\sqrt{b}\log N) + \log b\,\mathsf{M}_{\mathrm{int}}(\log N)\log\log N\big)$$

bit operations. The space complexity is $O(\sqrt{b}\log N)$ bits.

Proof. Let $c = \lfloor\sqrt{b}\rfloor$ and let us consider the polynomials of $\mathbb{Z}/N\mathbb{Z}[X]$ given by $f(X) = X + 1 - c$ and

$$F(X) = \prod_{k=0}^{c-1}(X + ck + 1) = f(X + c^2)\cdots f(X + 2c)f(X + c).$$

Our first goal is to compute the values

$$F(0) = \prod_{k=0}^{c-1}(ck + 1) \bmod N, \ldots, F(c-1) = \prod_{k=0}^{c-1}(ck + c) \bmod N.$$

To this effect, we want to apply Theorem 8 with $n = 1$, $\alpha = c$, $\beta = 1$, and $k = c$. This can be done under suitable invertibility conditions: by Lemma 7, $O(\sqrt{b})$ integers bounded by $c^2 \le b$ must be invertible modulo N. All these integers are easily computable and less than N. Using Lemma 12, we can test whether one of these integers is not invertible modulo N and, if this is the case, find one such integer in

$$O\big(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N) + \log b\,\mathsf{M}_{\mathrm{int}}(\log N)\log\log N\big)$$

bit operations. Then, by trial division, we can find a prime factor of N bounded by b in $O(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N))$ bit operations; in this case, the result is proved. Thus, we can now assume that all invertibility conditions are satisfied.

By Lemma 6, computing $\mathsf{D}(c, 1, c) \in \mathbb{Z}/N\mathbb{Z}$ has a cost of $O(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N))$; computing its inverse has a cost in $O(\mathsf{M}_{\mathrm{int}}(\log N)\log\log N)$. Applying Theorem 8, we deduce that all the values $F(i)$, for $i = 0, \ldots, c-1$, can be computed in time $O(\mathsf{M}_{\mathrm{int}}(\sqrt{b}\log(bN)))$; since $b < N$, this is in $O(\mathsf{M}_{\mathrm{int}}(\sqrt{b}\log N))$.

Suppose that N admits a prime factor bounded by c^2; then, some $F(i)$ is not invertible modulo N. By Lemma 12, such an i can be found in time

$$O\big(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N) + \log b\,\mathsf{M}_{\mathrm{int}}(\log N)\log\log N\big).$$

Since $F(i)$ is not invertible, $\gcd(ck+i+1, N)$ is nontrivial for some $k \le c-1$. Applying Lemma 12 again, the element $ck + i + 1$ can be found in time

$$O\big(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N) + \log b\,\mathsf{M}_{\mathrm{int}}(\log N)\log\log N\big).$$

By definition, $ck + i + 1 \le b$, so by trial division, we can then find a prime factor of N bounded by b in $O(\sqrt{b}\,\mathsf{M}_{\mathrm{int}}(\log N))$ bit operations.

At this point, if we have not found any prime divisor, then we have certified that N has no prime divisor in $2, \ldots, c^2$, so we finally inspect the range $c^2 + 1, \ldots, b$. Since $b - c^2 \in O(\sqrt{b})$, and since all these numbers are in $O(b)$, we can finish using trial division with the same time complexity bounds as above. The space complexity is dominated by those of Theorem 8 and Lemma 12. ∎

The proof of Theorem 11 follows from successive applications of the lemma above with increasing values of b. Starting with $b = 2$, the algorithm of Lemma 13 is run

with the same value of b until no more factors smaller than b remain in N; then the value of b is doubled. The algorithm stops as soon as $b \geq \sqrt{N}$.

The space complexity estimate comes immediately. However, analyzing the time complexity requires more work. Indeed, since there are at most $(\log N)$ prime factors of N, a rough upper bound on the runtime of the whole factorization would be $(\log N)$ times the runtime of Lemma 13 with $b = \sqrt{N}$, which is too high for our purposes. We show now that this $(\log N)$ factor can in fact be avoided, thus proving Theorem 11.

When Lemma 13 is run with a given parameter b, all prime divisors of N less than $b/2$ have already been found and divided out. Therefore the primes that can be detected are greater than $b/2$; since their product is bounded by N, we deduce that the number of runs of Lemma 13 for the parameter b is upper-bounded by $O(\log N/\log b)$.

In the complexity estimate of Lemma 13, the sum of all $\mathsf{M}_{\mathrm{int}}(\log N)\log b \log\log N$ terms is bounded by a polynomial in $(\log N)$, so its contribution in the total runtime is negligible. We are thus left with the problem of evaluating the following quantity:

$$\sum_{i=1}^{\lceil(\log N)/2\rceil} \frac{\log N}{\log(2^i)} \mathsf{M}_{\mathrm{int}}(2^{i/2}\log N) \leq \mathsf{M}_{\mathrm{int}}\left(\log N \sum_{i=1}^{\lceil(\log N)/2\rceil} \left\lceil \frac{\log N}{i} \right\rceil 2^{i/2}\right),$$

where the upper bound follows from the first assumption of (2). Then, the sum is upper-bounded by a constant times $N^{1/4}$, giving the runtime of Theorem 11.

6. Computing one term in a linear recurrence. We now address the following problem: given an $n \times n$ matrix $M(X)$ with entries in $R[X]$, all of them of degree at most 1, and a vector of initial conditions U_0, define the sequence (U_i) of elements in R^n by the linear recurrence

$$U_{i+1} = M(i+1)U_i \text{ for all } i \geq 0.$$

Our question is to compute the vector U_N for some $N \in \mathbb{N}$. In the introduction, we gave the complexity of Chudnovsky and Chudnovsky's algorithm for this task. Our modification is similar to the one for integer factorization in the previous section.

THEOREM 14. *Let N be a positive integer and $s = \lfloor \log_4 N \rfloor$, such that $2, \ldots, 2^s+1$ are units in R. Given the inverses of $\mathsf{D}(1, 2^t, 2^t)$, $t \leq s$, U_N can be computed in*

$$O\big(\mathsf{MM}(n)\sqrt{N} + n^2\,\mathsf{M}(\sqrt{N})\big)$$

operations, in space $O(n^2\sqrt{N})$. If R is effective, then the bit complexity is

$$O\big(\mathsf{MM}_R(n)\sqrt{N} + n^2\mathsf{M}_R(\sqrt{N}) + n^2\ell\sqrt{N}\min(\log N, \log n)\big),$$

using $O(n^2\ell\sqrt{N} + \mathsf{SM}_R(n) + \mathsf{S}_R(\sqrt{N}))$ bits.

Proof. The proof is divided into two steps. We begin by proving the assertion in the particular case when N is a power of 4; then we treat the general case. Dealing with powers of 4 makes it possible to control the list of elements that need to be units.

- *The case when N is a power of 4.* Let us suppose that $k = 2^s$ and $N = k^2$, so that $N = 4^s$. Let $M_k(X)$ be the $n \times n$ matrix over $R[X]$ defined by

  $$M_k(X) = M(X+k)\cdots M(X+1);$$

  then, the requested output U_N can be obtained by the equation

  $$U_N = M_k(k(k-1))\cdots M_k(k)\,M_k(0)\,U_0. \tag{5}$$

Taking $\alpha = 1$ and $\beta = k = 2^s$, Lemma 7 shows that condition $\mathsf{H}(1, 2^s, 2^s)$ of Theorem 8 is satisfied. Thus, $M_k(0), M_k(k), \ldots, M_k((k-1)k)$ can be computed within the required time and space complexities. Then, by formula (5), the result is obtained by performing $\sqrt{N}$ successive matrix-vector products, which has a cost in both time and space of $O(n^2\sqrt{N})$. There is no additional complication in the Turing model, since in the row-major representation of matrices, matrix-vector products do not involve expensive tape movements.

- *The general case.* Let $N = \sum_{i=0}^{s} N_i 4^i$ be the 4-adic expansion of N, with $N_i \in \{0, 1, 2, 3\}$ for all i. Given any $t \geq 0$, we will denote by $\lceil N \rceil^t$ the integer $\sum_{i=0}^{t-1} 4^i N_i$. Using this notation, we define a sequence $(V_t)_{0 \leq t \leq s}$ as follows: We let $V_0 = U_0$ and, for $0 \leq t \leq s$ we set

$$\begin{aligned} &V_{t+1} = M(\lceil N \rceil^t + 4^t N_t) \cdots M(\lceil N \rceil^t + 1) V_t \quad \text{if } N_t \in \{1, 2, 3\}, \\ &V_{t+1} = V_t \quad \text{if } N_t = 0. \end{aligned} \tag{6}$$

One checks that $V_t = U_{\lceil N \rceil^t}$ for all t, and in particular that $V_{s+1} = U_N$. We will compute all V_t successively. Suppose thus that the term V_t is known. If N_t is zero, we have nothing to do. Otherwise, we let $V_{t+1}^{(0)} = V_t$, and, for $1 \leq j \leq N_t$, we let

$$M^{(j)}(X) = M\left(X + \lceil N \rceil^t + 4^t(j-1)\right).$$

Then we define $V_{t+1}^{(j)}$ by

$$V_{t+1}^{(j)} = M^{(j)}(4^t) \cdots M^{(j)}(1) V_{t+1}^{(j-1)}, \quad j = 1, \ldots, N_t.$$

By (6), we have $V_{t+1}^{(N_t)} = V_{t+1}$. Thus, passing from V_t to V_{t+1} amounts to computing N_t selected terms of a linear recurrence of the special form treated in the previous case, for indices of the form $4^t \leq 4^s$. With all necessary assumptions being satisfied, using the complexity result therein and the fact that all N_t are bounded by 3 we see that the total cost of the general case is thus

$$O\left(\sum_{t=0}^{s} \left(\mathsf{MM}(n) 2^t + n^2\, \mathsf{M}(2^t)\right)\right) = O\left(\mathsf{MM}(n) 2^s + n^2 \left(\sum_{t=0}^{s} \mathsf{M}(2^t)\right)\right).$$

Using the fact that $2^s \leq \sqrt{N} \leq 2^{s+1}$ and the assumptions on the function M, we easily deduce that the whole complexity fits into the bound $O\big(\mathsf{MM}(n)\sqrt{N} + n^2\, \mathsf{M}\big(\sqrt{N}\big)\big)$, and the bound concerning the memory requirements follows. As before, porting this algorithm on a Turing machine does not raise any difficulty. □

7. Computing several terms in a linear recurrence. We now study the case when several terms in a linear recurrence are questioned; this will be applied in the next section for the computation of the Cartier–Manin operator. We use the same notation as before: $M(X)$ is an $n \times n$ matrix whose entries are degree 1 polynomials over a ring R; we consider the sequence (U_i) defined for all $i \geq 0$ by $U_{i+1} = M(i+1)U_i$, and let U_0 be a vector in R^n. Given r indices $N_1 < N_2 < \cdots < N_r$, we want to compute all the values $U_{N_i}, 1 \leq i \leq r$.

An obvious solution is to repeatedly apply Theorem 14 to each interval $[N_i, N_{i-1}]$, leading to a time complexity of

$$O\left(\mathsf{MM}(n)\sum_{i=1}^{r}\sqrt{N_i - N_{i-1}} + n^2\sum_{i=1}^{r}\mathsf{M}\left(\sqrt{N_i - N_{i-1}}\right)\right)$$

and with a memory requirement of $O(n^2 \max_{i=1}^{r}\sqrt{N_i - N_{i-1}})$, where we have set $N_0 = 0$. In special cases, this might be close to optimal, for instance if $N_1, \ldots, N_{r-1}$ are small compared to N_r. However, in most cases it is possible to improve on this: we now present an alternative algorithm, which improves on the time complexity, at the cost, however, of increased memory requirements.

THEOREM 15. *Let $N_1 < N_2 < \cdots < N_r = N$ be positive integers. Suppose that $2, \ldots, 2^s + 1$ are units in R, where $s = \lfloor \log_4 N \rfloor$, and that the inverse of $\mathsf{D}(1, 2^s, 2^s)$ is known. Suppose also that $r < N^{\frac{1}{2}-\varepsilon}$, with $0 < \varepsilon < \frac{1}{2}$. Then $U_{N_1}, \ldots, U_{N_r}$ can be computed within*

$$O\big(\mathsf{MM}(n)\sqrt{N} + n^2\mathsf{M}(\sqrt{N})\big)$$

ring operations in space $O(n^2\sqrt{N})$. If R is effective, the bit complexity is in

$$O\big(\mathsf{MM}_R(n)\sqrt{N} + n^2\mathsf{M}_R(\sqrt{N}) + n^2\ell\sqrt{N}\min(\log N, \log n)\big),$$

and the space complexity is $O(n^2\ell\sqrt{N} + \mathsf{SM}_R(n) + \mathsf{S}_R(\sqrt{N}))$ bits.

In other words, the complexity of computing several terms is essentially the same as that of computing the one of largest index, as long as the total number of terms to compute is not too large. If the N_i form an arithmetic progression, and if the multiplication function M is essentially linear, we gain a factor of $\sqrt{r}$ compared to the naive approach. In the limiting case, where $r = \sqrt{N}$, and for fixed size n, our algorithm is optimal up to logarithmic factors, as it takes a time essentially linear in the size of the output.

Proof. The algorithm proceeds as follows: we start by applying Theorem 8 with $k \approx \sqrt{N}$, so as to compute k terms in the sequence with indices separated by intervals of size about $\sqrt{N}$. If all indices N_i have been reached, then we are done. Otherwise, we have reached r indices that are within a distance of about $\sqrt{N}$ of $N_1, \ldots, N_r$. A recursive refining procedure is then done simultaneously for all these r indices. When we get close enough to the wanted indices, we finish with a divide-and-conquer method. The refining and the final steps make use of Corollary 10.

We now describe more precisely the first step (named Step 0), the ith refining step, and the final step. In what follows, given $k \in \mathbb{N}$, we denote by $M_k(X)$ the polynomial matrix $M_k(X) = M(X + k) \cdots M(X + 1)$.

Step 0. Let $k_0 = 2^s$. As a preliminary, from the inverse of $\mathsf{D}(1, 2^s, 2^s)$, we deduce using Lemma 1 the inverses of all integers $1, \ldots, k_0$ in R. This will be used later on.

Define $N_j^{(0)} = k_0\lfloor \frac{N_j}{k_0} \rfloor$ for $j \le r$, so that $N_j^{(0)} \le N_j < N_j^{(0)} + k_0$. Step 0 consists of computing $U_{N_1^{(0)}}, \ldots, U_{N_r^{(0)}}$. This is done by computing all vectors $U_{k_0}, U_{2k_0}, \ldots, U_{4k_0^2}$; this sequence contains all requested vectors, since all wanted indices are multiples of k_0 and upper-bounded by $4k_0^2$. Lemma 7 shows that the assumptions of Theorem 8 with $\alpha = 1$ and $\beta = k = k_0$ are satisfied. We use it to compute the matrices

$$M_{k_0}(0), M_{k_0}(k_0), \ldots, M_{k_0}((k_0 - 1)k_0);$$

the vector U_0 is then successively multiplied by these matrices, yielding the vectors $U_{k_0}, U_{2k_0}, \dots, U_{k_0^2}$. To complete the sequence, we repeat three times the same strategy, starting with $V_0 = U_{k_0^2}$ and shifting the matrix $M(X)$ accordingly. Among all the resulting vectors, we collect the requested values $U_{N_1^{(0)}}, \dots, U_{N_r^{(0)}}$.

Step i. We assume that after step $(i-1)$, we are given an integer k_{i-1} and indices $N_j^{(i-1)}$, where for all $j \le r$, we have $N_j^{(i-1)} \le N_j < N_j^{(i-1)} + k_{i-1}$. We also suppose that $U_{N_1^{(i-1)}}, \dots, U_{N_r^{(i-1)}}$ are known. In other words, we know r vectors whose indices are within a distance of k_{i-1} of the wanted ones. Then, the ith refining step is as follows. Set

$$k_i = \left\lceil \sqrt{r k_{i-1}} \right\rceil;$$

then the new terms that we want to compute correspond to the indices

$$N_1^{(i)} = N_1^{(i-1)} + k_i \left\lfloor \frac{N_1 - N_1^{(i-1)}}{k_i} \right\rfloor, \dots, N_r^{(i)} = N_r^{(i-1)} + k_i \left\lfloor \frac{N_r - N_r^{(i-1)}}{k_i} \right\rfloor,$$

which satisfies the induction assumption for entering step $(i+1)$. To compute these values, we evaluate the new polynomial matrix $M_{k_i}(X)$ at the points

$$\begin{aligned}
\mathbf{N}_1^{(i)} &= \left\{ N_1^{(i-1)},\ N_1^{(i-1)} + k_i, \dots, N_1^{(i-1)} + \left(\left\lfloor \frac{k_{i-1}}{k_i} \right\rfloor - 1 \right) k_i \right\} \\
&\ \ \vdots \qquad\qquad\qquad\qquad \vdots \\
\mathbf{N}_r^{(i)} &= \left\{ N_r^{(i-1)},\ N_r^{(i-1)} + k_i, \dots, N_r^{(i-1)} + \left(\left\lfloor \frac{k_{i-1}}{k_i} \right\rfloor - 1 \right) k_i \right\}.
\end{aligned}$$

There are $r\lfloor \frac{k_{i-1}}{k_i} \rfloor \le k_i$ points of evaluation. We have already computed the inverses of $1, \dots, k_0$ in R; in particular, we know the inverses of $1, \dots, k_i$, so Corollary 10 can be applied to perform the evaluation. Then, for all $j \le r$, we successively multiply $U_{N_j^{(i-1)}}$ by the values taken by $M_{k_i}(X)$ at all indices in $\mathbf{N}_j^{(i)}$. By construction, $N_j^{(i)} - k_i$ belongs to $\mathbf{N}_j^{(i)}$, so we obtain in particular the requested value $U_{N_j^{(i)}}$.

Final step. The refining process stops when k_i is close to r, namely $k_i \le 2r$. In this situation, we have at our disposal r indices $N_1', \dots, N_r'$ such that $N_j' \le N_j < N_j' + 2r$ for all j, and such that all values $U_{N_1'}, \dots, U_{N_r'}$ are known. Then another recursive algorithm is used: $M_r(X)$ is computed, evaluated at all N_j' using Corollary 10, and used to reduce all gaps that were of size between r and $2r$; again, the invertibility conditions cause no problem. As a result, all the gaps are now of size at most r. Then $M_{\frac{r}{2}}(X)$ is computed and used to reduce the gaps to at most $\frac{r}{2}$, and so on until we get the result.

It remains to perform the complexity analysis. The cost of Step 0 is dominated by that of evaluating the matrix $M_{k_0}(X)$; by Theorem 8, it can be done in

$$O\big(\mathsf{MM}(n)\sqrt{N} + n^2 \mathsf{M}(\sqrt{N})\big)$$

operations in R, and space $O(n^2\sqrt{N})$. By Corollary 10, the ith refining step has a cost of

$$O\left(\mathsf{MM}(n) k_i + n^2 \mathsf{M}(k_i) \log k_i\right)$$

operations, and requires $O(n^2 k_i + k_i \log k_i)$ temporary memory allocation; the total cost of this phase is thus

$$O\left(\sum_{i=1}^{i_{\max}} \mathsf{MM}(n)k_i + n^2\mathsf{M}(k_i)\log k_i\right),$$

where $i_{\max}$ is the number of refining steps. Our hypotheses on the function M show that this fits into the bound

$$O\left(\mathsf{MM}(n)\sum_{i=1}^{i_{\max}} k_i + n^2\mathsf{M}\left(\sum_{i=1}^{i_{\max}} k_i\right)\log\left(\sum_{i=1}^{i_{\max}} k_i\right)\right).$$

An easy induction shows that k_i is at most $(4r)^{\frac{2^i-1}{2^i}} N^{\frac{1}{2^{i+1}}}$, whence $i_{\max} = O(\log\log N)$. Furthermore, using $r < N^{\frac{1}{2}-\varepsilon}$, we get

$$k_i \le 4N^{\frac{1}{2}}\left(N^{-\varepsilon}\right)^{\frac{2^i-1}{2^i}}.$$

For any $0 < \ell \le 1$ we have $\sum_{i=1}^{i_{\max}} \ell^{\frac{2^i-1}{2^i}} \le i_{\max}\ell^{\frac{1}{2}}$; therefore we deduce

$$\sum_{i=1}^{i_{\max}} k_i = O\big(N^{\frac{1-\varepsilon}{2}}\log\log N\big).$$

Using the second assumption in (2), the cost of the refining steps is negligible compared with that of the first step, since the $N^{-\varepsilon/2}$ compensates for all logarithmic factors. The memory requirement also is negligible compared with that of Step 0.

Using Corollary 10, the cost of the first reduction in the final step is

$$O\left(\mathsf{MM}(n)r + n^2\mathsf{M}(r)\log r\right)$$

ring operations and $O(n^2 r + r\log r)$ temporary space. Due to our hypotheses on the function M, the second reduction costs at most half as much, the third reduction costs at most $\frac{1}{4}$ of it, etc. Summing up, we see that the whole cost of the final step is bounded by twice that of the first reduction, which is itself less than the cost of Step 0.

There is no complication in the Turing model. Indeed, between the calls to Theorem 8, to Corollary 10, and the successive matrix-vector products, there is no need to reorganize data, so the tape movements' cost is negligible. Furthermore, our assumptions on M_R and S_R imply that, as in the arithmetic model, the time and space costs of Step 0 are predominant. □

8. Application to the Cartier–Manin operator. Let $\mathcal{C}$ be a hyperelliptic curve of genus g defined over the finite field $\mathbb{F}_{p^d}$ with p^d elements, where $p > 2$ is prime. We suppose that the equation of $\mathcal{C}$ is of the form $y^2 = f(x)$, where f is monic and square free, of degree $2g+1$. The generalization to hyperelliptic curves of the Hasse invariant for elliptic curves is the Hasse–Witt matrix [21]: Let h_k be the coefficient of degree x^k in the polynomial $f^{(p-1)/2}$. The Hasse–Witt matrix is the $g \times g$ matrix with coefficients in $\mathbb{F}_{p^d}$ given by $H = (h_{ip-j})_{1\le i,j\le g}$. It represents, in a suitable basis, the operator on differential forms introduced by Cartier [10]; Manin [28] showed that this matrix is strongly related to the action of the Frobenius endomorphism on

the p-torsion part of the Jacobian of $\mathcal{C}$. The article [50] provides a complete survey about these facts; they are summarized in the following theorem.

THEOREM 16 (Manin). *Let $H_\pi = HH^{(p)} \cdots H^{(p^{d-1})}$, where the notation $H^{(q)}$ means elementwise raising to the power q. Let $\kappa(t)$ be the characteristic polynomial of the matrix H_π and $\chi(t)$ the characteristic polynomial of the Frobenius endomorphism of the Jacobian of $\mathcal{C}$. Then $\chi(t) \equiv (-1)^g t^g \kappa(t) \bmod p$.*

This result provides a method to compute the characteristic polynomial of the Frobenius endomorphism. As such, it can be used in a point-counting algorithm, which is the main application we have in mind; see the end of this section for more comments.

To compute the entries of the Hasse–Witt matrix, the obvious solution consists of expanding the product $f^{(p-1)/2}$. Using binary powering, this can be done in $O(\mathsf{M}(gp))$ base ring operations, whence a time complexity that is essentially linear in p, if g is kept constant. In what follows, we show how to obtain a complexity essentially linear in $\sqrt{p}$ using the results of the previous sections. We will make the additional assumption that the constant term of f is not zero; otherwise, the problem is actually simpler.

Introduction of a linear recurrent sequence. In [14], Flajolet and Salvy already treated the question of computing a selected coefficient in a high power of some given polynomial, as an answer to a SIGSAM challenge. The key point of their approach is that $h = f^{(p-1)/2}$ satisfies the following first-order linear differential equation

$$fh' - \frac{p-1}{2} f'h = 0.$$

This shows that coefficients of h satisfy a linear recurrence of order $2g+1$, with polynomial coefficients of degree 1. Explicitly, denote by h_k the coefficient of degree k of h, and for convenience, set $h_k = 0$ for $k < 0$. Similarly, the coefficient of degree k of f is denoted by f_k. Then the differential equation above implies that, for all k in $\mathbb{Z}$,

$$(k+1)f_0 h_{k+1} + \left(k - \frac{p-1}{2}\right) f_1 h_k + \cdots + \left(k - 2g - \frac{(2g+1)(p-1)}{2}\right) f_{2g+1} h_{k-2g} = 0.$$

We set $U_k = [h_{k-2g}, h_{k-2g+1}, \ldots, h_k]^t$, and let $A(k)$ be the companion matrix:

$$A(k) = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ddots & 1 \\ \frac{f_{2g+1}((2g+1)(p-1)/2-(k-2g-1))}{f_0 k} & \cdots & \cdots & \cdots & \frac{f_1((p-1)/2-k+1)}{f_0 k} \end{pmatrix}.$$

The initial vector $U_0 = [0, \ldots, 0, f_0^{(p-1)/2}]^t$ can be computed using binary powering techniques in $O(\log p)$ operations; then for $k \geq 0$, we have $U_{k+1} = A(k+1)U_k$. Thus, to answer our specific question, it suffices to note that the vector U_{ip-1} gives the coefficients h_{ip-j} for $j = 1, \ldots, g$ that form the ith row of the Hasse–Witt matrix of $\mathcal{C}$.

Theorems 14 and 15 cannot be directly applied to this sequence, because $A(k)$ has entries that are rational functions, not polynomials. Though the algorithm could be adapted to handle the case of rational functions, we rather use the very specific form of the matrix $A(k)$, so only a small modification is necessary. Let us define a new

sequence V_k by $V_k = f_0^k k! U_k$. Then, this sequence is linearly generated and we have $V_{k+1} = B(k+1)V_k$, where $B(k) = f_0 k A(k)$. Therefore, the entries of the matrix $B(k)$ are polynomials of degree at most 1. Note also that the denominators $f_0^k k!$ satisfy the recurrence relation $f_0^{k+1}(k+1)! = (f_0(k+1)) \cdot (f_0^k k!)$.

We can apply the results of the previous sections to compute separately the required vectors V_k, as well as the corresponding denominators: specifically, we will compute $V_{p-1}, V_{2p-1}, \ldots, V_{gp-1}$ as well as $f_0^{p-1}(p-1)!, \ldots, f_0^{gp-1}(gp-1)!$. Then the vectors $U_{p-1}, U_{2p-1}, \ldots, U_{gp-1}$ are deduced using the relation $f_0^k k!\, U_k = V_k$.

Lifting to characteristic zero. A difficulty arises from the fact that the characteristic is too small compared to the degrees we are aiming for, so $p!$ is zero in $\mathbb{F}_{p^d}$. The workaround is to do computations in the unramified extension K of $\mathbb{Q}_p$ of degree d, whose residual field is $\mathbb{F}_{p^d}$. The ring of integers of K will be denoted by $\mathcal{O}_K$, so that any element of $\mathcal{O}_K$ can be reduced modulo p to give an element of $\mathbb{F}_{p^d}$. On the other hand, K has characteristic 0, so p is invertible in K.

We consider an arbitrary lift of f to $\mathcal{O}_K[X]$. The reformulation in terms of linear recurrent sequence made in the paragraph above can be performed over K; the coefficients of $f^{(p-1)/2}$ are computed as elements of K and then projected back onto $\mathbb{F}_{p^d}$. This is possible, as they all belong to $\mathcal{O}_K$. We separately compute the values in K of the vectors V_{ip-1} and the denominators $f_0^{ip-1}(ip-1)!$, for $i = 1, \ldots, g$. To this effect, we can apply any of the strategies mentioned in section 7.

The first one is the plain iteration based on Theorem 14; iterating g times, it performs $O\left(\mathsf{MM}(g) g\sqrt{p} + g^3 \mathsf{M}(\sqrt{p})\right)$ operations in K and requires storing $O(g^2\sqrt{p})$ elements. The second strategy is to apply Theorem 15. In this case, we need to have $g \leq (gp)^{1/2-\varepsilon'}$, for some $0 < \varepsilon' < \frac{1}{2}$; this is equivalent to imposing that $g \leq p^{1-\varepsilon}$ for some $0 < \varepsilon < 1$. Then, with increased memory requirements of $O(g^2\sqrt{gp})$, the number of operations in K reduces to $O\left(\mathsf{MM}(g)\sqrt{gp} + g^2 \mathsf{M}(\sqrt{gp})\right)$.

Computing at fixed precision. We do not want to compute in K at arbitrary precision, since this is not an effective ring (even in a much weaker sense than the one we defined). For our purposes, it suffices to truncate all computations modulo a suitable power of p. To evaluate the required precision of the computation, we need to check when the algorithm operates a division by p.

To compute the vectors V_{ip-1} and the denominators $f_0^{ip-1}(ip-1)!$, for $i = 1, \ldots, g$, we use either Theorem 14 or Theorem 15. In the worst case, it might be required to invert all integers up to $\sqrt{gp}$. With the condition $g \leq p^{1-\varepsilon}$, these numbers are strictly smaller than p, and they are thus units in $\mathcal{O}_K$. Hence no division by p occurs in this first phase. Then, for all $i = 1, \ldots, g$, to deduce U_{ip-1} from V_{ip-1}, we need to divide by $f_0^{ip-1}(ip-1)!$. The element f_0 is a unit in $\mathcal{O}_K$, so the only problem comes from the factorial. If $i < p$, then the p-adic valuation of $(ip-1)!$ is exactly $i-1$. Therefore the worst case is $i = g$, for which we have to divide by p^{g-1}. Hence computing the vectors V_{ip-1} modulo p^g is enough to know the vectors U_{ip-1} modulo p, and then to deduce the Hasse–Witt matrix.

Overall complexity. Since the ring $R = \mathcal{O}_K/p^g\mathcal{O}_K$ is isomorphic to $(\mathbb{Z}/p^g\mathbb{Z})[X]/P$ for some monic polynomial P of degree d, Lemma 3 shows that R is an effective ring with

- $\ell = d\lceil g \log p \rceil$,
- $\mathsf{m}_R \in O(\mathsf{M}_{\mathrm{int}}(dg \log(dp)))$ and $\mathsf{s}_R \in O(dg \log(dp))$,
- $\mathsf{M}_R(k) \in O(\mathsf{M}_{\mathrm{int}}(dkg \log(dkp)))$ and $\mathsf{S}_R(k) \in O(dkg \log(dkp))$,
- $\mathsf{MM}_R(n) \in O(n^{\log 7} \mathsf{m}_R)$ and $\mathsf{SM}_R(n) \in O(n^2\ell + \mathsf{s}_R)$.

From the results of sections 6 and 7, we deduce the following theorem on the complexity of computing the Hasse–Witt matrix. We give two variants that follow the two strategies described above. The first one follows from applying g times Theorem 14 with $n = 2g+1, N = p, \ell = d\lceil g \log p \rceil$; the second one come from applying Theorem 15 with $n = 2g+1, N = gp, \ell = d\lceil g \log p \rceil$. Both theorems require us to compute the inverses of some integers in R as prerequisites; the cost of their computation is negligible.

THEOREM 17. *Let $p > 2$ be a prime, $d \geq 0$, and $\mathcal{C}$ be a hyperelliptic curve defined over $\mathbb{F}_{p^d}$ by the equation $y^2 = f(x)$, with f of degree $2g+1$. Assuming $g < p^{1-\varepsilon}$ for some $0 < \varepsilon < 1$, one can compute the Hasse–Witt matrix of $\mathcal{C}$ using one of the two following strategies:*

1. *A memory-efficient strategy gives a complexity of*

$$O\left(g^{1+\log 7} p^{\frac{1}{2}} \mathsf{M}_{\mathrm{int}}(dg \log(dp)) + g^3 \mathsf{M}_{\mathrm{int}}(dgp^{\frac{1}{2}} \log(dp)) + dg^4 p^{\frac{1}{2}} \log g \log p\right)$$

   bit operations and $O(dg^3 p^{\frac{1}{2}} \log p + dgp^{\frac{1}{2}} \log d)$ storage.
2. *A time-efficient strategy gives a complexity of*

$$O\left(g^{\frac{1}{2}+\log 7} p^{\frac{1}{2}} \mathsf{M}_{\mathrm{int}}(dg \log(dp)) + g^2 \mathsf{M}_{\mathrm{int}}(dg^{\frac{3}{2}} p^{\frac{1}{2}} \log(dgp)) + dg^{\frac{7}{2}} p^{\frac{1}{2}} \log g \log p\right)$$

   bit operations, with $O(dg^{\frac{7}{2}} p^{\frac{1}{2}} \log p + dg^{\frac{3}{2}} p^{\frac{1}{2}} \log d)$ storage.

The matrix H already gives information on the curve $\mathcal{C}$: for instance, H is invertible if and only if the Jacobian of $\mathcal{C}$ is ordinary [50, Corollary 2.3]. However, as stated in Theorem 16, the matrix H_π, and in particular its characteristic polynomial κ, tells much more and is required if the final goal is point-counting.

From now on, all operations are done in the effective ring $R' = \mathbb{F}_{p^d}$; hence the cost of the basic operations becomes $\mathsf{m}_{R'} \in O(\mathsf{M}_{\mathrm{int}}(d \log(dp)))$ and $\mathsf{s}_{R'} \in O(d \log(dp))$. The matrix H_π is the "norm" of H and as such can be computed with a binary powering algorithm. For simplicity, we assume that d is a power of 2; then, denoting

$$H_{\pi,i} = H H^{(p)} \cdots H^{(p^{2^i-1})},$$

we have

$$H_{\pi,i+1} = H_{\pi,i} \cdot (H_{\pi,i})^{(p^{2^i})}.$$

Hence computing $H_{\pi,i+1}$ from $H_{\pi,i}$ costs one matrix multiplication and 2^i matrix conjugations. A matrix conjugation consists of raising all the entries to the power p; therefore it costs $O(g^2 \mathsf{M}_{\mathrm{int}}(d \log(dp)) \log p)$ bit operations. The matrix we need to compute is $H_\pi = H_{\pi, \log d}$. Hence the cost of computing H_π is

$$O\left(\left(dg^2 \log p + g^{\log 7} \log d\right) \mathsf{M}_{\mathrm{int}}(d \log(dp))\right)$$

bit operations. The general case, where d is not a power of 2, is handled by adjusting the recursive step according to the binary expansion of d and yields the same complexity up to a constant factor.

The cost of the characteristic polynomial computation of an $n \times n$ matrix defined over an effective ring can be bounded by $O(n^4)$ operations in the ring using a sequential version of Berkowitz's algorithm [1]. This adds a negligible $O(g^4 \mathsf{M}_{\mathrm{int}}(d \log(dp)))$ contribution to the complexity.

If we are interested only in the complexity in p and d, i.e., if we assume that the genus is fixed, then the two strategies of Theorem 17 become equivalent, up to constant factors. Then, to summarize, the reduction modulo p of the characteristic polynomial χ of the Frobenius endomorphism can be computed in time

$$O\big(\mathsf{M}_{\mathrm{int}}(dp^{\frac{1}{2}}\log(dp)) + d\log p\,\mathsf{M}_{\mathrm{int}}(d\log(dp))\big)$$

bit operations and $O(dp^{\frac{1}{2}}\log(dp))$ storage.

Case of large genus. If $g \geq p$, then our analysis is no longer valid. In this paragraph, we assume that the function $\mathsf{M}_{\mathrm{int}}$ is essentially linear, i.e., we do not count logarithmic factors. Then the cost of strategy 1 of Theorem 17 is of order $dg^4p^{\frac{1}{2}}$ bit operations, the cost of strategy 2 is of order $dg^{\frac{7}{2}}p^{\frac{1}{2}}$, and that of the naive algorithm is in $O(dgp)$.

It turns out that for $p^{1/6} < g < p^{1-\varepsilon}$, the algorithms of Theorems 14 and 15 are not the fastest. Assume that $g > p^{1/6}$. Then it follows that $g^4p^{\frac{1}{2}} > gp$, and therefore the naive algorithm is faster than strategy 1 of Theorem 17. If further $g > p^{1/5}$, then $g^{\frac{7}{2}}p^{\frac{1}{2}} > gp$, and the naive algorithm is also faster than strategy 2. Thus, whatever the strategy used in Theorem 17, the parameter range for which our algorithms are interesting is far from the limit induced by the technical condition $g < p^{1-\varepsilon}$.

Combination with other point-counting algorithms. Computing the characteristic polynomial of H_π is not enough to deduce the group order of the Jacobian of $\mathcal{C}$, since only $\chi \bmod p$ is computed. We now survey different ways to complete the computation; we give rough complexity estimates, neglecting the logarithmic factors.

If p is small compared to g or d, p-adic algorithms [24, 37] have the best asymptotic complexity. These algorithms compute χ modulo high powers of p, so they necessarily recompute the information that has been obtained via the Cartier–Manin operator. Hence, our approach is of no interest here.

Consider next the extensions of Schoof's algorithm [34]. These algorithms have a complexity that is polynomial in $d\log p$ and exponential in g. For fixed g, our algorithm will be faster only if p is small compared to d, so that the power of d in the complexity of Schoof's algorithm can compensate the $\sqrt{p}$ complexity of our method. But in that case, our algorithm gives only very small information, and therefore the overall complexity of point counting is unchanged.

The combination with approaches based on the baby steps/giant steps algorithm (or low memory variants) is more fruitful, and can be of practical interest. Indeed, as far as we know, there is no implementation of any Schoof-like approach for genus greater than 2, and even for genus 2, the current record computations [19, 29] are obtained by combining many methods, including the baby steps/giant steps approach. Here is thus a short description of known approaches using BSGS ideas:

1. *BSGS method*: This is the generic method for finding the order of a group. If the order is known to be in an interval of width w, then the complexity is in $O(\sqrt{w})$. In the case of the Jacobian of $\mathcal{C}$, Hasse–Weil bounds give $w = O(p^{d(g-\frac{1}{2})})$, so the complexity is in $O(p^{d(\frac{g}{2}-\frac{1}{4})})$.
2. Computing the number of points of $\mathcal{C}$ in small degree extensions of $\mathbb{F}_{p^d}$ reduces the width of the search interval. Counting (naively) the points of $\mathcal{C}$ up to extension degree k costs $O(p^{dk})$, and the cost of the BSGS algorithm becomes $O(q^{d(\frac{g}{2}-\frac{k+1}{4})})$. This method (and additional practical improvements) is from [45]. We call it "approximation method" below.
3. When χ is known modulo some integer M, the group order is also known modulo M and therefore the BSGS method can be sped up by a factor of

$\sqrt{M}$. In [29] it is shown that in some cases a full factor M can be gained by doing a BSGS search on the coefficients of χ instead of just the group order. We abbreviate this method as MCT (from the names of the authors).

Thus, in genus 2, the complexity of the BSGS algorithm is in $O(p^{\frac{3d}{4}})$. For prime fields, our $O(p^{\frac{1}{2}})$ method is faster and gives essentially the complete information. For extension fields, our method gives the characteristic polynomial modulo p at a cost of $O(p^{\frac{1}{2}})$, from which we can recover the whole characteristic polynomial using the MCT algorithm at a cost of $O(p^{\frac{3d}{4}-1})$. Thus, for $d = 2$, the complexity is improved from $O(p^{\frac{3}{2}})$ to $O(p^{\frac{1}{2}})$, and for $d = 3$, from $O(p^{\frac{9}{4}})$ to $O(p^{\frac{5}{4}})$.

In genus 3, using the approximation method with $k = 1$ yields a complexity in $O(p^d)$. For prime fields, our method yields most of the information, the remaining part being computable using BSGS in time $O(p^{\frac{1}{4}})$. Hence, the cost drops from $O(p)$ to $O(p^{\frac{1}{2}})$; this is of practical interest, since the $O(p)$ algorithm is currently used for genus 3 point-counting over prime fields. For extension fields, the complexity drops from $O(p^d)$ to $O(p^{d-\frac{1}{2}})$.

The complexities for small degrees and genera are summarized in the following table. For each parameter set (g, d), there are two columns: the left-hand column describes the previously best known combination of methods; the right-hand one gives the new best combination with our algorithm (written "CM"). In each column we put an X in front of the algorithms that are used in the combination, and at the bottom list the total complexity.

	$g=2$						$g=3$					
	$d=1$		$d=2$		$d=3$		$d=1$		$d=2$		$d=3$	
BSGS	X		X		X		X	X	X	X	X	X
Approx.							X		X	X	X	X
MCT				X		X						
CM		X		X		X		X		X		X
Cplx.	$p^{3/4}$	$p^{1/2}$	$p^{3/2}$	$p^{1/2}$	$p^{9/4}$	$p^{5/4}$	p	$p^{1/2}$	p^2	$p^{3/2}$	p^3	$p^{5/2}$

Computer experiments. We have implemented our algorithm using Shoup's NTL C++ library [42]. NTL does not provide any arithmetic of local fields or rings, but allows one to work in finite extensions of rings of the form $\mathbb{Z}/p^g\mathbb{Z}$, as long as no divisions by p occur; the divisions by p are well isolated in the algorithm, so we could handle them separately. Furthermore, NTL multiplies polynomials defined over this kind of structure using an asymptotically fast FFT-based algorithm.

To illustrate that our method can be used as a tool in point-counting algorithms, we have computed the Zeta function of a (randomly chosen) genus 2 curve defined over $\mathbb{F}_{p^3}$, with $p = 2^{32} - 5$. Such a Jacobian has therefore about 2^{192} elements and should be suitable for cryptographic use if the group order has a large prime factor. Note that previous computations were limited to p of order 2^{23} [29].

The characteristic polynomial χ of the Frobenius endomorphism was computed modulo p in 3 hours and 41 minutes, using 1 GB of memory, on an AMD Athlon MP 2200+. Then we used the Schoof-like algorithms of [19] to compute χ modulo $128 \times 9 \times 5 \times 7$, and finally we used the modified BSGS algorithm of [29] to finish the computation. These other parts were implemented in Magma [5] and were performed in about 15 days of computation on an Alpha EV67 at 667 MHz. This computation was meant as an illustration of the possible use of our method, so little time was spent optimizing our code. In particular, the Schoof-like part and the final BSGS computations are done using a generic code that is not optimized for extension fields. Still, to our knowledge, on the same computers, such a computation would not have been possible with previous algorithms.

Appendix. Computations in the Turing model. In this appendix we discuss basic complexity results for polynomials and matrices over effective rings, in the multitape Turing machine model. We do not consider the operations used to control the computations, like incrementing an index in a loop: this is done on separate tapes, and the corresponding cost is negligible.

Proof of Lemma 3. Let $N \in \mathbb{N}$, $R_0 = \mathbb{Z}/N\mathbb{Z}$, and $R = R_0[T]/P$, with $P \in R_0[T]$ monic of degree m. We show here how to make R an effective ring. Elements of R_0 will be represented as integers in $0, \ldots, N-1$, and elements of R as sequences of m elements of R_0; representing such an element requires $\ell = m\lceil \log N \rceil$ bits.

Polynomials in $R_0[T]$ are multiplied as polynomials in $\mathbb{Z}[T]$; then their coefficients are reduced modulo N. Using Kronecker's substitution [15, Corollary 8.27], the multiplication in degree d is done in time $\mathsf{M}_{\mathrm{int}}(d\log(dN))$ and space $O(d\log(dN))$; the subsequent reduction is done by fast integer Euclidean division, using Cook's algorithm [12], which adds a negligible cost. Using Cook's algorithm again, Euclidean division in degree d in $R_0[T]$ can be done in time $O(\mathsf{M}_{\mathrm{int}}(d\log(dN)))$ and space $O(d\log(dN))$. In particular, taking $d = m$, this establishes the bounds on m_R and s_R given in the lemma.

Polynomials in $R[X]$ are multiplied as polynomials in $\mathbb{Z}[T, X]$, and then reduced modulo N and P, where the product in $\mathbb{Z}[T, X]$ is reduced to an integer product by bivariate Kronecker's substitution. In degree d, this yields time and space complexities M_R and S_R of, respectively, $O(\mathsf{M}_{\mathrm{int}}(dm\log(dmN)))$ and $O(dm\log(dmN))$.

We finally discuss matrix multiplication, contenting ourselves with the description of Strassen's algorithm [46, 15] for matrices of size $n = 2^k$ (which is enough to establish our claim). Each step of the algorithm requires us to compute 14 linear combinations of the 4 quadrants of the input matrices before entering recursive calls; 4 linear combinations of the 7 subproducts are performed after the recursive calls.

At each step in the recursion, the data has to be reorganized. The row-major representation of each input matrix is replaced with the consecutive row-major representations of its four quadrants, from which the linear combinations can be performed; a similar unfolding is done after the recursive calls. Taking into account the cost of this reorganization does not alter the complexity of this algorithm. This yields estimates for MM_R and SM_R, respectively, in $O(n^{\log 7}\mathsf{m}_R)$ and $O(n^{\log 7}\ell + \mathsf{s}_R)$, the term s_R standing for temporary memory used for scalar multiplications.

Finally, checking all required conditions on m_R, s_R, M_R, S_R, MM_R, and SM_R is straightforward.

Proof of Lemma 4. Let now R be an effective ring, with elements represented on ℓ bits. We prove here the assertions in Lemma 4.

- Trading inverses for multiplications: proof of Lemma 4, item 1. We use the notation of the proof of Lemma 1. Looking at the proof, one sees that all quantities R_i can be computed and stored on a tape T_1 in a single forward sweep of the input $r_0, \ldots, r_d$; reading the input backward, we compute all S_i and store them on a tape T_2. Finally, the output values s_i are computed by a single forward sweep of T_1 and T_2. The time complexity is $O(d\mathsf{m}_R)$ for multiplications, plus $O(d\ell)$ for tape movements; hence it fits in $O(d\mathsf{m}_R)$. The space complexity is $O(d\ell)$ bits for storage, plus s_R temporary bits for multiplications.
- Computing constants: proof of Lemma 4, items 2 and 3. We apply the same formulas as in the proof of Lemma 2. The cost of all operations is in $O(d\mathsf{m}_R)$; it is easy to check that the tape movements contribute with a negligible $O(d\ell)$ cost. As above, the space complexity is $O(d\ell)$ bits for storage,

plus s_R temporary bits for multiplications.

- Evaluation and interpolation: proof of Lemma 4, items 4 and 5. Let $r_0, \dots, r_d$ be in R. For simplicity, we suppose that the number of points is a power of 2, that is, $d+1 = 2^k$; the general case is handled similarly and presents only notational difficulties. All algorithms below are classical [15, Chapter 10]; our focus is on their adaptation in the Turing model.

For $i \le d$, set $A_{i,k} = X - r_i \in R[X]$; then, for $0 \le j \le k-1$ and $0 \le i \le 2^j - 1$, set $A_{i,j} = A_{2i,j+1}A_{2i+1,j+1}$. These polynomials will be arranged in a "subproduct tree," where $A_{2i,j+1}$ and $A_{2i+1,j+1}$ are the children of $A_{i,j}$. We now show how to compute this tree, writing A_j for the sequence $A_{0,j}, \dots, A_{2^j-1,j}$. Note that the sum of the degrees of the polynomials in A_j is $d+1$.

Given the sequence A_j, one can compute the extended sequence A_j, A_{j-1} in $O(\mathsf{M}_R(d))$ bit operations and space $O(d\ell + \mathsf{S}_R(2^{k-j}))$. It suffices to read the input once and to compute on the fly the products $A_{2i,j}A_{2i+1,j}$, storing them on an auxiliary tape, before appending all results to the input; the cost estimate follows from the superadditivity of M_R. Applying this $k = \log d$ times, one can compute the sequences $A_k, \dots, A_0$ in time $O(\mathsf{M}_R(d)\log d)$ and space $O(\ell d \log d + \mathsf{S}_R(d))$.

The evaluation algorithm uses the subproduct tree as follows. Let $P = P_{0,0}$ be of degree at most d, and set $P_{2i,j+1} = P_{i,j} \bmod A_{2i,j+1}$ and $P_{2i+1,j+1} = P_{i,j} \bmod A_{2i+1,j+1}$, for $0 \le i \le 2^j - 1$ and $0 \le j \le k$. We write P_j for the sequence $P_{0,j}, \dots, P_{2^j-1,j}$.

On input the sequences P_j and A_{j+1}, given on two distinct tapes, one can compute P_j, P_{j+1} in $O(\mathsf{M}_R(d))$ bit operations and space $O(\ell d + \mathsf{S}_R(2^{k-j}))$: we read once the input sequences and compute on the fly the remainders $P_{2i,j+1}$ and $P_{2i+1,j+1}$, storing them on an auxiliary tape; then they are appended to the sequence P_j. The estimates for Euclidean division and the superadditivity property then give the complexity estimate. Applying this $k = \log d$ times, given P and the sequence $A_k, \dots, A_0$, one can compute all $P_{i,k} = P(r_i)$ in time $O(\mathsf{M}_R(d)\log d)$ and space $O(\ell d \log d + \mathsf{S}_R(d))$.

It remains to deal with interpolation. Difficulties come from the inversion of quantities associated with the sample points. We thus suppose that $a_i = i$ for all $i \le d$ (this is what is used in this article), that $2, \dots, d$ are units in R, and that their inverses are known. Interpolating a polynomial P at $0, \dots, d$ is done by computing $\sum_{i \le d} P_i \prod_{j \ne i}(X - j)$, where $P_i = P(i)/\delta(i,d)$. The inverses of all $\delta(i,d)$ can be computed in time $O(d\mathsf{m}_R)$, by Lemma 4, item 2. Then, from [15, Chapter 10], the sum can be computed "going up" the subproduct tree, just as evaluation amounts to "going down" the tree. One checks that as above, it can be performed in time $O(\mathsf{M}_R(d)\log d)$ and space $O(\ell d \log d + \mathsf{S}_R(d))$.

Matrix of vectors and vectors of matrices. Let R be an effective ring, with elements represented using ℓ bits. Two representations for matrices with vector entries are used in this paper:

1. the row-major representation, where each entry is a vector over R, say of size k;
2. the vector representation, through a sequence of k scalar matrices, each in row-major representation.

In the Turing model, we must take care of data contiguity. We now give an algorithm that converts efficiently from one representation to the other; we start with a lemma on matrix transposition.

LEMMA 18. *In row-major representation, the transpose of an $m \times n$ matrix A can be computed in bit complexity $O(\ell mn \min(\log m, \log n))$ and space $O(\ell mn)$.*

Proof. Suppose that $n \leq m$. We first copy A from the input tape to an auxiliary tape, and pad on the fly the end of each line with an arbitrary symbol to make the column dimension equal to a power of 2. Thus we obtain an $m \times n'$ matrix A', where n' is a power of 2; the cost is in $O(mn\ell)$. We describe now a recursive algorithm that transposes A'; the transpose of A can be deduced as the top-left $n \times m$ submatrix of the transpose of A', and it can be copied on the output tape at a cost of $O(mn\ell)$.

We are thus reduced to transposing an $m \times n$ matrix A with n a power of 2. First, note that if $n = 1$, then the representations of A and of its transpose are the same. For $n \geq 2$ we proceed as follows. Let A_1 be the submatrix of A formed of the $n/2$ first coefficients of each row and A_2 the submatrix of A formed of their $n/2$ last coefficients. Then, the row-major representation of the transpose of A is the row-major representation of the transpose of A_1 followed by the row-major representation of the transpose of A_2. Hence computing the transpose of A amounts to the following operations:

- Uninterleaving: put A_1 followed by A_2 on a tape in place of the original A, using a temporary auxiliary tape.
- Recursively call to replace A_1 by its transpose at the same place, using a temporary auxiliary tape, and do the same with A_2.

The number $T(m, n)$ of tape movements verifies an equation of the form

$$T(m, n) \leq \lambda \ell mn + 2T(m, n/2),$$

for some constant λ. Therefore the overall cost is $O(\ell mn \log n)$ and the number of cells visited on each tape is at most ℓmn. This concludes the proof in the case $n \leq m$.

In the case $m \leq n$, we use essentially the same recursive algorithm but with the matrix split in two blocks of complete rows. Hence the algorithm for size m decomposes in two recursive calls at size $m/2$ and one subsequent step of interleaving the resulting matrices. In this way the $\log n$ factor is replaced with $\log m$. ∎

COROLLARY 19. *Let M be an $n \times n$ matrix, with entries in R^k. Switching between the two possible representations of M has bit complexity in $O(\ell n^2 k \min(\log n, \log k))$ and space complexity in $O(\ell n^2 k)$.*

Proof. Let M be represented on tape as a matrix of vectors. We can see the data of this tape as the row-major representation of an $n^2 \times k$ matrix over R. Let us compute the transpose of this matrix using the algorithm of Lemma 18. We obtain the representation of a $k \times n^2$ matrix over R; for $i \leq k$, its ith entry is the row-major representation of the $n \times n$ matrix made of the ith entries of M. ∎

Acknowledgments. We thank Bruno Salvy for his comments on this paper, Joachim von zur Gathen and Jürgen Gerhard for answering our questions on the complexity of integer factorization, and the referees for their helpful comments.

REFERENCES

[1] J. BERKOWITZ, *On computing the determinant in small parallel time using a small number of processors*, Inform. Process. Lett., 18 (1984), pp. 147–150.

[2] J. L. BORDEWIJK, *Inter-reciprocity applied to electrical networks*, Appl. Sci. Res. B., 6 (1956), pp. 1–74.

[3] A. BORODIN, *Time space tradeoffs (getting closer to the barrier?)*, in Proceedings of the 4th International Symposium on Algorithms and Computation, Lecture Notes in Comput. Sci. 762, Springer-Verlag, London, 1993, pp. 209–220.

[4] A. BORODIN AND R. T. MOENCK, *Fast modular transforms*, Comput. Systems Sci., 8 (1974), pp. 366–386.

[5] W. Bosma, J. Cannon, and C. Playoust, *The Magma algebra system.* I. *The user language*, J. Symbolic Comput., 24 (1997), pp. 235–265. See also http://www.maths.usyd.edu.au.

[6] A. Bostan, P. Gaudry, and É. Schost, *Linear recurrences with polynomial coefficients and computation of the Cartier-Manin operator on hyperelliptic curves*, in Proceedings of the International Conference on Finite Fields and Applications (Toulouse, 2003), Lecture Notes in Comput. Sci. 2948, Springer, Berlin, 2004, pp. 40–58.

[7] A. Bostan, G. Lecerf, and É. Schost, *Tellegen's principle into practice*, in Proceedings of the International Conference on Symbolic and Algebraic Computation, ACM Press, New York, 2003, pp. 37–44.

[8] P. Bürgisser, M. Clausen, and M. A. Shokrollahi, *Algebraic Complexity Theory*, Grundlehren der Math. Wiss. 315, Springer-Verlag, Berlin, 1997.

[9] D. G. Cantor and E. Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Inform., 28 (1991), pp. 693–701.

[10] P. Cartier, *Une nouvelle opération sur les formes différentielles*, C. R. Acad. Sci. Paris, 244 (1957), pp. 426–428.

[11] D. V. Chudnovsky and G. V. Chudnovsky, *Approximations and complex multiplication according to Ramanujan*, in Ramanujan Revisited (Urbana-Champaign, IL, 1987), Academic Press, Boston, MA, 1988, pp. 375–472.

[12] S. Cook, *On the Minimum Computation Time of Functions*, Ph.D. thesis, Harvard University, Cambridge, MA, 1966.

[13] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Comput., 9 (1990), pp. 251–280.

[14] P. Flajolet and B. Salvy, *The SIGSAM challenges: Symbolic asymptotics in practice*, SIGSAM Bull., 31 (1997), pp. 36–47.

[15] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, Cambridge, UK, 1999.

[16] J. von zur Gathen and V. Shoup, *Computing Frobenius maps and factoring polynomials*, Comput. Complexity, 2 (1992), pp. 187–224.

[17] P. Gaudry and N. Gürel, *Counting points in medium characteristic using Kedlaya's algorithm*, Experiment. Math., 12 (2003), pp. 395–402.

[18] P. Gaudry and R. Harley, *Counting points on hyperelliptic curves over finite fields*, in Algorithmic Number Theory (ANTS-IV), Lecture Notes in Comput. Sci. 1838, Springer, Berlin, 2000, pp. 313–332.

[19] P. Gaudry and É. Schost, *Construction of secure random curves of genus* 2 *over prime fields*, in Advances in Cryptology (EUROCRYPT 2004), C. Cachin and J. Camenisch, eds., Lecture Notes in Comput. Sci. 3027, Springer, Berlin, 2004, pp. 239–256.

[20] G. Hanrot, M. Quercia, and P. Zimmermann, *The middle product algorithm,* I. *Speeding up the division and square root of power series*, Appl. Algebra Engrg. Comm. Comput., 14 (2004), pp. 415–438.

[21] H. Hasse and E. Witt, *Zyklische unverzweigte Erweiterungskörper vom Primzahlgrade p über einem algebraischen Funktionenkörper der Charakteristik p*, Monatsch. Math. Phys., 43 (1936), pp. 477–492.

[22] E. Horowitz, *A fast method for interpolation using preconditioning*, Inform. Process. Lett., 1 (1972), pp. 157–163.

[23] E. Kaltofen, R. M. Corless, and D. J. Jeffrey, *Challenges of symbolic computation: My favorite open problems*, J. Symbolic Comput., 29 (2000), pp. 891–919.

[24] K. S. Kedlaya, *Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology*, J. Ramanujan Math. Soc., 16 (2001), pp. 323–338.

[25] D. E. Knuth, *The analysis of algorithms*, in Actes du Congrès International des Mathématiciens (Nice, 1970), Tome 3, Gauthier-Villars, Paris, 1971, pp. 269–274.

[26] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, *The number field sieve*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, ACM, New York, 1990, pp. 564–572.

[27] H. W. Lenstra, Jr., and C. Pomerance, *A rigorous time bound for factoring integers*, J. Amer. Math. Soc., 5 (1992), pp. 483–516.

[28] J. I. Manin, *The Hasse-Witt matrix of an algebraic curve*, Trans. Amer. Math. Soc., 45 (1965), pp. 245–264.

[29] K. Matsuo, J. Chao, and S. Tsujii, *An improved baby step giant step algorithm for point counting of hyperelliptic curves over finite fields*, in Algorithmic Number Theory (ANTS-V), Lecture Notes in Comput. Sci. 2369, Springer, Berlin, 2002, pp. 461–474.

[30] J. McKee and R. Pinch, *Old and new deterministic factoring algorithms*, in Algorithmic Number Theory (Talence, 1996), Lecture Notes in Comput. Sci. 1122, Springer, Berlin, 1996, pp. 217–224.

[31] R. T. Moenck and A. Borodin, *Fast modular transforms via division*, in Proceedings of the Thirteenth Annual IEEE Symposium on Switching and Automata Theory (University of Maryland, College Park, MD), 1972, pp. 90–96.
[32] P. L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Math. Comp., 48 (1987), pp. 243–264.
[33] P. L. Montgomery, *An FFT Extension of the Elliptic Curve Method of Factorization*, Ph.D. thesis, University of California, Los Angeles CA, 1992.
[34] J. Pila, *Frobenius maps of abelian varieties and finding roots of unity in finite fields*, Math. Comp., 55 (1990), pp. 745–763.
[35] J. M. Pollard, *Theorems on factorization and primality testing*, Proc. Cambridge Philos. Soc., 76 (1974), pp. 521–528.
[36] C. Pomerance, *Analysis and comparison of some integer factoring algorithms*, in Computational Methods in Number Theory, Part I, Math. Centre Tracts 154, Math. Centrum, Amsterdam, 1982, pp. 89–139.
[37] T. Satoh, *The canonical lift of an ordinary elliptic curve over a finite field and its point counting*, J. Ramanujan Math. Soc., 15 (2000), pp. 247–270.
[38] A. Schönhage, *Schnelle Berechnung von Kettenbruchentwicklungen*, Acta Inform., 1 (1971), pp. 139–144.
[39] A. Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik* 2, Acta Inform., 7 (1977), pp. 395–398.
[40] A. Schönhage, A. F. W. Grotefeld, and E. Vetter, *Fast Algorithms*, Bibliographisches Institut, Mannheim, 1994.
[41] A. Schönhage and V. Strassen, *Schnelle Multiplikation großer Zahlen*, Computing, 7 (1971), pp. 281–292.
[42] V. Shoup, *NTL: A library for doing number theory*. http://www.shoup.net/ntl (2005).
[43] V. Shoup, *A fast deterministic algorithm for factoring polynomials over finite fields of small characteristic*, in Proceedings of the International Conference on Symbolic and Algebraic Computation, ACM Press, New York, 1991, pp. 14–21.
[44] J. H. Silverman, *The Arithmetic of Elliptic Curves*, Graduate Texts in Math. 106, Springer-Verlag, New York, 1996.
[45] A. Stein and H. Williams, *Some methods for evaluating the regulator of a real quadratic function field*, Experiment. Math., 8 (1999), pp. 119–133.
[46] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.
[47] V. Strassen, *Die Berechnungskomplexität von elementarsymmetrischen Funktionen und von Interpolationskoeffizienten*, Numer. Math., 20 (1972/73), pp. 238–251.
[48] V. Strassen, *Einige Resultate über Berechnungskomplexität*, Jber. Deutsch. Math.-Verein., 78 (1976/77), pp. 1–8.
[49] B. Tellegen, *A general network theorem, with applications*, Philips Res. Rep., 7 (1952), pp. 259–269.
[50] N. Yui, *On the Jacobian varieties of hyperelliptic curves over fields of characteristic* $p > 2$, J. Algebra, 52 (1978), pp. 378–410.