Implementations of efficient univariate polynomial matrix algorithms and application to bivariate resultants

Seung Gyu Hyun, Vincent Neiger, Éric Schost

Implementations of efficient univariate polynomial matrix algorithms and application to bivariate resultants

Seung Gyu Hyun
University of Waterloo
Waterloo, ON, Canada
kevin.hyun@edu.uwaterloo.ca

Vincent Neiger
Univ. Limoges, CNRS, XLIM, UMR 7252
F-87000 Limoges, France
vincent.neiger@unilim.fr

Éric Schost
University of Waterloo
Waterloo, ON, Canada
eschost@uwaterloo.ca

Abstract

Complexity bounds for many problems about matrices with univariate polynomial entries have been improved in the last few years. Still, for most recent algorithms, efficient implementations are not yet available. This leaves open the question of the practical impact of these algorithms on potential applications, which include decoding some error-correcting codes and solving polynomial systems or structured linear systems.

In this paper, we describe the implementation of some of the most fundamental algorithms for polynomial matrices: multiplication, truncated inversion, approximants, interpolants, kernels, linear system solving, and determinant. Our work currently focuses on prime fields with a word-size modulus and is based on Shoup's C++ library NTL. We combine these new tools to implement variants of Villard's recent algorithm for the resultant of generic bivariate polynomials (ISSAC 2018), and exhibit parameter ranges for which they outperform previous state of the art.

CCS Concepts

• **Mathematics of computing** → **Computations on matrices**; **Computations on polynomials**; • **Computing methodologies** → **Algebraic algorithms**.

Keywords

Polynomial matrices, algorithms, implementation, resultant.

1 Introduction

Recent years have witnessed a host of activity on fast algorithms for polynomial matrices and their applications. Consider for example the following contributions from the last 10 years (hereafter, $\mathbb{K}$ is a field and $\mathbb{K}[x]$ is the algebra of univariate polynomials over $\mathbb{K}$):

- Minimal approximant bases [17, 57] were used to compute kernel bases [58], giving the first efficient deterministic algorithm for linear system solving over $\mathbb{K}[x]$.
- Basis reduction [17, 19] played a key role in accelerating the decoding of one-point Hermitian codes [39] and in designing deterministic determinant and Hermite form algorithms [33].
- Progress on minimal interpolant bases [26, 27] led to the best known complexity bound for list-decoding Reed-Solomon codes and folded Reed-Solomon codes [27, Sec. 2.4 to 2.7].
- Coppersmith's block Wiedemann algorithm and its extensions [8, 30, 52] were used in a variety of contexts, from integer factorization [48] to polynomial system solving [25, 53].

At the core of these improvements, in addition to the algorithms explicitly mentioned, one also finds techniques such as high-order lifting [45] and partial linearization [46],[19, Sec. 6].

For many of these operations, no implementation of the latest algorithms is available and no experimental evidence has been given regarding their practical behavior. Our goal is to partly remedy this issue, by providing implementations for a core of fundamental algorithms such as polynomial matrix multiplication, approximant and interpolant bases, etc., upon which one may implement higher level algorithms. As an illustration, we describe the performance of slightly modified versions of Villard's recent breakthroughs on bivariate resultant and characteristic polynomial computation [53].

Our implementation is based on Shoup's Number Theory Library (NTL) [44], and is dedicated to polynomial matrix arithmetic over $\mathbb{K} = \mathbb{F}_p$ for a word-size prime p. Particular attention was paid to performance issues, so that our library compares favorably with previous work for those operations where comparisons were possible. Our code is available at https://github.com/vneiger/pml.

Overview. Basic ingredients for polynomial matrix algorithms are efficient arithmetic in $\mathbb{K}[x]$ and efficient matrix arithmetic over $\mathbb{K}$; in Section 2, we review some related algorithms and discuss their NTL implementations. Then, we describe how we implemented a further key building block, polynomial matrix multiplication.

Section 3 presents the next major part of our work, concerning algorithms for *approximant bases* and *interpolant bases*. Algorithms for the former are well-known [2, 17, 28, 57], and the latter were studied in [3, 26, 27, 51]; we focus here on a version of interpolants which is slightly less general but allows for a simpler and more efficient algorithm. In particular, we show that with this version, algorithms for interpolant bases can be as efficient or even faster than those for approximant bases, and that both can be used interchangeably in several contexts. In Section 4, we discuss algorithms for minimal kernel bases, linear system solving, determinant, and row reduction. Finally, Section 5 uses several of these tools to study the practical behavior of Villard's bivariate resultant algorithm [53].

To describe the cost of these algorithms, we use an algebraic complexity model, counting all operations in the base field at unit cost. While standard, this point of view fails to describe some parts of the implementation (Chinese Remaindering-based algorithms, such as the so-called 3-primes FFT, cannot be described in such a manner), but we believe that this is a minor issue.

Implementation choices. NTL is a C++ library for polynomial and matrix arithmetic over rings such as $\mathbb{Z}$, $\mathbb{Z}/n\mathbb{Z}$, etc., and is often seen as the *de facto* reference point for fast algorithms in such contexts. In our case, other libraries could serve as a starting point, a natural choice being the combination of FFLAS-FFPACK [12, 49] and LinBox [50]. While NTL offers an extensive array of algorithms for polynomials, together with some linear algebra operations (such as matrix multiplication and Gaussian elimination over prime fields), FFLAS-FFPACK implements a large variety of linear algebra algorithms over finite fields and LinBox includes polynomial matrix multiplication and approximant bases. Deciding factors were the availability of many polynomial operations in NTL such as fast extended GCD and fast modular computations when the same modulus is used multiple times; the fact that NTL offers native support for FFT primes up to 60 bits; and, above all, our goal to implement recent resultant algorithms and compare their performance to the state of the art, which happens to be NTL's built-in routines.

In our implementation, the base field is a prime finite field $\mathbb{F}_p$; we rely on NTL's `lzz_p` class. At the time of writing, on standard x86_64 platforms, NTL v11.3.1 uses `unsigned long`'s as its primary data type for `lzz_p`, supporting moduli up to 60 bits long.

For such fields, one can directly compare performance timings and cost bounds, since most polynomial matrix algorithms in the literature are analyzed in the algebraic complexity model. Furthermore, computing modulo primes is at the core of a general approach consisting in solving problems over $\mathbb{Z}$ or $\mathbb{Q}$ by means of reduction modulo sufficiently many primes. In this case, the primes are chosen so as to satisfy several, partly conflicting, objectives. We may want them to support Fourier Transforms of high orders. Linear algebra modulo each prime should be fast, so we may wish these primes to be small enough to support vectorized matrix arithmetic (for example with AVX instructions). On the other hand, using larger primes makes it possible to use fewer of them; also, for randomized algorithms, this reduces the likelihood of unlucky choices.

As a result, while all NTL `lzz_p` moduli are supported, our implementation puts an emphasis on three families: small FFT primes that support AVX-based matrix multiplication (such primes have at most 23 bits); arbitrary size FFT primes (at most 60 bits); arbitrary moduli (at most 60 bits). Very small fields such as $\mathbb{F}_2$ or $\mathbb{F}_3$ are supported, but no particular effort was made to optimize the implementation for such cases (for instance, NTL provides a dedicated class for arithmetic over $\mathbb{F}_2$, but we currently do not exploit it).

Experiments. All runtimes below are in seconds and were measured on an Intel Core i7-4790 CPU with 32GB RAM, using the version 11.3.1 of NTL. Unless specified otherwise, timings are obtained modulo a random 60 bit prime. Runtimes were measured on a single thread; currently, most parts of our code do not explicitly exploit multi-threading. Since most algorithms have two or more input parameters, we do not give plots but tables showing a few selected timings, with the best time(s) in bold; for more timings we refer the reader to https://github.com/vneiger/pml/tree/master/benchmarks.

2 Basic polynomial and matrix arithmetic

We review basic algorithms for polynomials and matrices, and related complexity results that hold over an abstract field $\mathbb{K}$, and we describe how we implemented these operations. Hereafter, for $d \geq 0$, $\mathbb{K}[x]_d$ is the set of elements of $\mathbb{K}[x]$ of degree less than d.

2.1. Polynomial multiplication. Multiplication in $\mathbb{K}[x]$ and Fast Fourier Transform (FFT) are cornerstones of most algorithms in this paper. Let $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ be a function such that polynomials of degree at most d in $\mathbb{K}[x]$ can be multiplied in $\mathsf{M}(d)$ operations in $\mathbb{K}$. If $\mathbb{K}$ supports FFT, we can take $\mathsf{M}(d) \in O(d \log(d))$, and otherwise, $\mathsf{M}(d) \in O(d \log(d) \log\log(d))$ [13, Chapter 8]; as in this reference, we assume that $d \mapsto \mathsf{M}(d)/d$ is increasing.

A useful variant of multiplication in $\mathbb{K}[x]$ is the *middle product* [5, 20]: for integers c and d, and F in $\mathbb{K}[x]_c$ and G in $\mathbb{K}[x]_{c+d}$, $\textsc{MiddleProduct}(F, G, c, d)$ returns the slice with coefficients of degrees $c, \ldots, c + d - 1$ of the product FG; a common case is with $c = d$. The direct approach computes the whole product and extracts the slice. Yet, the *transposition principle* [31] shows that the middle product can be computed in time $\mathsf{M}(c, d) + O(c+d)$, saving a constant factor (roughly a factor 2 when $c = d$, if FFT multiplication is used).

Polynomial matrix algorithms frequently use fast evaluation and interpolation at multiple points. In general, subproduct tree techniques [13, Chapter 10] allow one to do evaluation and interpolation of polynomials in $\mathbb{K}[x]_d$ at d points in $O(\mathsf{M}(d) \log(d))$ operations. For special sets of points, one can do better: if we know α in $\mathbb{K}$ of order at least d, then evaluation and interpolation at the geometric progression $(1, \alpha, \ldots, \alpha^{d-1})$ can both be done in time $O(\mathsf{M}(d))$ [6].

In NTL, multiplication in $\mathbb{F}_p[x]$ uses either naive, Karatsuba, or FFT techniques, depending on p and on the degree (NTL provides FFT primes with roots of unity of order 2^{25}, and supports arbitrary user-chosen FFT primes). FFT multiplication uses the TFT algorithm of [21] and Harvey's improvements on arithmetic mod p [22]. For primes p that do not support Fourier transforms, multiplication is done by means of either 3-primes FFT techniques [13, Chapter 8] or Schönhage and Strassen's algorithm. We implemented middle products for naive, Karatsuba and FFT multiplication, closely following [5, 20], as well as evaluation/interpolation algorithms for general sets of points and for geometric progressions.

2.2. Matrix multiplication. For cost analyses, let ω be such that $n \times n$ matrices over any ring can be multiplied by a bilinear algorithm doing $O(n^\omega)$ ring operations. The naive algorithm does exactly n^3 multiplications. First improvements due to Winograd and Waksman [54, 55] reduced the number of operations to $n^3/2 + O(n^2)$ if 2 is a unit. Strassen's and Winograd's recursive algorithms [47, 56] have $\omega = \log_2(7)$; the best known bound is $\omega \leq 2.373$ [9, 34]. Note that, using blocking, rectangular matrices of sizes $(m \times n)$ and $(n \times p)$ can be multiplied in $O(mnp \min(m, n, p)^{\omega-3})$ ring operations.

Unlike libraries such as FFLAS-FFPACK whose matrix multiplication relies on an external BLAS library, NTL implements its own arithmetic for matrices over $\mathbb{F}_p$. It chooses one of several implementations depending on the bitsize of p, the matrix dimensions, the available processor instructions, etc. On our platforms, for dimensions up to a few thousands, timings for matrix multiplication are very close between NTL and FFLAS for primes of bitsize about 20, while NTL is slightly faster for primes of bitsize closer to 60.

2.3. Polynomial matrix multiplication. In what follows, we write $\mathsf{MM}(n, d)$ for a function such that two $n \times n$ matrices of degree at most d can be multiplied in $\mathsf{MM}(n, d)$ operations in $\mathbb{K}$; we make the assumption that $d \mapsto \mathsf{MM}(n, d)/d$ is increasing for all n.

It follows from the definitions above that $\mathsf{MM}(n, d) \in O(n^\omega \mathsf{M}(d))$, which is in $\tilde{O}(n^\omega d)$. Yet, better bounds on $\mathsf{MM}(n, d)$ are known:

- $O(n^\omega d \log(d) + n^2 d \log(d) \log(\log(d)))$ for any field $\mathbb{K}$ [7];
- $O(n^\omega d 4^{\log^*(d)} + n^2 d \log(d) 8^{\log^*(d)})$ if $\mathbb{K}$ is finite [23, Sec. 8];
- $O(n^\omega d + n^2 \mathsf{M}(d))$ if an element α in $\mathbb{K}$ of order more than $2d$ is known [6, Thm. 2.4].
- $O(n^\omega d + n^2 d \log(d))$ if $\mathbb{K}$ supports FFT in degree $2d$.

The last two bounds are obtained by evaluation/interpolation, either at the geometric progression $1, \alpha, \ldots, \alpha^{2d}$ or at roots of unity.

Finally, we mention a polynomial analogue of an integer matrix multiplication algorithm due to Doliskani *et al.* [11]. It is also based on evaluation/interpolation, but these are done by plain multiplication by Vandermonde and inverse Vandermonde matrices. Then, the corresponding part of the cost (e.g. $O(n^2\mathsf{M}(d))$ for geometric progressions) is replaced by the cost of multiplying matrices over $\mathbb{K}$ in sizes roughly $(d \times d)$ by $(d \times n^2)$; this is in $O(n^2 d^{\omega-1})$ if $d \le n^2$. For moderate values of d, where $\mathsf{M}(d)$ is not in the FFT regime yet, this allows us to leverage fast matrix multiplication over $\mathbb{K}$.

We implemented and compared various algorithms for matrix multiplication over $\mathbb{F}_p[x]$. For matrices of degree less than 5, we use dedicated routines based on Karatsuba's and Montgomery's formulas [36]; for matrices of small size (up to 10, depending on p), we use Waksman's algorithm. For other inputs, most of our efforts were spent on variants of the evaluation/interpolation scheme.

For FFT primes, we use evaluation/interpolation at roots of unity. For general primes, we use either evaluation/interpolation at geometric progressions (if such points exist in $\mathbb{F}_p$), or our adaptation of the algorithm of [11], or 3-primes multiplication (as for polynomials, we lift the product from $\mathbb{F}_p[x]$ to $\mathbb{Z}[x]$, where it is done modulo up to 3 FFT primes). No single variant outperformed or underperformed all others for all sizes and degrees, so thresholds were experimentally determined to switch between these options, with different values for small (less than 23 bits) and for large primes.

Middle product versions of these algorithms were implemented, and are used in approximant basis algorithms (Section 3.1) and Newton iteration (Section 4.3). Multiplier classes are available, that store values of a matrix $\mathbf{A}$ for cases where repeated multiplications by $\mathbf{A}$ are needed; they are used in Dixon's algorithm for linear system solving (Section 4.3).

Finally, in Table 1, we show timings for our polynomial matrix multiplication and LinBox' one. We used random $m \times m$ matrices of degree d, with p either a 20 bit FFT prime or a 60 bit prime. For the 20 bit FFT prime, LinBox' implementation slightly outperforms ours for matrices of dimension about 20 and more; in all other cases, the timings are either similar or in favor of our implementation.

Table 1: *Polynomial matrix multiplication*

		20 bit FFT prime			60 bit prime		
m	d	ours	Linbox	ratio	ours	Linbox	ratio
8	131072	**1.098**	1.510	0.73	**3.577**	13.59	0.26
32	4096	0.604	**0.492**	1.22	**2.000**	5.330	0.38
128	1024	2.69	**1.973**	1.36	**15.73**	23.13	0.68
512	128	6.085	**4.006**	1.52	**41.57**	50.62	0.82

3 Approximant bases and interpolant bases

These bases are matrix generalizations of Padé approximation and play an important role in many higher-level algorithms. For $\mathbf{F}$ in $\mathbb{K}[x]^{m\times n}$ and M non-constant in $\mathbb{K}[x]$, they are bases of the $\mathbb{K}[x]$-module $\mathcal{A}_M(\mathbf{F})$ of all $\boldsymbol{p}$ in $\mathbb{K}[x]^{1\times m}$ such that $\boldsymbol{p}\mathbf{F} = 0 \bmod M$. Specifically, *approximant bases* are for $M = x^d$ and *interpolant bases* for $M = \prod_i (x - \alpha_i)$ for d distinct points $\alpha_1, \ldots, \alpha_d$ in $\mathbb{K}$. (Here, we do not consider more general cases from the literature, for example with several moduli $M_1, \ldots, M_n$, one for each column of $\boldsymbol{p}\mathbf{F}$.)

Since $\mathcal{A}_M(\mathbf{F})$ is free of rank m, such a basis is represented row-wise by a nonsingular $\mathbf{P}$ in $\mathbb{K}[x]^{m\times m}$. The algorithms below return $\mathbf{P}$ in *$\boldsymbol{s}$-ordered weak Popov form* (also known as $\boldsymbol{s}$-quasi Popov form [4]), for a given *shift* $\boldsymbol{s} = (s_1, \ldots, s_m)$ in $\mathbb{Z}^m$. Shifts allow us to set degree constraints on the sought basis $\mathbf{P}$, and they inherently occur in a general approach for computing bases of solutions to equations (approximants, interpolants, kernels, etc.). Approximant basis algorithms often require $\mathbf{P}$ to be in *$\boldsymbol{s}$-reduced form* [51]; although the $\boldsymbol{s}$-ordered weak Popov form is stronger, obtaining it involves minor changes in these algorithms, without impact on performance according to our experiments. Besides, recent literature shows that having $\mathbf{P}$ in this stronger form yields information (via the pivots) which is valuable for further computations with $\mathbf{P}$ [26, 28], in particular for finding bases in *$\boldsymbol{s}$-Popov form* [4].

From the shift $\boldsymbol{s}$, the *$\boldsymbol{s}$-degree* of $\boldsymbol{p} = [p_i]_i \in \mathbb{K}[x]^{1\times m}$ is defined as $\mathrm{rdeg}_{\boldsymbol{s}}(\boldsymbol{p}) = \max_{1\le i\le m}(\deg(p_i) + s_i)$, which extends to matrices: $\mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{P})$ is the list of $\boldsymbol{s}$-degrees of the rows of $\mathbf{P}$. Then, the *$\boldsymbol{s}$-pivot* of $\boldsymbol{p}$ is its rightmost entry p_i such that $\mathrm{rdeg}_{\boldsymbol{s}}(\boldsymbol{p}) = \deg(p_i) + s_i$, and a nonsingular matrix $\mathbf{P}$ is in $\boldsymbol{s}$-ordered weak Popov form if the $\boldsymbol{s}$-pivots of its rows are located on the diagonal.

To simplify cost bounds below, we make use of the function

$$\mathsf{MM}'(m, d) = \sum_{i=0}^{\log_2(d)} 2^i \mathsf{MM}(m, d/2^i),$$

which is in $O(\mathsf{MM}(m, d)\log(d))$, and thus in $\tilde{O}(m^\omega d)$.

3.1. Approximant bases. As written above, for $\mathbf{F}$ in $\mathbb{K}[x]^{m\times n}$ and d in $\mathbb{Z}_{>0}$, an *approximant basis* for $(\mathbf{F}, d)$ is a nonsingular $m \times m$ matrix whose rows form a basis of $\mathcal{A}_{x^d}(\mathbf{F})$.

We implemented minor variants of the algorithms M-Basis (iterative, via matrix multiplication) and PM-Basis (divide and conquer, via polynomial matrix multiplication) from [17]. The lowest-level function, M-Basis-1, handles order $d = 1$ in time $O(\mathrm{rank}(\mathbf{F})^{\omega-2}mn)$; here, since we work modulo x, the matrix $\mathbf{F}$ is over $\mathbb{K}$. Our implementation follows [28, Algo. 1], which has the following signature.

> *Algorithm 1:* M-Basis-1$(\mathbf{F}, \boldsymbol{s})$
> *Input:* matrix $\mathbf{F}$ in $\mathbb{K}^{m\times n}$, shift $\boldsymbol{s}$ in $\mathbb{Z}^m$
> *Output:* the $\boldsymbol{s}$-Popov approximant basis for $(\mathbf{F}, 1)$

We chose this version rather than those in [17, 18] because its output is in $\boldsymbol{s}$-ordered weak Popov form, only at the price of an additional row permutation. This property suffices to ensure that M-Basis and PM-Basis return bases in this form as well.

In our implementation, the matrix $\mathbf{L}$ in [28, Algo. 1] is directly obtained from NTL's Gaussian elimination. In most cases we call it once (via the `kernel` function), yet for some rare "bad" inputs we use a second call (via the `image` function) to ensure that $\mathbf{L}$ has the required echelon form. This minor issue is because NTL does not provide matrix decompositions such as LSP or PLUQ; here, relying on FFPACK may prove valuable, at least for small primes p.

Our implementation of M-Basis follows the original design [17] (see also [18, Sec. 3.2]), with d iterations, each computing the so-called *residual* $\mathbf{R}$ and updating $\mathbf{P}$ via multiplication by a basis $\mathbf{Q}$ obtained by M-Basis-1 on $\mathbf{R}$. We also follow [17] for PM-Basis, using a threshold T such that M-Basis is called for orders $d \le T$. Building PM-Basis directly upon M-Basis-1, i.e. choosing $T = 1$, achieves the same asymptotic cost bound but is slower in practice.

The cost of M-Basis is $O((m^\omega + m^{\omega-1}n)d^2)$ operations in $\mathbb{K}$ and that of PM-Basis is $O((1 + n/m)\mathsf{MM}'(m, d))$ [17]. These algorithms

Input: matrix $\mathbf{F}$ in $\mathbb{K}[x]^{m\times n}$, order d in $\mathbb{Z}_{>0}$, shift $\boldsymbol{s}$ in $\mathbb{Z}^m$
Output: an $\boldsymbol{s}$-ordered weak Popov approximant basis for $(\mathbf{F}, d)$

Algorithm 2: M-Basis$(\mathbf{F}, d, \boldsymbol{s})$

1. $\mathbf{P} \leftarrow$ identity matrix in $\mathbb{K}[x]^{m\times m}$, and $\boldsymbol{t} \leftarrow$ copy of $\boldsymbol{s}$
2. *For* $k = 0, \ldots, d-1$:
  a. $\mathbf{R} \in \mathbb{K}^{m\times n} \leftarrow$ coefficient of $\mathbf{PF}$ of degree k
  b. $\mathbf{Q} \in \mathbb{K}[x]^{m\times m} \leftarrow$ M-Basis-1$(\mathbf{R}, \boldsymbol{t})$
  c. $\mathbf{P} \leftarrow \mathbf{QP}$, and then $\boldsymbol{t} \leftarrow \mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{P})$

Algorithm 3: PM-Basis$(\mathbf{F}, d, \boldsymbol{s})$

1. *if* $d \le T$ *return* M-Basis$(\mathbf{F}, d, \boldsymbol{s})$
2. $\mathbf{P}_1 \leftarrow$ PM-Basis$(\mathbf{F} \bmod x^{\lceil d/2\rceil}, \lceil d/2\rceil, \boldsymbol{s})$
3. $\mathbf{R} \leftarrow$ MiddleProduct$(\mathbf{P}_1, \mathbf{F}, \lceil d/2\rceil, \lfloor d/2\rfloor)$
4. $\boldsymbol{t} \leftarrow \mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{P}_1)$
5. $\mathbf{P}_2 \leftarrow$ PM-Basis$(\mathbf{R}, \lfloor d/2\rfloor, \boldsymbol{t})$
6. *return* $\mathbf{P}_2\mathbf{P}_1$

form the bulk of the approximant bases algorithms we implemented; some implementation details and timings are given in Section 3.2.

We also implemented [28, Algo. 3] which returns $\boldsymbol{s}$-Popov bases at the price of a factor about 2 in performance. Future work includes making this overhead negligible for cases that arise in applications.

For completeness, we handle general approximants with several moduli (one per column of $\mathbf{F}$) by an iterative algorithm from [2, 51]; faster algorithms are more complex [26–28] and involve partial linearization techniques.

These techniques were introduced in [46, 57] to obtain faster algorithms when $n \ll m$, with cost $\tilde{O}(m^{\omega-1}nd)$ instead of $\tilde{O}(m^{\omega}d)$ achieved by PM-Basis. Implementing these techniques is work in progress and can bring substantial improvements. Experimental code, which focuses for simplicity on "generic" inputs for which the degrees in $\mathbf{P}$ can be predicted, led to significant speedups (Table 2).

Table 2: *Accelerating $n = 1$ using partial linearization*

m	d	PM-Basis	PM-Basis with linearization
4	65536	1.6693	**1.26891**
16	16384	1.8535	**0.89652**
64	2048	2.2865	**0.14362**
256	1024	36.620	**0.20660**

In Table 3, we compare timings for LinBox' and our implementations of PM-Basis, for a 20 bit FFT prime. In this case ours has a moderate advantage; for large primes and general primes LinBox was at a significant disadvantage, for reasons that are unclear to us.

Table 3: *Approximant basis with* PM-Basis *(20 bit FFT prime)*

m	n	d	ours	Linbox	ratio
8	4	128	**0.0024**	0.0050	0.48
8	4	131072	**7.9307**	15.2605	0.52
32	16	128	**0.0226**	0.0436	0.52
32	16	8192	**5.3096**	6.9188	0.77
128	64	32	**0.0780**	0.1274	0.61
128	64	2048	**19.6627**	27.5212	0.71
512	256	256	**37.2000**	41.1077	0.90

Approximant bases are often applied to solve block-Hankel systems [32]. Now, we compare this approach to the one which uses structured matrix algorithms; below, we use the solver from [24], which is based on NTL as well. We are not aware of previous comparisons of this kind. Precisely, we study two situations.

First, we call PM-Basis on $[\mathbf{F}^{\mathsf{T}} \quad -\mathbf{I}_m]^{\mathsf{T}}$ at order $2d$ with shift $(0, \ldots, 0)$, where $\mathbf{F}$ is an $m\times m$ matrix of degree $2d-1$, and we solve a system with $m \times m$ Hankel blocks of size $d \times d$ (the structured solver returns a random solution to the system). Our experiments show a clear advantage for approximant algorithms (see Table 4). The asymptotic costs being similar, the effects at play here are constant factor differences: approximant basis algorithms seem to be somewhat simpler and to better leverage the main building blocks (matrix arithmetic over $\mathbb{K}$ and univariate polynomial arithmetic).

Second, we consider a vector rational reconstruction setting: we call PM-Basis on $[\mathbf{F}^{\mathsf{T}} \quad -\mathbf{I}_m]^{\mathsf{T}}$ at order $(m+1)d$ with shift $(0, \ldots, 0)$, where $\mathbf{F}$ is a row vector of degree $(m+1)d-1$, and we solve a block system with $1 \times m$ Hankel blocks of size $md \times d$. The former uses $\tilde{O}(m^{\omega+1}d)$ operations while the latter costs $\tilde{O}(m^{\omega}d)$. Approximants are still faster up to dimension about 15, which is explained by the same arguments as in the previous paragraph. On the other hand, as predicted by the cost estimates, the block-Hankel solver is more efficient for larger dimensions.

Table 4: PM-Basis *vs. structured system solver*

		$m \times m$		$1 \times m$	
m	d	PM-Basis	solver	PM-Basis	solver
5	8000	**0.996**	8.23	**2.19**	3.820
12	1000	**0.687**	6.18	2.33	2.28
30	500	**2.84**	42.5	19.5	**11.5**

3.2. Interpolant bases.

For matrices $\mathbf{E} = (\mathbf{E}_1, \ldots, \mathbf{E}_d)$ in $\mathbb{K}^{m\times n}$ and pairwise distinct points $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$ in $\mathbb{K}$, consider

$$\mathcal{I}_{\boldsymbol{\alpha}}(\mathbf{E}) = \{\boldsymbol{p} \in \mathbb{K}[x]^{1\times m} \mid \boldsymbol{p}(\alpha_i)\mathbf{E}_i = 0 \text{ for } 1 \le i \le d\}.$$

An *interpolant basis* for $(\mathbf{E}, \boldsymbol{\alpha})$ is a matrix whose rows form a basis of the $\mathbb{K}[x]$-module $\mathcal{I}_{\boldsymbol{\alpha}}(\mathbf{E})$. Note that $\mathcal{I}_{\boldsymbol{\alpha}}(\mathbf{F}(\alpha_1), \ldots, \mathbf{F}(\alpha_d))$ coincides with $\mathcal{A}_M(\mathbf{F})$, for $\mathbf{F}$ in $\mathbb{K}[x]^{m\times n}$ and $M = \Pi_{i=1}^{d}(x - \alpha_i)$.

This definition is a specialization of those in [3, 27], which consider n sets of points, one for each of the n columns of $\mathbf{E}_1, \ldots, \mathbf{E}_d$: here, these sets are all equal. This more restrictive problem allows us to give faster algorithms than those in these references, by direct adaptations of the approximant basis algorithms presented above. Besides, Sections 4.1 and 4.2 will show that interpolant bases can often play the same role as approximant bases in applications.

In Algorithms 4 and 5, we describe the modified M-Basis and PM-Basis; we write $\boldsymbol{\alpha}_{i\ldots j}$ for the sublist $(\alpha_i, \alpha_{i+1}, \ldots, \alpha_j)$.

In the next proposition, we assume that $\mathsf{MM}(n, d)$ is in $\Omega(n^2\mathsf{M}(d))$ (instead, one may add an extra term $O(n^2\mathsf{M}(d)\log(d))$ in the cost).

Proposition 3.1. *Algorithm 5 is correct. For points in geometric progression, it costs $O(\mathsf{MM}'(m, d))$ if $n \le m$ and $O(\mathsf{MM}'(m, d) + m^{\omega-1}nd\log(d))$ otherwise. For general evaluation points, an extra cost $O(m^2\mathsf{M}(d)\log^2(d))$ is incurred.*

Proof. Correctness, including the specific form of the output, follows directly from the first and third items of [28, Lem. 2.4].

Step 1 costs $O((m^{\omega} + m^{\omega-1}n)T^2)$ and we enter it $O(d/T)$ times, for a total of $O((m^{\omega} + m^{\omega-1}n)d)$ since T is a constant. If the points $\boldsymbol{\alpha}$ are in geometric progression, evaluating the $\mathbf{P}_1(\alpha_i)$'s in Step 3 costs

Input: matrices $\mathbf{E} = (\mathbf{E}_1, \ldots, \mathbf{E}_d)$ in $\mathbb{K}^{m\times n}$, evaluation points $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_d)$ in $\mathbb{K}$, shift $\boldsymbol{s}$ in $\mathbb{Z}^m$
Output: an $\boldsymbol{s}$-ordered weak Popov interpolant basis for $(\mathbf{E}, \boldsymbol{\alpha})$

Algorithm 4: M-IntBasis$(\mathbf{E}, \boldsymbol{\alpha}, \boldsymbol{s})$

1. $\mathbf{P} \leftarrow$ identity matrix in $\mathbb{K}[x]^{m\times m}$, and $\boldsymbol{t} \leftarrow$ copy of $\boldsymbol{s}$
2. *For* $k = 0, \ldots, d-1$:
  a. $\mathbf{R} \in \mathbb{K}^{m\times n} \leftarrow \mathbf{P}(\alpha_i)\mathbf{E}_i$
  b. $\mathbf{Q} \in \mathbb{K}[x]^{m\times m} \leftarrow$ M-Basis-1$(\mathbf{R}, \boldsymbol{t})$
  c. $\mathbf{P} \leftarrow \mathbf{QP}$, and then $\boldsymbol{t} \leftarrow \mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{P})$

Algorithm 5: PM-IntBasis$(\mathbf{E}, \boldsymbol{\alpha}, \boldsymbol{s})$

1. *if* $d \le T$ *return* M-IntBasis$(\mathbf{E}, \boldsymbol{\alpha}, \boldsymbol{s})$
2. $\mathbf{P}_1 \leftarrow$ PM-IntBasis$(\mathbf{E}_{1\ldots\lceil d/2\rceil}, \boldsymbol{\alpha}_{1\ldots\lceil d/2\rceil}, \boldsymbol{s})$
3. $\mathbf{R} \leftarrow (\mathbf{P}_1(\alpha_{\lceil d/2\rceil+1})\mathbf{E}_{\lceil d/2\rceil+1}, \ldots, \mathbf{P}_1(\alpha_d)\mathbf{E}_d)$
4. $\boldsymbol{t} \leftarrow \mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{P}_1)$
5. $\mathbf{P}_2 \leftarrow$ PM-IntBasis$(\mathbf{R}, \boldsymbol{\alpha}_{\lceil d/2\rceil+1\ldots d}, \boldsymbol{t})$
6. return $\mathbf{P}_2\mathbf{P}_1$

$O(m^2\mathsf{M}(d))$; for general points, the cost is $O(m^2\mathsf{M}(d)\log(d))$. Then, the residual $\mathbf{R}$ is deduced in time $O(m^\omega d)$ if $n \le m$ and $O(m^{\omega-1}nd)$ otherwise. Finally, $\mathbf{P}_2\mathbf{P}_1$ can be computed in time $O(\mathsf{MM}(m,d))$. Summing up all costs gives the proposition. ◻

We now compare the performance of these algorithms against their approximant versions. All computations are done modulo a 60 bit (non-FFT) prime. Our current code uses the threshold $T = 32$ in the divide and conquer PM-Basis and PM-IntBasis: beyond this point, they are faster than the iterative M-Basis and M-IntBasis.

Unlike in most other functions, where elements of $\mathbb{K}[x]^{m\times n}$ are represented as matrices of polynomials (Mat<Vec<zz_p>> in NTL), in M-Basis and M-IntBasis we see them as polynomials with matrix coefficients (Vec<Mat<zz_p>>). Indeed, since these algorithms involve only matrix arithmetic over $\mathbb{K}$ (recall that $\deg(\mathbf{Q}) \le 1$), this turns out to be more cache-friendly and faster.

Besides, we implemented two variants for approximant bases: either the residual $\mathbf{R}$ is computed from $\mathbf{P}$ and $\mathbf{F}$ at each iteration, or we initialize a list of residuals with a copy of $\mathbf{F}$ and we update the whole list at each iteration using $\mathbf{Q}$. The second variant improves over the first when $n > m/2$, with significant savings when n is close to m. For interpolant bases, this did not lead to any gain.

Timings are showed in Table 5. For approximants, we use as input a random matrix in $\mathbb{K}[x]^{m\times n}$ of degree $d-1$; for interpolants, we use d random matrices in $\mathbb{K}^{m\times n}$. We focus on the common case $m \simeq 2n$, which arises for example in kernel algorithms (Section 4.2) and in fraction reconstruction, itself used in the block Wiedemann algorithm, in basis reduction, and in the resultant algorithm of [53].

Concerning iterative algorithms, we observe that interpolants are slightly faster than approximants, which is explained by the cost of computing the residual $\mathbf{R}$: it uses one Horner evaluation of $\mathbf{P}$ and one matrix product for interpolants, whereas for approximants it uses about $\min(k, \deg(\mathbf{P}))$ matrix products at iteration k.

As for the divide and conquer algorithms, interpolant bases with general points are slower, in some cases significantly, than the other two algorithms: although the complexity analysis predicted a disadvantage, we believe that our implementation of multipoint evaluation at general points could be improved to reduce this gap. For the other two algorithms, the comparison is less clear. There could be many factors at play here, but the main differences lie in the base case (Step 1) which calls the iterative algorithm, and in the computation of residuals (Step 3) which uses either middle products or geometric evaluation. It seems that FFT-based polynomial multiplication performs slightly better than geometric evaluation for small matrices and slightly worse for large matrices.

Table 5: *Approximant basis and interpolant basis. Timings for* M-Basis *(M),* M-IntBasis *(M-I),* PM-Basis *(PM),* PM-IntBasis *for general points (PM-I) and* PM-IntBasis *for geometric points (PM-Ig).*

m	n	d	M	M-I	d	PM	PM-I	PM-Ig
4	2	32	1.60e-4	1.42e-4	32768	**1.06**	6.81	1.47
16	8	32	1.98e-3	**1.55e-3**	4096	**1.82**	5.51	**1.92**
32	16	32	0.0104	**7.59e-3**	2048	**3.90**	8.18	**3.56**
64	32	32	0.0502	**0.0354**	1024	8.1	12.2	**6.38**
128	64	32	0.374	**0.253**	1024	45	56.7	**33.3**
256	128	32	2.92	**1.83**	1024	288	292	**198**

4 Higher-level algorithms

In this section we consider kernel computation, system solving, determinant computation, and basis reduction. For each of these problems, we discuss algorithms which rely on polynomial matrix multiplication, through either approximant/interpolant basis computation, or lifting techniques, or a combination of both for basis reduction. For many of these algorithms, this is the first implementation and experimental comparison we are aware of.

4.1. A note on matrix fraction reconstruction Given $\mathbf{H}$ in $\mathbb{K}(x)^{n\times n}$, a *left fraction description* of $\mathbf{H}$ is a pair of polynomial matrices $(\mathbf{Q}, \mathbf{R})$ in $\mathbb{K}[x]^{n\times n}$ such that $\mathbf{H} = \mathbf{Q}^{-1}\mathbf{R}$. It is *minimal* if $\mathbf{Q}$ and $\mathbf{R}$ have unimodular left matrix GCD and $\mathbf{Q}$ is in reduced form (*right fraction descriptions* are defined similarly). Besides, $\mathbf{H}$ is said to be *strictly proper* if the numerator of each of its entries has degree less than the corresponding denominator.

Such a description of $\mathbf{H}$ is often computed from the power series expansion of $\mathbf{H}$ at sufficient precision, using an approximant basis. Yet, for resultant computations in Section 5.2, we would like to use an interpolant basis to obtain this description from sufficiently many values of $\mathbf{H}$. We now state the validity of this approach; this is a matrix version of rational function reconstruction [13, Chap. 5.7].

Proposition 4.1. *Let* $\mathbf{H}$ *be in* $\mathbb{K}(x)^{n\times n}$ *be strictly proper and suppose* $\mathbf{H}$ *admits left and right fraction descriptions of degrees at most* D*, for some* $D \in \mathbb{Z}_{>0}$*. For* M *in* $\mathbb{K}[x]$ *of degree at least* $2D$ *and such that all denominators in* $\mathbf{H}$ *are invertible modulo* M*, define*

$$\mathbf{F} = \begin{bmatrix} \mathbf{H} \bmod M \\ -\mathbf{I}_n \end{bmatrix} \in \mathbb{K}[x]^{2n\times n}.$$

Then, if $\mathbf{P} \in \mathbb{K}[x]^{2n\times 2n}$ *is a* $\mathbf{0}$*-ordered weak Popov basis of* $\mathcal{A}_M(\mathbf{F})$*, the first* n *rows of* $\mathbf{P}$ *form a matrix* $[\mathbf{Q}\ \ \mathbf{R}]$ *such that* $(\mathbf{Q}, \mathbf{R})$ *is a minimal left fraction description of* $\mathbf{H}$*, with* $\mathbf{Q}$ *in* $\mathbf{0}$*-ordered weak Popov form.*

The proof given in [17, Lem 3.7] for the specific $M = x^{2D+1}$ extends to any modulus M; using an ordered weak Popov form (rather than a reduced form) allows us both to know *a priori* that the first n rows are those of degree at most D, and to use degree $2D$ instead of $2D + 1$ (since $\deg(\mathbf{R}) < \deg(\mathbf{Q})$ is ensured by this form).

In particular, if $M = \prod_{i=1}^{2D}(x - \alpha_i)$ for pairwise distinct points $(\alpha_1, \ldots, \alpha_{2D})$, the interpolant basis algorithms in Section 3.2 compute a minimal left fraction description of $\mathbf{H}$ from $\mathbf{H}(\alpha_1), \ldots, \mathbf{H}(\alpha_d)$.

4.2. Kernel basis We implemented two kernel basis algorithms: first, one which uses a single approximant basis computation at an order sufficiently large so that the basis contains a kernel basis (based on Lemma 4.2); second, the divide and conquer algorithm of [58], which computes several approximant bases at smaller order and combines the recursively obtained kernel bases via multiplication. We ensured that our algorithms return a kernel basis in shifted ordered weak Popov form, again without impact on performance. Furthermore, in both cases, we designed and implemented variants which rely on interpolant bases instead of approximant bases.

Lemma 4.2. *Let* $\mathbf{F}$ *be in* $\mathbb{K}[x]^{m\times n}$ *of degree* $d \geq 0$*, let* $\boldsymbol{s}$ *be in* $\mathbb{N}^m$*, and let* δ *in* $\mathbb{Z}_{>0}$ *be an upper bound on the* $\boldsymbol{s}$*-degree of any* $\boldsymbol{s}$*-reduced left kernel basis of* $\mathbf{F}$*; for example,* $\delta = nd + \max(\boldsymbol{s}) - \min(\boldsymbol{s}) + 1$*. Let* M *be in* $\mathbb{K}[x]$ *of degree at least* $\delta + d$*, and* $\mathbf{P}$ *in* $\mathbb{K}[x]^{m\times m}$ *be an* $\boldsymbol{s}$*-reduced basis of* $\mathcal{A}_M(\mathbf{F})$*. Then, the submatrix of* $\mathbf{P}$ *formed by its rows of* $\boldsymbol{s}$*-degree less than* δ *is an* $\boldsymbol{s}$*-reduced left kernel basis for* $\mathbf{F}$*.*

Proof. Let $\mathbf{K} \in \mathbb{K}[x]^{k\times m}$ be this submatrix, which is $\boldsymbol{s}$-reduced since $\mathbf{P}$ is. Since $\min(\boldsymbol{s}) \geq 0$, we have $\deg(\mathbf{K}) \leq \max(\mathrm{rdeg}_{\boldsymbol{s}}(\mathbf{K})) < \delta$, hence $\deg(\mathbf{KF}) < \delta + d \leq \deg(M)$. From $\mathbf{KF} = \mathbf{0} \bmod M$, we obtain that $\mathbf{KF}$ is zero. It remains to observe that $\mathbf{K}$ generates the kernel of $\mathbf{F}$: since this set is included in $\mathcal{A}_M(\mathbf{F})$, any basis of it is a left multiple of $\mathbf{P}$, and in particular a basis of $\boldsymbol{s}$-degree less than δ is a left multiple of the submatrix $\mathbf{K}$, according to the predictable degree property [29, Thm. 6.3-13]. The validity of the suggested bound δ follows from [58, Thm. 3.4] for $(d, \ldots, d)$-reduced kernel bases; then, compared to these, for an arbitrary $\boldsymbol{s}$ the degree of $\boldsymbol{s}$-reduced bases cannot increase by more than $\max(\boldsymbol{s}) - \min(\boldsymbol{s})$. □

In particular, one may find $\mathbf{P}$ via PM-IntBasis at $d + \delta$ points or via PM-Basis at order $d+\delta$; for $n \leq m$, this costs $O(\mathsf{MM}'(m, d+\delta))$. The approximant-based direct approach is folklore [58, Sec. 2.3], yet explicit statements in the literature focus on shifts linked to the degrees in $\mathbf{F}$, with better bounds δ [58, Lem. 3.3], [37, Lem. 4.3].

The algorithm of [58] is more efficient, at least when the entries of $\boldsymbol{s}$ are close to the corresponding row degrees of $\mathbf{F}$; for a uniform shift, it costs $\tilde{O}(m^\omega \lceil nd/m \rceil)$ operations. We obtained significant practical improvements over the plain implementation of [58, Algo. 1] thanks to the following observation: if $n \leq m/2$, for a vast majority of input $\mathbf{F}$, the approximant basis at Step 2 of [58, Algo. 1], computed at order more than $2s$, contains the sought kernel basis. Furthermore, this can be easily tested by checking well-chosen degrees, and then the algorithm can exit early, avoiding the further recursive calls. We took advantage of this via the following modifications: we use order $2s + 1$ rather than $3s$ (see [58, Rmk. 3.5] for a discussion on this point), and when $n > m/2$ we directly reduce the number of columns via the divide and conquer scheme in [58, Thm. 3.15].

The use of approximants here follows the idea in Lemma 4.2: row vectors of small degree which are in $\mathcal{A}_M(\mathbf{F})$ for a large degree M must be in the kernel of $\mathbf{F}$. Thus, one can directly replace approximant bases with interpolant bases in [58, Algo. 1], up to modifying Step 8 accordingly (dividing by the appropriate polynomial M).

Timings for the two approaches are showed in Table 6. The input matrix $\mathbf{F}$ is chosen at random of degree d over a 60 bit prime, and the shift is uniform. As expected, [58, Algo. 1] is faster than the direct approach when $n > 1$, and the differences between interpolant and approximant variants follow those observed in Section 3.

Table 6: *Minimal kernel basis*

			direct		divide and conquer	
m	n	d	approx.	int.	approx.	int.
8	1	8192	**1.36**	2.00	**1.35**	2.00
8	4	8192	7.22	6.60	**2.16**	**2.49**
8	7	8192	14.1	14.4	**4.64**	5.63
32	16	1024	86.3	63.1	**3.75**	**3.51**
32	31	1024	142	118	**8.27**	**8.09**
128	1	256	**14.0**	**14.6**	**14.0**	**14.6**
128	64	256	2720	1827	16.8	**11.8**
128	127	256	>1h	>1h	43.8	**35.6**

4.3. Linear system solving. For systems $\mathbf{A}\boldsymbol{v} = \boldsymbol{b}$, with $\mathbf{A}$ in $\mathbb{K}[x]^{m\times n}$, $\boldsymbol{b}$ in $\mathbb{K}[x]^{m\times 1}$ and $\boldsymbol{v}$ in $\mathbb{K}(x)^{n\times 1}$, we implemented two families of algorithms. The first one uses lifting techniques, assuming $\mathbf{A}$ is square, nonsingular, with $\mathbf{A}(0)$ invertible; in this case, the algorithm returns a pair $(\boldsymbol{u}, f)$ in $\mathbb{K}[x]^{n\times 1}\times\mathbb{K}[x]$ such that $\mathbf{A}\boldsymbol{u} = f\boldsymbol{b}$ and f has minimal degree. The second approach is based on kernel computation and works for any matrix $\mathbf{A}$; under the assumptions above it has a similar output.

Lifting techniques. Under the above assumptions, our lifting algorithm is standard: if $\mathbf{A}$ and $\boldsymbol{b}$ have degree at most d, we first compute the truncated inverse $\mathbf{S} = \mathbf{A}^{-1} \bmod x^{d+1}$ by matrix Newton iteration [42]. Then, we use Dixon's algorithm [10] to compute $\boldsymbol{v} \bmod x^{2nd} = \mathbf{A}^{-1}\boldsymbol{b} \bmod x^{2nd}$; it consists of roughly $2n$ steps, each involving a matrix-vector product using either $\mathbf{A}$ or $\mathbf{S}$. Then, vector rational reconstruction is applied to recover $(\boldsymbol{u}, f)$ from $\boldsymbol{v}$. The cost of this algorithm is $O(\mathsf{MM}(n, d))$ for the truncated inverse of $\mathbf{A}$ and $O(n^3\mathsf{M}(d))$ for Dixon's algorithm; overall this is in $\tilde{O}(n^3 d)$.

To reduce the exponent in n, Storjohann introduced the *high-order lifting* algorithm [45]. The core of this algorithm is the computation of $\Theta(\log(n))$ slices $\mathbf{S}_0, \mathbf{S}_1, \ldots$ of the power series expansion of $\mathbf{A}^{-1}$, where the coefficients of $\mathbf{S}_i$ are the coefficients of degree $(2^i-1)d-2^i+1, \ldots, (2^i+1)d-2^i-1$ in $\mathbf{A}^{-1}$. These matrices are computed recursively, each step involving 4 matrix products; the other steps of the algorithm, that use these $\mathbf{S}_i$ to compute $\boldsymbol{v} \bmod x^{2nd}$, are cheaper, so the runtime is $O(\mathsf{MM}(n, d)\log(n)) \subset \tilde{O}(n^\omega d)$.

Using kernel bases. For this second approach, let $\mathbf{A}$ be any matrix in $\mathbb{K}[x]^{m\times n}$ and $\boldsymbol{b}$ be in $\mathbb{K}[x]^{m\times 1}$. The algorithm simply computes $\mathbf{K} \in \mathbb{K}[x]^{(n+1)\times k}$, a right kernel basis of the augmented matrix $[\mathbf{A} \mid \boldsymbol{b}] \in \mathbb{K}[x]^{m\times(n+1)}$. The matrix $\mathbf{K}$ generates, via $\mathbb{K}(x)$-linear combinations of its columns, all solutions $\boldsymbol{v} \in \mathbb{K}(x)^{n\times 1}$ to $\mathbf{A}\boldsymbol{v} = \boldsymbol{b}$.

In particular, if $\mathbf{K}$ is empty (i.e. $k = 0$, which requires $m \geq n$), or if the last row of $\mathbf{K}$ is zero, then the system has no solution. Furthermore, if $\mathbf{A}$ is square and nonsingular, $\mathbf{K}$ has a single column $[\boldsymbol{u}^\mathsf{T} \mid f]^\mathsf{T}$, where $\boldsymbol{u} \in \mathbb{K}[x]^{n\times 1}$ and $f \in \mathbb{K}[x]$, with f of minimal degree (otherwise, $\mathbf{K}$ would not be a basis).

Our implementation uses the kernel algorithm of [58]; to exploit it best, we choose the input shift $\boldsymbol{s} = (\boldsymbol{d}, d)$, where $d = \deg(\boldsymbol{b})$ and

$\boldsymbol{d} \in \mathbb{N}^n$ is the tuple of column degrees of $\mathbf{A}$ (zero columns of $\mathbf{A}$ are discarded while computing $\boldsymbol{d}$).

Implementation. We implemented the approaches described above: lifting with Dixon's algorithm, high-order lifting, and via kernel. Table 7 shows timings for randomly chosen $m \times m$ matrix $\mathbf{A}$ and $m \times 1$ vector $\boldsymbol{b}$, both of degree d, over a 60 bit prime field. In this case the lifting algorithms apply (with high probability). On such inputs, Dixon's algorithm usually does best. High-order lifting, although theoretically faster, is outperformed, mainly because it performs $\Theta(\log(n))$ matrix products (we will however see that this algorithm plays an important role for basis reduction). The kernel based approach is moderately slower than Dixon's algorithm, but has the advantage of working without any assumption on $\mathbf{A}$.

Table 7: *Linear system solving*

m	d	Dixon	high-order lifting	kernel
16	1024	**1.53**	2.39	2.07
32	1024	**4.94**	13.8	9.45
64	1024	**19.5**	94.7	46.5
128	512	**55.2**	266	108

4.4. Determinant.

We implemented four algorithms, taking as input a square $m \times m$ matrix $\mathbf{A}$. For timings showed in Table 8, we used a random such matrix of degree d over a 60 bit prime.

The most basic one uses expansion by minors, which turns out to be the fastest option up to dimension about 6.

The second one assumes that we have an element α in $\mathbb{K}$ of order at least $md + 1$, and uses evaluation/interpolation at the geometric progression $1, \alpha, \dots, \alpha^{md}$; this costs $O(m^3 \mathsf{M}(d) + m^{\omega+1} d)$ operations in $\mathbb{K}$. For dimensions between 7 and about 20, it is often the fastest variant, sometimes competing with the third.

The third one consists in solving a linear system with random right-hand side of degree d; this yields a solution $(\boldsymbol{u}, f)$ with f sought determinant up to a constant [40]. For dimensions exceeding 20, this is the fastest method (assuming that $\mathbf{A}(0)$ is invertible).

The last one is an experimental implementation of the algorithm of [33] based on triangularization, which runs in $\tilde{O}(m^\omega d)$. For the moment, it only supports the generic case where the Hermite form of $\mathbf{A}$ has diagonal $(1, \dots, 1, \det(\mathbf{A}))$, or in other words, all so-called "row bases" computed in that algorithm are identity. This allows us to temporarily circumvent the lack of a row basis implementation, while still being able to observe meaningful timings. Indeed, we believe that timings for a complete implementation called on such "generic" input will be similar to the timings presented here in many cases of interest, where one can easily detect whether the algorithm has made a wrong prediction about the row basis being identity (for example if the input matrix is reduced, which is the case if it is the denominator of a minimal fraction description). This recursive determinant algorithm calls expansion by minors as a base case for small dimensions; for larger dimensions, it is generally slightly slower than the third method.

4.5. Basis reduction.

Our implementation of the algorithm of [17] takes as input a nonsingular matrix $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$ of degree d such that $\mathbf{A}(0)$ is invertible, and returns a reduced form of $\mathbf{A}$. The algorithm first computes a slice $\mathbf{S}$ of $2d$ consecutive coefficients of degree about md in the power series expansion of $\mathbf{A}^{-1}$, then uses PM-Basis to reconstruct a fraction description $\mathbf{S}^{-1} = \mathbf{R}^{-1}\mathbf{Q}$, and then returns $\mathbf{R}$. A Las Vegas randomized version is mentioned in [17], to remove the assumption on $\mathbf{A}(0)$: we will implement it for large enough $\mathbb{K}$, but for smaller fields this requires to work in an extension of $\mathbb{K}$, which is currently beyond the scope of our work.

Table 8: *Determinant*

m	d	minors	evaluation	linsolve	triangular
4	65536	**0.7751**	2.014	8.460	**0.7826**
16	4096	∞	**4.14**	5.023	7.38
32	4096	∞	34.5	**25.4**	41.6
64	2048	∞	127	**68.6**	100
128	512	∞	244	**96.6**	**99.0**

In our experiments, to create the input $\mathbf{A}$, we started from a random $m \times m$ matrix of degree $d/3$ (which is reduced with high probability), and we left-multiplied it by a lower unit triangular matrix and then by an upper one, both chosen at random of degree $d/3$. Table 9 shows timings for both steps, with the first step either based on Newton iteration or on high-order lifting; the displayed total time is when using the faster of the two. We conclude that for reduction, as opposed to the above observations for system solving, it is crucial to rely on high-order lifting. Indeed, it improves over Newton iteration already for dimension 8, and the gap becomes quite significant when the dimension grows.

Table 9: *Basis reduction*

m	d	Newton	high-order	reconstruct	total
4	24574	**1.251**	1.688	8.772	10.02
8	6142	2.617	**2.244**	8.851	11.09
16	1534	4.457	**3.044**	8.506	11.55
32	382	11.147	**4.858**	7.977	12.83
64	94	30.62	**5.509**	5.833	11.34
128	22	84.47	**3.973**	5.357	9.22
128	94	371.1	**29.17**	37.23	66.41

5 Applications to bivariate resultants

We conclude this paper with algorithms originating from Villard's recent breakthrough on computing the determinant of structured polynomial matrices [53]. Fix a field $\mathbb{K}$ and consider the two following questions: computing the resultant of two polynomials F, G in $\mathbb{K}[x, z]$ with respect to z, and computing the characteristic polynomial of an element A in $\mathbb{K}[z]/(P)$, for some P in $\mathbb{K}[z]$.

The second problem is a particular case of the former, since the characteristic polynomial of A modulo P is the resultant of $x - A(z)$ and $P(z)$ with respect to z, up to a nonzero constant. Let n be an upper bound on the degree in z of the polynomials we consider, and d be a bound on their degree in x (so in the second problem, $d = 1$). Villard proved that for generic inputs, both problems can be solved in $\tilde{O}(n^{2-1/\omega} d) \subset \tilde{O}(n^{1.58} d)$ operations in $\mathbb{K}$. For the first problem, the best previous bound is $\tilde{O}(n^2 d)$, obtained either by evaluation/interpolation techniques or Reischert's algorithm [41]. For the second problem, the previous record was $\tilde{O}(n^{\omega_2/2})$, where ω_2 is the exponent of matrix multiplication in size $(s, s) \times (s, s^2)$, with $\omega_2/2 \leq 1.63$ [35]. Note that these bounds apply to all inputs.

We show how the work we presented above allows us to put Villard's ideas to practice, and outperform the previous state of the art for large input sizes. This is however not straightforward: in both cases, this required modifications of Villard's original designs (for the second case, using an algorithm from [38]).

5.1. Overview of the approach. In [53], Villard designed the following algorithm to find the determinant of a matrix $\mathbf{P}$ over $\mathbb{K}[x]$.

> *Algorithm 6:* DETERMINANT($\mathbf{P}, m$)
> *Input:* nonsingular $\mathbf{P}$ in $\mathbb{K}[x]^{\nu\times\nu}$; parameter $m \in \{1,\ldots,\nu\}$
> *Output:* det($\mathbf{P}$)
> **1.** compute $\bar{\mathbf{H}} = \mathbf{H} \bmod x^{2\lceil \nu/m\rceil d+1}$, where d is the degree of $\mathbf{P}$ and $\mathbf{H}$ is the $m\times m$ top-right quadrant of $\mathbf{P}^{-1} \in \mathbb{K}(x)^{\nu\times\nu}$
> **2.** compute a minimal left fraction description $(\mathbf{Q},\mathbf{R})$ of $\mathbf{H}$, using $\bar{\mathbf{H}}$
> **3.** return det($\mathbf{Q}$)

The parameter m is chosen so as to minimize the theoretical cost. The correctness of the algorithm follows from the next properties, which do not hold for an arbitrary nonsingular $\mathbf{P}$: the matrix $\mathbf{H}$ is strictly proper and admits a left fraction description $\mathbf{H} = \mathbf{Q}^{-1}\mathbf{R}$ such that det($\mathbf{P}$) = det($\mathbf{Q}$), for $\mathbf{Q}$ and $\mathbf{R}$ in $\mathbb{K}[x]^{m\times m}$ of degree at most $\lceil \nu/m\rceil d$ (see Section 4.1 for definitions). In [53], they are proved to hold for generic instances of the problems discussed here.

Once sufficiently many terms of the expansion of $\mathbf{H}$ have been obtained in Step 1, the denominator $\mathbf{Q}$ is recovered by an approximant basis algorithm and its determinant is computed by a general algorithm in Steps 2 and 3, which cost $\tilde{O}(m^\omega(\nu d/m))$.

While the algorithm applies to any nonsingular matrix $\mathbf{P}$ satisfying the properties above, in general it does not improve over previously known methods (see Section 4.4). Indeed, the fastest known algorithm for obtaining $\bar{\mathbf{H}}$ costs $\tilde{O}(\nu^\omega d)$ operations via high-order lifting (see for example [19, Thm. 1]).

However, sometimes $\mathbf{P}$ has some structure which helps to speed up the first step. Villard pointed out that when $\mathbf{P}$ is the Sylvester matrix of two bivariate polynomials, then $\mathbf{P}^{-1}$ is a Toeplitz-like matrix which can be described succinctly as $\mathbf{P}^{-1} = \mathbf{L}_1\mathbf{U}_1 + \mathbf{L}_2\mathbf{U}_2$; here, $\mathbf{L}_1, \mathbf{L}_2$ (resp. $\mathbf{U}_1, \mathbf{U}_2$) are lower (resp. upper) triangular Toeplitz matrices with entries in $\mathbb{K}(x)$. Hence, we start by computing the first columns $\boldsymbol{c}_1$ of $\mathbf{L}_1$ and $\boldsymbol{c}_2$ of $\mathbf{L}_2$ as well as the first rows $\boldsymbol{r}_1$ of $\mathbf{U}_1$ and $\boldsymbol{r}_2$ of $\mathbf{U}_2$, all of them modulo $x^{2\lceil \nu/m\rceil d+1}$; then, $\bar{\mathbf{H}}$ is directly obtained via the above formula for $\mathbf{P}^{-1}$, using $\tilde{O}(m\nu d)$ operations. Computing these rows and columns is done by solving systems with matrices $\mathbf{P}$ and $\mathbf{P}^\mathsf{T}$ and very simple right-hand sides [53, Prop. 5.1], with power series coefficients, in time $\tilde{O}(\nu^2 d/m)$.

Altogether, taking $m = \nu^{1/\omega}$ minimizes the cost, yielding the runtime $\tilde{O}(\nu^{2-1/\omega}d)$. In the case of bivariate resultants described above, the Sylvester matrix of F and G has size $\nu = 2n$, hence the cost bound $\tilde{O}(n^{2-1/\omega}d)$.

5.2. Resultant of generic bivariate polynomials. We implemented the algorithm described in the previous section to compute the resultant of generic F, G in $\mathbb{K}[x,z]$; first experiments showed that obtaining $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{r}_1, \boldsymbol{r}_2$ was a bottleneck. These vectors have power series entries and are solutions of linear systems whose matrix is the Sylvester matrix of F and G or its transpose: they were obtained via Hensel lifting techniques, following [13, Ch. 15.4].

To get better performance, we designed a minor variant of Villard's algorithm: instead of computing the power series expansion of $\mathbf{H}$ modulo x^δ, where $\delta = 2\lceil \nu/m\rceil d + 1$, we compute values of $\mathbf{H}$ at δ points. We choose these points in geometric progression and use the interpolant basis algorithm of Section 3.2 to recover $\mathbf{Q}$ and $\mathbf{N}$, as detailed in Section 4.1. The value of $\mathbf{H}$ at $x = \alpha$ is computed following the same approach as above, but over $\mathbb{K}$ instead of $\mathbb{K}[[x]]$. In particular, our implementation directly relies on NTL's extended GCD algorithm over $\mathbb{K} = \mathbb{F}_p$ to compute the vectors $\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{r}_1, \boldsymbol{r}_2$.

Table 10: *Resultant of generic bivariate polynomials*

$n = d$	Direct	Algo. 6	$n = d$	Direct	Algo. 6
100	**1.75**	3.48	600	797	**653**
200	**17.4**	29.3	700	1343	**1081**
300	**72.3**	106	800	2121	**1388**
400	182	182	900	3203	**1760**

Table 10 compares our implementation to a direct approach: computing the resultant by evaluation/interpolation. Note that the latter approach, while straightforward conceptually, is the state of the art. In a close match with the analysis above, the parameter m was set to $\lceil n^{0.4}\rceil$, since this gave us the best runtimes. As an example, for $d = 300$, the cost of each individual steps were 65s for computing structured inversions, and 40s for obtaining $\mathbf{Q}$ and its determinant, which is a good balance. The base field was a prime field with a 60 bit general prime.

Input polynomials are chosen at random with partial degree n both in x and in z; such polynomials have total degree $2n$, and their resultant has degree $2n^2$. The largest examples have quite significant sizes, but such degrees are not unheard-of in applications, as for instance in the genus-2 point counting algorithms of [1, 14–16]. Overall, with $n = d$, we observe a crossover point around $n = 400$.

5.3. Characteristic polynomial. We consider the computation of the characteristic polynomial of an element A in $\mathbb{K}[z]/(P)$, for some monic P in $\mathbb{K}[z]$ of degree n. The algorithm we implemented, and which we sketch below, is from [38] and assumes that A and P are generic.

As explained previously, this problem is a particular case of a bivariate resultant, but we rely on another point of view that allows for a better asymptotic cost. Indeed, the characteristic polynomial of A modulo P is by definition the characteristic polynomial of the matrix $\mathbf{M}$ of multiplication by A modulo P. In other words, it is the determinant of the degree-1 matrix $\mathbf{P} = x\mathbf{I} - \mathbf{M} \in \mathbb{K}[x]^{n\times n}$.

The genericity assumption ensures that $\mathbf{M}$ is invertible, hence the power series expansion of $\mathbf{P}^{-1}$ is $\sum_{k\ge 0} -\mathbf{M}^{-k-1}x^k$. Here, we use the top-left $m\times m$ quadrant $\mathbf{H}$ of $\mathbf{P}^{-1}$; it has entries $h_{i,j} \in \mathbb{K}[[x]]$, where

$$h_{i,j,k} := \mathrm{coeff}(h_{i,j}, x^k) = \mathrm{coeff}(-z^j A^{-k-1} \bmod P, z^i).$$

for $0 \le i, j < m$ and for all $k \ge 0$.

A direct implementation of this idea does not improve on the runtime given in Section 5.1, since it computes $A^{-k-1} \bmod P$ for all $0 \le k < \delta = 2\lceil n/m\rceil$ and therefore costs $\Omega(n^2/m)$. It turns out that baby-steps giant-steps techniques allow one to compute $h_{i,j,k}$ for $0 \le i, j < m$ and $0 \le k < \delta$ in $\tilde{O}(\delta^{(\omega-1)/2}n + mn)$ operations in $\mathbb{K}$. Taking $m = \lceil n^{1/3}\rceil$ minimizes the overall cost, resulting in the runtime $\tilde{O}(n^{(\omega+2)/3}) \subset O(n^{1.46})$.

Table 11 compares our implementation to NTL's built-in characteristic polynomial algorithm, with random inputs A and P. For

Table 11: *Characteristic polynomial modulo P*

n	m	NTL	new algorithm
5000	5	**0.143**	0.225
20000	8	**1.43**	1.62
40000	8	4.69	**4.42**
60000	10	8.45	**8.34**
80000	10	16.6	**12.1**
100000	10	23.1	**17.4**

such inputs, NTL uses Shoup's algorithm for power projection [43], which runs in time $\tilde{O}(n^{(\omega+1)/2})$.

Acknowledgements. Hyun was supported by a Mitacs Globalink award. The second author would like to thank the institute INS2I of CNRS for its support through its program for young researchers. Schost was supported by an NSERC Discovery Grant.

References

[1] S. Abélard, P. Gaudry, and P.-J. Spaenlehauer. 2018. Counting points on genus-3 hyperelliptic curves with explicit real multiplication. In *ANTS-XIII*.

[2] B. Beckermann and G. Labahn. 1994. A Uniform Approach for the Fast Computation of Matrix-Type Padé Approximants. *SIAM J. Matrix Anal. Appl.* 15, 3 (1994), 804–823.

[3] B. Beckermann and G. Labahn. 2000. Fraction-free computation of matrix rational interpolant and matrix gcds. *SIAM J. Matrix Anal. Appl.* 22, 1 (2000), 114–144.

[4] B. Beckermann, G. Labahn, and G. Villard. 1999. Shifted Normal Forms of Polynomial Matrices. In *ISSAC'99*. ACM, 189–196.

[5] A. Bostan, G. Lecerf, and É. Schost. 2003. Tellegen's Principle Into Practice. In *ISSAC'03*. ACM, 37–44.

[6] A. Bostan and É. Schost. 2005. Polynomial evaluation and interpolation on special sets of points. *J. Complexity* 21, 4 (2005), 420–446.

[7] D. G. Cantor and E. Kaltofen. 1991. On fast multiplication of polynomials over arbitrary algebras. *Acta Inform.* 28, 7 (1991), 693–701.

[8] D. Coppersmith. 1994. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comp.* 62, 205 (1994), 333–350.

[9] D. Coppersmith and S. Winograd. 1990. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.* 9, 3 (1990), 251–280.

[10] J. Dixon. 1982. Exact solution of linear equations using p-adic expansions. *Numer. Math.* 40 (1982), 137–141.

[11] J. Doliskani, P. Giorgi, R. Lebreton, and É. Schost. 2018. Simultaneous conversions with the residue number system using linear algebra. *ACM Trans. Math. Software* 44, 3 (2018), 1–21.

[12] J.-G. Dumas, P. Giorgi, and C. Pernet. 2008. Dense Linear Algebra over Word-Size Prime Fields: The FFLAS and FFPACK Packages. *ACM Trans. Math. Softw.* 35, 3, Article 19 (2008), 42 pages.

[13] J. von zur Gathen and J. Gerhard. 2013. *Modern Computer Algebra (third edition)*. Cambridge University Press.

[14] P. Gaudry and R. Harley. 2000. Counting points on hyperelliptic curves over finite fields. In *ANTS-IV (LNCS)*, Vol. 1838. Springer-Verlag, 313–332.

[15] P. Gaudry and É. Schost. 2004. Construction of secure random curves of genus 2 over prime fields. In *Eurocrypt'04 (LNCS)*, Vol. 3027. Springer, 239–256.

[16] P. Gaudry and É. Schost. 2012. Point-counting in genus 2 over prime fields. *J. Symbolic Comput.* 47, 4 (2012), 368âĂŞ400.

[17] P. Giorgi, C.-P. Jeannerod, and G. Villard. 2003. On the complexity of polynomial matrix computations. In *ISSAC'03*. ACM, 135–142.

[18] P. Giorgi and R. Lebreton. 2014. Online order basis algorithm and its impact on the block Wiedemann algorithm. In *ISSAC'14*. ACM, 202–209.

[19] S. Gupta, S. Sarkar, A. Storjohann, and J. Valeriote. 2012. Triangular x-basis decompositions and derandomization of linear algebra algorithms over $K[x]$. *J. Symbolic Comput.* 47, 4 (2012), 422–453.

[20] G. Hanrot, M. Quercia, and P. Zimmermann. 2004. The Middle Product Algorithm, I. *Appl. Algebra Engrg. Comm. Comp.* 14, 6 (2004), 415–438.

[21] D. Harvey. 2009. A cache-friendly truncated FFT. *Theoretical Computer Science* 410, 27-29 (2009), 2649–2658.

[22] D. Harvey. 2014. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation* 60 (2014), 113–119.

[23] D. Harvey, J. van der Hoeven, and G. Lecerf. 2017. Faster Polynomial Multiplication over Finite Fields. *J. ACM* 63, 6, Article 52 (2017), 23 pages. https://doi.org/10.1145/3005344

[24] Seung Gyu Hyun, Romain Lebreton, and Éric Schost. 2017. Algorithms for structured linear systems solving and their implementation. In *ISSAC'17*. ACM Press, 205–212.

[25] S. G. Hyun, V. Neiger, H. Rahkooy, and É. Schost. 2017. Block-Krylov techniques in the context of sparse-FGLM algorithms. (2017). arXiv:1712.04177

[26] C.-P. Jeannerod, V. Neiger, É. Schost, and G. Villard. 2016. Fast computation of minimal interpolation bases in Popov form for arbitrary shifts. In *ISSAC'16*. ACM, 295–302.

[27] C.-P. Jeannerod, V. Neiger, É. Schost, and G. Villard. 2017. Computing minimal interpolation bases. *J. Symbolic Comput.* 83 (2017), 272–314.

[28] C.-P. Jeannerod, V. Neiger, and G. Villard. 2018. Fast computation of approximant bases in canonical form. *Journal of Symbolic Computation* (2018). to appear.

[29] T. Kailath. 1980. *Linear Systems*. Prentice-Hall.

[30] E. Kaltofen. 1995. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comp.* 64, 210 (1995), 777–806.

[31] M. Kaminski, D.G. Kirkpatrick, and N.H. Bshouty. 1988. Addition requirements for matrix and transposed matrix products. *J. Algorithms* 9, 3 (1988), 354–364.

[32] G. Labahn, D. K. Choi, and S. Cabay. 1990. The inverses of block Hankel and block Toeplitz matrices. *SIAM J. Comput.* 19, 1 (1990), 98–123.

[33] G. Labahn, V. Neiger, and W. Zhou. 2017. Fast, deterministic computation of the Hermite normal form and determinant of a polynomial matrix. *J. Complexity* (in press) (2017).

[34] F. Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *ISSAC'14*. ACM, 296–303.

[35] F. Le Gall and F. Urrutia. 2018. Improved rectangular matrix multiplication using powers of the Coppersmith-Winograd tensor. In *SODA '18*. SIAM, 1029–1046.

[36] P. L. Montgomery. 2005. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. Comput.* 54, 3 (2005), 362–369.

[37] Vincent Neiger, Johan Rosenkilde, and Grigory Solomatov. 2018. Computing Popov and Hermite Forms of Rectangular Polynomial Matrices. In *ISSAC'18*. ACM, New York, NY, USA, 295–302.

[38] V. Neiger, B. Salvy, É. Schost, and G. Villard. 2019. In preparation.

[39] J. S. R. Nielsen and P. Beelen. 2015. Sub-quadratic decoding of one-point Hermitian codes. *IEEE Trans. Inf. Theory* 61, 6 (June 2015), 3225–3240.

[40] Victor Pan. 1988. Computing the determinant and the characteristic polynomial of a matrix via solving linear systems of equations. *Inform. Process. Lett.* 28, 2 (1988), 71–75.

[41] D. Reischert. 1997. Asymptotically fast computation of subresultants. In *ISSAC'97*. ACM, 233–240.

[42] G. Schulz. 1933. Iterative Berechung der reziproken Matrix. *Zeitschrift für Angewandte Mathematik und Mechanik* 13, 1 (1933), 57–59.

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

[44] V. Shoup. 2018. NTL: A Library for doing Number Theory, version 11.3.2. `http://www.shoup.net`.

[45] A. Storjohann. 2003. High-order lifting and integrality certification. *J. Symbolic Comput.* 36, 3-4 (2003), 613–648.

[46] A. Storjohann. 2006. Notes on computing minimal approximant bases. In *Challenges in Symbolic Computation Software (Dagstuhl Seminar Proceedings)*.

[47] V. Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13 (1969), 354–356.

[48] The CADO-NFS Development Team. 2017. CADO-NFS, An implementation of the number field sieve algorithm. http://cado-nfs.gforge.inria.fr/ Release 2.3.0.

[49] The FFLAS-FFPACK Group. 2019. FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package, version 2.3.2. http://github.com/linbox-team/fflas-ffpack.

[50] The LinBox Group. 2018. LinBox: Linear algebra over black-box matrices, version 1.5.3. https://github.com/linbox-team/linbox/.

[51] M. Van Barel and A. Bultheel. 1992. A general module theoretic framework for vector M-Padé and matrix rational interpolation. *Numer. Algorithms* 3 (1992), 451–462.

[52] G. Villard. 1997. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In *ISSAC'97*. ACM, 32–39.

[53] G. Villard. 2018. On computing the resultant of generic bivariate polynomials. In *ISSAC'18*. ACM, 391–398.

[54] A. Waksman. 1970. On Winograd's Algorithm for Inner Products. *IEEE Transactions On Computers* C-19 (1970), 360–361.

[55] S. Winograd. 1968. A New Algorithm for Inner Product. *IEEE Trans. Comput.* C-17, 7 (1968), 693–694.

[56] S. Winograd. 1971. On multiplication of 2×2 matrices. *Linear Algebra Appl.* 4 (1971), 381–388.

[57] W. Zhou and G. Labahn. 2012. Efficient Algorithms for Order Basis Computation. *J. Symbolic Comput.* 47, 7 (2012), 793–819.

[58] W. Zhou, G. Labahn, and A. Storjohann. 2012. Computing Minimal Nullspace Bases. In *ISSAC'12*. ACM, 366–373.