On a fast and nearly division-free algorithm for the characteristic polynomial

Fredrik Johansson

ON A FAST AND NEARLY DIVISION-FREE ALGORITHM FOR THE CHARACTERISTIC POLYNOMIAL

FREDRIK JOHANSSON

Abstract. We review the Preparata-Sarwate algorithm, a simple $O(n^{3.5})$ method for computing the characteristic polynomial, determinant and adjugate of an $n \times n$ matrix using only ring operations together with exact divisions by small integers. The algorithm is a baby-step giant-step version of the more well-known Faddeev-Leverrier algorithm. We make a few comments about the algorithm and evaluate its performance empirically.

1. Introduction

Let R be a commutative ring. Denote by $\omega > 2$ an exponent of matrix multiplication, meaning that we can multiply two $n \times n$ matrices using $O(n^\omega)$ ring operations (additions, subtractions and multiplications). Given a matrix $A \in R^{n \times n}$, how fast can we compute its characteristic polynomial, determinant and adjugate (or where applicable, inverse), without dividing by elements in R?

The obvious division-free algorithm is cofactor expansion, which uses $O(n!)$ operations. It is mainly interesting for $n \leq 4$ and for sparse symbolic matrices. The first published efficient method is the Faddeev-Leverrier algorithm [29, 15, 8, 16]; Alg. 2.2.7 in [7]), which solves the problem using roughly n matrix multiplications, for a complexity of $O(n^{\omega+1})$. The Faddeev-Leverrier algorithm requires some divisions, but it is *nearly division-free* in the sense that all divisors are fixed small integers $1, 2, \ldots, n$ rather than general elements of R, and the divisions are *exact* in the sense that the quotients remain in R.

The Berkowitz algorithm [3] achieves the same complexity using $O(n^4)$ operations, or $O(n^{\omega+1} \log n)$ with fast matrix multiplication, without performing any divisions. In practice, the Berkowitz algorithm is faster than the Faddeev-Leverrier algorithm by a constant factor and is now widely used in computer algebra systems for linear algebra over rings where divisions are problematic.

The complexity of the Faddeev-Leverrier algorithm can be improved using a baby-step giant-step technique, leading to a method with $O(n^{\omega+0.5}+n^3)$ complexity which is asymptotically better than the Berkowitz algorithm. This method was apparently first discovered by S. Winograd who did not publish the result, and then independently discovered and published by Preparata and Sarwate [25].

Unfortunately, many references to Faddeev-Leverrier algorithm in the literature present it as an $O(n^4)$ or $O(n^{\omega+1})$ algorithm without mentioning the improvement of Preparata and Sarwate. Berkowitz [3] claims that a baby-step giant-step Faddeev-Leverrier algorithm is possible but only cites private communication with S. Winograd without giving an explicit algorithm. The present author is not aware of any software using the Preparata-Sarwate algorithm.

There are perhaps three reasons for the relative obscurity of the method.[1] First of all, the Faddeev-Leverrier algorithm is numerically unstable [26, 31] making it virtually useless for ordinary numerical computation, and it is in any case much slower than standard $O(n^3)$ methods such as Gaussian elimination and Hessenberg reduction which are numerically stable. Second, faster methods are known for the most commonly used exact fields and rings such as $\mathbb{F}_q$, $\mathbb{Q}$ and $\mathbb{Z}$. Third, Kaltofen and Villard [20, 21, 30] have achieved an even lower division-free complexity of $O(n^{3.2})$ for the determinant or adjugate and $O(n^{3.29})$ for the characteristic polynomial with classical multiplication ($\omega = 3$), and about $O(n^{2.72})$ and $O(n^{2.84})$ for the respective tasks using the fastest currently known matrix multiplication algorithm due to Le Gall [12]. The Kaltofen-Villard algorithm is far more complicated than the Preparata-Sarwate algorithm, however.

The contribution of this note is twofold. First, we give explicit pseudocode for a version of the Preparata-Sarwate algorithm (Algorithm 2) that may serve as a convenient reference for future implementers. The code makes a superficial change to the algorithm as it is usually presented [25, 6, 1], halving memory usage (this is not an entirely negligible change since the Preparata-Sarwate algorithm uses $O(n^{2.5})$ memory). Second, we discuss the applicability of the algorithm and perform computational experiments comparing several algorithms over various rings.

2. The Faddeev-Leverrier algorithm and its refinement

We first recall the original Faddeev-Leverrier algorithm (Algorithm 1) for the characteristic polynomial $p_A(x) = c_n x^n + \cdots + c_1 x + c_0$ of a matrix $A \in R^{n \times n}$.

Algorithm 1 Faddeev-Leverrier algorithm

Input: $A \in R^{n \times n}$ where $n \geq 1$ and R is a commutative ring, R having a unit element and characteristic 0 or characteristic coprime to $1, 2, \ldots, n$

Output: $(p_A(x),\ \det(A),\ \mathrm{adj}(A))$

$c_n = 1, \quad B \leftarrow I, \quad k \leftarrow 1$
while $k \leq n - 1$ **do**
    $B \leftarrow AB$
    $c_{n-k} \leftarrow -\frac{1}{k} \operatorname{Tr}(B)$
    $B \leftarrow B + c_{n-k}\, I$
    $k \leftarrow k + 1$
end while
$c_0 \leftarrow -\frac{1}{n} \operatorname{Tr}(AB)$
return $(c_0 + c_1 x + \ldots + c_n x^n,\ (-1)^n c_0,\ (-1)^{n+1} B)$

Algorithm 1 is based on the recursion $c_{n-k} = -\frac{1}{k} \sum_{j=1}^{k} c_{n-k+j} \operatorname{Tr}(A^j)$: we compute a sequence of matrices (stored in the variable B) through repeated multiplication by A, and in each step extract a trace. The determinant and the adjugate

[1] Indeed, the present author also reinvented the same method. The first version of this note presented it simply as a "baby-step giant-step Faddeev-Leverrier algorithm", mentioning Berkowitz's attribution of the idea to S. Winograd, but without citing the concrete algorithm already given in [25]. The author is grateful to Eric Schost for pointing out the relevant prior art.

matrix appear as byproducts of this process: $(-1)^n \det(A)$ is the last coefficient c_0, and $(-1)^{n+1}\operatorname{adj}(A)$ is the penultimate entry in the matrix sequence.[2]

It is easy to see that Algorithm 1 performs $O(n)$ matrix multiplications and $O(n^2)$ additional arithmetic operations. The condition on the characteristic of R ensures that we can divide exactly by each k, i.e. $(xk)/k = x$ holds for $x \in R, k \leq n$.

Algorithm 2 Preparata-Sarwate algorithm (slightly modified)

Input: $A \in R^{n\times n}$ with the same conditions as in Algorithm 1
Output: $(p_A(x),\ \det(A),\ \operatorname{adj}(A))$

$m \leftarrow \lfloor\sqrt{n}\rfloor$
Precompute the matrices $A^1, A^2, A^3, \ldots, A^m$
Precompute the traces $t_k = \operatorname{Tr}(A^k)$ for $k = 1, \ldots, m$
$c_n = 1, \quad B \leftarrow I, \quad k \leftarrow 1$
while $k \leq n-1$ **do**
    $m \leftarrow \min(m, n-k)$
    $c_{n-k} \leftarrow -\frac{1}{k}\operatorname{Tr}(A, B)$
    for $j \leftarrow 1, 2, \ldots, m-1$ **do**
        $c_{n-k-j} \leftarrow \operatorname{Tr}(A^{j+1}, B)$ ▷ Using precomputed power of A
        for $i \leftarrow 0, 1, \ldots, j-1$ **do**
            $c_{n-k-j} \leftarrow c_{n-k-j} + t_{j-i}c_{n-k-i}$
        end for
        $c_{n-k-j} \leftarrow c_{n-k-j}/(-k-j)$
    end for
    $B \leftarrow A^m B$ ▷ Using precomputed power of A
    for $j \leftarrow 0, 1, \ldots, m-1$ **do**
        $B \leftarrow B + c_{n-k-j}A^{m-j-1}$ ▷ Using precomputed power, or $A^0 = I$
    end for
    $k \leftarrow k + m$
end while
$c_0 \leftarrow -\frac{1}{n}\operatorname{Tr}(A, B)$
return $(c_0 + c_1 x + \ldots + c_n x^n,\ (-1)^n c_0,\ (-1)^{n+1}B)$

Algorithm 1 computes a sequence of n matrices but only extracts a small amount of unique information (the trace) from each matrix. In such a scenario, we can often save time using a baby step giant-step approach in which we only compute $O(\sqrt{n})$ products explicitly (see [23, 4, 3, 17] for other examples of this technique). Preparata and Sarwate [25] improve Algorithm 1 by choosing $m \approx \sqrt{n}$ and precomputing the powers $A^1, A^2, \ldots, A^m$ and $A^m, A^{2m}, A^{3m}, \ldots$. The key observation is that we can compute $\operatorname{Tr}(AB)$ using $O(n^2)$ operations without forming the complete matrix product AB, by simply evaluating the dot products for the main diagonal of AB. We denote this *product trace* operation by $\operatorname{Tr}(A, B)$. Algorithm 2 presents a version of this method. It is clear from inspection that this version performs roughly $m + n/m \approx 2\sqrt{n}$ matrix multiplications of size $n \times n$, and $O(n^3)$ arithmetic operations in the remaining steps.

[2]The code can be tightened assuming $n \geq 2$, in which case the line before the start of the loop can be changed to $\{c_n = 1,\ c_{n-1} = -\operatorname{Tr}(A),\ B \leftarrow A + c_{n-1}I,\ k \leftarrow 2\}$. We can change the loop condition to $k \leq n$ and remove the line after the loop if we omit returning $\operatorname{adj}(A)$.

Algorithm 2 is a small modification of the Preparata-Sarwate algorithm as it is usually presented. Instead of computing the giant-step powers $A^m, A^{2m}, A^{3m}, \ldots$ explicitly, we expand the loop in Algorithm 1 to group m iterations, using multiplications by A^m to update the running sum over both the powers and their traces. This version performs essentially the same number of operations while using half the amount of memory.

As an observation for implementations, the matrix-matrix multiplications and product traces are done with invariant operands that get recycled $O(\sqrt{n})$ times. This can be exploited for preconditioning purposes, for instance by packing the data more efficiently for arithmetic operations. We also note that the optimal m may depend on the application, and a smaller value will reduce memory consumption at the expense of requiring more matrix multiplications.

3. Applicability and performance evaluation

When, if ever, does it make sense to use Algorithm 2? If R is a field, then the determinant, adjugate and characteristic polynomial can be computed in $O(n^\omega)$ operations or $O(n^3)$ classically allowing divisions [10, 24, 2, 22]. Most commonly encountered rings are integral domains, in which case we can perform computations in the fraction field and in some cases simply clear denominators. This does not automatically render an $O(n^{3.5})$ algorithm obsolete, since naively counting operations may not give the whole picture. Nevertheless, we can immediately discard some applications:

- For computing over $\mathbb{R}$ and $\mathbb{C}$ in ordinary floating-point arithmetic, the Faddeev-Leverrier algorithm is slower and far less numerically stable than textbook techniques such as reduction to Hessenberg form and Gaussian elimination with $O(n^3)$ or better complexity, as we already noted in the introduction.
- For finite fields, classical $O(n^3)$ methods using divisions have no drawbacks, and linear algebra with $O(n^{2.81})$ Strassen complexity is well established [9]. Over rings with small characteristic, the applicability of Algorithm 2 is in any case limited due to the integer divisions.

Generally speaking, division-free or nearly division-free algorithms are interesting for rings and fields R where dividing recklessly can lead to coefficient explosion (for example, $\mathbb{Q}$) or in which testing for zero is problematic (for example, exact models of $\mathbb{R}$). The optimal approach in such situations is usually to avoid computing directly in R or its fraction field, for example using modular arithmetic and interpolation techniques, but such indirect methods are more difficult to implement and must typically be designed on a case by case basis. By contrast, Algorithm 2 is easy to use anywhere. We will now look at some implementation experiments.

3.1. Integers.

For exact linear algebra over $\mathbb{Z}$ and $\mathbb{Q}$, the best methods are generally fraction-free versions of classical algorithms (such as the Bareiss version of Gaussian elimination) for small n, and multimodular or p-adic methods for large n (see for example [10, 24]). We do not expect Algorithm 2 to beat those algorithms, but it is instructive to examine its performance. Table 1 shows timings for computing a determinant, inverse or characteristic polynomial of an $n \times n$ matrix over $\mathbb{Z}$ with random entries in $-10, \ldots, 10$, using the following algorithms:

- FFLU: fraction-free LU factorization using the Bareiss algorithm.

TABLE 1. Time in seconds to compute characteristic polynomial (C), determinant (D), adjugate/inverse (A) of an $n \times n$ matrix over $\mathbb{Z}$ with random elements in $-10, \ldots, 10$, using various algorithms.

n	FFLU D	ModDet D	FFLU2 DA	ModInv A	ModCP CD	Berk CD	Alg1 CDA	Alg2 CDA
10	0.0000060	0.000021	0.000015	0.00012	0.000016	0.000035	0.000015	0.000030
20	0.000036	0.000078	0.00043	0.00096	0.00011	0.0010	0.00086	0.00061
50	0.0023	0.0012	0.011	0.016	0.0039	0.048	0.052	0.017
100	0.039	0.0068	0.14	0.18	0.055	0.84	1.1	0.22
200	0.64	0.044	2.3	1.8	0.89	16	27	4.1
300	3.4	0.15	13	9.0	4.6	94	174	20
400	12	0.38	45	22	15	321	696	66
500	32	0.77	127	52	37	900	2057	150

- FFLU2: as above, but using the resulting decomposition to compute A^{-1} (equivalently determining $\operatorname{adj}(A)$) by solving $AA^{-1} = I$.
- ModDet: a multimodular algorithm for the determinant.
- ModInv: a multimodular algorithm for the inverse matrix.
- ModCP: a multimodular algorithm for the characteristic polynomial.
- Berk: the Berkowitz algorithm for the characteristic polynomial.
- Alg1: original Faddeev-Leverrier algorithm, Algorithm 1.
- Alg2: modified Preparata-Sarwate algorithm, Algorithm 2.

We implemented Alg1 and Alg2 on top of Flint [13], while all the other tested algorithms are builtin Flint methods.

3.1.1. *Observations.* Alg2 clearly outperforms both Alg1 and Berk for large n, making it the best algorithm for computing the characteristic using direct arithmetic in $\mathbb{Z}$ (the modular algorithm is, as expected, superior). Alg2 is reasonably competitive for computing the inverse or adjugate matrix, coming within a factor 2-3$\times$ of FFLU2 and ModInv in this example. For determinants, the gap to the FFLU algorithm is larger, and the modular determinant algorithm is unmatched.

3.2. **Number fields.** Exact linear algebra over algebraic number fields $\mathbb{Q}(a)$ is an interesting use case for division-free algorithms since coefficient explosion is a significant problem for classical $O(n^3)$ algorithms. As in the case of $\mathbb{Z}$ and $\mathbb{Q}$, modular algorithms are asymptotically more efficient than working over $\mathbb{Q}(a)$ directly, but harder to implement. Here we compare the following algorithms:

- Sage: the `charpoly` method in SageMath [28], which implements a special-purpose algorithm for cyclotomic fields based on modular computations and reconstruction using the Chinese remainder theorem.
- Hess: Hessenberg reduction for the characteristic polynomial
- Dani: Danilevsky's algorithm for the characteristic polynomial.
- LU: LU factorization to compute the determinant.
- FFLU: fraction-free LU factorization using the Bareiss algorithm.
- LU2 and FFLU2: as above, but using the resulting decomposition to compute A^{-1} (equivalently determining $\operatorname{adj}(A)$) by solving $AA^{-1} = I$.
- Berk (Berkowitz), Alg1 and Alg2 as in the previous section.

TABLE 2. Time in seconds to compute characteristic polynomial (C), determinant (D), adjugate/inverse (A) of a matrix over a cyclotomic number field.

n	Sage CD	Hess CD	Dani CD	LU D	FFLU D	LU2 DA	FFLU2 DA	Berk CD	Alg1 CDA	Alg2 CDA
Input: $n \times n$ matrix over $\mathbb{Q}(\zeta_{20})$, entries $\sum_k (p/q)\zeta_{20}^k$, random $\lvert p\rvert \leq 10$, $1 \leq q \leq 10$.										
10	0.038	0.31	0.16	0.024	0.0059	0.21	0.11	0.010	0.0073	0.010
20	0.12	19	6.7	0.22	0.067	2.6	1.4	0.28	0.15	0.16
30	0.39	200	67	0.93	0.31	12	6.8	2.0	1.1	0.8
40	1.1		353	2.8	0.9	37	22	7.5	3.7	2.6
50	1.9			7.0	2.3	88	56	22	8.7	5.7
60	3.4			15	4.7	182	119	54	19	12
70	5.1			29	8.6	337	230	114	39	22
80	7.5			53	15	581	409	208	67	34
90	11			89	24			397	144	54
100	15			144	41			608	670	130
120	24			322	83			1439	3013	420
Input: $n \times n$ DFT matrix over $\mathbb{Q}(\zeta_n)$, entries $A_{i,j} = \zeta_n^{(i-1)(j-1)}$.										
10	0.010	0.0018	0.0016	0.00017	0.00022	0.00076	0.0014	0.00061	0.00075	0.00059
20	0.039	0.0019	0.0024	0.0017	0.0046	0.0071	0.038	0.020	0.020	0.0070
50	1.3	0.17	0.13	0.065	0.80	0.28	6.0	8.2	2.0	0.49
100	22	5.4	22	0.89	43	5.3	335	803	223	29
150	78	22	7.9	4.4	214	19	1423	7259	933	138
200	333	1928	140	31	1655	192				1687

With the exception of the Sage function, we implemented all the algorithms using Antic [14] for number field arithmetic and Flint for other operations. We perform fast matrix multiplication by packing number field elements into integers and multiplying matrices over $\mathbb{Z}$ via Flint. Our implementations of LU, LU2, Alg1 and Alg2 benefit from matrix multiplication while Hess, Dani, FFLU, FFLU2 and Berk do not. The benchmark is therefore not representative of the performance that ideally should be achievable with these algorithms, although it is fair in the sense that the implementation effort for Alg1 and Alg2 was minimal while the other algorithms would require much more code to speed up using block strategies.

Table 2 compares timings for two kinds of input: random matrices over a fixed cyclotomic field, and discrete Fourier transform (DFT) matrices which have special structure. Choosing cyclotomic fields allows us to compare with the dedicated algorithm for characteristic polynomials in Sage; the corresponding method for generic number fields in Sage is far slower. All the other algorithms make no assumptions about the field.

3.2.1. *Observations.* There are no clear winners since there is a complex interplay between operation count, multiplication algorithms, matrix structure and coefficient growth. Modular algorithms are the best solution in general for large n, but implementations for number fields are complex and less readily available in current software than for $\mathbb{Z}$ and $\mathbb{Q}$.

TABLE 3. Time in seconds to compute characteristic polynomial (C), determinant (D), adjugate/inverse (A) of a random $n \times n$ matrix in real ball arithmetic. The respective algorithms were run with $333 + p$ bits of precision, with p chosen to give roughly 100-digit output accuracy.

n	Hess CD	Hess2 CD	Dani CD	Eig CD	LU D	LU2 D	Berk CD	Alg1 CDA	Alg2 CDA
10	0.00021	0.00038	0.00022	0.017	0.000068	0.00023	0.00080	0.0010	0.00078
20	0.0021	0.0032	0.0020	0.18	0.00052	0.0015	0.0039	0.017	0.0092
50	0.045	0.057	0.048	4.6	0.0078	0.019	0.22	0.61	0.21
100	0.64	0.69	0.61	56	0.062	0.15	6.3	9.7	2.4
150	3.5	3.5	3.0	245	0.23	0.44	52	52	11
200	12	11	10		0.59	1.0	224	176	34
250	31	29	25		1.4	1.9	687	460	73
300	66	59	53		2.5	3.2	1804	1075	160
350	135	115	110		4.4	5.0	4033	2107	306
p	$10n$	$6n$	$10n$	0	n	0	$6n$	$6n$	$6n$

Among the non-modular algorithms, the $O(n^3)$ division-heavy Hessenberg and Danilevsky algorithms are nearly useless due to coefficient explosion for generic input, but both perform well on the DFT matrix. The LU and FFLU algorithms have more even performance but alternate with each other for the advantage depending on the matrix. Alg2 has excellent average performance for the determinant, characteristic polynomial as well as the adjugate matrix considering the large variability between the algorithms for different input. It is highly competitive for computing the inverse or adjugate of the random matrix.

3.3. **Ball arithmetic.** Division-free algorithms are useful when computing rigorously over $\mathbb{R}$ and $\mathbb{C}$ in interval arithmetic or ball arithmetic. The reason is that we cannot test whether elements are zero, so algorithms like Gaussian elimination and Hessenberg reduction fail when they need to branch upon zero pivot elements or zero vectors. Although zeros will not occur for random input, they are likely to occur for structured matrices arising in applications. *A posteriori* verification of approximate numerical solutions or perturbation analysis is in principle the best workaround [27], but it is sometimes useful to fall back to more direct division-free methods, especially when working in very high precision.

Table 3 shows timings for computing a determinant or characteristic polynomial with 100-digit accuracy using the following algorithms implemented in ball arithmetic. The input is taken to be an $n \times n$ real matrix with uniformly random entries in $[0, 1]$. For this experiment, we only focus on the determinant and characteristic polynomial (the conclusions regarding matrix inversion would be similar to those regarding the determinant).

- Hess: Hessenberg reduction using Gaussian elimination.
- Hess2: Hessenberg reduction using Householder reflections.
- LU: LU factorization using Gaussian elimination.
- LU2: approximate computation of the determinant using LU factorization followed by *a posteriori* verification.

- Eig: approximate computation of the eigenvalues using the QR algorithm followed by *a posteriori* verification and reconstruction of the characteristic polynomial from its roots.
- Berk (Berkowitz), Alg1 and Alg2 as in the previous section.

All algorithms were implemented in Arb[18] which uses the accelerated dot product and matrix multiplication algorithms described in [19]. The LU, LU2, Alg1 and Alg2 implementations benefit from fast matrix multiplication while Hess, Hess2, Eig and Berk do not.

The methods LU2 and Eig are numerically stable: the output balls are precise to nearly full precision for well-conditioned input. All other algorithms are unstable in ball arithmetic and lose $O(n)$ digits of accuracy. At least on this example, the rate of loss is almost the same for Hess2, Berk, Alg1 and Alg2, while LU is more stable and Hess and Dani are less stable. To make the comparison meaningful, we set the working precision (shown in the table) to an experimentally determined value so that all algorithms enclose the determinant with around 100 digits of accuracy.

3.3.1. *Observations.* For computing the characteristic polynomial in high-precision ball arithmetic, it seems prudent to try Hessenberg reduction and fall back to a division-free algorithm when it fails due to encountering a zero vector. The Berkowitz algorithm is the best fallback for small n, while Alg2 wins for large n ($n \approx 50$, although the cutoff will vary). On this particular benchmark, Alg2 runs only about $4\times$ slower than Hessenberg reduction, making it an interesting *one-size-fits-all* algorithm. The verification method (Eig) gives the best results if the precision is constrained, but is far more expensive than the other methods.

For computing the determinant alone, all the division-free methods are clearly inferior to methods based on LU factorization in this setting. The only advantage of the division-free algorithms is that they are foolproof while LU factorization requires some attention to implement correctly.

Better methods for computing the characteristic polynomial in ball arithmetic or interval arithmetic are surely possible. For the analogous problem of computing the characteristic polynomial over $\mathbb{Q}_p$, see [5].

3.4. **Polynomial quotient rings.** At first glance Algorithm 2 seems to hold potential for working over multivariate polynomial quotient rings. Such rings need not be integral domains and division can be very expensive (requiring Gröbner basis computations). Unfortunately, in most examples we have tried, Algorithm 2 performs worse than both the Berkowitz algorithm and Algorithm 1, presumably because repeated multiplication by the initial matrix A is much cheaper than multiplication by a power of A which generally will have much larger entries. There may be special classes of matrices for which the method performs well, however.

4. Discussion

We find that the Preparata-Sarwate algorithm often outperforms the Berkowitz algorithm in practice, in some circumstances even being competitive with $O(n^3)$ algorithms. We can therefore recommend it for more widespread adoption.

Galil and Pan [11, 1] have further refined the Preparata-Sarwate algorithm to eliminate the $O(n^3)$ complexity term which dominates asymptotically with a sufficiently fast matrix multiplication algorithm. We have not tested this method since those $O(n^3)$ operations are negligible in practice.

An interesting problem is whether it is possible to design a division-free or nearly division-free algorithm with better than $O(n^4)$ classical complexity that minimizes the observed problems with growing entries, particularly in multivariate rings.

References

[1] Jounaidi Abdeljaoued and Henri Lombardi. *Méthodes matricielles-Introduction à la complexité algébrique*, volume 42. Springer Science & Business Media, 2003.

[2] Jounaidi Abdeljaoued and Gennadi I. Malaschonok. Efficient algorithms for computing the characteristic polynomial in a domain. *Journal of Pure and Applied Algebra*, 156(2-3):127–145, 2001.

[3] Stuart J. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18(3):147–150, March 1984.

[4] R. P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. *Journal of the ACM*, 25(4):581–595, October 1978.

[5] Xavier Caruso, David Roe, and Tristan Vaccon. Characteristic polynomials of p-adic matrices. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*. ACM, July 2017.

[6] B. Codenotti and M. Leoncini. Parallelism and fast solution of linear systems. *Computers & Mathematics with Applications*, 19(10):1–18, 1990.

[7] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer Berlin Heidelberg, 1996.

[8] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal on Computing*, 5(4):618–623, December 1976.

[9] Jean-Guillaume Dumas and Clément Pernet. Computational linear algebra over finite fields. *arXiv preprint arXiv:1204.3735*, 2012.

[10] Jean-Guillaume Dumas, Clément Pernet, and Zhendong Wan. Efficient computation of the characteristic polynomial. In *Proceedings of the 2005 international symposium on Symbolic and algebraic computation - ISSAC'05*. ACM Press, 2005.

[11] Zvi Galil and Victor Pan. Parallel evaluation of the determinant and of the inverse of a matrix. *Information Processing Letters*, 30(1):41–45, January 1989.

[12] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation - ISSAC'14*. ACM Press, 2014.

[13] William B. Hart. Fast Library for Number Theory: An introduction. In *Mathematical Software – ICMS 2010*, pages 88–91. Springer Berlin Heidelberg, 2010.

[14] William B. Hart. ANTIC: Algebraic number theory in C. *Computeralgebra-Rundbrief: Vol. 56*, 2015.

[15] Gilbert Helmberg, Peter Wagner, and Gerhard Veltkamp. On Faddeev-Leverrier's method for the computation of the characteristic polynomial of a matrix and of eigenvectors. *Linear Algebra and its Applications*, 185:219–233, 1993.

[16] Shui-Hung Hou. Classroom note: A simple proof of the Leverrier–Faddeev characteristic polynomial algorithm. *SIAM Review*, 40(3):706–709, January 1998.

[17] Fredrik Johansson. A fast algorithm for reversion of power series. *Mathematics of Computation*, 84(291):475–484, May 2014.

[18] Fredrik Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, August 2017.

[19] Fredrik Johansson. Faster arbitrary-precision dot product and matrix multiplication. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. IEEE, June 2019.

[20] Erich Kaltofen. On computing determinants of matrices without divisions. In *Papers from the international symposium on Symbolic and algebraic computation - ISSAC'92*. ACM Press, 1992.

[21] Erich Kaltofen and Gilles Villard. On the complexity of computing determinants. *Computational Complexity*, 13(3-4):91–130, February 2005.

[22] Vincent Neiger and Clément Pernet. Deterministic computation of the characteristic polynomial in the time of matrix multiplication. *arXiv preprint arXiv:2010.04662*, 2020.

[23] Michael S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, March 1973.

[24] Clément Pernet and Arne Storjohann. Faster algorithms for the characteristic polynomial. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation - ISSAC'07*. ACM Press, 2007.
[25] F.P. Preparata and D.V. Sarwate. An improved parallel processor bound in fast matrix inversion. *Information Processing Letters*, 7(3):148–150, April 1978.
[26] Rizwana Rehman and Ilse CF Ipsen. La Budde's method for computing characteristic polynomials. *arXiv preprint arXiv:1104.3769*, 2011.
[27] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, May 2010.
[28] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.0)*, 2020. `https://www.sagemath.org`.
[29] Urbain Le Verrier. Sur les variations séculaire des élements des orbites pour les sept planétes principales. *J. de Math*, (s 1):5, 1840.
[30] Gilles Villard. Kaltofen's division-free determinant algorithm differentiated for matrix adjoint computation. *Journal of Symbolic Computation*, 46(7):773–790, July 2011.
[31] James Hardy Wilkinson. *The algebraic eigenvalue problem*, volume 87. Clarendon press Oxford, 1965.

Inria Bordeaux-Sud-Ouest and Institut Math. Bordeaux, U. Bordeaux,
33400 Talence, France
Email address: `fredrik.johansson@gmail.com`