Vincent Neiger .... LIP6, Sorbonne Université, France

joint work with
Bruno Salvy, Gilles Villard .... Inria/CNRS, ENS Lyon, France
Seung Gyu Hyun, Éric Schost .... U. Waterloo, Canada

faster modular composition of polynomials

Algorithmic Number Theory seminar
Institut de Mathématiques de Bordeaux, France
23 January 2024

outline

- **context and contribution**

- **minimal polynomial**

- **modular composition**

- **implementation aspects**

outline

context and contribution

- complexity and software
- minpoly & modular composition
- summary of contributions

minimal polynomial

modular composition

implementation aspects

“fast”: measuring efficiency

efficient algorithms for polynomials, matrices, power series, . . .
with coefficients in some base field $\mathbb{K}$

- low **complexity bound**
- low **execution time**

low memory usage, power consumption, . . .

prime field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$
field extension $\mathbb{F}_p[x]/\langle f(x) \rangle$
rational numbers $\mathbb{Q}$

"fast": measuring efficiency

efficient algorithms for polynomials, matrices, power series, . . .
with coefficients in some base field $\mathbb{K}$

- low **complexity bound**
- low **execution time**

low memory usage, power consumption, . . .

prime field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$
field extension $\mathbb{F}_p[x]/\langle f(x) \rangle$
rational numbers $\mathbb{Q}$

algebraic complexity bounds
$\rightsquigarrow$ count number of operations in $\mathbb{K}$

- 👍 standard complexity model for algebraic computations
- 👍 accurate for finite fields $\mathbb{K} = \mathbb{F}_p$
- 👎 ignores coefficient growth, e.g. over $\mathbb{K} = \mathbb{Q}$

“fast”: measuring efficiency

efficient algorithms for polynomials, matrices, power series, . . .
with coefficients in some base field $\mathbb{K}$

- low **complexity bound**
- low **execution time**

low memory usage, power consumption, . . .

prime field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$
field extension $\mathbb{F}_p[x]/\langle f(x) \rangle$
rational numbers $\mathbb{Q}$

practical performance
$\rightsquigarrow$ measure software running time

this talk:

- working over $\mathbb{K} = \mathbb{F}_p$ with word-size prime p
- Intel Core i7-7600U @ 2.80GHz, no multithreading

modular composition

polynomials a, f, g, h univariate over $\mathbb{K}$

modular composition

given g, a, h, compute $h(a) \bmod g$

modular composition

polynomials a, f, g, h univariate over $\mathbb{K}$

modular composition

given g, a, h, compute $h(a) \bmod g$

minimal polynomial

given g, a, compute f such that $f(a) = 0 \bmod g$

modular composition

polynomials a, f, g, h univariate over $\mathbb{K}$

modular composition

given g, a, h, compute $h(a) \bmod g$

minimal polynomial

given g, a, compute f such that $f(a) = 0 \bmod g$

related problems: power projections & inverse composition

[reminder] matrices: multiplication

$$\mathbf{M} = \begin{bmatrix} 28 & 68 & 75 & 70 \\ 38 & 25 & 75 & 55 \\ 24 & 1 & 56 & 28 \end{bmatrix} \in \mathbb{K}^{3\times 4} \longrightarrow 3 \times 4 \text{ matrix over } \mathbb{K} \text{ (here } \mathbb{F}_{97})$$

fundamental operations on $m \times m$ matrices:

- addition is "quadratic": $O(m^2)$ operations in $\mathbb{K}$
- naive multiplication is cubic: $O(m^3)$

[Strassen'69]

breakthrough: **subcubic** matrix multiplication

- complexity exponent $\omega \approx 2.81$
- used in practice for $m \geqslant$ a few 100s
  in NTL, FLINT, fflas-ffpack...

i.e. $O(m^{\omega})$ complexity

- best-known exponent $\omega \approx 2.373$
  [Le Gall'14] [Alman-Williams'20]
- "galactic" algorithms: strongly impractical as such

[reminder] polynomials: multiplication

$p = 87x^7 + 74x^6 + 60x^5 + 46x^4 + 16x^3 + 41x^2 + 86x + 69$

$p \in \mathbb{K}[x]_{<8}$ $\longrightarrow$ univariate polynomial in x of degree < 8 over $\mathbb{K}$

fundamental operations on polynomials of degree $< d$:

- addition and Horner's evaluation are linear: $O(d)$
- naive multiplication is quadratic: $O(d^2)$

[Karatsuba'62] $M(d) \in O(d^{1.58})$

breakthrough: **subquadratic** polynomial multiplication

[Schönhage-Strassen'71] [Nussbaumer'80] [Cantor-Kaltofen'91] $M(d) \in O(d \log(d) \log\log(d))$

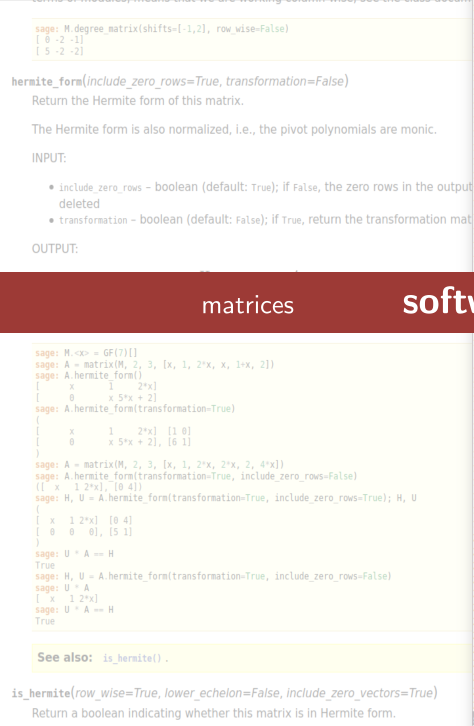
breakthrough: **quasi-linear** polynomial multiplication

research still active, with recent progress by [Harvey-van der Hoeven-Lecerf]

- change of representation by evaluation-interpolation
- used in practice as soon as $d \approx 100$ $(\mathbb{K} = \mathbb{F}_p)$
- FFT techniques using (virtual) roots of unity

note: $M(d) \in O(d \log(d))$
if provided a “good” root of unity

matrices **software** polynomials



```
sage: M.degree_matrix(shifts=[-1,2], row_wise=False)
[ 0 -2 -1]
[ 5 -2 -2]
```

hermite_form(*include_zero_rows=True, transformation=False*)

Return the Hermite form of this matrix.

The Hermite form is also normalized, i.e., the pivot polynomials are monic.

INPUT:

- `include_zero_rows` – boolean (default: `True`); if `False`, the zero rows in the output deleted
- `transformation` – boolean (default: `False`); if `True`, return the transformation mat

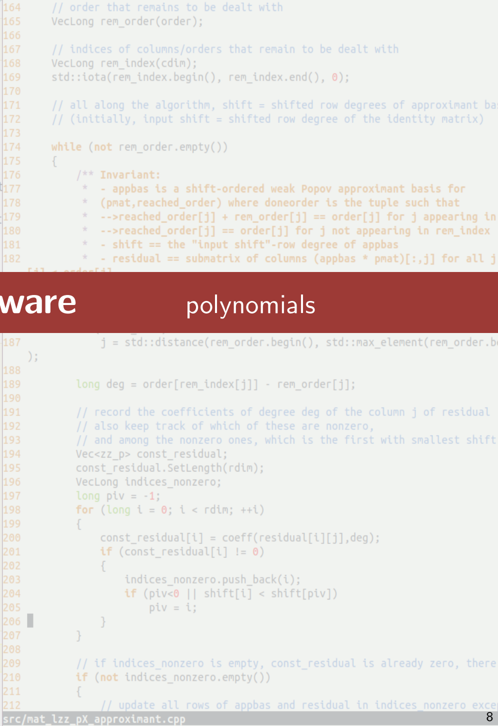
OUTPUT:

```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

is_hermite(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
    // order that remains to be dealt with
    VecLong rem_order(order);

    // indices of columns/orders that remain to be dealt with
    VecLong rem_index(cdim);
    std::iota(rem_index.begin(), rem_index.end(), 0);

    // all along the algorithm, shift = shifted row degrees of approximant ba
    // (initially, input shift = shifted row degree of the identity matrix)

    while (not rem_order.empty())
    {
        /** Invariant:
         *  - appbas is a shift-ordered weak Popov approximant basis for
         *  (pmat,reached_order) where doneorder is the tuple such that
         *  -->reached_order[j] + rem_order[j] == order[j] for j appearing in
         *  -->reached_order[j] == order[j] for j not appearing in rem_index
         *  - shift == the "input shift"-row degree of appbas
         *  - residual == submatrix of columns (appbas * pmat)[:,j] for all j
```

```
            j = std::distance(rem_order.begin(), std::max_element(rem_order.be
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
```

src/mat_lzz_pX_approximant.cpp

open-source mathematics software system

sage *Python/Cython*

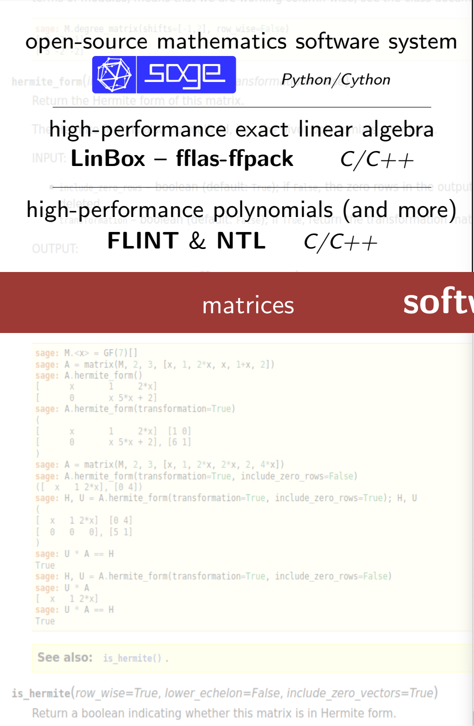
high-performance exact linear algebra

LinBox – fflas-ffpack *C/C++*

high-performance polynomials (and more)

FLINT & NTL *C/C++*

matrices **software** polynomials



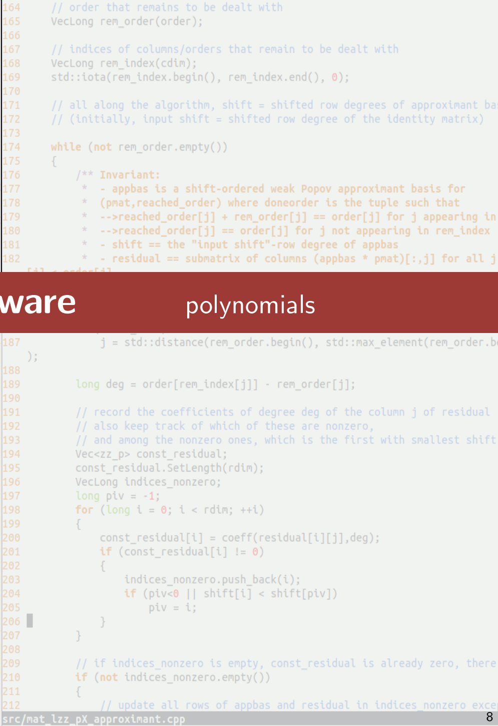
Return the Hermite form of this matrix.

```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

`is_hermite`(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
    // order that remains to be dealt with
    VecLong rem_order(order);

    // indices of columns/orders that remain to be dealt with
    VecLong rem_index(cdim);
    std::iota(rem_index.begin(), rem_index.end(), 0);

    // all along the algorithm, shift = shifted row degrees of approximant ba
    // (initially, input shift = shifted row degree of the identity matrix)

    while (not rem_order.empty())
    {
        /** Invariant:
         *  - appbas is a shift-ordered weak Popov approximant basis for
         *  (pmat,reached_order) where doneorder is the tuple such that
         *  -->reached_order[j] + rem_order[j] == order[j] for j appearing in
         *  -->reached_order[j] == order[j] for j not appearing in rem_index
         *  - shift == the "input shift"-row degree of appbas
         *  - residual == submatrix of columns (appbas * pmat)[:,j] for all j
```

```
            j = std::distance(rem_order.begin(), std::max_element(rem_order.be
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
```

src/mat_lzz_pX_approximant.cpp

open-source mathematics software system

sage *Python/Cython*

high-performance exact linear algebra

LinBox – fflas-ffpack *C/C++*

high-performance polynomials (and more)

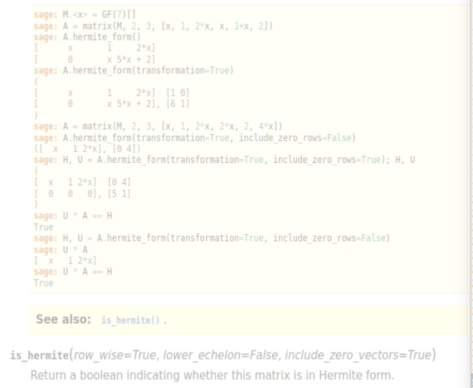
FLINT & NTL *C/C++*

- choice of algorithms
- data structures and storage
- cache efficiency
- SIMD vectorization instructions
- multithreading, GPU programming

matrices

software

polynomials

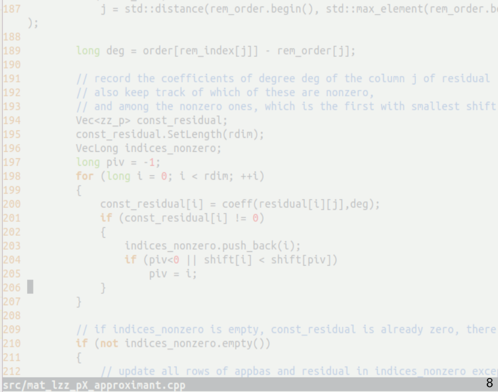


```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

is_hermite(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
            j = std::distance(rem_order.begin(), std::max_element(rem_order.be
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
src/mat_lzz_pX_approximant.cpp
```

open-source mathematics software system

sage *Python/Cython*

---

high-performance exact linear algebra

LinBox – fflas-ffpack *C/C++*

high-performance polynomials (and more)

FLINT & NTL *C/C++*

- choice of algorithms
- data structures and storage
- cache efficiency
- SIMD vectorization instructions
- multithreading, GPU programming

matrices **software** polynomials

what you can compute in about 1 second

with fflas-ffpack with NTL

`PLUQ`	$m = 3800$	1.00s
`LinSys`	$m = 3800$	1.00s
`MatMul`	$m = 3000$	0.97s
`Inverse`	$m = 2800$	1.01s
`CharPoly`	$m = 2000$	1.09s

`PolMul`	$d = 7 \times 10^6$	1.03s
`Division`	$d = 4 \times 10^6$	0.96s
`XGCD`	$d = 2 \times 10^5$	0.99s
`MinPoly`	$d = 2 \times 10^5$	1.10s
`MPeval`	$d = 1 \times 10^4$	1.01s

univariate polynomials: computational problems

most problems have quasi-linear complexity

thanks to reductions to `PolMul`

$O(M(d))$

- addition $f + g$, multiplication $f * g$
- division with remainder $f = qg + r$
- truncated inverse $f^{-1} \bmod x^d$
- extended GCD $fu + gv = \gcd(f, g)$

$O(M(d) \log(d))$

- multipoint eval. $f \mapsto f(x_1), \dots, f(x_d)$
- interpolation $f(x_1), \dots, f(x_d) \mapsto f$
- Padé approximation $f = \frac{p}{q} \bmod x^d$
- minpoly of linearly recurrent sequence

PETER BÜRGISSER
MICHAEL CLAUSEN
M. AMIN SHOKROLLAHI

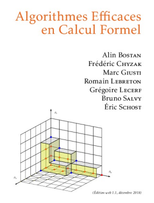
Volume 315

Grundlehren der mathematischen Wissenschaften

A Series of Comprehensive Studies in Mathematics

ALGEBRAIC COMPLEXITY THEORY

Springer-Verlag Berlin Heidelberg GmbH



Algorithmes Efficaces en Calcul Formel

Alin Bostan
Frédéric Chyzak
Marc Giusti
Romain Lebreton
Grégoire Lecerf
Bruno Salvy
Éric Schost

(Édition web 1.1, décembre 2018)

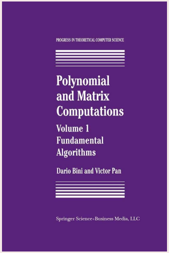
Modern Computer Algebra

Third Edition

Joachim von zur Gathen and Jürgen Gerhard

CAMBRIDGE

more information – www.cambridge.org/9781107039032



PROGRESS IN THEORETICAL COMPUTER SCIENCE

Polynomial and Matrix Computations

Volume 1
Fundamental Algorithms

Dario Bini and Victor Pan

Springer Science+Business Media, LLC

univariate polynomials: computational problems

most problems have quasi-linear complexity

thanks to reductions to `PolMul`

$O(M(d))$

- addition $f+g$, multiplication $f*g$
- division with remainder $f=qg+r$
- truncated inverse $f^{-1} \bmod x^d$

$O(M(d)\log(d))$

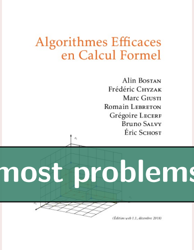
- multipoint eval. $f \mapsto f(x_1),\dots,f(x_d)$
- interpolation $f(x_1),\dots,f(x_d) \mapsto f$
- Padé approximation $f=\frac{p}{q} \bmod x^d$
- extended GCD $fu+gv=\gcd(f,g)$
- minpoly of linearly recurrent sequence

PETER BÜRGISSER
MICHAEL CLAUSEN
M. AMIN SHOKROLLAHI

Volume 315
Grundlehren der mathematischen Wissenschaften
A Series of Comprehensive Studies in Mathematics

ALGEBRAIC COMPLEXITY THEORY

Springer-Verlag Berlin Heidelberg GmbH



Algorithmes Efficaces
en Calcul Formel

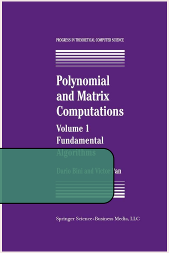
Alin Bostan
Frédéric Chyzak
Marc Giusti
Romain Lebreton
Grégoire Lecerf
Bruno Salvy
Éric Schost

(Édition web 1.1, décembre 2018)

Modern Computer Algebra
Third Edition
Joachim von zur Gathen and Jürgen Gerhard

CAMBRIDGE

more information – www.cambridge.org/9781107039032



PROGRESS IN THEORETICAL COMPUTER SCIENCE

Polynomial
and Matrix
Computations
Volume 1
Fundamental
Algorithms

Dario Bini and Victor an

Springer Science+Business Media, LLC

most problems, except. . .

univariate polynomials: open problems

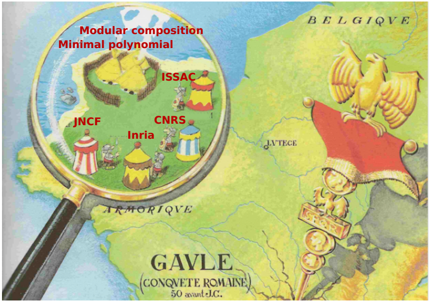
modular composition

given g, a, h, compute $h(a) \bmod g$

minimal polynomial

given g, a, compute f such that $f(a) = 0 \bmod g$

related problems: power projections & inverse composition



The year is 2024 A.D.

Basic Polynomial Algebra is entirely occupied by Computer Algebraists.

Well not entirely!

One small village of indomitable open problems still holds out against the invaders. And life is not easy for the scientists who garrison the fortified camps of ISSAC, JNCF, Inria, CNRS…

complexity improvements

[V.Neiger - B.Salvy - É.Schost - G.Villard, J.ACM 2024]

for generic input || using randomization

minimal polynomial
modular composition $\Big\}$ **in** $\tilde{O}(n^{(\omega+2)/3})$

exponent $(\omega+2)/3$: 1.67 for $\omega = 3$, 1.6 for $\omega = 2.8$, 1.46 for $\omega = 2.38$

previous work (composition)
- naive: $\tilde{O}(n^2)$
- [Brent-Kung 1978]: $O(n^{(\omega+1)/2})$

previous work (minpoly)
- naive: $\tilde{O}(n^\omega)$ or $\tilde{O}(n^2)$
- [Shoup 1994]: $O(n^{(\omega+1)/2})$

exponent $(\omega+1)/2$: 2 for $\omega = 3$, 1.9 for $\omega = 2.8$, 1.69 for $\omega = 2.38$

breakthough [Kedlaya-Umans 2011]:
composition in $\tilde{O}(n \log(q))$ bit operations, over $\mathbb{K} = \mathbb{F}_q$

quasi-linear bit complexity, yet currently impractical [van der Hoeven-Lecerf 2020]

software improvements

efficient implementation for the minimal polynomial
for large degrees, outperforms the state of the art

implementation for modular composition: work in progress

field $\mathbb{K} = \mathbb{F}_p$, prime p with 60 bits
Intel Core i7-7600U @ 2.80GHz

random input polynomials ⇒ "generic"

	general prime		FFT prime	
n	NTL	new	NTL	new
5k	0.349	0.496	0.130	0.208
20k	3.13	3.19	1.21	1.39
80k	31.5	23.6	13.9	10.7
320k	311	178	158	91.0

uses many types of computations on matrices over $\mathbb{K}[x]$
$\rightsquigarrow$ relies on the Polynomial Matrix Library

- multiplication for various parameters
- matrix-Padé approximation
- matrix division with remainder
- determinant
- system solving
- kernel

`https://github.com/vneiger/pml`

outline

- context and contribution
  - complexity and software
  - minpoly & modular composition
  - summary of contributions
- minimal polynomial
- modular composition
- implementation aspects

outline

context and contribution

- complexity and software
- minpoly & modular composition
- summary of contributions

minimal polynomial

- minimal polynomial. . .
- using power projections. . .
- and blocking + baby step-giant step

modular composition

implementation aspects

[reminder] minimal polynomial $\bmod g(x)$

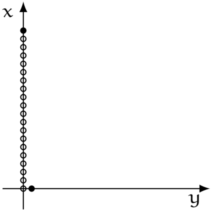
ideal $\mathcal{I} = \langle g(x), y - a(x) \rangle$:
set of all $F(x, y)$ such that $F(x, a(x)) = 0 \bmod g(x)$

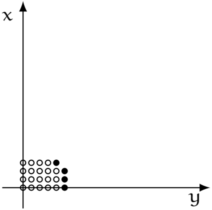
minimal polynomial = $f(y)$ of smallest degree in $\mathcal{I}$

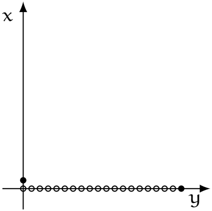
example: $f(y) = (y-1)^{16}$ is the minpoly
of $a(x) = x^2 + 1$ modulo $g(y) = x^{32}$

relation to bivariate resultant, and specific ideal bases

$$\mathcal{I} \;=\; \langle g(x), y \;-\; a(x) \rangle \;=\; \langle f(y), x \;-\; b(y) \rangle$$







using power projections

[Shoup 1994, 1999]

0. choose random vector $[\ell_1 \ \cdots \ \ell_n] \in \mathbb{K}^n$

$\rightarrow$ defines a linear form $\ell : \mathbb{K}[x]/\langle g \rangle \rightarrow \mathbb{K}$

1. compute linear recurrent sequence

$\ell(1), \ell(a \bmod g), \ldots, \ell(a^{2n-1} \bmod g)$

2. compute minimal recurrence relation $f(y)$

via Berlekamp-Massey / Padé approximation

minpoly $f(y)$

$\Downarrow$

$f(a) = 0 \bmod g$

$\Downarrow$

$f(y) =$ relation for $(a^k \bmod g)_k$

$\Downarrow$

$f(y) =$ relation for $(\ell(a^k \bmod g))_k$

using power projections

[Shoup 1994, 1999]

0. choose random vector $[\ell_1 \;\; \cdots \;\; \ell_n] \in \mathbb{K}^n$

→ defines a linear form $\ell : \mathbb{K}[x]/\langle g \rangle \to \mathbb{K}$

1. compute linear recurrent sequence

$\ell(1), \ell(a \bmod g), \ldots, \ell(a^{2n-1} \bmod g)$

2. compute minimal recurrence relation $f(y)$

via Berlekamp-Massey / Padé approximation

minpoly $f(y)$

$\Downarrow$

$f(a) = 0 \bmod g$

$\Downarrow$

$f(y)$ = relation for $(a^k \bmod g)_k$

$\Downarrow$

$f(y)$ = relation for $(\ell(a^k \bmod g))_k$

→ related to algorithm of [Wiedemann 1986]:

$$\ell(a^k \bmod g) = \begin{bmatrix} \ell_1 & \cdots & \ell_n \end{bmatrix} \; \mathbf{A}^k \; \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $\mathbf{A} \in \mathbb{K}^{n \times n}$ is the “multiplication matrix” of $a(x)$ modulo $g(x)$

for generic $a(x)$ and $g(0) \neq 0$, choose $\ell = [1 \quad 0 \quad \cdots \quad 0]$

then $\ell(a^k \bmod g)$ = constant coeff of $a^k \bmod g$

new minpoly algorithm:
blocking & baby-step giant-step

block Wiedemann approach [Coppersmith 1994]

iterating projection by $1 \times n$ vector on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2n-1}$
$\Rightarrow$ iterating projection by $m \times n$ matrix on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2d-1}$

choose $m \ll n$ and take $d = n/m$

new minpoly algorithm:
blocking & baby-step giant-step

block Wiedemann approach [Coppersmith 1994]

iterating projection by $1 \times n$ vector on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2n-1}$
$\Rightarrow$ iterating projection by $m \times n$ matrix on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2d-1}$

choose $m \ll n$ and take $d = n/m$

1. compute linear recurrent matrix sequence:

$$\mathbf{I}_m, \quad \begin{bmatrix}\mathbf{I}_m & \mathbf{0}\end{bmatrix} \mathbf{A} \begin{bmatrix}\mathbf{I}_m \\ \mathbf{0}\end{bmatrix}, \quad \ldots, \quad \begin{bmatrix}\mathbf{I}_m & \mathbf{0}\end{bmatrix} \mathbf{A}^{2d-1} \begin{bmatrix}\mathbf{I}_m \\ \mathbf{0}\end{bmatrix}$$

2. compute minimal matrix recurrence relation $\mathbf{P}(y) \in \mathbb{K}[y]^{m \times m}$
   via matrix-Berlekamp-Massey / matrix-Padé, complexity $O\tilde{}(m^\omega d)$

new minpoly algorithm:
blocking & baby-step giant-step

block Wiedemann approach [Coppersmith 1994]

iterating projection by $1 \times n$ vector on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2n-1}$
$\Rightarrow$ iterating projection by $m \times n$ matrix on powers $\mathbf{A}^0, \mathbf{A}^1, \ldots, \mathbf{A}^{2d-1}$

choose $m \ll n$ and take $d = n/m$

1. compute linear recurrent matrix sequence:

$$\mathbf{I}_m, \quad \begin{bmatrix}\mathbf{I}_m & \mathbf{0}\end{bmatrix} \mathbf{A} \begin{bmatrix}\mathbf{I}_m \\ \mathbf{0}\end{bmatrix}, \quad \ldots, \quad \begin{bmatrix}\mathbf{I}_m & \mathbf{0}\end{bmatrix} \mathbf{A}^{2d-1} \begin{bmatrix}\mathbf{I}_m \\ \mathbf{0}\end{bmatrix}$$

2. compute minimal matrix recurrence relation $\mathbf{P}(y) \in \mathbb{K}[y]^{m \times m}$

via matrix-Berlekamp-Massey / matrix-Padé, complexity $\tilde{O}(m^\omega d)$

step *1*: computing coefficient i of $x^j a^k$ mod g, for $i, j < m$, $k < 2d$
$\rightarrow$ **new baby-step giant-step** in $\tilde{O}(md^{(\omega+1)/2})$

- $f(y) = \det(\mathbf{P}(y))$ is the minimal polynomial of a modulo g
- $\mathbf{P}(y)$ is a good basis of $\mathcal{I} = \langle g(x), y - a(x)\rangle$

good: $\deg(P) \leqslant d$, Popov form, predictable degrees, ...

[reminder] polynomial matrices

$$\mathbf{A} = \begin{bmatrix} 3x+4 & x^3+4x+1 & 4x^2+3 \\ 5 & 5x^2+3x+1 & 5x+3 \\ 3x^3+x^2+5x+3 & 6x+5 & 2x+1 \end{bmatrix} \in \mathbb{K}[x]^{3\times 3}$$

3×3 matrix of degree 3
with entries in $\mathbb{K}[x] = \mathbb{F}_7[x]$

operations on $\mathbb{K}[x]^{m\times m}_{<d}$

- combination of matrix and polynomial computations
- addition in $O(m^2 d)$, naive multiplication in $O(m^3 d^2)$

[Cantor-Kaltofen'91]

multiplication in $O(m^\omega d\log(d) \;+\; m^2 d\log(d)\log\log(d))$

$\in O(m^\omega \mathsf{M}(d)) \subset O\tilde{~}(m^\omega d)$

2×2 matrices in XGCD, Padé approximation, Berlekamp-Massey, Toeplitz linear systems...

$\rightsquigarrow$ $m \times m$ matrix versions of these problems

- some problems&techniques shared with matrices over $\mathbb{K}$
- some problems&techniques specific to entries in $\mathbb{K}[x]$

polynomial matrices: main computational problems

reductions of most problems to polynomial matrix multiplication

matrix $m \times m$ of degree d → $O\tilde{~}(m^\omega d)$

of "average" degree $\frac{D}{m}$ → $O\tilde{~}(m^\omega \frac{D}{m})$

classical matrix operations

- multiplication
- inversion $O\tilde{~}(m^3 d)$
- kernel, system solving
- rank, determinant

univariate relations

- Hermite-Padé approximation
- vector rational interpolation
- syzygies, modular equations

transformation to normal forms

- triangularization: Hermite form
- row reduction: Popov form
- diagonalization: Smith form

polynomial matrices: two open questions

deterministic Smith form

$$\begin{bmatrix} & & \\ & \mathbf{A} & \\ & & \end{bmatrix} \longrightarrow \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_m \end{bmatrix}$$

s_{i+1} divides s_i

- complexity $O\tilde{}(m^\omega d)$ [Storjohann'03]
- Las Vegas randomized algorithm
- requires large field $\mathbb{K}$

deterministic algo in $O\tilde{}(m^\omega d)$?

polynomial matrices: two open questions

deterministic Smith form

$$\begin{bmatrix} & & \\ & \mathbf{A} & \\ & & \end{bmatrix} \longrightarrow \begin{bmatrix} s_1 & & & \\ & s_2 & & \\ & & \ddots & \\ & & & s_m \end{bmatrix}$$

s_{i+1} divides s_i

- complexity $\tilde{O}(m^\omega d)$ [Storjohann'03]
- Las Vegas randomized algorithm
- requires large field $\mathbb{K}$

deterministic algo in $\tilde{O}(m^\omega d)$?

algebraic approximants

$$\begin{tikzcd} p_1 a_1 + p_2 a_2 + \cdots + p_m a_m = 0 \bmod f(y) \arrow[d, "\text{structured } a_i\text{'s}"] \\ p_1 1 + p_2 a + \cdots + p_m a^{m-1} = 0 \bmod f(y) \end{tikzcd}$$

- most algorithms ignore the structure
- recent progress [Villard'18]+this talk
- restrictive: genericity, specific m

how to leverage this structure?

outline

context and contribution

- complexity and software
- minpoly & modular composition
- summary of contributions

minimal polynomial

- minimal polynomial...
- using power projections...
- and blocking + baby step-giant step

modular composition

implementation aspects

outline

- **context and contribution**
  - complexity and software
  - minpoly & modular composition
  - summary of contributions
- **minimal polynomial**
  - minimal polynomial...
  - using power projections...
  - and blocking + baby step-giant step
- **modular composition**
  - previously existing algorithms
  - approach for generic input
  - randomizing via change of basis
- **implementation aspects**

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $\tilde{O}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $\tilde{O}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

[Paterson-Stockmeyer 1971, Brent-Kung 1978]

rely on matrix multiplication using “slices” of length $\nu = \sqrt{n}$
$h(y) = S_0(y) + y^{\nu} S_1(y) + y^{2\nu} S_2(y) + \cdots + y^{(\nu-1)\nu} S_{\nu-1}(y)$

define $\alpha = a^{\nu} \bmod g$

$$h(a) = S_0(a) + \alpha S_1(a) + \alpha^2 S_2(a) + \cdots + \alpha^{\nu-1} S_{\nu-1}(a) \mod g$$

complexity: $\tilde{O}(n^{3/2})$ for $O(\sqrt{n})$ multiplications by a and α modulo g
$+\, O(n^{(\omega+1)/2})$ for matrix multiplication

in practice:
- much faster than naive approach
- $\tilde{O}(n^{3/2})$ regime lasts until largish n

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

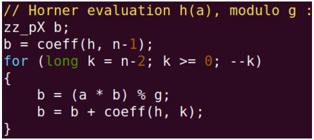
$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $O\tilde{}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm



```
// Horner evaluation h(a), modulo g :
zz_pX b;
b = coeff(h, n-1);
for (long k = n-2; k >= 0; --k)
{
    b = (a * b) % g;
    b = b + coeff(h, k);
}
```

n	Horner	Horner with precomputations	NTL built-in Brent-Kung
100	0.00229	0.00227	0.000441
200	0.0162	0.00691	0.00110
400	0.117	0.0278	0.00312
800	0.637	0.116	0.00944
1600	2.52	0.515	0.0281
3200	10.4	2.23	0.0884
6400	45.8	9.61	0.273

field $\mathbb{K} = \mathbb{F}_p$, prime p with 60 bits
NTL 11.4.3 on Intel Core i7-7600U @ 2.80GHz

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $\tilde{O}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

$$\begin{aligned}
h(a) &= S_0(a) + \alpha S_1(a) + \alpha^2 S_2(a) + \cdots + \alpha^{\nu-1} S_{\nu-1}(a) \\
&= \begin{bmatrix} 1 & \alpha & \cdots & \alpha^{\nu-1} \end{bmatrix} \begin{bmatrix} S_0(a) \\ S_1(a) \\ \vdots \\ S_{\nu-1}(a) \end{bmatrix} \\
&= \begin{bmatrix} 1 & \alpha & \cdots & \alpha^{\nu-1} \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & \cdots & S_{0,\nu-1} \\ S_{1,0} & S_{1,1} & \cdots & S_{1,\nu-1} \\ \vdots & \vdots & & \vdots \\ S_{\nu-1,0} & S_{\nu-1,1} & \cdots & S_{\nu-1,\nu-1} \end{bmatrix} \begin{bmatrix} 1 \\ a \\ \vdots \\ a^{\nu-1} \end{bmatrix}
\end{aligned}$$

recall: $\alpha = a^{\nu} \bmod g$

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $\tilde{O}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

$$h(a) = S_0(a) + \alpha S_1(a) + \alpha^2 S_2(a) + \cdots + \alpha^{\nu-1} S_{\nu-1}(a)$$

recall: $\alpha = a^\nu \bmod g$

$$= \begin{bmatrix} 1 & \alpha & \cdots & \alpha^{\nu-1} \end{bmatrix} \begin{bmatrix} S_0(a) \\ S_1(a) \\ \vdots \\ S_{\nu-1}(a) \end{bmatrix}$$

length ν vectors over $\mathbb{K}[x]_{<n}$

$\nu \times \nu$ matrix over $\mathbb{K}$

$$= \begin{bmatrix} 1 & \alpha & \cdots & \alpha^{\nu-1} \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} & \cdots & S_{0,\nu-1} \\ S_{1,0} & S_{1,1} & \cdots & S_{1,\nu-1} \\ \vdots & \vdots & & \vdots \\ S_{\nu-1,0} & S_{\nu-1,1} & \cdots & S_{\nu-1,\nu-1} \end{bmatrix} \begin{bmatrix} 1 \\ a \\ \vdots \\ a^{\nu-1} \end{bmatrix}$$

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $O\tilde{}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

$$h(a) = S_0(a) + \alpha S_1(a) + \alpha^2 S_2(a) + \cdots + \alpha^{\nu-1} S_{\nu-1}(a)$$

recall: $\alpha = a^\nu \bmod g$

$$= \begin{bmatrix} 1 & \alpha & \cdots & \alpha^{\nu-1} \end{bmatrix} \begin{bmatrix} S_0(a) \\ S_1(a) \\ \vdots \\ S_{\nu-1}(a) \end{bmatrix}$$

length ν vectors over $\mathbb{K}[x]_{<n}$

$\nu \times \nu$ matrix over $\mathbb{K}$

$$\begin{bmatrix} S_{0,0} & S_{0,1} & \cdots & S_{0,\nu-1} \\ S_{1,0} & S_{1,1} & \cdots & S_{1,\nu-1} \\ \vdots & \vdots & & \vdots \\ S_{\nu-1,0} & S_{\nu-1,1} & \cdots & S_{\nu-1,\nu-1} \end{bmatrix} \begin{bmatrix} 1 \\ a \\ \vdots \\ a^{\nu-1} \end{bmatrix}$$

$=$ matrix multiplication $(n \times \sqrt{n}) * (\sqrt{n} \times \sqrt{n}) * (\sqrt{n} \times n)$

input: $g(x)$ of degree n, $a(x)$ of degree $< n$, $h(y)$ of degree $< n$
output: $h(a(x)) \bmod g(x)$

$$h(a) \bmod g \;=\; h_0 + h_1(a \bmod g) + h_2(a^2 \bmod g) + \cdots + h_{n-1}(a^{n-1} \bmod g)$$

complexity: $\tilde{O}(n^2)$ for $O(n)$ multiplications by a modulo g
in practice: constant-factor speedup via precomputations on a and g

naive via Horner evaluation

classical composition algorithms

baby-step giant-step algorithm

a bivariate extension of modular composition:
input: $g(x)$ and $a(x)$ of degree n
$H(x, y)$ with $\deg_x < m$ and $\deg_y < d = n/m$
output: $H(x, a(x)) \bmod g(x)$

case discussed until now: $m = 1$, $d = n$

- algorithm: generalizes Brent-Kung [Nüsken-Ziegler 2004]
- complexity : $O(md^{(\omega+1)/2})$

modular composition, step 1: matrix minpoly

summary of the **minpoly algorithm**:

- specialization of first step of bivariate resultant [Villard 2018]
- accelerated by baby-step giant-step → $O\tilde{~}(md^{(\omega+1)/2} + m^{\omega} d)$
- genericity or randomization required for efficiency

computes an $m \times m$ polynomial matrix $\mathbf{P}(y)$ of degree $\leqslant d$
whose columns are minimal polynomial vectors of a mod g

change of representation

univariate vector	⟷	bivariate polynomial
$\begin{bmatrix} F_0(y) \\ F_1(y) \\ \vdots \\ F_{m-1}(y) \end{bmatrix}$	⟷	$F(x,y) = \sum_{i<m} F_i(y) x^i$

Popov basis of submodule $\mathcal{I} \cap \mathbb{K}[x,y]_{\deg_x < m}$ ⟷ Gröbner basis of ideal in $\mathbb{K}[x,y]$ $\mathcal{I} = \langle g(x), y - a(x) \rangle$

modular composition, step 1: matrix minpoly

summary of the **minpoly algorithm**:

- specialization of first step of bivariate resultant [Villard 2018]
- accelerated by baby-step giant-step → $O\tilde{~}(md^{(\omega+1)/2} + m^{\omega} d)$
- genericity or randomization required for efficiency

computes an $m \times m$ polynomial matrix $\mathbf{P}(y)$ of degree $\leqslant d$ whose columns are minimal polynomial vectors of a mod g

change of representation

univariate vector ⟷ bivariate polynomial

$$\begin{bmatrix} F_0(y) \\ F_1(y) \\ \vdots \\ F_{m-1}(y) \end{bmatrix} \longleftrightarrow F(x,y) = \textstyle\sum_{i<m} F_i(y) x^i$$

columns of $\mathbf{P}(y)$ ⇒ $F(x,a) = 0 \bmod g$ i.e. $F \in \mathcal{I}$

Popov basis of submodule $\mathcal{I} \cap \mathbb{K}[x,y]_{\deg_x < m}$ ⟷ Gröbner basis of ideal in $\mathbb{K}[x,y]$ $\mathcal{I} = \langle g(x), y - a(x) \rangle$

modular composition, step 2: balance degrees

$$\text{composition } h(y) \to b(x) = h(a) \bmod g = h(a) + F(x,a) \bmod g = H(x,a) \bmod g$$

$H(x,y) = h(y) + F(x,y)$ for any $F(x,y)$ generated by $\mathbf{P}(y)$

step 2: find $H(x,y)$ such that $\begin{cases} \deg_x(H) < m, \quad \deg_y(H) < d \\ h(a) = H(x,a) \bmod g \end{cases}$

modular composition, step 2: balance degrees

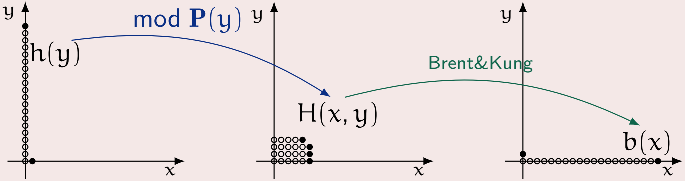
$$\begin{aligned}\text{composition } h(y) \to b(x) &= h(a) \bmod g \\ &= h(a) + F(x,a) \bmod g \\ &= H(x,a) \bmod g\end{aligned}$$

$H(x, y) = h(y) + F(x, y)$ for any $F(x, y)$ generated by $\mathbf{P}(y)$

step 2: find $H(x, y)$ such that $\begin{cases} \deg_x(H) < m, \quad \deg_y(H) < d \\ h(a) = H(x, a) \bmod g \end{cases}$

step 3: computing $H(x, a) \bmod g$ costs $\tilde{O}(md^{(\omega+1)/2})$

extending Brent&Kung's approach [Nüsken-Ziegler'04]



modular composition, step 2: balance degrees

$$\text{composition } h(y) \to b(x) = h(a) \bmod g$$
$$= h(a) + F(x,a) \bmod g$$
$$= H(x,a) \bmod g$$

$H(x,y) = h(y) + F(x,y)$ for any $F(x,y)$ generated by $\mathbf{P}(y)$

step 2: find $H(x,y)$ such that $\begin{cases} \deg_x(H) < m, \quad \deg_y(H) < d \\ h(a) = H(x,a) \bmod g \end{cases}$

step 3: computing $H(x,a) \bmod g$ costs $\tilde{O}(md^{(\omega+1)/2})$

extending Brent&Kung's approach [Nüsken-Ziegler'04]

finding $H(x,y)$: matrix division with remainder

$$\begin{bmatrix} h(y) \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{P}(y)\mathbf{Q}(y) + \left.\begin{bmatrix} H_0(y) \\ H_1(y) \\ \vdots \\ H_{m-1}(y) \end{bmatrix}\right\} \text{degree} < d$$

complexity $\tilde{O}(m^\omega d)$

complexity minimized for $m = n^{1/3}, d = n^{2/3}$

$\tilde{O}(n^{(\omega+2)/3})$

genericity and randomization

non-generic $a(x) \longrightarrow$

- $\mathcal{I} \cap \mathbb{K}[x,y]_{\deg_x < m}$ might not generate $\mathcal{I}$ (not an issue)
- finding a basis of $\mathcal{I} \cap \mathbb{K}[x,y]_{\deg_x < m}$ seems more difficult

randomization by **change of basis**

take a random $\gamma \in \mathbb{K}[x]/\langle g(x)\rangle$
w.h.p. γ has minimal polynomial $\mu(y)$ of degree n
$\Rightarrow 1, \gamma, \gamma^2, \ldots, \gamma^{n-1}$ is a basis of $\mathbb{K}[x]/\langle g(x)\rangle$

$$\Rightarrow \text{isomorphism} \quad \begin{array}{ccc} \mathbb{K}[x]/\langle g(x)\rangle & \to & \mathbb{K}[y]/\langle \mu(y)\rangle \\ a(x) & \mapsto & \alpha(y) \text{ such that } \alpha(\gamma) = a \bmod g \end{array}$$

algorithm:

1. compute $\alpha(y)$ and $\mu(y)$
2. compute $\beta(y) = h(\alpha(y)) \bmod \mu(y)$
3. compute $b(x) = \beta(\gamma(x)) \bmod g(x)$

outline

- **context and contribution**
  - complexity and software
  - minpoly & modular composition
  - summary of contributions
- **minimal polynomial**
  - minimal polynomial...
  - using power projections...
  - and blocking + baby step-giant step
- **modular composition**
  - previously existing algorithms
  - approach for generic input
  - randomizing via change of basis
- **implementation aspects**

outline

- **context and contribution**
  - complexity and software
  - minpoly & modular composition
  - summary of contributions
- **minimal polynomial**
  - minimal polynomial. . .
  - using power projections. . .
  - and blocking + baby step-giant step
- **modular composition**
  - previously existing algorithms
  - approach for generic input
  - randomizing via change of basis
- **implementation aspects**
  - framework for polynomial matrices
  - matrix fraction reconstruction
  - system solving and determinant

software development for polynomial matrices

```
sage: M.degree_matrix(shifts=[-1,2], row_wise=False)
[ 0 -2 -1]
[ 5 -2 -2]
```

hermite_form(*include_zero_rows=True, transformation=False*)

Return the Hermite form of this matrix.

The Hermite form is also normalized, i.e., the pivot polynomials are monic.

INPUT:

- `include_zero_rows` – boolean (default: `True`); if `False`, the zero rows in the output
  deleted
- `transformation` – boolean (default: `False`); if `True`, return the transformation mat

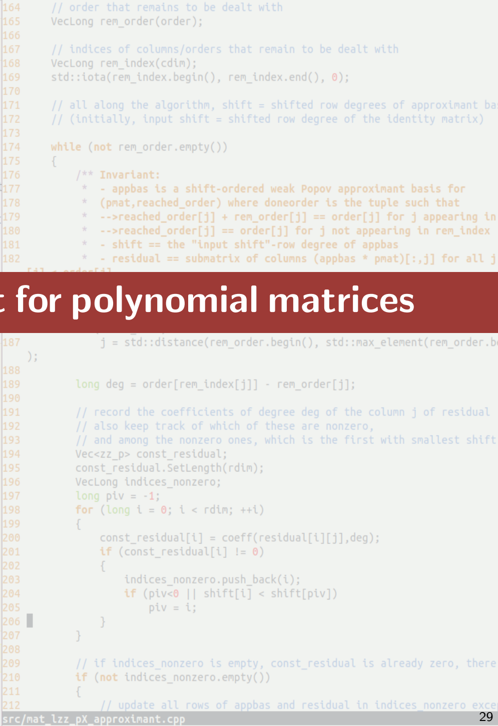
OUTPUT:

```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

is_hermite(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
    // order that remains to be dealt with
    VecLong rem_order(order);

    // indices of columns/orders that remain to be dealt with
    VecLong rem_index(cdim);
    std::iota(rem_index.begin(), rem_index.end(), 0);

    // all along the algorithm, shift = shifted row degrees of approximant ba
    // (initially, input shift = shifted row degree of the identity matrix)

    while (not rem_order.empty())
    {
        /** Invariant:
         *  - appbas is a shift-ordered weak Popov approximant basis for
         *  (pmat,reached_order) where doneorder is the tuple such that
         *  -->reached_order[j] + rem_order[j] == order[j] for j appearing in
         *  -->reached_order[j] == order[j] for j not appearing in rem_index
         *  - shift == the "input shift"-row degree of appbas
         *  - residual == submatrix of columns (appbas * pmat)[:,j] for all j

            j = std::distance(rem_order.begin(), std::max_element(rem_order.b
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
src/mat_lzz_pX_approximant.cpp
```

open-source mathematics software system

sage

Python/Cython

goals: **complete, robust, available**

(more than 60k downloads per month)

software development for polynomial matrices

```
sage: M.degree_matrix(shifts=[-1,2], row_wise=False)
[ 0 -2 -1]
[ 5 -2 -2]
```

INPUT:

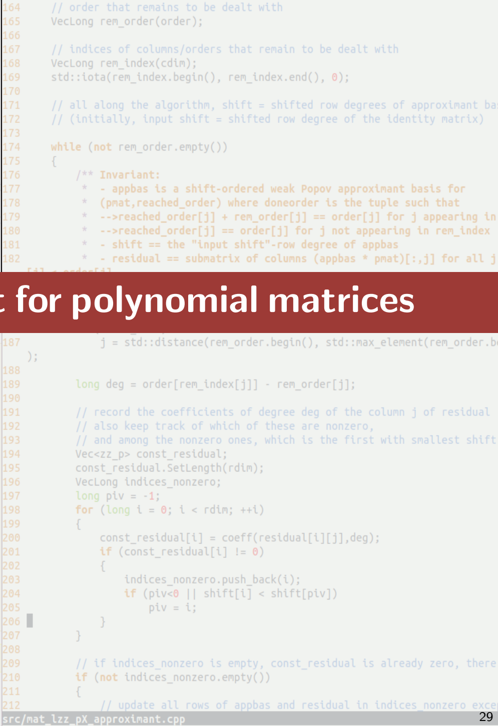
OUTPUT:

```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

is_hermite(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
    // order that remains to be dealt with
    VecLong rem_order(order);

    // indices of columns/orders that remain to be dealt with
    VecLong rem_index(cdim);
    std::iota(rem_index.begin(), rem_index.end(), 0);

    // all along the algorithm, shift = shifted row degrees of approximant ba
    // (initially, input shift = shifted row degree of the identity matrix)

    while (not rem_order.empty())
    {
        /** Invariant:
         * - appbas is a shifted-ordered weak Popov approximant basis for
         * (pmat,reached_order) where doneorder is the tuple such that
         * -->reached_order[j] + rem_order[j] == order[j] for j appearing in
         * -->reached_order[j] == order[j] for j not appearing in rem_index
         * - shift == the "input shift"-row degree of appbas
         * - residual == submatrix of columns (appbas * pmat)[:,j] for all j
[j] < order[j]

            j = std::distance(rem_order.begin(), std::max_element(rem_order.be
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
```

src/mat_lzz_pX_approximant.cpp

open-source mathematics software system

sage   *Python/Cython*

goals: **complete, robust, available**

(more than 60k downloads per month)

high-performance exact linear algebra

LinBox – fflas-ffpack   *C/C++*

goal: **optimized basic operations**

memory cost, vectorization, multithreading

software development for polynomial matrices

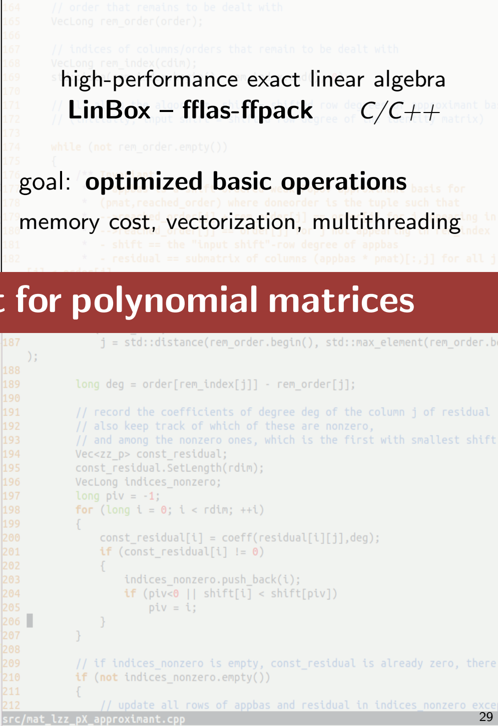
```
sage: M.degree_matrix(shifts=[-1,2], row_wise=False)
[ 0 -2 -1]
[ 5 -2 -2]
```

```
sage: M.<x> = GF(7)[]
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [0 4])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [0 4]
[  0   0   0], [5 1]
)
sage: U * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U * A
[  x   1 2*x]
sage: U * A == H
True
```

See also: `is_hermite()` .

`is_hermite`(*row_wise=True, lower_echelon=False, include_zero_vectors=True*)

Return a boolean indicating whether this matrix is in Hermite form.



```
    // order that remains to be dealt with
    VecLong rem_order(order);

    // indices of columns/orders that remain to be dealt with
    VecLong rem_index(cdim);
```

```
            j = std::distance(rem_order.begin(), std::max_element(rem_order.be
);

        long deg = order[rem_index[j]] - rem_order[j];

        // record the coefficients of degree deg of the column j of residual
        // also keep track of which of these are nonzero,
        // and among the nonzero ones, which is the first with smallest shift
        Vec<zz_p> const_residual;
        const_residual.SetLength(rdim);
        VecLong indices_nonzero;
        long piv = -1;
        for (long i = 0; i < rdim; ++i)
        {
            const_residual[i] = coeff(residual[i][j],deg);
            if (const_residual[i] != 0)
            {
                indices_nonzero.push_back(i);
                if (piv<0 || shift[i] < shift[piv])
                    piv = i;
            }
        }

        // if indices_nonzero is empty, const_residual is already zero, there
        if (not indices_nonzero.empty())
        {
            // update all rows of appbas and residual in indices_nonzero exce
```

src/mat_lzz_pX_approximant.cpp

open-source mathematics software system

sage *Python/Cython*

goals: **complete, robust, available**

(more than 60k downloads per month)

high-performance exact linear algebra

LinBox – fflas-ffpack *C/C++*

goal: **optimized basic operations**

memory cost, vectorization, multithreading

software development for polynomial matrices

Polynomial Matrix Library *C/C++*

403 files, 59k lines of code, including 17k lines of comments

`https://github.com/vneiger/pml`

[Hyun-Neiger-Schost '19]

- current version based on NTL
- work-in-progress version based on FLINT
- **welcome** comments, suggestions, contributions

"hey, this doesn't work!"
"yo, plans for implementing this?"
"how to decode RS codes with PML?"

wide range of algorithms

efficiency = state of the art

kernel, high-order lifting,
system solving, reduced form. . .

multiplication

most fundamental nontrivial operation ⇒ must be thoroughly optimized

various algorithms + use of thresholds:

- specific code for very small size or very small degree
(they arise in recursive calls)
- specific algorithm for large size & small degree
(used in matrix sequence computation for obtaining balanced bases)
- evaluation/interpolation + matrix multiplication over $\mathbb{K}$
(FFT points, geometric points, 3-primes FFT, . . .)

		20 bit FFT prime			60 bit prime		
m	d	ours	Linbox	ratio	ours	Linbox	ratio
8	131072	**1.034**	1.231	0.84	**3.067**	10.48	0.29
32	8192	**0.653**	0.776	0.84	**2.782**	8.510	0.33
128	2048	**3.079**	3.544	0.87	**20.84**	38.66	0.54
512	128	**3.623**	4.329	0.84	**31.54**	47.17	0.67

+ middle product versions

+ allowing precomputations (for repeated multiplication with the same matrix)

fraction reconstruction

reconstruct a matrix of degree $\leqslant d$ as a fraction with degrees $\leqslant d/2$

i.e. matrix version of Padé approximation: $\mathbf{F} = \mathbf{P}^{-1}\mathbf{Q} \bmod x^{d}$

$\rightsquigarrow$ used in block Wiedemann, matrix Berlekamp-Massey, basis reduction, ...

- using M-Basis / PM-Basis [Giorgi-Jeannerod-Villard 2003]
- performance similar to or better than state-of-the-art (LinBox)

$\rightsquigarrow$ depends on: bitsize of p, matrix dimensions, matrix degrees

- interpolant variants also implemented, and often slightly faster

m	n	d	ours	Linbox	ratio
8	4	131072	**6.091**	12.74	0.48
32	16	8192	**3.602**	5.665	0.64
128	64	2048	**13.61**	18.66	0.73
512	256	256	**32.08**	37.31	0.86

m	n	d	M	M-I	d	PM	PM-I	PM-Ig
8	4	32	4.31e-4	**3.54e-4**	32768	**4.36**	20.7	**4.38**
32	16	32	9.41e-3	**6.47e-3**	4096	**6.91**	17.0	**6.18**
128	64	32	0.333	**0.229**	1024	31.9	41.7	**25.7**
256	128	32	2.49	**1.46**	256	33.3	28.1	**24.2**

linear system solving over $\mathbb{F}_p[x]$

- Dixon's method turned out as the most efficient [Dixon 1982]
- kernel based solver is not far behind, and more general
- high-order lifting solver [Storjohann 2003] seems slower

m	d	Dixon	high-order lifting	kernel
16	1024	**0.695**	2.39	1.96
32	1024	**2.88**	13.8	8.06
128	512	**37.2**	266	84.2

determinant

- expansion by minors for small dimensions
- evaluation/interpolation at sufficiently many points
- solving a linear system with random right-hand side [Pan, 1988]
- triangularizing the matrix via kernel bases [Labahn-Neiger-Zhou, 2017]

m	d	minors	evaluation	linsolve	triangular
4	65536	**0.673**	1.90	5.78	**0.686**
16	4096	∞	**3.75**	**3.52**	6.12
32	4096	∞	26.5	**15.3**	32.4
64	2048	∞	109	**35.9**	71.0
128	512	∞	`out of memory`	**40.7**	71.8

outline

context and contribution

- complexity and software
- minpoly & modular composition
- summary of contributions

minimal polynomial

- minimal polynomial. . .
- using power projections. . .
- and blocking + baby step-giant step

modular composition

- previously existing algorithms
- approach for generic input
- randomizing via change of basis

implementation aspects

- framework for polynomial matrices
- matrix fraction reconstruction
- system solving and determinant

conclusion and perspectives

faster algorithms: minimal polynomial & modular composition

- also for power projections and inverse composition
- improved cost bound $O\tilde{~}(n^{(\omega+2)/3})$ (generic or randomized)
- baby steps-giant steps + univariate polynomial matrices

elaborating upon Villard's block Wiedemann with structured projections

- competitive practical performance for large degrees

perspectives & open questions

- improve practical performance further and wider
- further study impacts on related topics

Guruswami-Sudan decoding, bivariate resultants, algebraic approximants, guessing, ...

- open: exploit bivariate multiplication to reach $O\tilde{~}(n^{(\omega+3)/4})$?
- very much open: any new idea towards quasi-linear complexity??