

Vincent Neiger

LIP6, Sorbonne Université, France

**designing and exploiting fast algorithms
for univariate polynomial matrices**

Journées Nationales de Calcul Formel
Centre International de Rencontre Mathématiques
Marseille Luminy, France, 4 March 2024

outline

computer algebra

polynomial matrices

first algorithms

exercises

outline

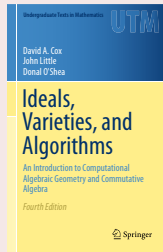
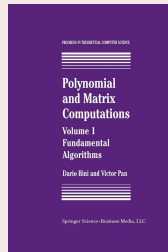
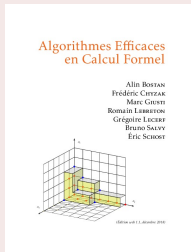
▶ **computer algebra**

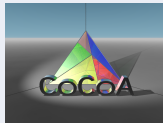
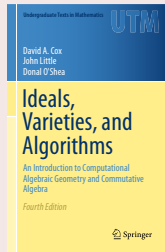
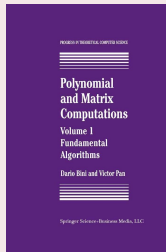
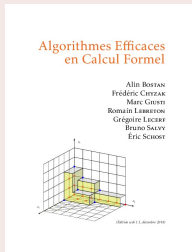
- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

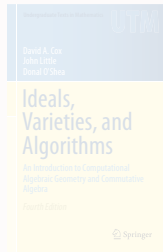
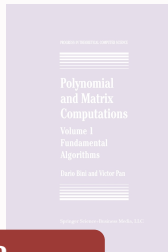
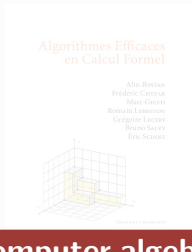
▶ **polynomial matrices**

▶ **first algorithms**

▶ **exercises**





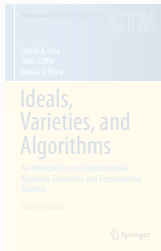
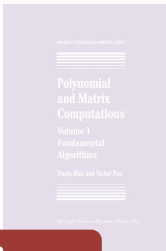
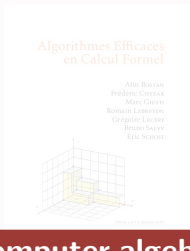


computer algebra

**algorithm design
and software implementations
for exact computations
with mathematical objects**

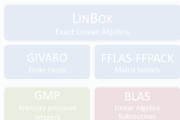


Euclid's GCD -300



computer algebra

algorithm design
and software implementations
for exact computations
with mathematical objects



Euclid's GCD -300

Gaussian elimination 179

computer algebra

algorithm design
and software implementations
for exact computations
with mathematical objects

MAGNUM
COMPUTER • ALGEBRA

RIIP



Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

MAGNUM
COMPUTER • ALGEBRA

RIIP



Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

FFT 1805, '65

MAGNUM
COMPUTER • ALGEBRA



Euclid's GCD -300

Gaussian elimination 179

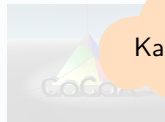
Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

FFT 1805, '65

MAGNUM
COMPUTER • ALGEBRA



Karatsuba '62

Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

FFT 1805, '65

MAGNUM
COMPUTER • ALGEBRA

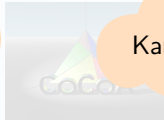


Strassen '69

Karatsuba '62



SymPy



Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

FFT 1805, '65

Buchberger '76

Strassen '69

Karatsuba '62

MAGNUM
COMPUTER • ALGEBRA

Maple

WOLFRAM
MATHEMATICA 12

SDG

LinBox
Linear Algebra

FELAS/FFPAC
Matrix arithmetic

GMP
Arbitrary precision
Integers

BLAS
Linear Algebra
Subroutines

SymPy

GoCode

Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

computer algebra

algorithm design
and **software implementations**
for **exact computations**
with **mathematical objects**

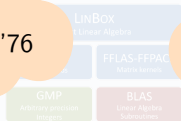
LLL '82, NFS '88

FFT 1805, '65

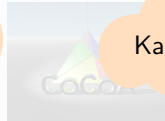
Buchberger '76

Strassen '69

Karatsuba '62



SymPy



Euclid's GCD -300

Gaussian elimination 179

Newton's method 1669

Principal Discoveries of Efficient Methods of Computing the DFT

Researcher(s)	Date	Sequence Lengths	Number of DFT Values	Application
C. F. Gauss [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. Carlini [28]	1828	12	—	Harmonic analysis of barometric pressure
A. Smith [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. Everett [23]	1860	12	5	Modeling underground temperature deviations
C. Runge [7]	1903	2^k	All	Harmonic analysis of functions
K. Stumpff [16]	1939	$2^k, 3^k$	All	Harmonic analysis of functions
Danielson and Lanczos [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. Thomas [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. Good [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
Cooley and Tukey [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. Winograd [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

FFT 1805, '65

Karatsuba '62

Euclid's GCD -300

Gaussian elimination 179

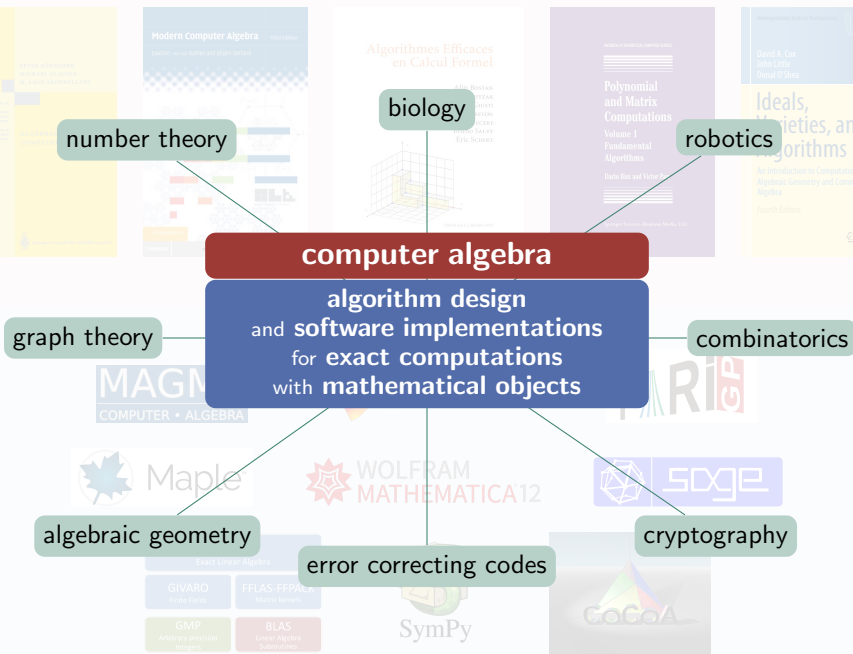
Newton's method 1669

Principal Discoveries of Efficient Methods of Computing the DFT

Researcher(s)	Date	Sequence Lengths	Number of DFT Values	Application
C. F. Gauss [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. Carlini [28]	1828	12	—	Harmonic analysis of barometric pressure
A. Smith [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. Everett [23]	1860	12	5	Modeling underground temperature deviations
C. Runge [7]	1903	2^k	All	Harmonic analysis of functions
K. Stumpff [16]	1939	$2^k, 3^k$	All	Harmonic analysis of functions
Danielson and Lanczos [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. Thomas [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. Good [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
Cooley and Tukey [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. Winograd [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

FFT 1805, '65

Karatsuba '62



computer algebra

algorithm design
and software implementations
for **exact** computations
with mathematical objects

number theory

biology

robotics

graph theory

combinatorics

algebraic geometry

error correcting codes

cryptography

this Friday:

- ▶ Alexandre Guillemot: validated **numerical** methods + **interval** arithmetic
- ▶ Tom Hubrecht: **correct rounding** of power function

error correcting codes



cryptographic protocols



XXth-XXIst centuries: digital data & interconnected networks
integrity – confidentiality

discrete structures: **exact** and **intensive** computations

- ▶ matrices of large size, with sparsity or structure
- ▶ polynomials and polynomial matrices in one variable
- ▶ polynomials in several variables

goal of computer algebra

fast algorithms: complexity & efficient implementations

error correcting codes



cryptographic protocols



XXth-XXIst centuries: digital data & interconnected networks
integrity – confidentiality

discrete structures: **exact** and **intensive** computations

- ▶ matrices of large size, with sparsity or structure
- ▶ polynomials and polynomial matrices in one variable
- ▶ polynomials in several variables

goal of computer algebra

fast algorithms: complexity & efficient implementations

general methodology: **reductions** to efficient **basic operations**

- ▶ IntMul: integer multiplication
- ▶ MatMul: matrix multiplication
- ▶ PolMul: univariate polynomial multiplication

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 $\mathbb{F}_p[x]/\langle f(x) \rangle$
rationals \mathbb{Q} , number fields, ...

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 $\mathbb{F}_p[x]/\langle f(x) \rangle$
rationals \mathbb{Q} , number fields, ...

algebraic complexity (upper) bounds

\rightsquigarrow count number of operations in \mathbb{K}

- 👍 standard complexity model for algebraic computations
- 👍 often well correlated to implementation timings (e.g. over $\mathbb{K} = \mathbb{F}_p$)
- 👎 ignores coefficient growth (e.g. over $\mathbb{K} = \mathbb{Q}$)

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 $\mathbb{F}_p[x]/\langle f(x) \rangle$
rationals \mathbb{Q} , number fields, ...

practical performance

↔ measure software running time

⚠ strongly influenced by the quality of the implementation

this talk:

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

matrices: multiplication

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

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

matrices: multiplication

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

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** $\omega \approx 2.81$ — i.e. $O(m^\omega)$ complexity
- ▶ **used in practice** for $m \geq$ a few 100s
in NTL, FLINT, fflas-ffpack...

[Coppersmith-Winograd 1990]

- ▶ best-known exponent $\omega \approx 2.3719$

[Le Gall'14] [Alman-Williams'20] [Duan-Wu-Zhou'23]

- ▶ “galactic” algorithms: strongly impractical as such

reductions: a new hope



Strassen, in his seminal 1969 paper

“Gaussian Elimination is Not Optimal”

sent a clear message to the scientific community:

Natural, obvious and centuries-old methods for solving important computational problems may be far from the fastest.

reductions: a new hope



Strassen, in his seminal 1969 paper
"Gaussian Elimination is Not Optimal"
sent a clear message to the scientific
community:

Natural, obvious and centuries-old
methods for solving important
computational problems may be
far from the fastest.



Abel Prize Laureates - Jeroen Zuiddam - Matrix multiplication and Shannon capacity



Centrum Wiskunde & Informatica
1.16K subscribers

Subscribe



Share



189 views 1 year ago Abel Prize Laureates Lectures (8 April 2022)

Asymptotic spectrum duality in computer science and discrete mathematics: Matrix multiplication and Shannon capacity

Abstract: ...more

0 Comments

Sort by

reductions: a new hope



Strassen, in his seminal 1969 paper "Gaussian Elimination is Not Optimal" sent a clear message to the scientific community:

Natural, obvious and centuries-old methods for solving important computational problems may be far from the fastest.



Abel Prize Laureates - Jeroen Zuiddam - Matrix multiplication and Shannon capacity



Centrum Wiskunde & Informatica
1.16K subscribers

Subscribe



1



Share



189 views 1 year ago Abel Prize Laureates Lectures (8 April 2022)

Asymptotic spectrum duality in computer science and discrete mathematics: Matrix multiplication and Shannon capacity


Abstract: ...more

0 Comments



Sort by

reductions: a new hope




Justin Bieber - Sorry (PURPOSE : The Movement)

Justin Bieber
 72.6M subscribers

3,772,923,806 views 8 years ago #JustinBieber #Vevo #Sorry
'Purpose' Available Everywhere Now!
iTunes: <http://smarturl.it/PurposeDix?IQid=VE...>
Stream & Add To Your Spotify Playlist: <http://smarturl.it/sPurpose?IQid=VEVO...> ...more

874,077 Comments Sort by



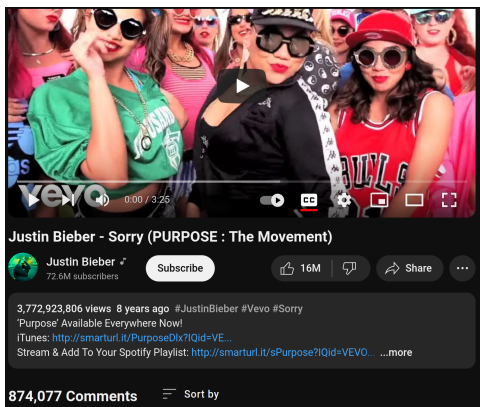
PSY - GANGNAM STYLE(강남스타일) M/V

officialpsy
 18.6M subscribers

5,064,558,413 views 11 years ago #강남스타일 #PSY #GANGNAMSTYLE
PSY - '1 LUV IT' M/V @ [PSY - '1 LUV IT' M/V](#)
PSY - 'New Face' M/V @ [PSY - 'New Face' M/V](#)
...more

5,360,274 Comments Sort by

reductions: a new hope

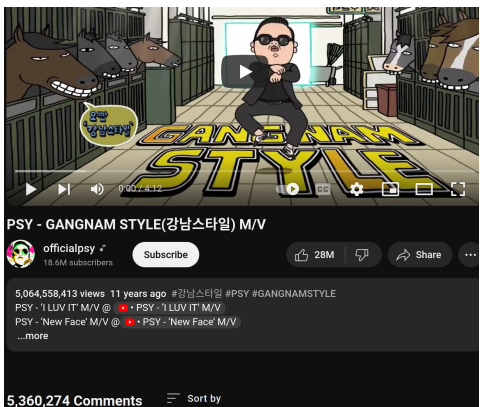


Justin Bieber - Sorry (PURPOSE : The Movement)

Justin Bieber 72.6M subscribers

3,772,923,806 views 8 years ago #JustinBieber #Vevo #Sorry
'Purpose' Available Everywhere Now!
iTunes: <http://smarturl.it/PurposeDix?IQid=VE...>
Stream & Add To Your Spotify Playlist: <http://smarturl.it/sPurpose?IQid=VEVO...> ...more

874,077 Comments Sort by



PSY - GANGNAM STYLE(강남스타일) M/V

officialpsy 18.6M subscribers

5,064,558,413 views 11 years ago #강남스타일 #PSY #GANGNAMSTYLE
PSY · '1 LUV IT' M/V @ · PSY · '1 LUV IT' M/V
PSY · 'New Face' M/V @ · PSY · 'New Face' M/V
...more

5,360,274 Comments Sort by

take-home messages:

- ▶ bibliometric indicators measure quantity, and there exist counterexamples to “quantity = quality”
- ▶ design fast algorithms for the most basic routines → MatMul
- ▶ design efficient reductions to them for other tasks → LinSys, Det, Inverse

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

polynomials: multiplication

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

$p \in \mathbb{K}[x]_{<8}$ \rightarrow 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 $d \approx 100$
- ▶ **FFT techniques** using (virtual) roots of unity

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

reductions strike back

```
318 long IsFFTPrime(long n, long& w)
319 +-- 74 lines: {-----
393
394
395 static
396 void NextFFTPrime(long& q, long& w, long index)
397 +-- 45 lines: {-----
442
443
444 long CalcMaxRoot(long p)
445 +-- 13 lines: {-----
458
459
460
461
462 +-- 5 lines: #ifndef NTL_WIZARD_HACK-----
467
468 void UseFFTPrime(long index)
469 +-- 36 lines: {-----
505
506
507 +-- 15 lines: #ifdef NTL_FFT_LAZYMUL -----
522
523
524
525 □
526 +--2687 lines: #ifdef NTL_FFT_LAZYMUL-----
3213
```

▶ small prime FFT in NTL:

↪ about **5500 lines** of C++

↪ target operation: FFT

(including 1200 lines for vectorized version
and 1100 for machine word arithmetic...)

reductions strike back

```
3250 void DivRem(zz_pX& q, zz_pX& r, const zz_pX& a,
3251 +--- 6 lines: {-----
3257
3258 void div(zz_pX& q, const zz_pX& a, const zz_pX&
3259 +--- 6 lines: {-----
3265
3266 void div(zz_pX& q, const zz_pX& a, zz_p b)
3267 +--- 5 lines: {-----
3272
3273
3274 void rem(zz_pX& r, const zz_pX& a, const zz_pX&
3275 +--- 6 lines: {-----
3281
3282 []
3283
3284 long operator==(const zz_pX& a, long b)
3285 +--- 20 lines: {-----
3305
3306 long operator==(const zz_pX& a, zz_p b)
3307 +--- 11 lines: {-----
3318
3319 void power(zz_pX& x, const zz_pX& a, long e)
3320 +--- 40 lines: {-----
3360
3361 void reverse(zz_pX& x, const zz_pX& a, long hi)
3362 +--- 13 lines: {-----
3375
3376 NTL_END_IMPL
```

▶ small prime FFT in NTL:

↪ about **5500 lines** of C++

↪ target operation: FFT

(including 1200 lines for vectorized version
and 1100 for machine word arithmetic...)

▶ polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$:

↪ about **5500 lines** as well

↪ target operations include:

- multiplication, truncated inversion, division,
- interpolation, multipoint evaluation,
- XGCD, Berlekamp-Massey, resultant,
- power projection, modular composition, ...

reductions strike back

```
3165 void FFTDiv(zz_pX& q, const zz_pX& a, const zz_pX& b)
3166 {
3167
3168     long n = deg(b);
3169     long m = deg(a);
3170     long k;
3171
3172 +---- 4 lines: if (m < n) {-----
3176
3177 +---- 6 lines: if (m >= 3*n) {-----
3183
3184     zz_pX P1, P2, P3;
3185
3186     CopyReverse(P3, b, 0, n);
3187     InvTrunc(P2, P3, m-n+1);
3188     CopyReverse(P1, P2, 0, m-n);
3189
3190     k = NextPowerOfTwo(2*(m-n)+1);
3191
3192     fftRep R1(INIT_SIZE, k), R2(INIT_SIZE, k);
3193
3194     TofftRep(R1, P1, k);
3195     TofftRep(R2, a, k, n, m);
3196     mul(R1, R1, R2);
3197     FromfftRep(q, R1, m-n, 2*(m-n));
3198 }
```

▶ small prime FFT in NTL:

↪ about **5500 lines** of C++

↪ target operation: FFT

(including 1200 lines for vectorized version and 1100 for machine word arithmetic...)

▶ polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$:

↪ about **5500 lines** as well

↪ target operations include:

- . multiplication, truncated inversion, division,
- . interpolation, multipoint evaluation,
- . XGCD, Berlekamp-Massey, resultant,
- . power projection, modular composition, ...

▶ reductions are often

- . concise and readable
- . close to the pseudocode

reductions strike back

```
3165 void FFTDiv(zz_pX& q, const zz_pX& a, const zz_pX& b)
3166 {
3167
3168     long n = deg(b);
3169     long m = deg(a);
3170     long k;
3171
3172 +---- 4 lines: if (m < n) {-----
3176
3177 +---- 6 lines: if (m >= 3*n) {-----
3183
3184     zz_pX P1, P2, P3;
3185
3186     CopyReverse(P3, b, 0, n);
3187     InvTrunc(P2, P3, m-n+1);
3188     CopyReverse(P1, P2, 0, m-n);
3189
3190     k = NextPowerOfTwo(2*(m-n)+1);
3191
3192     fftRep R1(INIT_SIZE, k), R2(INIT_SIZE, k);
3193
3194     TofftRep(R1, P1, k);
3195     TofftRep(R2, a, k, n, m);
3196     mul(R1, R1, R2);
3197     FromfftRep(q, R1, m-n, 2*(m-n));
3198 }
```

► $m \leftarrow \deg(A)$ and $n \leftarrow \deg(B)$

► if $m < n$, return $(0, A)$

► set reversals $\tilde{A} \leftarrow x^m A(1/x)$
and $\tilde{B} \leftarrow x^n B(1/x)$

► find $\tilde{Q} = \tilde{A}/\tilde{B} \pmod{x^{m-n+1}}$ by
power series inversion and product

► reverse \tilde{Q} to obtain Q

reductions strike back

concentrate efforts on: basic routines + good reductions

```
3165 void FFTDiv(zz_pX& q, const zz_pX& a, const zz_pX& b)
3166 {
3167
3168     long n = deg(b);
3169     long m = deg(a);
3170     long k;
3171
3172 +---- 4 lines: if (m < n) {-----
3176
3177 +---- 6 lines: if (m >= 3*n) {-----
3183
3184     zz_pX P1, P2, P3;
3185
3186     CopyReverse(P3, b, 0, n);
3187     InvTrunc(P2, P3, m-n+1);
3188     CopyReverse(P1, P2, 0, m-n);
3189
3190     k = NextPowerOfTwo(2*(m-n)+1);
3191
3192     fftRep R1(INIT_SIZE, k), R2(INIT_SIZE, k);
3193
3194     TofftRep(R1, P1, k);
3195     TofftRep(R2, a, k, n, m);
3196     mul(R1, R1, R2);
3197     FromfftRep(q, R1, m-n, 2*(m-n));
3198 }
```

► $m \leftarrow \deg(A)$ and $n \leftarrow \deg(B)$

► if $m < n$, return $(0, A)$

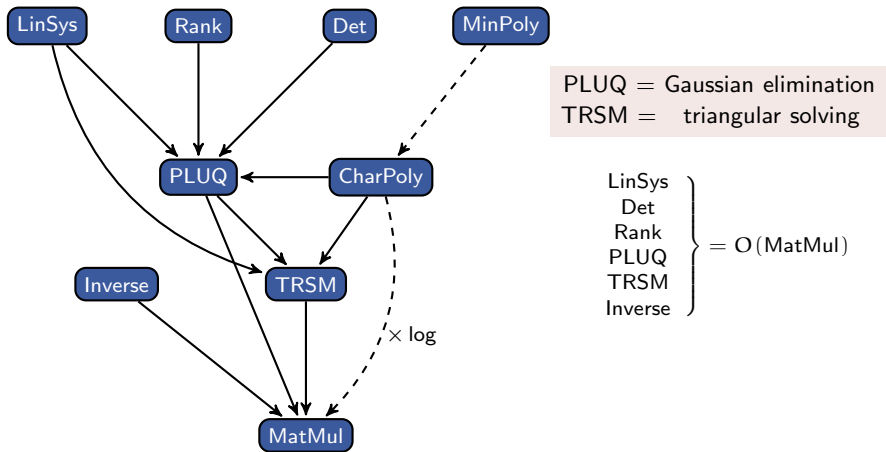
► set reversals $\tilde{A} \leftarrow x^m A(1/x)$
and $\tilde{B} \leftarrow x^n B(1/x)$

► find $\tilde{Q} = \tilde{A}/\tilde{B} \pmod{x^{m-n+1}}$ by
power series inversion and product

► reverse \tilde{Q} to obtain Q

matrices: main computational problems

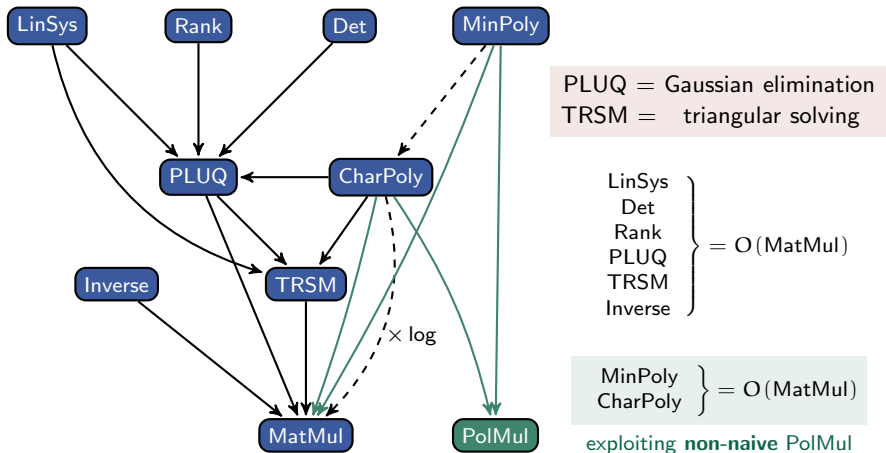
reductions of most problems to matrix multiplication



not closed:
open:

matrices: main computational problems

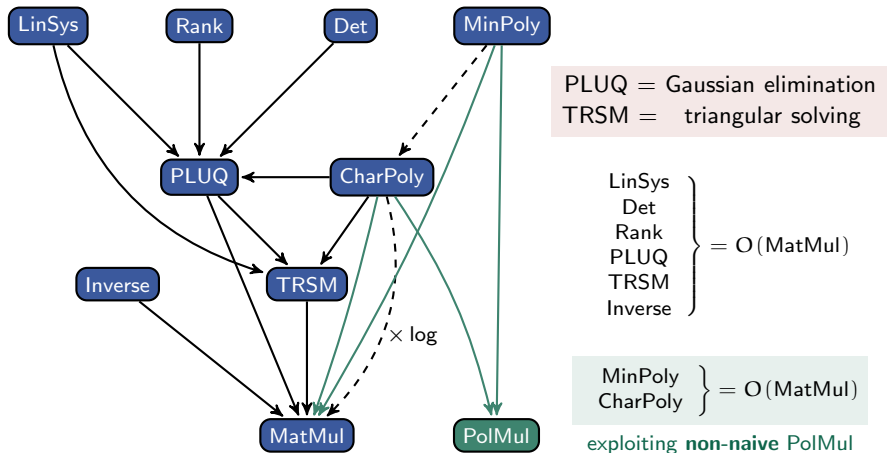
reductions of most problems to matrix multiplication



not closed:
open:

matrices: main computational problems

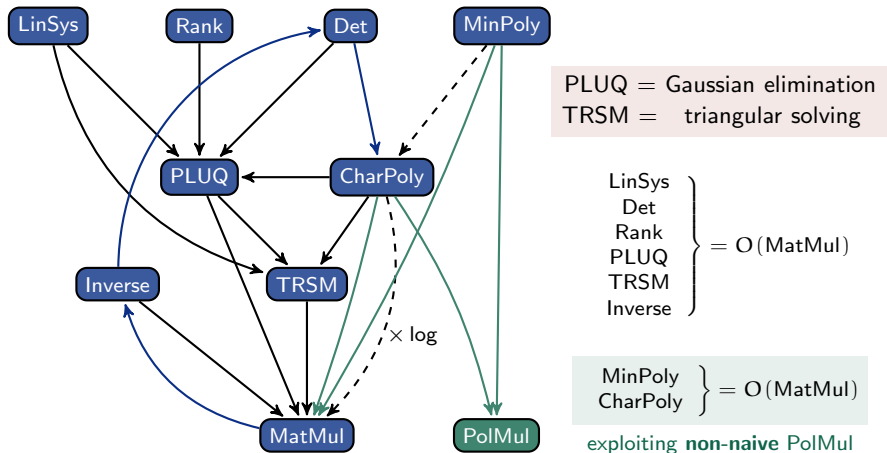
reductions of most problems to matrix multiplication



not closed: is Frobenius normal form in $O(\text{MatMul})$?
open:

matrices: main computational problems

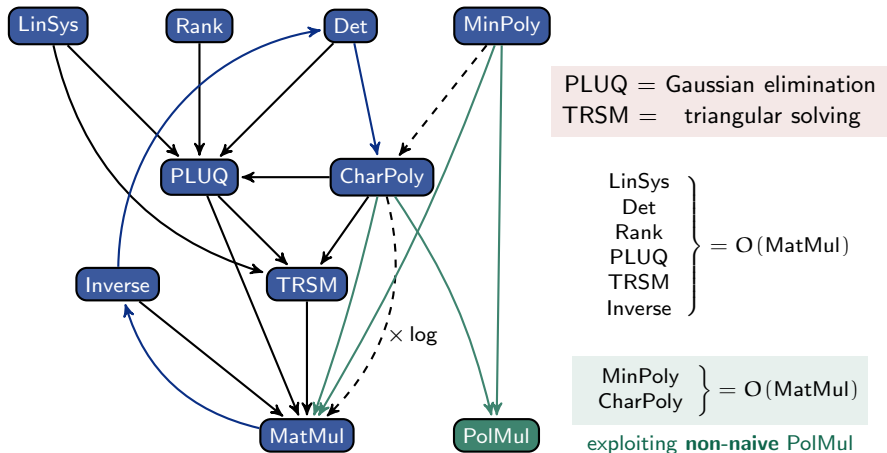
reductions of most problems to matrix multiplication



not closed: is Frobenius normal form in $O(\text{MatMul})$?
open:

matrices: main computational problems

reductions of most problems to matrix multiplication



not closed: is Frobenius normal form in $O(\text{MatMul})$?
open: is linear system solving as hard as multiplication?

bonus: some notes/references

[Jeannerod-Pernet-Storjohann 2013] doi.org/10.1016/j.jsc.2013.04.004

- ▶ explicit reductions between inversion & MatMul & Gaussian elimination / echelonization
- ▶ constants in the $O(\cdot)$ complexities when using classical matrix multiplication ($\omega = 3$) or Strassen's multiplication

“not closed”: it is open, but

- ▶ there is a randomized algorithm for Frobenius form computation which has complexity $O(\text{MatMul})$

[Pernet-Storjohann 2007] <http://www.cs.uwaterloo.ca/~astorjoh/cpoly.pdf>

- ▶ recent developments give new insight concerning core operations typically used in Frobenius form algorithms

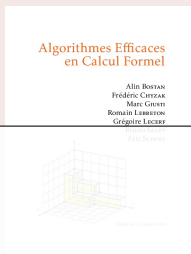
charpoly in $O(\text{MatMul})$: [Neiger-Pernet 2021] [doi.org/10.1016/S0885-064X\(22\)00005-X](https://doi.org/10.1016/S0885-064X(22)00005-X)
Krylov iterates in $O(\text{MatMul})$: [Neiger-Pernet-Villard 2024] hal.science/hal-04445355

polynomials: main computational problems

most problems have **quasi-linear complexity**

thanks to reductions to Po1Mu1 — did we mention the importance of good reductions?

- ▶ addition $f + g$, multiplication $f * g$
- ▶ **division** with remainder $f = qg + r$
- ▶ truncated **inverse** $f^{-1} \bmod x^d$
- ▶ extended **GCD** $fu + gv = \text{gcd}(f, g)$
- ▶ **multipoint eval.** $f \mapsto f(\alpha_1), \dots, f(\alpha_d)$
- ▶ **interpolation** $f(\alpha_1), \dots, f(\alpha_d) \mapsto f$
- ▶ Padé **approximation** $f = \frac{p}{q} \bmod x^d$
- ▶ minpoly of linearly **recurrent sequence**



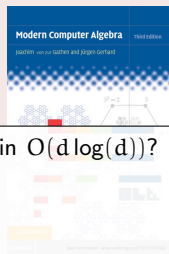
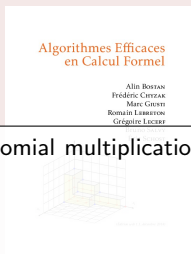
polynomials: main computational problems

most problems have **quasi-linear complexity**

thanks to reductions to Po1Mu1 — did we mention the importance of good reductions?

$O(M(d))$

- ▶ addition $f + g$, multiplication $f * g$
- ▶ **division** with remainder $f = qg + r$
- ▶ truncated **inverse** $f^{-1} \bmod x^d$
- ▶ extended **GCD** $fu + gv = \gcd(f, g)$
- ▶ **multipoint eval.** $f \mapsto f(\alpha_1), \dots, f(\alpha_d)$
- ▶ **interpolation** $f(\alpha_1), \dots, f(\alpha_d) \mapsto f$
- ▶ Padé **approximation** $f = \frac{p}{q} \bmod x^d$
- ▶ minpoly of linearly **recurrent sequence**



open: polynomial multiplication in $O(d \log(d))$?
open:
open:
open:

polynomials: main computational problems

most problems have **quasi-linear complexity**

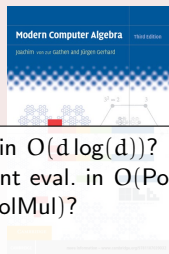
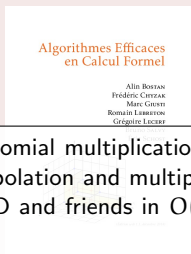
thanks to reductions to PolMul — did we mention the importance of good reductions?

$O(M(d))$

- ▶ 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.** $f \mapsto f(\alpha_1), \dots, f(\alpha_d)$
- ▶ **interpolation** $f(\alpha_1), \dots, f(\alpha_d) \mapsto f$
- ▶ Padé **approximation** $f = \frac{p}{q} \bmod x^d$
- ▶ minpoly of linearly **recurrent sequence**



- open:** polynomial multiplication in $O(d \log(d))$?
- open:** interpolation and multipoint eval. in $O(\text{PolMul})$?
- open:** XGCD and friends in $O(\text{PolMul})$?
- open:**

polynomials: main computational problems

most problems have **quasi-linear complexity**

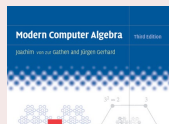
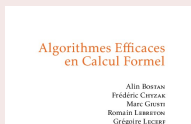
thanks to reductions to PolMul — did we mention the importance of good reductions?

$O(M(d))$

- ▶ 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.** $f \mapsto f(\alpha_1), \dots, f(\alpha_d)$
- ▶ **interpolation** $f(\alpha_1), \dots, f(\alpha_d) \mapsto f$
- ▶ Padé **approximation** $f = \frac{p}{q} \bmod x^d$
- ▶ minpoly of linearly **recurrent sequence**



- open:** polynomial multiplication in $O(d \log(d))$?
- open:** interpolation and multipoint eval. in $O(\text{PolMul})$?
- open:** XGCD and friends in $O(\text{PolMul})$?
- open:** modular composition $f(g) \bmod h$ closer to $O(\text{PolMul})$?

bonus: some notes/references

polynomial multiplication in $O(d \log(d))$?

- ▶ remains open over an arbitrary field, concerning algebraic complexity
- ▶ solved when the field possesses suitable roots of unity for FFT
- ▶ method of choice in practice (using several primes and CRT if needed) when working over prime finite fields $\mathbb{Z}/p\mathbb{Z}$
- ▶ recent progress in the bit complexity model
[Harvey-van der Hoeven 2019] <https://doi.org/10.1016/j.jco.2019.03.004>
[Harvey-van der Hoeven 2022] <https://doi.org/10.1145/3505584>

interpolation and multipoint evaluation in $O(\text{PolMul})$?

- ▶ remains open for an arbitrary set of points, with no assumption, but:
- ▶ by design, solved for FFT points, when available
- ▶ more generally, solved for points forming a geometric sequence
[Bostan-Schost 2005] <https://doi.org/10.1016/j.jco.2004.09.009>
- ▶ in many applications of interpolation/evaluation, one can choose the points, in which case $O(\text{PolMul})$ is feasible

open-source mathematics software system



Python/Cython

high-performance exact linear algebra

INPUT: **LinBox – fflas-ffpack** C/C++

high-performance polynomials (and more)

OUTPUT: **NTL & FLINT** C/C++

matrices

software

polynomials

```
sage: M.<M> = GF(7)[[
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[[ 0      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.

```
164 // order that remains to be dealt with
165 VecLong rem_order(order);
166
167 // indices of columns/orders that remain to be dealt with
168 VecLong rem_index(cdn);
169 std::iota(rem_index.begin(), rem_index.end(), 0);
170
171 // all along the algorithm, shift = shifted row degrees of approximant basis
172 // (initially, input shift = shifted row degree of the identity matrix)
173
174 while (not rem_order.empty())
175 {
176     /** Invariant:
177     * - appbas is a shift-ordered weak Popov approximant basis for
178     * (pmat, reached_order) where doneorder is the tuple such that
179     * -->reached_order[j] + rem_order[j] == order[j] for j appearing in
180     * -->reached_order[j] == order[j] for j not appearing in rem_index
181     * - shift == the "input shift"-row degree of appbas
182     * - residual == submatrix of columns (appbas * pmat)[:,:j] for all j
183     * in rem_order[j]
```

```
187     j = std::distance(rem_order.begin(), std::max_element(rem_order.begin(),
188 );
189     long deg = order[rem_index[j]] - rem_order[j];
190
191 // record the coefficients of degree deg of the column j of residual
192 // also keep track of which of these are nonzero,
193 // and among the nonzero ones, which is the first with smallest shift
194 Vec<zz_p> const_residual;
195 const_residual.SetLength(rdn);
196 VecLong indices_nonzero;
197 long piv = -1;
198 for (long i = 0; i < rdn; ++i)
199 {
200     const_residual[i] = coeff(residual[i][j], deg);
201     if (const_residual[i] != 0)
202     {
203         indices_nonzero.push_back(i);
204         if (piv < 0 || shift[i] < shift[piv])
205             piv = i;
206     }
207 }
208
209 // if indices_nonzero is empty, const_residual is already zero, there
210 if (not indices_nonzero.empty())
211 {
212     // update all rows of appbas and residual in indices_nonzero except
src/mat_lzz_pX_approximant.cpp
```

open-source mathematics software system



Python/Cython

high-performance exact linear algebra

INPUT: **LinBox – fflas-ffpack** C/C++

high-performance polynomials (and more)

OUTPUT: **NTL & FLINT** C/C++

matrices

software

polynomials

```
sage: M.<M> = 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^T * A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U^T * A
[ [ x 1 2*x]
sage: U^T * 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.

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

```
187         j = std::distance(rem_order.begin(), std::max_element(rem_order.begin(),
188 );
189
190     long deg = order[rem_index[j]] - rem_order[j];
191
192     // record the coefficients of degree deg of the column j of residual
193     // also keep track of which of these are nonzero,
194     // and among the nonzero ones, which is the first with smallest shift
195     Vec<zz_p> const_residual;
196     const_residual.SetLength(rdim);
197     VecLong indices_nonzero;
198     long piv = -1;
199     for (long i = 0; i < rdim; ++i)
200     {
201         const_residual[i] = coeff(residual[i][j], deg);
202         if (const_residual[i] != 0)
203         {
204             indices_nonzero.push_back(i);
205             if (piv < 0 || shift[i] < shift[piv])
206                 piv = i;
207         }
208     }
209
210     // if indices_nonzero is empty, const_residual is already zero, there
211     // is no need to update it
212     if (not indices_nonzero.empty())
213     {
214         // update all rows of appbas and residual in indices nonzero except
215         // row i
216         for (long i = 0; i < rdim; ++i)
217         {
218             if (i == piv) continue;
219             const zz_p x = const_residual[i];
220             const zz_p y = const_residual[piv];
221             const zz_p z = x / y;
222             for (long j = 0; j < rdim; ++j)
223             {
224                 const zz_p w = const_residual[j];
225                 const zz_p v = w - z * const_residual[piv];
226                 const_residual[j] = v;
227                 if (j == i) continue;
228                 const zz_p u = w - z * const_residual[piv];
229                 const zz_p t = u / v;
230                 for (long k = 0; k < rdim; ++k)
231                 {
232                     const zz_p s = const_residual[k];
233                     const zz_p r = s - t * const_residual[piv];
234                     const_residual[k] = r;
235                 }
236             }
237         }
238     }
239     // update the residual in indices_nonzero
240     for (long i = 0; i < rdim; ++i)
241     {
242         if (i == piv) continue;
243         const zz_p x = const_residual[i];
244         const zz_p y = const_residual[piv];
245         const zz_p z = x / y;
246         for (long j = 0; j < rdim; ++j)
247         {
248             const zz_p w = const_residual[j];
249             const zz_p v = w - z * const_residual[piv];
250             const_residual[j] = v;
251         }
252     }
253     // update the residual in indices_nonzero
254     for (long i = 0; i < rdim; ++i)
255     {
256         if (i == piv) continue;
257         const zz_p x = const_residual[i];
258         const zz_p y = const_residual[piv];
259         const zz_p z = x / y;
260         for (long j = 0; j < rdim; ++j)
261         {
262             const zz_p w = const_residual[j];
263             const zz_p v = w - z * const_residual[piv];
264             const_residual[j] = v;
265         }
266     }
267     // update the residual in indices_nonzero
268     for (long i = 0; i < rdim; ++i)
269     {
270         if (i == piv) continue;
271         const zz_p x = const_residual[i];
272         const zz_p y = const_residual[piv];
273         const zz_p z = x / y;
274         for (long j = 0; j < rdim; ++j)
275         {
276             const zz_p w = const_residual[j];
277             const zz_p v = w - z * const_residual[piv];
278             const_residual[j] = v;
279         }
280     }
281     // update the residual in indices_nonzero
282     for (long i = 0; i < rdim; ++i)
283     {
284         if (i == piv) continue;
285         const zz_p x = const_residual[i];
286         const zz_p y = const_residual[piv];
287         const zz_p z = x / y;
288         for (long j = 0; j < rdim; ++j)
289         {
290             const zz_p w = const_residual[j];
291             const zz_p v = w - z * const_residual[piv];
292             const_residual[j] = v;
293         }
294     }
295     // update the residual in indices_nonzero
296     for (long i = 0; i < rdim; ++i)
297     {
298         if (i == piv) continue;
299         const zz_p x = const_residual[i];
300         const zz_p y = const_residual[piv];
301         const zz_p z = x / y;
302         for (long j = 0; j < rdim; ++j)
303         {
304             const zz_p w = const_residual[j];
305             const zz_p v = w - z * const_residual[piv];
306             const_residual[j] = v;
307         }
308     }
309     // update the residual in indices_nonzero
310     for (long i = 0; i < rdim; ++i)
311     {
312         if (i == piv) continue;
313         const zz_p x = const_residual[i];
314         const zz_p y = const_residual[piv];
315         const zz_p z = x / y;
316         for (long j = 0; j < rdim; ++j)
317         {
318             const zz_p w = const_residual[j];
319             const zz_p v = w - z * const_residual[piv];
320             const_residual[j] = v;
321         }
322     }
323     // update the residual in indices_nonzero
324     for (long i = 0; i < rdim; ++i)
325     {
326         if (i == piv) continue;
327         const zz_p x = const_residual[i];
328         const zz_p y = const_residual[piv];
329         const zz_p z = x / y;
330         for (long j = 0; j < rdim; ++j)
331         {
332             const zz_p w = const_residual[j];
333             const zz_p v = w - z * const_residual[piv];
334             const_residual[j] = v;
335         }
336     }
337     // update the residual in indices_nonzero
338     for (long i = 0; i < rdim; ++i)
339     {
340         if (i == piv) continue;
341         const zz_p x = const_residual[i];
342         const zz_p y = const_residual[piv];
343         const zz_p z = x / y;
344         for (long j = 0; j < rdim; ++j)
345         {
346             const zz_p w = const_residual[j];
347             const zz_p v = w - z * const_residual[piv];
348             const_residual[j] = v;
349         }
350     }
351     // update the residual in indices_nonzero
352     for (long i = 0; i < rdim; ++i)
353     {
354         if (i == piv) continue;
355         const zz_p x = const_residual[i];
356         const zz_p y = const_residual[piv];
357         const zz_p z = x / y;
358         for (long j = 0; j < rdim; ++j)
359         {
360             const zz_p w = const_residual[j];
361             const zz_p v = w - z * const_residual[piv];
362             const_residual[j] = v;
363         }
364     }
365     // update the residual in indices_nonzero
366     for (long i = 0; i < rdim; ++i)
367     {
368         if (i == piv) continue;
369         const zz_p x = const_residual[i];
370         const zz_p y = const_residual[piv];
371         const zz_p z = x / y;
372         for (long j = 0; j < rdim; ++j)
373         {
374             const zz_p w = const_residual[j];
375             const zz_p v = w - z * const_residual[piv];
376             const_residual[j] = v;
377         }
378     }
379     // update the residual in indices_nonzero
380     for (long i = 0; i < rdim; ++i)
381     {
382         if (i == piv) continue;
383         const zz_p x = const_residual[i];
384         const zz_p y = const_residual[piv];
385         const zz_p z = x / y;
386         for (long j = 0; j < rdim; ++j)
387         {
388             const zz_p w = const_residual[j];
389             const zz_p v = w - z * const_residual[piv];
390             const_residual[j] = v;
391         }
392     }
393     // update the residual in indices_nonzero
394     for (long i = 0; i < rdim; ++i)
395     {
396         if (i == piv) continue;
397         const zz_p x = const_residual[i];
398         const zz_p y = const_residual[piv];
399         const zz_p z = x / y;
400         for (long j = 0; j < rdim; ++j)
401         {
402             const zz_p w = const_residual[j];
403             const zz_p v = w - z * const_residual[piv];
404             const_residual[j] = v;
405         }
406     }
407     // update the residual in indices_nonzero
408     for (long i = 0; i < rdim; ++i)
409     {
410         if (i == piv) continue;
411         const zz_p x = const_residual[i];
412         const zz_p y = const_residual[piv];
413         const zz_p z = x / y;
414         for (long j = 0; j < rdim; ++j)
415         {
416             const zz_p w = const_residual[j];
417             const zz_p v = w - z * const_residual[piv];
418             const_residual[j] = v;
419         }
420     }
421     // update the residual in indices_nonzero
422     for (long i = 0; i < rdim; ++i)
423     {
424         if (i == piv) continue;
425         const zz_p x = const_residual[i];
426         const zz_p y = const_residual[piv];
427         const zz_p z = x / y;
428         for (long j = 0; j < rdim; ++j)
429         {
430             const zz_p w = const_residual[j];
431             const zz_p v = w - z * const_residual[piv];
432             const_residual[j] = v;
433         }
434     }
435     // update the residual in indices_nonzero
436     for (long i = 0; i < rdim; ++i)
437     {
438         if (i == piv) continue;
439         const zz_p x = const_residual[i];
440         const zz_p y = const_residual[piv];
441         const zz_p z = x / y;
442         for (long j = 0; j < rdim; ++j)
443         {
444             const zz_p w = const_residual[j];
445             const zz_p v = w - z * const_residual[piv];
446             const_residual[j] = v;
447         }
448     }
449     // update the residual in indices_nonzero
450     for (long i = 0; i < rdim; ++i)
451     {
452         if (i == piv) continue;
453         const zz_p x = const_residual[i];
454         const zz_p y = const_residual[piv];
455         const zz_p z = x / y;
456         for (long j = 0; j < rdim; ++j)
457         {
458             const zz_p w = const_residual[j];
459             const zz_p v = w - z * const_residual[piv];
460             const_residual[j] = v;
461         }
462     }
463     // update the residual in indices_nonzero
464     for (long i = 0; i < rdim; ++i)
465     {
466         if (i == piv) continue;
467         const zz_p x = const_residual[i];
468         const zz_p y = const_residual[piv];
469         const zz_p z = x / y;
470         for (long j = 0; j < rdim; ++j)
471         {
472             const zz_p w = const_residual[j];
473             const zz_p v = w - z * const_residual[piv];
474             const_residual[j] = v;
475         }
476     }
477     // update the residual in indices_nonzero
478     for (long i = 0; i < rdim; ++i)
479     {
480         if (i == piv) continue;
481         const zz_p x = const_residual[i];
482         const zz_p y = const_residual[piv];
483         const zz_p z = x / y;
484         for (long j = 0; j < rdim; ++j)
485         {
486             const zz_p w = const_residual[j];
487             const zz_p v = w - z * const_residual[piv];
488             const_residual[j] = v;
489         }
490     }
491     // update the residual in indices_nonzero
492     for (long i = 0; i < rdim; ++i)
493     {
494         if (i == piv) continue;
495         const zz_p x = const_residual[i];
496         const zz_p y = const_residual[piv];
497         const zz_p z = x / y;
498         for (long j = 0; j < rdim; ++j)
499         {
500             const zz_p w = const_residual[j];
501             const zz_p v = w - z * const_residual[piv];
502             const_residual[j] = v;
503         }
504     }
505     // update the residual in indices_nonzero
506     for (long i = 0; i < rdim; ++i)
507     {
508         if (i == piv) continue;
509         const zz_p x = const_residual[i];
510         const zz_p y = const_residual[piv];
511         const zz_p z = x / y;
512         for (long j = 0; j < rdim; ++j)
513         {
514             const zz_p w = const_residual[j];
515             const zz_p v = w - z * const_residual[piv];
516             const_residual[j] = v;
517         }
518     }
519     // update the residual in indices_nonzero
520     for (long i = 0; i < rdim; ++i)
521     {
522         if (i == piv) continue;
523         const zz_p x = const_residual[i];
524         const zz_p y = const_residual[piv];
525         const zz_p z = x / y;
526         for (long j = 0; j < rdim; ++j)
527         {
528             const zz_p w = const_residual[j];
529             const zz_p v = w - z * const_residual[piv];
530             const_residual[j] = v;
531         }
532     }
533     // update the residual in indices_nonzero
534     for (long i = 0; i < rdim; ++i)
535     {
536         if (i == piv) continue;
537         const zz_p x = const_residual[i];
538         const zz_p y = const_residual[piv];
539         const zz_p z = x / y;
540         for (long j = 0; j < rdim; ++j)
541         {
542             const zz_p w = const_residual[j];
543             const zz_p v = w - z * const_residual[piv];
544             const_residual[j] = v;
545         }
546     }
547     // update the residual in indices_nonzero
548     for (long i = 0; i < rdim; ++i)
549     {
550         if (i == piv) continue;
551         const zz_p x = const_residual[i];
552         const zz_p y = const_residual[piv];
553         const zz_p z = x / y;
554         for (long j = 0; j < rdim; ++j)
555         {
556             const zz_p w = const_residual[j];
557             const zz_p v = w - z * const_residual[piv];
558             const_residual[j] = v;
559         }
560     }
561     // update the residual in indices_nonzero
562     for (long i = 0; i < rdim; ++i)
563     {
564         if (i == piv) continue;
565         const zz_p x = const_residual[i];
566         const zz_p y = const_residual[piv];
567         const zz_p z = x / y;
568         for (long j = 0; j < rdim; ++j)
569         {
570             const zz_p w = const_residual[j];
571             const zz_p v = w - z * const_residual[piv];
572             const_residual[j] = v;
573         }
574     }
575     // update the residual in indices_nonzero
576     for (long i = 0; i < rdim; ++i)
577     {
578         if (i == piv) continue;
579         const zz_p x = const_residual[i];
580         const zz_p y = const_residual[piv];
581         const zz_p z = x / y;
582         for (long j = 0; j < rdim; ++j)
583         {
584             const zz_p w = const_residual[j];
585             const zz_p v = w - z * const_residual[piv];
586             const_residual[j] = v;
587         }
588     }
589     // update the residual in indices_nonzero
590     for (long i = 0; i < rdim; ++i)
591     {
592         if (i == piv) continue;
593         const zz_p x = const_residual[i];
594         const zz_p y = const_residual[piv];
595         const zz_p z = x / y;
596         for (long j = 0; j < rdim; ++j)
597         {
598             const zz_p w = const_residual[j];
599             const zz_p v = w - z * const_residual[piv];
600             const_residual[j] = v;
601         }
602     }
603     // update the residual in indices_nonzero
604     for (long i = 0; i < rdim; ++i)
605     {
606         if (i == piv) continue;
607         const zz_p x = const_residual[i];
608         const zz_p y = const_residual[piv];
609         const zz_p z = x / y;
610         for (long j = 0; j < rdim; ++j)
611         {
612             const zz_p w = const_residual[j];
613             const zz_p v = w - z * const_residual[piv];
614             const_residual[j] = v;
615         }
616     }
617     // update the residual in indices_nonzero
618     for (long i = 0; i < rdim; ++i)
619     {
620         if (i == piv) continue;
621         const zz_p x = const_residual[i];
622         const zz_p y = const_residual[piv];
623         const zz_p z = x / y;
624         for (long j = 0; j < rdim; ++j)
625         {
626             const zz_p w = const_residual[j];
627             const zz_p v = w - z * const_residual[piv];
628             const_residual[j] = v;
629         }
630     }
631     // update the residual in indices_nonzero
632     for (long i = 0; i < rdim; ++i)
633     {
634         if (i == piv) continue;
635         const zz_p x = const_residual[i];
636         const zz_p y = const_residual[piv];
637         const zz_p z = x / y;
638         for (long j = 0; j < rdim; ++j)
639         {
640             const zz_p w = const_residual[j];
641             const zz_p v = w - z * const_residual[piv];
642             const_residual[j] = v;
643         }
644     }
645     // update the residual in indices_nonzero
646     for (long i = 0; i < rdim; ++i)
647     {
648         if (i == piv) continue;
649         const zz_p x = const_residual[i];
650         const zz_p y = const_residual[piv];
651         const zz_p z = x / y;
652         for (long j = 0; j < rdim; ++j)
653         {
654             const zz_p w = const_residual[j];
655             const zz_p v = w - z * const_residual[piv];
656             const_residual[j] = v;
657         }
658     }
659     // update the residual in indices_nonzero
660     for (long i = 0; i < rdim; ++i)
661     {
662         if (i == piv) continue;
663         const zz_p x = const_residual[i];
664         const zz_p y = const_residual[piv];
665         const zz_p z = x / y;
666         for (long j = 0; j < rdim; ++j)
667         {
668             const zz_p w = const_residual[j];
669             const zz_p v = w - z * const_residual[piv];
670             const_residual[j] = v;
671         }
672     }
673     // update the residual in indices_nonzero
674     for (long i = 0; i < rdim; ++i)
675     {
676         if (i == piv) continue;
677         const zz_p x = const_residual[i];
678         const zz_p y = const_residual[piv];
679         const zz_p z = x / y;
680         for (long j = 0; j < rdim; ++j)
681         {
682             const zz_p w = const_residual[j];
683             const zz_p v = w - z * const_residual[piv];
684             const_residual[j] = v;
685         }
686     }
687     // update the residual in indices_nonzero
688     for (long i = 0; i < rdim; ++i)
689     {
690         if (i == piv) continue;
691         const zz_p x = const_residual[i];
692         const zz_p y = const_residual[piv];
693         const zz_p z = x / y;
694         for (long j = 0; j < rdim; ++j)
695         {
696             const zz_p w = const_residual[j];
697             const zz_p v = w - z * const_residual[piv];
698             const_residual[j] = v;
699         }
700     }
701     // update the residual in indices_nonzero
702     for (long i = 0; i < rdim; ++i)
703     {
704         if (i == piv) continue;
705         const zz_p x = const_residual[i];
706         const zz_p y = const_residual[piv];
707         const zz_p z = x / y;
708         for (long j = 0; j < rdim; ++j)
709         {
710             const zz_p w = const_residual[j];
711             const zz_p v = w - z * const_residual[piv];
712             const_residual[j] = v;
713         }
714     }
715     // update the residual in indices_nonzero
716     for (long i = 0; i < rdim; ++i)
717     {
718         if (i == piv) continue;
719         const zz_p x = const_residual[i];
720         const zz_p y = const_residual[piv];
721         const zz_p z = x / y;
722         for (long j = 0; j < rdim; ++j)
723         {
724             const zz_p w = const_residual[j];
725             const zz_p v = w - z * const_residual[piv];
726             const_residual[j] = v;
727         }
728     }
729     // update the residual in indices_nonzero
730     for (long i = 0; i < rdim; ++i)
731     {
732         if (i == piv) continue;
733         const zz_p x = const_residual[i];
734         const zz_p y = const_residual[piv];
735         const zz_p z = x / y;
736         for (long j = 0; j < rdim; ++j)
737         {
738             const zz_p w = const_residual[j];
739             const zz_p v = w - z * const_residual[piv];
740             const_residual[j] = v;
741         }
742     }
743     // update the residual in indices_nonzero
744     for (long i = 0; i < rdim; ++i)
745     {
746         if (i == piv) continue;
747         const zz_p x = const_residual[i];
748         const zz_p y = const_residual[piv];
749         const zz_p z = x / y;
750         for (long j = 0; j < rdim; ++j)
751         {
752             const zz_p w = const_residual[j];
753             const zz_p v = w - z * const_residual[piv];
754             const_residual[j] = v;
755         }
756     }
757     // update the residual in indices_nonzero
758     for (long i = 0; i < rdim; ++i)
759     {
760         if (i == piv) continue;
761         const zz_p x = const_residual[i];
762         const zz_p y = const_residual[piv];
763         const zz_p z = x / y;
764         for (long j = 0; j < rdim; ++j)
765         {
766             const zz_p w = const_residual[j];
767             const zz_p v = w - z * const_residual[piv];
768             const_residual[j] = v;
769         }
770     }
771     // update the residual in indices_nonzero
772     for (long i = 0; i < rdim; ++i)
773     {
774         if (i == piv) continue;
775         const zz_p x = const_residual[i];
776         const zz_p y = const_residual[piv];
777         const zz_p z = x / y;
778         for (long j = 0; j < rdim; ++j)
779         {
780             const zz_p w = const_residual[j];
781             const zz_p v = w - z * const_residual[piv];
782             const_residual[j] = v;
783         }
784     }
785     // update the residual in indices_nonzero
786     for (long i = 0; i < rdim; ++i)
787     {
788         if (i == piv) continue;
789         const zz_p x = const_residual[i];
790         const zz_p y = const_residual[piv];
791         const zz_p z = x / y;
792         for (long j = 0; j < rdim; ++j)
793         {
794             const zz_p w = const_residual[j];
795             const zz_p v = w - z * const_residual[piv];
796             const_residual[j] = v;
797         }
798     }
799     // update the residual in indices_nonzero
800     for (long i = 0; i < rdim; ++i)
801     {
802         if (i == piv) continue;
803         const zz_p x = const_residual[i];
804         const zz_p y = const_residual[piv];
805         const zz_p z = x / y;
806         for (long j = 0; j < rdim; ++j)
807         {
808             const zz_p w = const_residual[j];
809             const zz_p v = w - z * const_residual[piv];
810             const_residual[j] = v;
811         }
812     }
813     // update the residual in indices_nonzero
814     for (long i = 0; i < rdim; ++i)
815     {
816         if (i == piv) continue;
817         const zz_p x = const_residual[i];
818         const zz_p y = const_residual[piv];
819         const zz_p z = x / y;
820         for (long j = 0; j < rdim; ++j)
821         {
822             const zz_p w = const_residual[j];
823             const zz_p v = w - z * const_residual[piv];
824             const_residual[j] = v;
825         }
826     }
827     // update the residual in indices_nonzero
828     for (long i = 0; i < rdim; ++i)
829     {
830         if (i == piv) continue;
831         const zz_p x = const_residual[i];
832         const zz_p y = const_residual[piv];
833         const zz_p z = x / y;
834         for (long j = 0; j < rdim; ++j)
835         {
836             const zz_p w = const_residual[j];
837             const zz_p v = w - z * const_residual[piv];
838             const_residual[j] = v;
839         }
840     }
841     // update the residual in indices_nonzero
842     for (long i = 0; i < rdim; ++i)
843     {
844         if (i == piv) continue;
845         const zz_p x = const_residual[i];
846         const zz_p y = const_residual[piv];
847         const zz_p z = x / y;
848         for (long j = 0; j < rdim; ++j)
849         {
850             const zz_p w = const_residual[j];
851             const zz_p v = w - z * const_residual[piv];
852             const_residual[j] = v;
853         }
854     }
855     // update the residual in indices_nonzero
856     for (long i = 0; i < rdim; ++i)
857     {
858         if (i == piv) continue;
859         const zz_p x = const_residual[i];
860         const zz_p y = const_residual[piv];
861         const zz_p z = x / y;
862         for (long j = 0; j < rdim; ++j)
863         {
864             const zz_p w = const_residual[j];
865             const zz_p v = w - z * const_residual[piv];
866             const_residual[j] = v;
867         }
868     }
869     // update the residual in indices_nonzero
870     for (long i = 0; i < rdim; ++i)
871     {
872         if (i == piv) continue;
873         const zz_p x = const_residual[i];
874         const zz_p y = const_residual[piv];
875         const zz_p z = x / y;
876         for (long j = 0; j < rdim; ++j)
877         {
878             const zz_p w = const_residual[j];
879             const zz_p v = w - z * const_residual[piv];
880             const_residual[j] = v;
881         }
882     }
883     // update the residual in indices_nonzero
884     for (long i = 0; i < rdim; ++i)
885     {
886         if (i == piv) continue;
887         const zz_p x = const_residual[i];
888         const zz_p y = const_residual[piv];
889         const zz_p z = x / y;
890         for (long j = 0; j < rdim; ++j)
891         {
892             const zz_p w = const_residual[j];
893             const zz_p v = w - z * const_residual[piv];
894             const_residual[j] = v;
895         }
896     }
897     // update the residual in indices_nonzero
898     for (long i = 0; i < rdim; ++i)
899     {
900         if (i == piv) continue;
901         const zz_p x = const_residual[i];
902         const zz_p y = const_residual[piv];
903         const zz_p z = x / y;
904         for (long j = 0; j < rdim; ++j)
905         {
906             const zz_p w = const_residual[j];
907             const zz_p v = w - z * const_residual[piv];
908             const_residual[j] = v;
909         }
910     }
911     // update the residual in indices_nonzero
912     for (long i = 0; i < rdim; ++i)
913     {
914         if (i == piv) continue;
915         const zz_p x = const_residual[i];
916         const zz_p y = const_residual[piv];
917         const zz_p z = x / y;
918         for (long j = 0; j < rdim; ++j)
919         {
920             const zz_p w = const_residual[j];
921             const zz_p v = w - z * const_residual[piv];
922             const_residual[j] = v;
923         }
924     }
925     // update the residual in indices_nonzero
926     for (long i = 0; i < rdim; ++i)
927     {
928         if (i == piv) continue;
929         const zz_p x = const_residual[i];
930         const zz_p y = const_residual[piv];
931         const zz_p z = x / y;
932         for (long j = 0; j < rdim; ++j)
933         {
934             const zz_p w = const_residual[j];
935             const zz_p v = w - z * const_residual[piv];
936             const_residual[j] = v;
937         }
938     }
939     // update the residual in indices_nonzero
940     for (long i = 0; i < rdim; ++i)
941     {
942         if (i == piv) continue;
943         const zz_p x = const_residual[i];
944         const zz_p y = const_residual[piv];
945         const zz_p z = x / y;
946         for (long j = 0; j < rdim; ++j)
947         {
948             const zz_p w = const_residual[j];
949             const zz_p v = w - z * const_residual[piv];
950             const_residual[j] = v;
951         }
952     }
953     // update the residual in indices_nonzero
954     for (long i = 0; i < rdim; ++i)
955     {
956         if (i == piv) continue;
957         const zz_p x = const_residual[i];
958         const zz_p y = const_residual[piv];
959         const zz_p z = x / y;
960         for (long j = 0; j < rdim; ++j)
961         {
962             const zz_p w = const_residual[j];
963             const zz_p v = w - z * const_residual[piv];
964             const_residual[j] = v;
965         }
966     }
967     // update the residual in indices_nonzero
968     for (long i = 0; i < rdim; ++i)
969     {
970         if (i == piv) continue;
971         const zz_p x = const_residual[i];
972         const zz_p y = const_residual[piv];
973         const zz_p z = x / y;
974         for (long j = 0; j < rdim; ++j)
975         {
976             const zz_p w = const_residual[j];
977             const zz_p v = w - z * const_residual[piv];
978             const_residual[j] = v;
979         }
980     }
981     // update the residual in indices_nonzero
982     for (long i = 0; i < rdim; ++i)
983     {
984         if (i == piv) continue;
985         const zz_p x = const_residual[i];
986         const zz_p y = const_residual[piv];
987         const zz_p z = x / y;
988         for (long j = 0; j < rdim; ++j)
989         {
990             const zz_p w = const_residual[j];
991             const zz_p v = w - z * const_residual[piv];
992             const_residual[j] = v;
993         }
994     }
995     // update the residual in indices_nonzero
996     for (long i = 0; i < rdim; ++i)
997     {
998         if (i == piv) continue;
999         const zz_p x = const_residual[i];
1000        const zz_p y = const_residual[piv];
1001        const zz_p z = x / y;
1002        for (long j = 0; j < rdim; ++j)
1003        {
1004            const zz_p w = const_residual[j];
1005            const zz_p v = w - z * const_residual[piv];
1006            const_residual[j] = v;
1007        }
1008    }
1009    // update the residual in indices_nonzero
1010    for (long i = 0; i < rdim; ++i)
1011    {
1012        if (i == piv) continue;
1013        const zz_p x = const_residual[i];
1014        const zz_p y = const_residual[piv];
1015        const zz_p z = x / y;
1016        for (long j = 0; j < rdim; ++j)
1017        {
1018            const zz_p w = const_residual[j];
1019            const zz_p v = w - z * const_residual[piv];
1020            const_residual[j] = v;
1021        }
1022    }
1023    // update the residual in indices_nonzero
1024    for (long i = 0; i < rdim; ++i)
1025    {
1026        if (i == piv) continue;
1027        const zz_p x = const_residual[i];
1028        const zz_p y = const_residual[piv];
1029        const zz_p z = x / y;
1030        for (long j = 0; j < rdim; ++j)
1031        {
1032            const zz_p w = const_residual[j];
1033            const zz_p v = w - z * const_residual[piv];
1034            const_residual[j] = v;
1035        }
1036    }
1037    // update the residual in indices_nonzero
1038    for (long i = 0; i < rdim; ++i)
1039    {
1040        if (i == piv) continue;
1041        const zz_p x = const_residual[i];
1042        const zz_p y = const_residual[piv];
1043        const zz_p z = x / y;
1044        for (long j = 0; j < rdim; ++j)
1045        {
1046            const zz_p w = const_residual[j];
```

open-source mathematics software system



SAGE

Python/Cython

high-performance exact linear algebra

INPUT: **LinBox – fflas-ffpack** C/C++

high-performance polynomials (and more)

OUTPUT: **NTL & FLINT** C/C++

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

open-source mathematics software system



Python/Cython

high-performance exact linear algebra

INPUT: **LinBox – fflas-ffpack** C/C++

high-performance polynomials (and more)

OUTPUT: **NTL & FLINT** C/C++

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

open-source mathematics software system



SAGE

Python/Cython

high-performance exact linear algebra

INPUT: **LinBox – fflas-ffpack** C/C++

high-performance polynomials (and more)

OUTPUT: **NTL & FLINT** 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

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

► repeated squaring: $O(m^\omega \log(k))$

output: A^k

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

- ▶ repeated squaring: $O(m^\omega \log(k))$
- ▶ using Frobenius form:
 $O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]
- ▶ improvement with polynomial matrices:
 $O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

can we reach $O(m^\omega)$?

▶ repeated squaring: $O(m^\omega \log(k))$

▶ using Frobenius form:

$O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]

▶ improvement with polynomial matrices:

$O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

can we reach $O(m^\omega)$?

► repeated squaring: $O(m^\omega \log(k))$

► using Frobenius form:

$O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]

► improvement with polynomial matrices:

$O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

Krylov iterates

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$

output: $v, Av, \dots, A^{m-1}v$

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

can we reach $O(m^\omega)$?

- ▶ repeated squaring: $O(m^\omega \log(k))$
- ▶ using Frobenius form:
 $O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]
- ▶ improvement with polynomial matrices:
 $O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

Krylov iterates

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$

output: $v, Av, \dots, A^{m-1}v$

- ▶ repeated matrix-vector products: $O(m^3)$
- ▶ via repeated squaring: $O(m^\omega \log(m))$
[Keller-Gehrig 1985]

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

can we reach $O(m^\omega)$?

- ▶ repeated squaring: $O(m^\omega \log(k))$
- ▶ using Frobenius form:
 $O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]
- ▶ improvement with polynomial matrices:
 $O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

Krylov iterates

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$

output: $v, Av, \dots, A^{m-1}v$

- ▶ repeated matrix-vector products: $O(m^3)$
- ▶ via repeated squaring: $O(m^\omega \log(m))$
[Keller-Gehrig 1985]
- ▶ with polynomial matrices: $O(m^\omega)$
[Zhou-Labahn-Storjohann 2012][Neiger-Pernet 2021]

constant matrices accelerated by polynomial matrices

matrix exponentiation

input: matrix $A \in \mathbb{K}^{m \times m}$,
integer $k > 0$

output: A^k

can we reach $O(m^\omega)$?

▶ repeated squaring: $O(m^\omega \log(k))$

▶ using Frobenius form:

$O(m^\omega \log(m) \log \log(m))$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Storjohann 2001]

▶ improvement with polynomial matrices:

$O(m^\omega \log \log(m)^2)$ if $\log(k) \in O(m)$
[Giesbrecht 1995] [Neiger-Pernet-Villard 2024]

Krylov iterates

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$

output: $v, Av, \dots, A^{m-1}v$

we do reach $O(m^\omega)$!

... can we do better?

▶ repeated matrix-vector products: $O(m^3)$

▶ via repeated squaring: $O(m^\omega \log(m))$
[Keller-Gehrig 1985]

▶ with polynomial matrices: $O(m^\omega)$
[Zhou-Labahn-Storjohann 2012][Neiger-Pernet 2021]


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

open-source mathematics software system



Python/Cython

The Hermitian form of a matrix, i.e., the pivot polynomials are monic.

INPUT:

goals: complete, robust, available

(more than 60k downloads per month)

OUTPUT:

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

```
int rdn;
```

high-performance exact linear algebra
LinBox – fflas-ffpack C/C++

```
while (not rem_order.empty())
```

```
{
```

goal: optimized basic operations

algorithms, vectorization, multithreading

```
* - shift == the "input shift"-row degree of appbas
```

```
* - residual == submatrix of columns [appbas * post][:i] for all i
```

software development for polynomial matrices

```
187 j = std::distance(rem_order.begin(), std::max_element(rem_order.b
);
188
189 long deg = order[rem_index[j]] - rem_order[j];
190
191 // record the coefficients of degree deg of the column j of residual
192 // also keep track of which of these are nonzero,
193 // and among the nonzero ones, which is the first with smallest shift
194 Vec<zz_p> const_residual;
195 const_residual.SetLength(rdn);
196 VecLong indices_nonzero;
197 long piv = -1;
198 for (long i = 0; i < rdn; ++i)
199 {
200     const_residual[i] = coeff(residual[i][j],deg);
201     if (const_residual[i] != 0)
202     {
203         indices_nonzero.push_back(i);
204         if (piv<0 || shift[i] < shift[piv])
205             piv = i;
206     }
207 }
208
209 // if indices_nonzero is empty, const_residual is already zero, there
210 if (not indices_nonzero.empty())
211 {
212     // update all rows of appbas and residual in indices nonzero exce
src/mat lzz pX approximant.cpp
```

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

open-source mathematics software system



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**
algorithms, vectorization, multithreading

software development for polynomial matrices

Polynomial Matrix Library C/C++

533 files, 72k lines of code, including 21k lines of comments

<https://github.com/vneiger/pml>

[Hyun-Neiger-Schost '19]

- ▶ most tools are 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 compute this determinant with PML?"

wide range of algorithms
efficiency = state of the art

system solving, determinant, kernel,
Hermite-Padé approximation,
high-order lifting, basis reduction...

```
is_hermite(row_wise=True, lower_echelon=False, include_zero_vectors=True)
Return a boolean indicating whether this matrix is in Hermite form.
```

```
Vec<long> ren_order(order);
// Indices of columns/orders that remain to be dealt with
Vec<long> ren_index(cdn);
// ...
while (not ren_order.empty())
// ...
j = std::distance(ren_order.begin(), std::max_element(ren_order
// ...
long phi = -1;
for (long i = 0; i < rdin; ++i)
// ...
// update all rows of appbas and residual in indices nonzero
src/mat_lzz_px_approxinant.cpp
```

outline

▶ **computer algebra**

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

▶ **polynomial matrices**

▶ **first algorithms**

▶ **exercises**

outline

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

exercises

polynomial matrices

basic definitions and properties

$\mathbb{K}[x]^{m \times n}$ = set of $m \times n$ matrices over $\mathbb{K}[x]$

called **polynomial matrices** in what follows

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

- ▶ structure: matrices over $\mathbb{K}[x]$ \longleftrightarrow free modules over $\mathbb{K}[x]$
similarly to: matrices over \mathbb{K} \longleftrightarrow vector spaces over \mathbb{K}
- ▶ basic operations: **addition and multiplication**
defined as usual (multiplication requires compatible dimensions)
- ▶ $\mathbb{K}[x]$ is not a field

polynomial matrices

basic definitions and properties

$\mathbb{K}[x]^{m \times n}$ = set of $m \times n$ matrices over $\mathbb{K}[x]$

called **polynomial matrices** in what follows

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

- ▶ structure: matrices over $\mathbb{K}[x] \longleftrightarrow$ free modules over $\mathbb{K}[x]$
similarly to: matrices over $\mathbb{K} \longleftrightarrow$ vector spaces over \mathbb{K}
- ▶ basic operations: **addition and multiplication**
defined as usual (multiplication requires compatible dimensions)
- ▶ $\mathbb{K}[x]$ is not a field

\rightsquigarrow algorithms may work in $\mathbb{K}(x)^{m \times n}$, but be careful with “**degree explosion**”!

polynomial matrices

examples you already know

large matrices with small degrees:

characteristic polynomial $\det(x\mathbf{I}_m - \mathbf{M}) \in \mathbb{K}[x]$ of a matrix $\mathbf{M} \in \mathbb{K}^{m \times m}$

\rightsquigarrow determinant of polynomial matrix $x\mathbf{I}_m - \mathbf{M} \in \mathbb{K}[x]^{m \times m}$

- ▶ fastest known algorithm uses this viewpoint [N.-Pernet, 2021]
- ▶ gradually transforms $x\mathbf{I}_m - \mathbf{M}$ to smaller matrices with larger degrees

polynomial matrices

examples you already know

large matrices with small degrees:

characteristic polynomial $\det(x\mathbf{I}_m - \mathbf{M}) \in \mathbb{K}[x]$ of a matrix $\mathbf{M} \in \mathbb{K}^{m \times m}$

\rightsquigarrow determinant of polynomial matrix $x\mathbf{I}_m - \mathbf{M} \in \mathbb{K}[x]^{m \times m}$

- ▶ fastest known algorithm uses this viewpoint [N.-Pernet, 2021]
- ▶ gradually transforms $x\mathbf{I}_m - \mathbf{M}$ to smaller matrices with larger degrees

small matrices with large degree:

extended GCD $uf + vg = \gcd(f, g)$ for polynomials $f, g \in \mathbb{K}[x]_{\leq d}$

\rightsquigarrow corresponds to a polynomial matrix transformation

$$\begin{bmatrix} u & v \\ \tilde{g} & \tilde{f} \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} \gcd(f, g) \\ 0 \end{bmatrix}$$

with the leftmost (polynomial) matrix of determinant in $\mathbb{K} \setminus \{0\}$

- ▶ fastest known “half-gcd” algorithms use this viewpoint [Knuth, 1970] [Schönhage, 1971] [Brent-Gustavson-Yun, 1980]

polynomial matrices

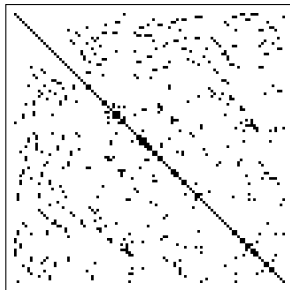
use in various situations

operations on sparse matrices

- ▶ solving sparse linear systems over \mathbb{K}
- ▶ computing the minimal polynomial / Frobenius form
- ▶ introducing parallelism in these computations

[Wiedemann 1986]
[Coppersmith 1993]
[Villard 1997]

example of sparse matrix in $\mathbb{K}^{m \times m}$
typical case: $O(m)$ nonzero entries



uses **polynomial matrix** generator
of linearly recurrent **matrix** sequence

polynomial matrices

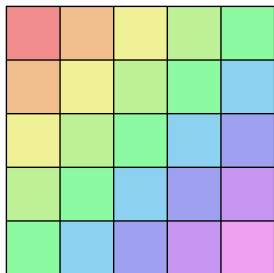
use in various situations

operations on structured matrices

- ▶ matrix-vector multiplication
- ▶ linear system solving
- ▶ nullspace computation

[Kailath-Kung-Morf 1979]
[Bostan et al. 2017]

example of Hankel matrix
↪ block-Hankel matrices
↪ Hankel-like matrices



uses **polynomial matrix** multiplication and
matrix-Padé approximation / **matrix-GCD**

polynomial matrices

use in various situations

bivariate interpolation and multipoint evaluation

problem: given points $(\alpha_1, \beta_1), \dots, (\alpha_n, \beta_n)$ in \mathbb{K}^2 ,

▶ given $p(x, y)$, compute $p(\alpha_i, \beta_i)$ for $1 \leq i \leq n$

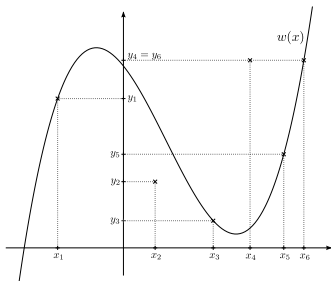
▶ find $p(x, y)$ of small degree such that $p(\alpha_i, \beta_i) = 0$

[Nüsken-Ziegler 2004]

[Beckermann 1992] [van Barel-Bultheel 1992]
[Marinari-Möller-Mora 1993]

bivariate interpolation = main step
in Reed-Solomon list-decoding
(univariate interpolation with errors)

[Guruswami-Sudan 1999] [Kötter-Vardy 2003]



uses **polynomial matrix** multiplication and
matrix rational reconstruction / **algebraic approximants**

polynomial matrices

seen as matrices over $\mathbb{K}(x)$

linear algebra viewpoint:

matrices in $\mathbb{K}[x]^{m \times n}$ are also in $\mathbb{K}(x)^{m \times n}$

(and $\mathbb{K}(x)$ is a field)

⇒ usual definition of **addition**, **multiplication**, **determinant**
these do not involve division anyway (... in algorithms?)

⇒ usual definition of **rank**
coincides with rank of free module

⇒ usual definition of **inverse**
with inverse over $\mathbb{K}(x)$

polynomial matrices

seen as matrices over $\mathbb{K}(x)$

linear algebra viewpoint:

matrices in $\mathbb{K}[x]^{m \times n}$ are also in $\mathbb{K}(x)^{m \times n}$

(and $\mathbb{K}(x)$ is a field)

\Rightarrow usual definition of **addition**, **multiplication**, **determinant**
these do not involve division anyway (... in algorithms?)

\Rightarrow usual definition of **rank**
coincides with rank of free module

\Rightarrow usual definition of **inverse**
with inverse over $\mathbb{K}(x)$

inverse is over $\mathbb{K}[x] \Leftrightarrow \det(\mathbf{A}) \in \mathbb{K} \setminus \{0\}$

def.: A is **unimodular**

polynomial matrices

seen as matrices over $\mathbb{K}(x)$

linear algebra viewpoint:

matrices in $\mathbb{K}[x]^{m \times n}$ are also in $\mathbb{K}(x)^{m \times n}$

(and $\mathbb{K}(x)$ is a field)

⇒ usual definition of **addition**, **multiplication**, **determinant**
these do not involve division anyway (... in algorithms?)

⇒ usual definition of **rank**
coincides with rank of free module

⇒ usual definition of **inverse**
with inverse over $\mathbb{K}(x)$

↔ algorithms may work in $\mathbb{K}(x)^{m \times n}$, but be careful with “**degree explosion**”!

exercise: Gaussian elimination is exponential-time

polynomial matrices

seen as matrices over $\mathbb{K}(x)$

linear algebra viewpoint:

matrices in $\mathbb{K}[x]^{m \times n}$ are also in $\mathbb{K}(x)^{m \times n}$

(and $\mathbb{K}(x)$ is a field)

- ▶ viewpoint **useful for definitions and properties**
- ▶ viewpoint **hardly usable for algorithms:**
ignores degree growth + too coarse cost bounds

- . cost of naive addition in $\mathbb{K}[x]^{m \times n}$ \longrightarrow $O(mn)$ additions in $\mathbb{K}(x)$
- . cost of naive multiplication in $\mathbb{K}[x]^{m \times m}$ \longrightarrow $O(m^3)$ ops in $\mathbb{K}(x)$

polynomial matrices

seen as matrices over $\mathbb{K}(x)$

linear algebra viewpoint:

matrices in $\mathbb{K}[x]^{m \times n}$ are also in $\mathbb{K}(x)^{m \times n}$

(and $\mathbb{K}(x)$ is a field)

- ▶ viewpoint **useful for definitions and properties**
- ▶ viewpoint **hardly usable for algorithms**:
ignores degree growth + too coarse cost bounds

- . cost of naive addition in $\mathbb{K}[x]^{m \times n}$ \longrightarrow $O(mn)$ additions in $\mathbb{K}(x)$
- . cost of naive multiplication in $\mathbb{K}[x]^{m \times m}$ \longrightarrow $O(m^3)$ ops in $\mathbb{K}(x)$

for algorithms&complexity, considering the degrees of entries is essential

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

$$\mathbf{A} = \begin{bmatrix} 3x + 4 & x^3 + 4x + 1 & 4x^2 + 3 \\ 5 & 5x^2 + 3x + 1 & 5x + 3 \\ 3x^3 + x^2 + 5x + 3 & 6x + 5 & 2x + 1 \end{bmatrix}$$
$$= \begin{bmatrix} 4 & 1 & 3 \\ 5 & 1 & 3 \\ 3 & 5 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ 5 & 6 & 2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 4 \\ 0 & 5 & 0 \\ 1 & 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} x^3$$

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

$$\mathbf{A} = \begin{bmatrix} 3x + 4 & x^3 + 4x + 1 & 4x^2 + 3 \\ 5 & 5x^2 + 3x + 1 & 5x + 3 \\ 3x^3 + x^2 + 5x + 3 & 6x + 5 & 2x + 1 \end{bmatrix}$$
$$= \begin{bmatrix} 4 & 1 & 3 \\ 5 & 1 & 3 \\ 3 & 5 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ 5 & 6 & 2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 4 \\ 0 & 5 & 0 \\ 1 & 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} x^3$$

A has degree 3; in general, $\deg(\mathbf{AB}) \leq \deg(\mathbf{A}) + \deg(\mathbf{B})$
e.g. $\deg(\mathbf{A}^2) = 6$, and $\deg(\mathbf{A}^3) = 8$, and $\deg(\mathbf{A}^4) = 11$

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

$$\mathbf{A} = \begin{bmatrix} 3x + 4 & x^3 + 4x + 1 & 4x^2 + 3 \\ 5 & 5x^2 + 3x + 1 & 5x + 3 \\ 3x^3 + x^2 + 5x + 3 & 6x + 5 & 2x + 1 \end{bmatrix}$$
$$= \begin{bmatrix} 4 & 1 & 3 \\ 5 & 1 & 3 \\ 3 & 5 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ 5 & 6 & 2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 4 \\ 0 & 5 & 0 \\ 1 & 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} x^3$$

degree growth enhances computational aspects

example: computing the N -th power \mathbf{A}^N

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[x]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[x]$

$$\mathbf{A} = \begin{bmatrix} 3x + 4 & x^3 + 4x + 1 & 4x^2 + 3 \\ 5 & 5x^2 + 3x + 1 & 5x + 3 \\ 3x^3 + x^2 + 5x + 3 & 6x + 5 & 2x + 1 \end{bmatrix}$$
$$= \begin{bmatrix} 4 & 1 & 3 \\ 5 & 1 & 3 \\ 3 & 5 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ 5 & 6 & 2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 4 \\ 0 & 5 & 0 \\ 1 & 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} x^3$$

degree growth enhances computational aspects

example: computing the N -th power \mathbf{A}^N

repeated squaring:

$$\left\{ \begin{array}{ll} \mathbf{A} \times \mathbf{A} & (\text{deg} = 3) \\ \mathbf{A}^2 \times \mathbf{A}^2 & (\text{deg} \leq 6) \\ \vdots & \vdots \\ \mathbf{A}^{\frac{N}{4}} \times \mathbf{A}^{\frac{N}{4}} & (\text{deg} \leq \frac{3N}{4}) \\ \mathbf{A}^{\frac{N}{2}} \times \mathbf{A}^{\frac{N}{2}} & (\text{deg} \leq \frac{3N}{2}) \end{array} \right.$$

find small recurrence + unroll it:

[Flajolet-Salvy 1997][Bostan-Neiger-Yurkevich 2023]

$O(N)$ operations in \mathbb{K}

- ▶ faster than multiplying $\mathbf{A}^{\frac{N}{2}} \times \mathbf{A}^{\frac{N}{2}}$
- ▶ does not require FFT
- ▶ prototype: $N = 2^{20} \rightsquigarrow 1.6\text{s vs. } 11.5\text{s}$

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

$$\mathbf{A} = \begin{bmatrix} 3x + 4 & x^3 + 4x + 1 & 4x^2 + 3 \\ 5 & 5x^2 + 3x + 1 & 5x + 3 \\ 3x^3 + x^2 + 5x + 3 & 6x + 5 & 2x + 1 \end{bmatrix}$$
$$= \begin{bmatrix} 4 & 1 & 3 \\ 5 & 1 & 3 \\ 3 & 5 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 4 & 0 \\ 0 & 3 & 5 \\ 5 & 6 & 2 \end{bmatrix} x + \begin{bmatrix} 0 & 0 & 4 \\ 0 & 5 & 0 \\ 1 & 0 & 0 \end{bmatrix} x^2 + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} x^3$$

- ▶ natural notion of **degree** of a polynomial matrix
- ▶ **addition** of $\mathbf{A}, \mathbf{B} \in \mathbb{K}[\chi]^{m \times n}$ is in $O(mnd)$ operations in \mathbb{K} where $d = \min(\deg(\mathbf{A}), \deg(\mathbf{B}))$
- ▶ some other polynomial operations available:
truncation $\mathbf{A} \bmod x^N$, **shift** $x^d \mathbf{A}$, **evaluation** $\mathbf{A}(\alpha)$ for $\alpha \in \mathbb{K}$

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

when $m = n$, $\mathbb{K}^{m \times m}$ is a (non-commutative) ring

derived from univariate polynomial algorithms:

- ▶ **multiplication** in $\mathbb{K}[\chi]^{m \times m}$ seen as a product of polynomials
complexity? $O(m^\omega M(d))$ is tempting... and true for best known $M(d)$
- ▶ **truncated inversion** via power series & Newton iteration
condition for invertibility? complexity?
- ▶ fast **Euclidean division with remainder**
conditions for feasibility? complexity?

polynomial matrices

seen as polynomials over $\mathbb{K}^{m \times n}$

polynomial viewpoint:

$\mathbb{K}[\chi]^{m \times n}$ is isomorphic to $\mathbb{K}^{m \times n}[\chi]$

algorithmically fruitful viewpoint, with some limitations

ignores heterogeneous degrees of matrix entries

consider $\mathbf{A} = \begin{bmatrix} f(\chi) & a_{01} & \cdots \\ a_{10} & a_{11} & \\ \vdots & & \ddots \end{bmatrix} \in \mathbb{K}[\chi]^{m \times m},$

$f(\chi)$ of degree d , other entries in \mathbb{K}

- ▶ data structure: $d + 1$ matrices in $\mathbb{K}^{m \times m}$
- ▶ size of representation: $m^2(d + 1)$ $\rightarrow m^2 + d?$
- ▶ adding two such matrices: $O(m^2(d + 1))$ $\rightarrow m^2 + d?$

outline

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

exercises

outline

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

- ▶ exploiting evaluation-interpolation
- ▶ extending algorithms for polynomials
- ▶ partial linearization techniques

exercises

first algorithms

fast multiplication

naive multiplication: $O(m^3 d^2)$ operations in \mathbb{K} $O(m^\omega M(d))?$

naive multiplication: $O(m^3 d^2)$ operations in \mathbb{K} $O(m^\omega M(d))?$

On fast multiplication of polynomials over arbitrary algebras

David G. Cantor¹ and Erich Kaltofen²★

¹ Department of Mathematics, University of California, Los Angeles, CA 90024-1555, USA

² Department of Computer Science, Rensselaer Polytechnic Institute, Troy,
NY 12180-3590, USA

Received January 22, 1988 / May 10, 1991

1 Introduction

In this paper we generalize the well-known Schönhage-Strassen algorithm for multiplying large integers to an algorithm for multiplying polynomials with coefficients from an arbitrary, not necessarily commutative, not necessarily associative, algebra \mathcal{A} . Our main result is an algorithm to multiply polynomials of degree $< n$ in $O(n \log n)$ algebra multiplications and $O(n \log n \log \log n)$ algebra additions/subtractions (we count a subtraction as an addition). The constant implied by the “ O ” does not depend upon the algebra \mathcal{A} . The parallel complexity of our algorithm, i.e., the depth of the corresponding arithmetic circuit, is

naive multiplication: $O(m^3 d^2)$ operations in \mathbb{K} $O(m^\omega M(d))$?

On fast multiplication of polynomials over arbitrary algebras

David G. Cantor¹ and Erich Kaltofen²★

¹ Department of Mathematics, University of California, Los Angeles, CA 90024-1555, USA

² Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA

Received January 22, 1988 / May 10, 1991

1 Introduction

In this paper we generalize the well-known Schönhage-Strassen algorithm for multiplying large integers to an algorithm for multiplying polynomials with coefficients from an arbitrary, not necessarily commutative, not necessarily associative, algebra \mathcal{A} . Our main result is an algorithm to multiply polynomials of degree $< n$ in $O(n \log n)$ algebra multiplications and $O(n \log n \log \log n)$ algebra additions/subtractions (we count a subtraction as an addition). The constant implied by the “ O ” does not depend upon the algebra \mathcal{A} . The parallel complexity of our algorithm, i.e., the depth of the corresponding arithmetic circuit, is

multiplication in $\mathbb{K}^{m \times m}[x]$ with degree $\leq d$:

- ▶ $O(d \log(d))$ multiplications in $\mathbb{K}^{m \times m}$
- ▶ $O(d \log(d) \log \log(d))$ additions in $\mathbb{K}^{m \times m}$

$$MM(m, d) \in O(m^\omega d \log(d) + m^2 d \log(d) \log \log(d))$$

first algorithms

exploiting evaluation-interpolation

exercise: multiplication, determinant, inversion

1. adapting the evaluation-interpolation paradigm to matrices in $\mathbb{K}[x]^{m \times m}$,

- ▶ give an explicit **multiplication** algorithm
- ▶ give a **determinant** algorithm
- ▶ give an **inversion** algorithm

computing the inverse over the fractions $\mathbb{K}(x)$

2. for each of these algorithms,

- ▶ give a required lower bound on the **cardinality of \mathbb{K}**
- ▶ state and prove an upper bound on the **complexity**

hint: use **known degree bounds** on the output

first algorithms

exploiting evaluation-interpolation

exercise: multiplication, determinant, inversion

1. adapting the evaluation-interpolation paradigm to matrices in $\mathbb{K}[x]^{m \times m}$,

▶ give an explicit **multiplication** algorithm

▶ give a **determinant** algorithm

▶ give an **inversion** algorithm

computing the inverse over the fractions $\mathbb{K}(x)$

2. for each of these algorithms,

▶ give a required lower bound on the **cardinality of \mathbb{K}**

▶ state and prove an upper bound on the **complexity**

multiplication: for large enough \mathbb{K} ,

$MM(m, d) \in O(m^\omega d + m^2 M(d))$ [Bostan-Schoot 2005]

\rightsquigarrow better than $m^\omega M(d)$

first algorithms

exploiting evaluation-interpolation

exercise: multiplication, determinant, inversion

1. adapting the evaluation-interpolation paradigm to matrices in $\mathbb{K}[\chi]^{m \times m}$,

▶ give an explicit **multiplication** algorithm

▶ give a **determinant** algorithm

▶ give an **inversion** algorithm

computing the inverse over the fractions $\mathbb{K}(\chi)$

2. for each of these algorithms,

▶ give a required lower bound on the **cardinality of \mathbb{K}**

▶ state and prove an upper bound on the **complexity**

	evaluation-interpolation, large \mathbb{K}	best known, unconditional
determinant	$O^{\sim}(m^{\omega+1}d)$	$O^{\sim}(m^{\omega}d)$
inversion	$O^{\sim}(m^{\omega+1}d)$	$O^{\sim}(m^3d)$
	reductions to PolMul&MatMul	reductions to PolMatMul

truncated inversion — from book “AECF”

62

3. Calculs rapides sur les séries

Entrée Un entier $N > 0$, $F \bmod X^N$ une série tronquée.

Sortie $F^{-1} \bmod X^N$.

Si $N = 1$, alors renvoyer f_0^{-1} , où $f_0 = F(0)$.

Sinon :

1. Calculer récursivement l'inverse G de $F \bmod X^{\lceil N/2 \rceil}$.
2. Renvoyer $G + (1 - GF)G \bmod X^N$.

Algorithme 3.2 – Inverse de série par itération de Newton.

Convergence quadratique pour l'inverse d'une série formelle

Lemme 3.2 Soient \mathbb{A} un anneau non nécessairement commutatif, $F \in \mathbb{A}[[X]]$ une série formelle de terme constant inversible et G une série telle que $G - F^{-1} = O(X^n)$ ($n \geq 1$). Alors la série

$$\mathcal{N}(G) = G + (1 - GF)G \quad (3.2)$$

vérifie $\mathcal{N}(G) - F^{-1} = O(X^{2n})$.

first algorithms

extending algorithms for polynomials

truncated inversion — results

consider a (square) polynomial matrix $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$

- ▶ \mathbf{A} is invertible as a power series
 \Leftrightarrow its constant term $\mathbf{A}(0) \in \mathbb{K}^{m \times m}$ is invertible
- ▶ if \mathbf{A} is invertible as a power series,
 computing $\mathbf{A}^{-1} \bmod x^N$ costs $O(\text{MM}(m, N))$ operations in \mathbb{K}

- ▶ no additional log: $\text{MM}(m, \frac{N}{2}) + \text{MM}(m, \frac{N}{4}) + \text{MM}(m, \frac{N}{8}) + \dots$
- ▶ excellent reduction to `PolMatMul`!
- ▶ timings with the Polynomial Matrix Library:

m	d	PolMatMul	TruncInv
10	20000	0.203	0.551
20	5000	0.225	0.639
40	2500	0.528	1.424
80	1250	1.227	3.653

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$,

compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

... are we not missing an assumption?

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$,

compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

... are we not missing an assumption?

rule 1: dividing by zero is generally a bad idea

rule 2: if you think you need to divide by zero, refer to rule 1

rule 3: neglecting to check that something is not zero does not make it nonzero

etc. etc.

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$,

compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

... are we not missing an assumption?

for a polynomial $p \in \mathcal{A}[x]$, over some ring \mathcal{A} , division by p is feasible

- ▶ if p is monic (leading coefficient $1_{\mathcal{A}}$)
- ▶ and more generally if the leading coefficient of p is invertible in \mathcal{A}

assumption: the leading coefficient of \mathbf{B} is invertible in $\mathbb{K}^{m \times m}$

recall $\mathbf{B} = \mathbf{B}_0 + \mathbf{B}_1x + \dots + \mathbf{B}_d x^d$ with $\mathbf{B}_i \in \mathbb{K}^{m \times m}$

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$ with $\text{lc}(\mathbf{B})$ invertible,
compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

► under this assumption, the usual fast Euclidean algorithm works

► recall:

1. reverse the equation,

2. compute quotient by truncated inverse multiplication

$$\tilde{\mathbf{Q}} = \tilde{\mathbf{B}}^{-1} \tilde{\mathbf{A}} \pmod{x^{d_{\mathbf{A}} - d_{\mathbf{B}} + 1}}$$

3. deduce remainder

► complexity is $O(\underbrace{\text{MM}(m, d_{\mathbf{A}} - d_{\mathbf{B}})}_{\text{find Q}} + \underbrace{\text{MM}(m, d_{\mathbf{B}})}_{\text{find R}})$

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$ with $\text{lc}(\mathbf{B})$ invertible,

compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

m	d_A	d_B	PolMatMul in deg d_B	TruncInv in deg d_B	QuoRem
10	40000	20000	0.203	0.551	1.873
20	10000	5000	0.225	0.639	2.164
40	5000	2500	0.528	1.424	6.468
80	2500	1250	1.227	3.653	15.59

first algorithms

extending algorithms for polynomials

division with remainder

problem:

given $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{m \times m}[x]$ with $\text{lc}(\mathbf{B})$ invertible,

compute $\mathbf{Q}, \mathbf{R} \in \mathbb{K}^{m \times m}[x]$ such that

$$\mathbf{A} = \mathbf{B}\mathbf{Q} + \mathbf{R} \quad \text{and} \quad \deg(\mathbf{R}) < \deg(\mathbf{B})$$

```
44 // step 1: reverse input matrices
45 row_reverse(Brev, B, rdegB);
46 row_reverse(buf, A, rdegA);
47
48 // step 2: compute quotient
49 // Qrev = Brev^{-1} R mod X^{d+1}
50 solve_series(Qrev, Brev, buf, d+1);
51 reverse(Q, Qrev, d);
52
53 // step 3: deduce remainder
54 // R = A - B*Q
55 multiply(buf, B, Q);
56 sub(R, A, buf);
```

► an efficient reduction, again

► rdegA vs. degree d ?

► `row_reverse` vs. `reverse` ?

► refinement of matrix degree:

row- or column-wise degrees

↪ improves applicability & complexity

e.g. division by $\mathbf{B} = \text{diag}(x^{d_1}, \dots, x^{d_m})$

first algorithms

refined degree measures — generalized division

row degree of a polynomial matrix

= the list of the maximum degree in each of its rows

for $\mathbf{A} = (a_{i,j}) \in \mathbb{K}[x]^{m \times n}$,

$$\begin{aligned} \text{rdeg}(\mathbf{A}) &= (\text{rdeg}(\mathbf{A}_{1,*}), \dots, \text{rdeg}(\mathbf{A}_{m,*})) \\ &= \left(\max_{1 \leq j \leq n} \deg(\mathbf{A}_{1,j}), \dots, \max_{1 \leq j \leq n} \deg(\mathbf{A}_{m,j}) \right) \in \mathbb{Z}^m \end{aligned}$$

first algorithms

refined degree measures — generalized division

row degree of a polynomial matrix

= the list of the maximum degree in each of its rows

column degree of a polynomial matrix

= the list of the maximum degree in each of its columns

first algorithms

refined degree measures — generalized division

row degree of a polynomial matrix

= the list of the maximum degree in each of its rows

column degree of a polynomial matrix

= the list of the maximum degree in each of its columns

sum of degrees of all entries $\leq \frac{n \times \text{sum of row degrees}}{m \times \text{sum of column degrees}} \leq mn \times \text{global degree}$

with notation:

$$\sum_{i,j} \deg(a_{ij}) \leq \frac{n|\text{rdeg}(\mathbf{A})|}{m|\text{cdeg}(\mathbf{A})|} \leq mn \deg(\mathbf{A})$$

first algorithms

refined degree measures — generalized division

row degree of a polynomial matrix

= the list of the maximum degree in each of its rows

column degree of a polynomial matrix

= the list of the maximum degree in each of its columns

sum of degrees of all entries $\leq \frac{n \times \text{sum of row degrees}}{m \times \text{sum of column degrees}} \leq mn \times \text{global degree}$

with notation:

$$\sum_{i,j} \deg(a_{ij}) \leq \frac{n|\text{rdeg}(\mathbf{A})|}{m|\text{cdeg}(\mathbf{A})|} \leq mn \deg(\mathbf{A})$$

consider \mathbf{A} with degree matrix

$$\begin{pmatrix} 100 & 5 & 20 & 1 \\ 100 & 5 & 20 & 1 \\ 100 & 5 & 20 & 1 \\ 100 & 5 & 20 & 1 \end{pmatrix}$$

determinant of \mathbf{A} : degree ≤ 126

\rightsquigarrow better than naive bound $4 \deg(\mathbf{A}) = 400$

first algorithms

refined degree measures — generalized division

row degree of a polynomial matrix

= the list of the maximum degree in each of its rows

column degree of a polynomial matrix

= the list of the maximum degree in each of its columns

$$\text{sum of degrees of all entries} \leq \frac{n \times \text{sum of row degrees}}{m \times \text{sum of column degrees}} \leq mn \times \text{global degree}$$

with notation:

$$\sum_{i,j} \deg(a_{ij}) \leq \frac{n|\text{rdeg}(\mathbf{A})|}{m|\text{cdeg}(\mathbf{A})|} \leq mn \deg(\mathbf{A})$$

more general division with remainder:

- ▶ take for $\text{lc}(\mathbf{B})$ **row-wise** leading coefficients
- ▶ if $\text{lc}(\mathbf{B})$ is invertible, division by \mathbf{B} is feasible
- ▶ with **row-wise** degree bounds on remainder

$$\begin{bmatrix} 4x^3 + 2x + 2 & 6x^3 + 2x^2 + 5 \\ 4x^2 + 2 & 3x^3 + x + 3 \end{bmatrix} = \begin{bmatrix} x & 0 \\ 0 & 2x^2 \end{bmatrix} \mathbf{Q} + \begin{bmatrix} 2 & 5 \\ 2 & x + 3 \end{bmatrix}$$

first algorithms

partial linearization techniques

reduce **unbalanced** degrees to some **average** degree

where degree means row degree, column degree, or related refined measures

[Storjohann 2006] [Zhou-Labahn 2012] [Jeannerod-Neiger-Villard 2020]

typical properties:

from a matrix $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$ with $D = |\text{rdeg}(\mathbf{A})| \ll m \deg(\mathbf{A})$
construct a matrix $\bar{\mathbf{A}} \in \mathbb{K}[x]^{m' \times m'}$ with

- ▶ a **slight increase** of matrix **dimension**: $m \leq m' \leq 2m$
- ▶ a **strong decrease** of matrix **degree**: $\deg(\bar{\mathbf{A}}) \leq 2 \frac{D}{m}$
- ▶ **preservation of the features** targeted by our computations

examples:

- ▶ product $\mathbf{A}\mathbf{B}$ easily deduced from product $\bar{\mathbf{A}}\bar{\mathbf{B}}$
- ▶ preservation of the determinant $\det(\mathbf{A}) = \det(\bar{\mathbf{A}})$
- ▶ inverse of $\bar{\mathbf{A}}$ contains inverse of \mathbf{A} as submatrix
- ▶ ...

first algorithms

partial linearization techniques

reduce **unbalanced** degrees to some **average** degree

basic illustration:

- ▶ let $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$ of degree $< d$,
- ▶ let $\mathbf{u} \in \mathbb{K}[x]^{m \times 1}$ of degree $< md$,

then the matrix-vector product $\mathbf{A}\mathbf{u}$ can be computed in
 $MM(m, d) + O(m^2d)$ operations in \mathbb{K}

what would be the cost of the “naive” multiplication? $\rightsquigarrow O(m^2M(md))$

algorithm:

[Lecerf 2001 (in communication + software)]

first algorithms

partial linearization techniques

reduce **unbalanced** degrees to some **average** degree

basic illustration:

▶ let $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$ of degree $< d$,

▶ let $\mathbf{u} \in \mathbb{K}[x]^{m \times 1}$ of degree $< md$,

then the matrix-vector product $\mathbf{A}\mathbf{u}$ can be computed in

$\text{MM}(m, d) + O(m^2d)$ operations in \mathbb{K}

what would be the cost of the “naive” multiplication? $\rightsquigarrow O(m^2M(md))$

algorithm:

[Lecerf 2001 (in communication + software)]

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{A} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{U}} \end{bmatrix} \begin{bmatrix} 1 \\ x^d \\ x^{2d} \\ \vdots \end{bmatrix}$$

where the columns of $\bar{\mathbf{U}} \in \mathbb{K}[x]^{m \times m}$ form the x^d -adic expansion of \mathbf{u}

\Rightarrow here $\deg(\bar{\mathbf{U}}) < d$

first algorithms

partial linearization techniques

reduce **unbalanced** degrees to some **average** degree

basic illustration:

- ▶ let $\mathbf{A} \in \mathbb{K}[x]^{m \times m}$ of degree $< d$,
- ▶ let $\mathbf{u} \in \mathbb{K}[x]^{m \times 1}$ of degree $< md$,

then the matrix-vector product $\mathbf{A}\mathbf{u}$ can be computed in
 $MM(m, d) + O(m^2d)$ operations in \mathbb{K}

what would be the cost of the “naive” multiplication? $\rightsquigarrow O(m^2M(md))$

algorithm:

[Lecerf 2001 (in communication + software)]

m	d	m d	via PolMatMul	matrix-vector
10	20000	200000	0.203	0.368
20	5000	100000	0.225	0.683
40	2500	100000	0.528	2.481
80	1250	100000	1.227	9.592

outline

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

- ▶ exploiting evaluation-interpolation
- ▶ extending algorithms for polynomials
- ▶ partial linearization techniques

exercises

outline

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

- ▶ exploiting evaluation-interpolation
- ▶ extending algorithms for polynomials
- ▶ partial linearization techniques

exercises

- ▶ evaluation-interpolation-based algorithms
- ▶ Krylov iterates via repeated squaring
- ▶ Krylov iterates in MatMul time

exercises

evaluation-interpolation-based algorithms

exercise: multiplication, determinant, inversion

1. adapting the evaluation-interpolation paradigm to matrices in $\mathbb{K}[x]^{m \times m}$,

- ▶ give an explicit **multiplication** algorithm
- ▶ give a **determinant** algorithm
- ▶ give an **inversion** algorithm

computing the inverse over the fractions $\mathbb{K}(x)$

2. for each of these algorithms,

- ▶ give a required lower bound on the **cardinality of \mathbb{K}**
- ▶ state and prove an upper bound on the **complexity**

hint: use **known degree bounds** on the output

exercises

evaluation-interpolation: multiplication

given \mathbf{A} and \mathbf{B} in $\mathbb{K}[\chi]^{m \times m}$ of degree $\leq d$,
we know that $\mathbf{C} = \mathbf{A}\mathbf{B}$ has degree at most $2d$, so:

1. **pick points:** pairwise distinct $\alpha_1, \dots, \alpha_{2d+1} \in \mathbb{K}$ $\text{Card}(\mathbb{K}) \geq 2d + 1$
2. **evaluate:** $\mathbf{A}(\alpha_i)$ and $\mathbf{B}(\alpha_i)$, for $i = 1, \dots, 2d + 1$ $O(m^2 M(d) \log(d))$
3. **multiply:** $\mathbf{A}(\alpha_i)\mathbf{B}(\alpha_i)$, for $i = 1, \dots, 2d + 1$ $O(m^\omega d)$
4. **interpolate:** find \mathbf{C} in $\mathbb{K}[\chi]^{m \times m}$ of degree $\leq 2d$ such that $\mathbf{C}(\alpha_i) = \mathbf{A}(\alpha_i)\mathbf{B}(\alpha_i)$, for $i = 1, \dots, 2d + 1$ $O(m^2 M(d) \log(d))$
5. return \mathbf{C}

excellent algorithm:

- . linear in d in the term $m^\omega d$ (recall Cantor-Kaltofen: $m^\omega d \log(d)$)
- . exponent ω of matrix multiplication
- . the $m^2 M(d) \log(d)$ term can be improved via points in geometric sequence
- . downside: restriction on \mathbb{K} (large degrees + small finite fields does happen)

exercises

evaluation-interpolation: determinant

given \mathbf{A} in $\mathbb{K}[x]^{m \times m}$ of degree $\leq d$,
we know that $\Delta = \det(\mathbf{A})$ has degree at most md , so:

1. **pick points:** pairwise distinct $\alpha_1, \dots, \alpha_{md+1} \in \mathbb{K}$
2. **evaluate:** $\mathbf{A}(\alpha_i)$ for $i = 1, \dots, md + 1$
3. **determinant:** $\beta_i = \det(\mathbf{A}(\alpha_i))$, for $i = 1, \dots, md + 1$
4. **interpolate:** find Δ in $\mathbb{K}[x]$ of degree $\leq md$ such that $\Delta(\alpha_i) = \beta_i$, for $i = 1, \dots, md + 1$
5. return Δ

$$\text{Card}(\mathbb{K}) \geq md + 1$$

$$O(m^3 M(d) \log(d))$$

$$O(m^{\omega+1} d)$$

$$O(M(md) \log(md))$$

- . quasi-linear in degree d : fast for large d , small m
- . exponent > 3 on matrix dimension m : slow for large m
- . best known today: $O^{\sim}(m^{\omega} d)$

exercises

evaluation-interpolation: inversion

given \mathbf{A} in $\mathbb{K}[x]^{m \times m}$ of degree $\leq d$,
we know that $\mathbf{C} = \mathbf{A}^{-1} = \frac{1}{\Delta} \mathbf{U}$ with
 $\deg(\Delta) \leq md$ and $\deg(\mathbf{U}) \leq (m-1)d$, so:

0. set $n = (2m-1)d + 1$ $n = \Theta(md)$
1. **pick points:** pairwise distinct $\alpha_1, \dots, \alpha_n \in \mathbb{K}$ $\text{Card}(\mathbb{K}) \geq (2m-1)d + 1$
2. **evaluate:** $\mathbf{A}(\alpha_i)$, for $i = 1, \dots, n$ $O(m^3 M(d) \log(d))$
3. **invert:** $\mathbf{A}(\alpha_i)^{-1}$, for $i = 1, \dots, n$ $O(m^{\omega+1} d)$
4. **interpolate:** using Cauchy interpolation find \mathbf{C} in $\mathbb{K}(X)^{m \times m}$ with all numerators of degree $\leq (m-1)d$ and all denominators of degree $\leq md$ such that $\mathbf{C}(\alpha_i) = \mathbf{A}(\alpha_i)^{-1}$, for $i = 1, \dots, n$ $O(m^2 M(md) \log(md))$
5. return \mathbf{C}

- . quasi-linear in degree d : fast for large d , small m
- . exponent > 3 on dimension m but recall size of \mathbf{A}^{-1} is typically $\Theta(m^3 d)$
- . best known today: $O(\tilde{m}^3 d)$, and even $O(\tilde{m}^\omega d)$ for factorized form
- . note: one could compute $\det(\mathbf{A})$ to avoid Cauchy interpolation

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

1. give an algorithm which costs $O(m^\omega \log(d) + m^{\omega-1}d)$ operations in \mathbb{K} , based on repeated squaring
2. prove that the generating series of $(A^k v)_{k \geq 0}$ rewrites as a fraction of polynomial matrices:

$$\sum_{k \geq 0} A^k v x^k = (I - xA)^{-1}v$$

3. using the kernel black box, give a complexity bound for finding $\lambda \in \mathbb{K}[x]$ and $u \in \mathbb{K}[x]^{m \times 1}$, both of degree $\leq m$, such that

$$\sum_{k \geq 0} A^k v x^k = u/\lambda$$

4. show that $(A^k v)_{0 \leq k < d}$ can be computed in $O(m^\omega + mM(d))$

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

1. give an algorithm which costs $O(m^\omega \log(d) + m^{\omega-1}d)$ operations in \mathbb{K} , based on repeated squaring

for simplicity, take d a power of 2

first compute $A^2, A^4, \dots, A^{d/2}$, cost $O(m^\omega \log(d))$

from v , compute Av

from $[v \ Av]$, compute $A^2[v \ Av] = [A^2v \ A^3v]$

from $[v \ Av \ A^2v \ A^3v]$, compute $A^4[v \ Av \ A^2v \ A^3v] = [A^4v \ A^5v \ A^6v \ A^7v]$

etc. . .

from $[A^k v]_{0 \leq k < d/2}$, compute $A^{d/2}[A^k v]_{0 \leq k < d/2} = [A^k v]_{d/2 \leq k < d}$

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

1. give an algorithm which costs $O(m^\omega \log(d) + m^{\omega-1}d)$ operations in \mathbb{K} , based on repeated squaring

the first $\min(\log(d), \log(m))$ products involve matrices of dimensions m or less, hence a total cost bounded by $O(m^\omega \log(d))$

the remaining products (if any) involve a lefthand operand of dimensions $m \times m$ and a righthand one of dimensions $m \times 2^k$, where k goes from about $\log_2(m)$ to for $\log_2(d)$
 \rightsquigarrow for a given k , the product costs $O(m^{\omega-1}2^k)$

\rightsquigarrow summing this over all k , with $\sum_{k \leq \log_2(d)} 2^k \in O(d)$, gives $O(m^{\omega-1}d)$

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

2. prove that the generating series of $(A^k v)_{k \geq 0}$ rewrites as a fraction of polynomial matrices:

$$\sum_{k \geq 0} A^k v x^k = (I - xA)^{-1} v$$

multiply the left-hand side by $I - xA$, this yields v

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

3. using the kernel black box, give a complexity bound for finding $\lambda \in \mathbb{K}[x]$ and $u \in \mathbb{K}[x]^{m \times 1}$, both of degree $\leq m$, such that

$$\sum_{k \geq 0} A^k v x^k = u / \lambda$$

- . consider $F = [I - xA \quad -v]$; this matrix has degree ≤ 1 and rank m (its leftmost $m \times m$ submatrix is nonsingular)
- . so, in $O(m^\omega)$, we can compute a nonzero element of degree $\leq m$ in its right kernel
- . this element can be written $\begin{bmatrix} u \\ \lambda \end{bmatrix}$, and $F \begin{bmatrix} u \\ \lambda \end{bmatrix} = 0$ rewrites as $(I - xA)u = v\lambda$
- . observe that λ cannot be zero (otherwise, u would be a nonzero vector in the right kernel of $I - xA$, which is not possible)
- . thus $(I - xA)^{-1}v = \frac{1}{\lambda}u$

exercises

problem (Krylov iterates):

input: matrix $A \in \mathbb{K}^{m \times m}$,
vector $v \in \mathbb{K}^{m \times 1}$
integer $d > 0$

output: $v, Av, \dots, A^{d-1}v$

kernel black box:

given a matrix $F \in \mathbb{K}[x]^{m \times (m+1)}$ of rank m and degree ≤ 1 , one can compute a nonzero element of degree $\leq m$ in the right kernel of F using $O(m^\omega)$ operations in \mathbb{K}

[refined analysis of Algo.1 in Zhou-Labahn-Storjohann 2012]

4. show that $(A^k v)_{0 \leq k < d}$ can be computed in $O(m^\omega + mM(d))$

- . these d vectors are the first d terms of the series $\sum_{k \geq 0} A^k v x^k$
- . we have seen that this series is equal to $\frac{1}{\lambda} \mathbf{u}$ (with \mathbf{u} and λ found in $O(m^\omega)$)
 \rightsquigarrow it suffices to expand \mathbf{u}/λ as a power series in precision d
- . since \mathbf{u} is a vector of m entries, this costs $O(mM(d))$

summary

computer algebra

- ▶ efficient algorithms and software
- ▶ for matrices over a field
- ▶ for univariate polynomials

polynomial matrices

- ▶ basic definitions and properties
- ▶ use in various situations
- ▶ seen as matrices / seen as polynomials

first algorithms

- ▶ exploiting evaluation-interpolation
- ▶ extending algorithms for polynomials
- ▶ partial linearization techniques

exercises

- ▶ evaluation-interpolation-based algorithms
- ▶ Krylov iterates via repeated squaring
- ▶ Krylov iterates in MatMul time