Storage of multivariate polynomials

M. Gastineau

IMCCE - Observatoire de Paris - CNRS UMR8028
77, avenue Denfert Rochereau
75014 PARIS
FRANCE

gastineau@imcce.fr

A small example with general computer algebra system

How fast can we multiply 2 polynomials ?

$$p \times (p+1) \text{ with } p = (1+x+y+z+t)^{16}$$

Execution on the same hardware and operating system

C.A.S.	*time (s)*
GiNaC 1.3.2	538.00
Maple 10	135.00
Maxima 5.9.2	73.40
Pari/GP 2.3.0	5.22
Singular 3.0.2	23.78

- Differences of these C.A.S.
  - algorithms
  - storage and memory management

General Computer Algebra Systems

- Store exponents as multiple precision integer

$$P(x) = 1 + x^{2^{128}}$$

➪ Most computations need small integer exponents

$$< 2^8 or < 2^{16} or < 2^{32}$$

➪ SAM uses hardware integers

- Only one representation to handle all polynomials

  - large generic objects

➪ SAM specializes the representation of the polynomials with their attribute

Contents

- Storage
  - Univariate polynomials
  - Multivariate polynomials

    $$P(x_1, x_2, ..., x_n) = \sum C x_1^{d_1} x_2^{d_2} ... x_n^{d_n}$$

  - Poisson series

    $$P(x_1, x_2, ..., x_n, \lambda_1, ..., \lambda_m) = \sum C x_1^{d_1} x_2^{d_2} ... x_n^{d_n} \begin{Bmatrix} \cos \\ \sin \end{Bmatrix} (k_1\lambda_1 + ... + k_m\lambda_m)$$

  - Coefficients

Univariate polynomials

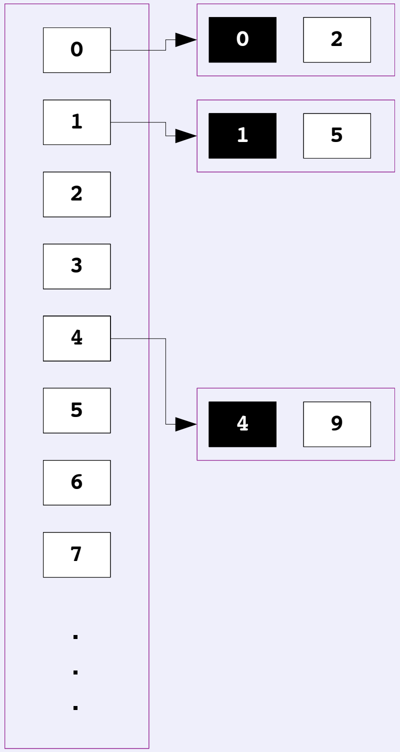
- K commutative ring
- polynomials $P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_D x^D$

  such that $D \in \mathbb{N}$
  $\forall j, a_j \in K$
  $\forall i > D, a_i = 0$

- assume that *P(x)* has *R* non-zero terms
-

Available representations for univariate polynomials

$$P(X) = 2 + 5X + 9X^4$$



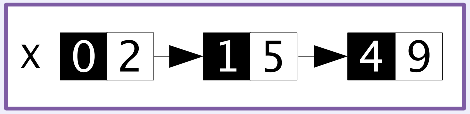
- Unordered structure : Hashmap
  - Fast insertion *O(1)*
  - Hashing function : key based on exponents
    - how to choose this function ?
  - Collision resolution
    - chaining : linked-list
    - open addressing : require to search an empty slot
  - Resize the hash table on large polynomial ?

Available representations for univariate polynomials

Sparse ordered structure

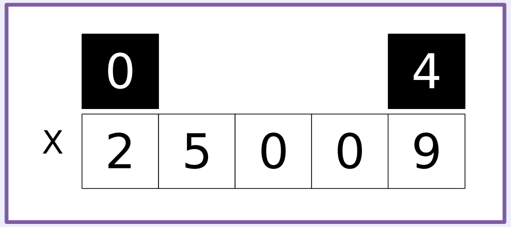
- List of monomials
  - search/insertion *O(R)*
  - efficient for very sparse polynomial

$$P(X) = 2 + 5X + 9X^4$$



Dense ordered structure

- vector
  - minimal degree d_{min} (such that $\forall i < d_{min}, a_i = 0$)
  - maximal degree d_{max} (such that $\forall i > d_{max}, a_i = 0$)
  - all coefficients between d_{min} and d_{max}
  - fast access *O(1)*

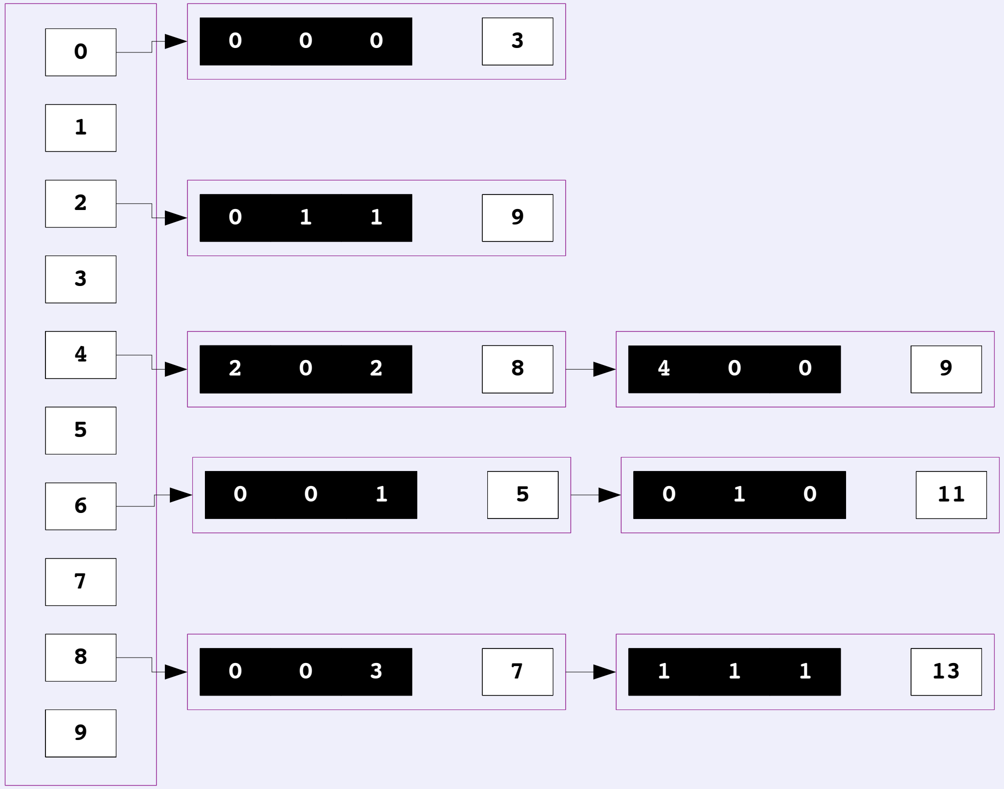


Available representations for multivariate polynomials

- Unordered structures
  - Hashmap
- Sparse ordered structure
  - List of monomials
  - Recursive list
  - Direct Acyclic Graph
  - Recursive vector
- Distributed ordered structure
  - Flat vector
  - (Compacted) Homogeneous blocks

Hash map

- hash key = exponents
- Collision resolution : chaining or cascading

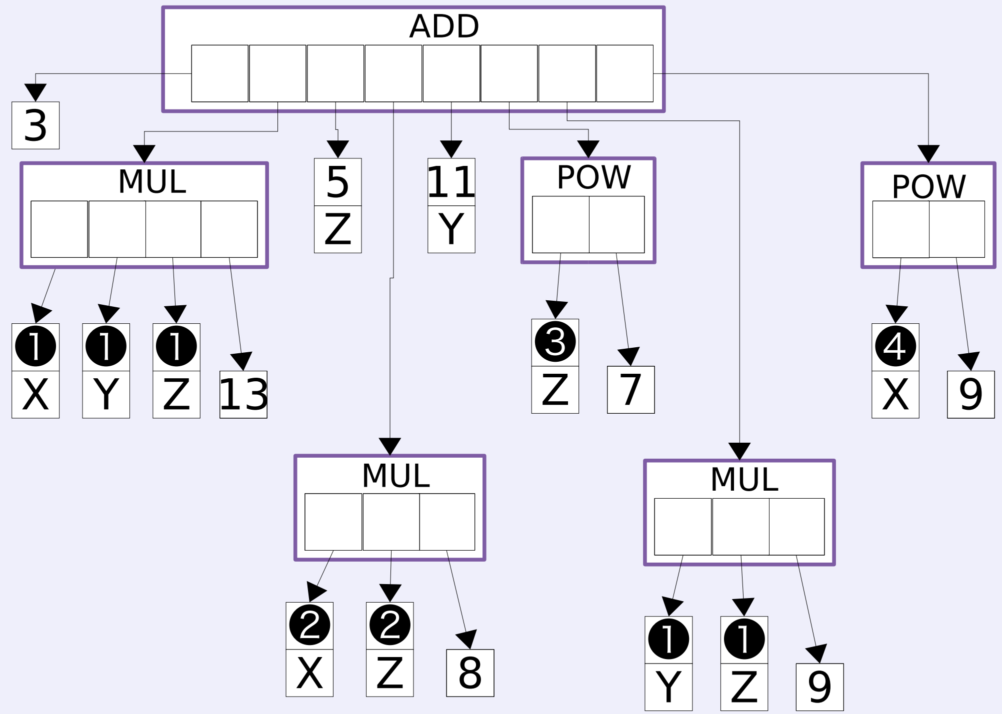


- Used by *GIAC* and *MAGMA*
- *Pros* : insertion/search usually in $O(1)$ but worst case in $O(R)$

General trees

- A node = an operator +,*,^

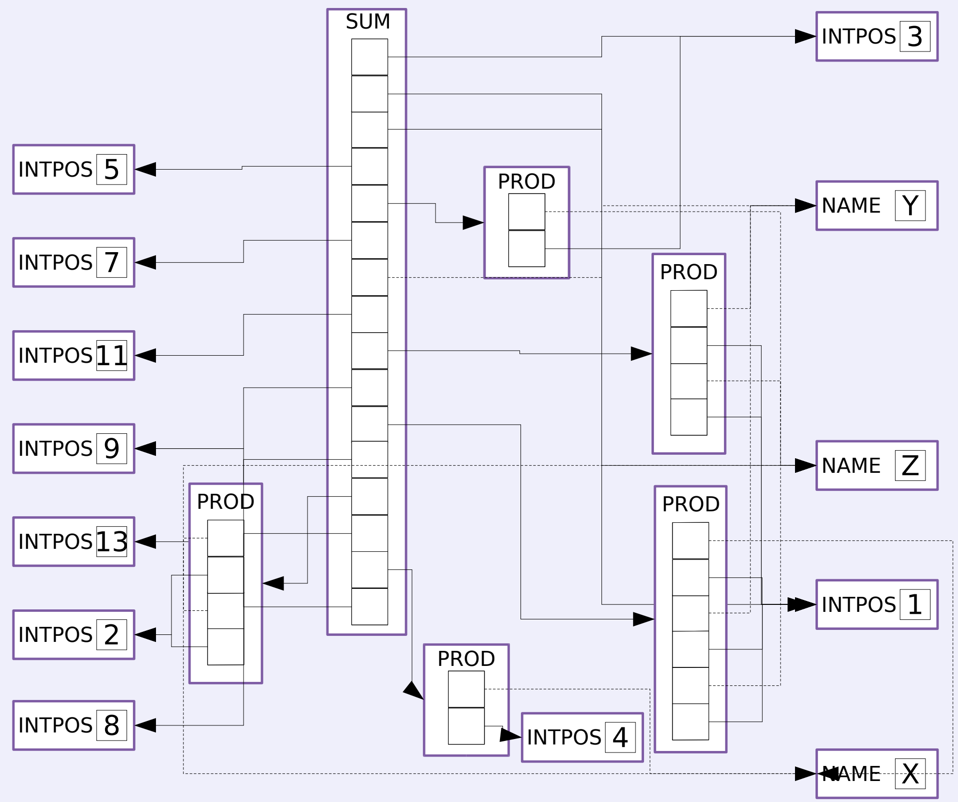
$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$



- Used by *GINAC*
- *Pros* : very easy to extend with your custom algebraic classes

Directed Acyclic Graph

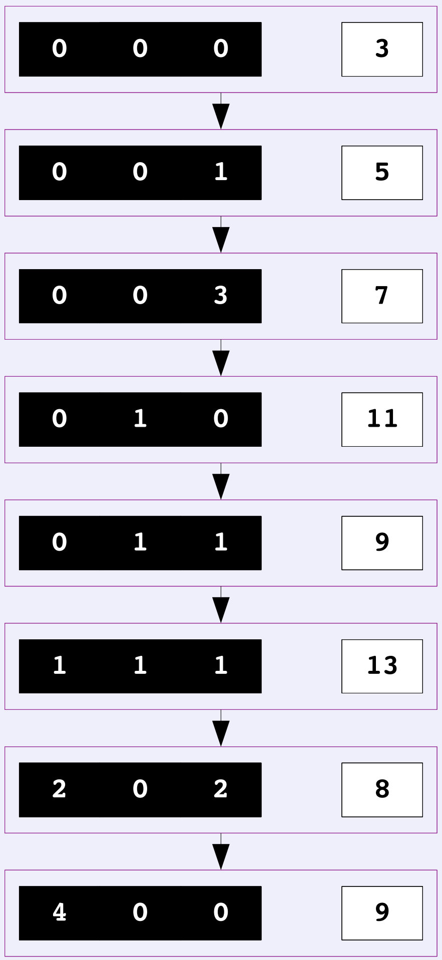
- No duplicated leafs
- couple of monomial and numeric coefficient



- Used by *MAPLE*
- *Pros : handle any type of expressions*

Monomial sorted lists

- Stores one monomial in an element of the sorted list
- an element = vector of exponents + coefficient



- Used by SINGULAR
- *Pros* : efficient for small polynomials
  access to the leading term
- *Cons* : insertion/search in $O(\mathrm{R})$

recursive representations

- polynomial in *n* variables could considered as polynomial in 1 variable with coefficients in the polynomial ring in *n-1* variables

$$P(x_1, x_2, ..., x_n) \in K[x_1, x_2, ..., x_n]$$

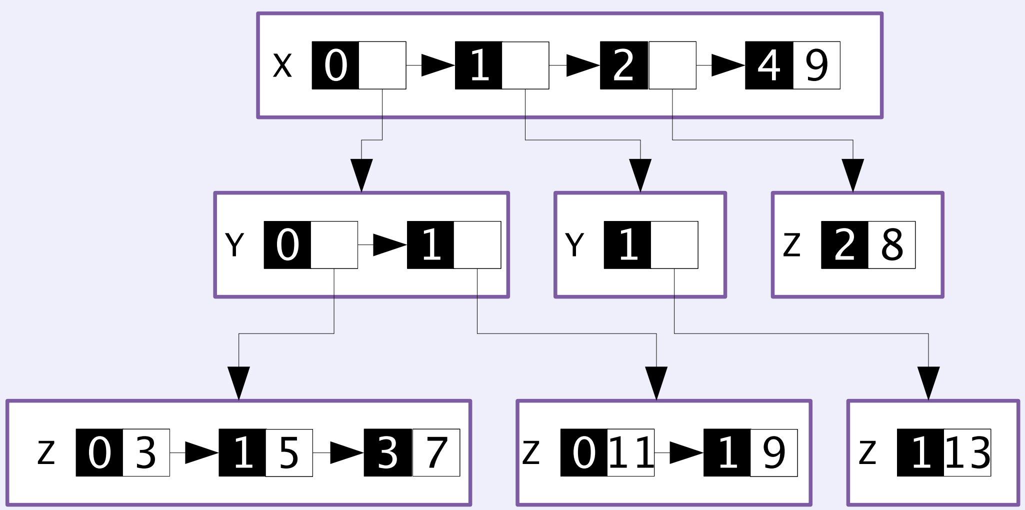
$$\downarrow$$

$$P(x_n) \in K[x_1, x_2, ..., x_{n-1}][x_n]$$

- same data structure as the univariate polynomials
  - handle very sparse polynomials.

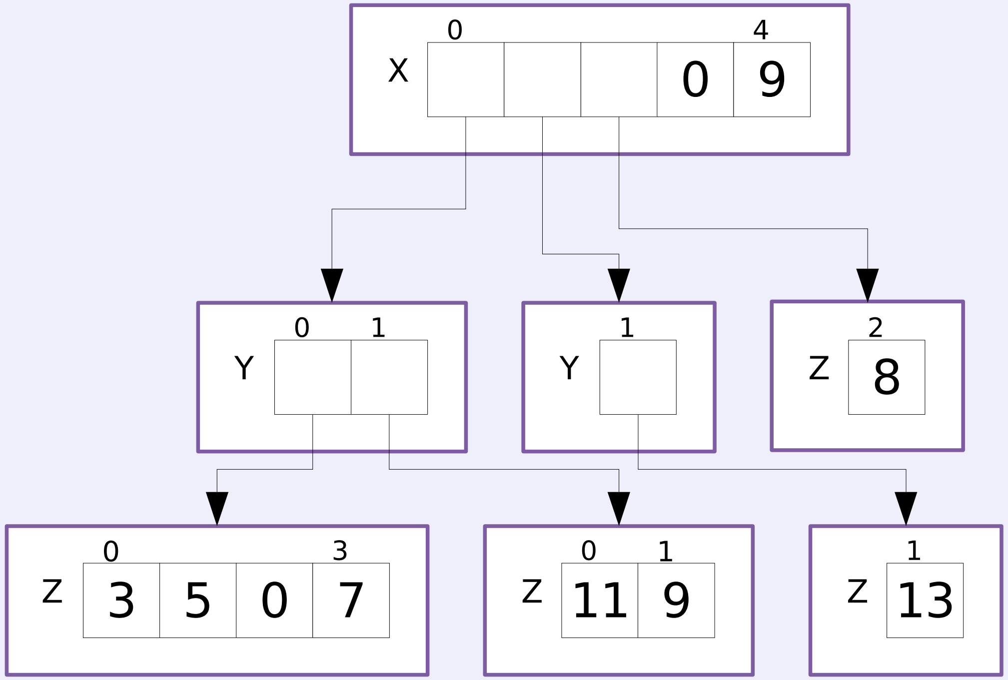
Recursive list

- Tuple of coefficient and exponent
- Contains only non zero coefficient
- Used by *TRIP*
- *Pros* : handle all polynomials
- *Cons* : inefficient with large degree and few variables
  e.g. $P(x,y) = y + (1+x)^{1000}$



Recursive vector

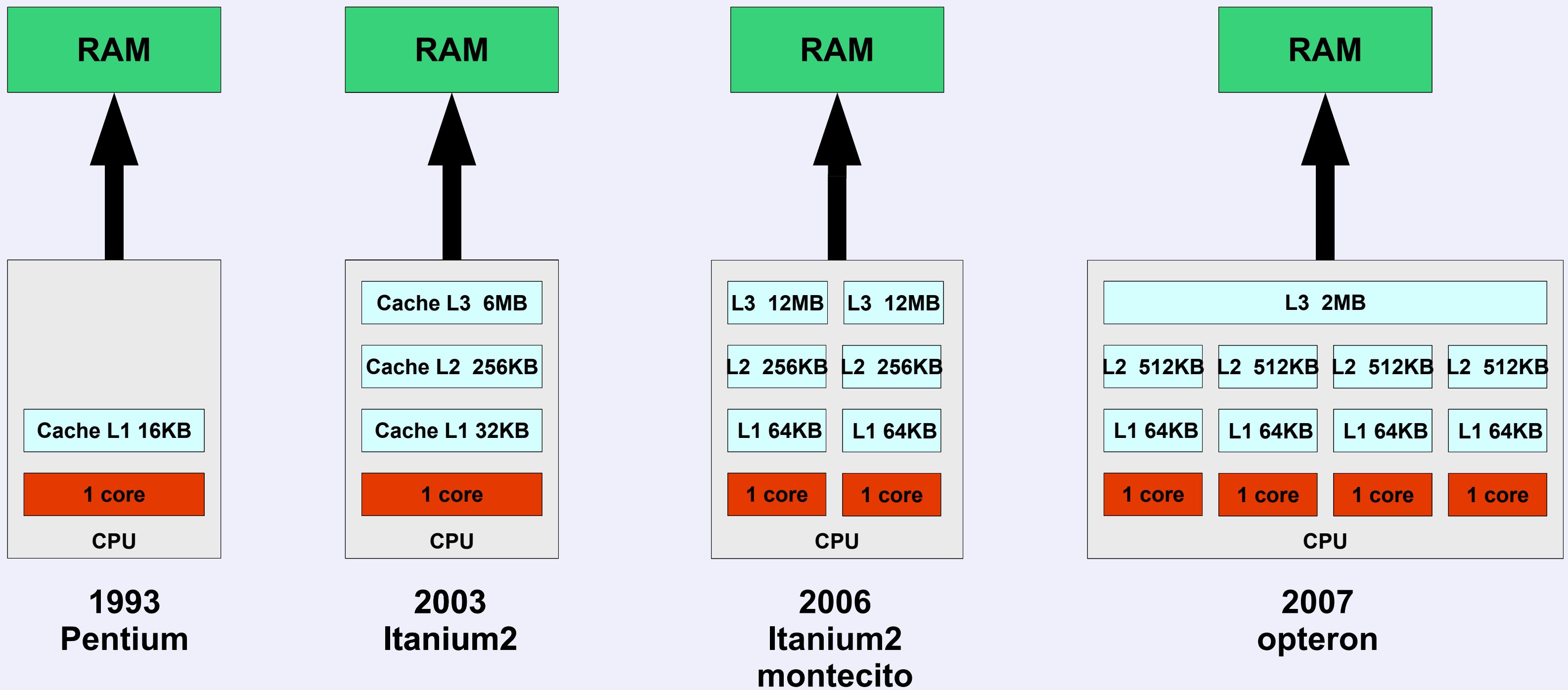
- Stores the minimal and maximal degree
- Store all coefficients into vectors
- Contains zero coefficients



- Used by *TRIP*
- *Pros* : Fast access to a monomial
- *Cons* : inefficient if polynomials have many zeros
  e.g. $P(x) = 1 + x^{1000}$

Sparse structures and processors

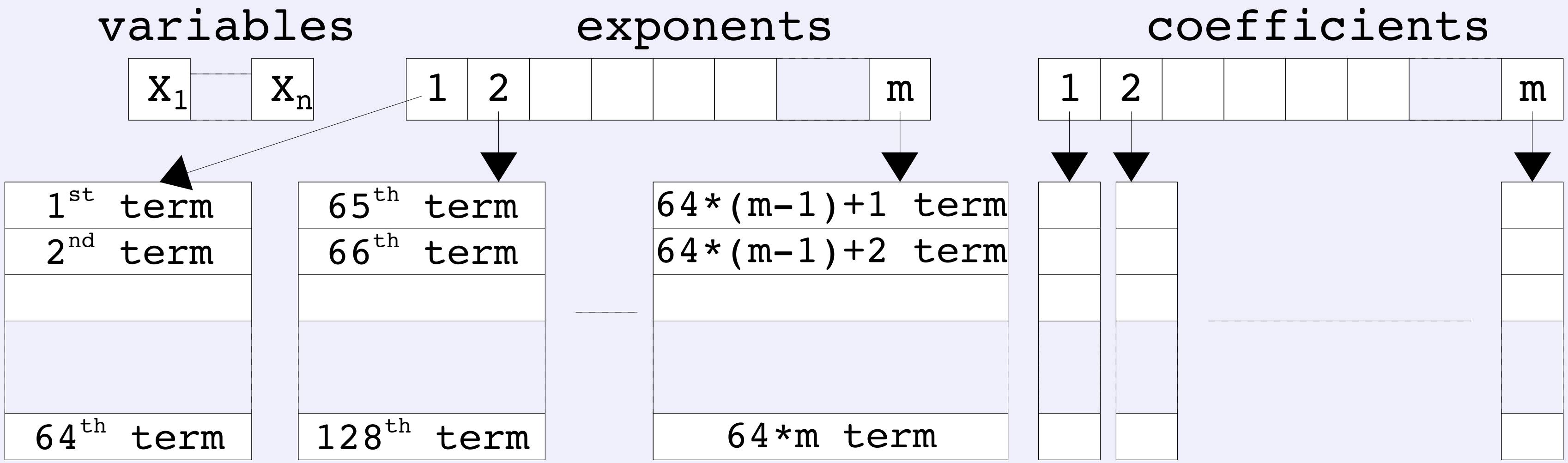
- Sparse structures ➪ less locality of data
- Modern CPU ➪ multiple levels of cache



L1 : 2 cycles
L2 : 5 cycles
L3 : 12~21 cycles
main memory : ~200 cycles

- Distributed structure : access data in order (prefetch) ➪ better locality

Flat vector

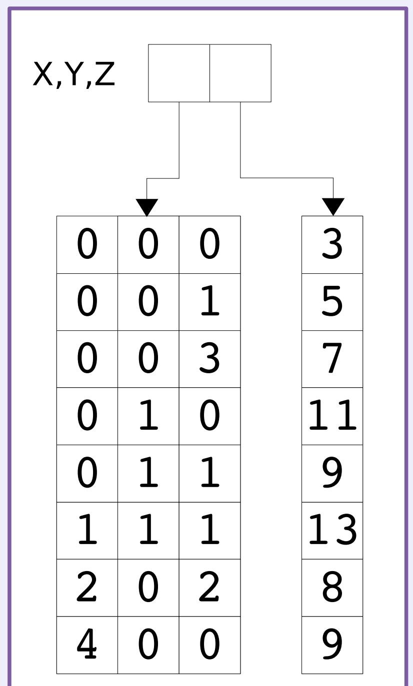


- Store exponents and coefficients in separate vectors
- polynomial has many terms ➪ split the vectors
- Terms are sorted on exponents
- Exponents are compressed with shift and mask operations : negligible overhead (only integer arithmetics)

Flat vector 2/2

- Length of each vector is variable
  ➪ insertion is faster than using a single large vector
- *Pros* : Use dichotomy to search a term
- *Cons* : many insertions are very expensive (many copies)

$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$



Homogeneous blocks 1/4

- $BH_\delta(x_1, ...x_N)$ = set of monomial of degree δ in $x_1, ..., x_N$

$$P(x_1, ..., x_N) = BH_0(x_1, ...x_N) + BH_1(x_1, ...x_N) + ... + BH_D(x_1, ...x_N)$$

0	0	4
0	1	3
0	2	2
0	3	1
0	4	0
1	0	3
1	1	2
1	2	1
1	3	0
2	0	2
2	1	1
2	2	0
3	0	1
3	1	0
4	0	0

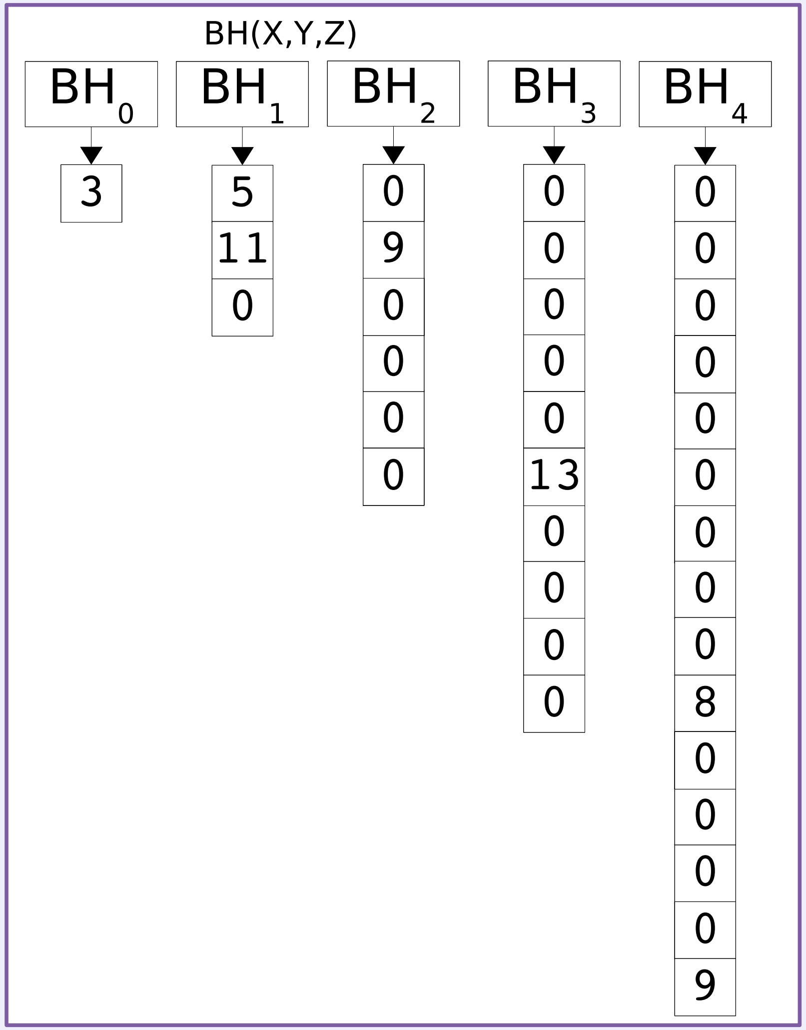
Homogeneous blocks 2/4

- Number of monomials in a block $C^{\delta}_{\delta+N-1} = \dfrac{(\delta+N-1)!}{\delta!(N-1)!}$

	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
δ=0	1	1	1	1	1	1	1	1
δ=1	3	4	5	6	7	8	9	10
δ=2	6	10	15	21	28	36	45	55
δ=3	10	20	35	56	84	120	165	220
δ=4	15	35	70	126	210	330	495	715
δ=5	21	56	126	252	462	792	1287	2002
δ=6	28	84	210	462	924	1716	3003	5005
δ=7	36	120	330	792	1716	3432	6435	11440
δ=8	45	165	495	1287	3003	6435	12870	24310
δ=9	55	220	715	2002	5005	11440	24310	48620
δ=10	66	286	1001	3003	8008	19448	43758	92378
δ=11	78	364	1365	4368	12376	31824	75582	167960
δ=12	91	455	1820	6188	18564	50388	125970	293930
δ=13	105	560	2380	8568	27132	77520	203490	497420
δ=14	120	680	3060	11628	38760	116280	319770	817190
δ=15	136	816	3876	15504	54264	170544	490314	1307504
δ=16	153	969	4845	20349	74613	245157	735471	2042975
δ=17	171	1140	5985	26334	100947	346104	1081575	3124550
δ=18	190	1330	7315	33649	134596	480700	1562275	4686825
δ=19	210	1540	8855	42504	177100	657800	2220075	6906900
δ=20	231	1771	10626	53130	230230	888030	3108105	10015005

Homogeneous blocks 3/4

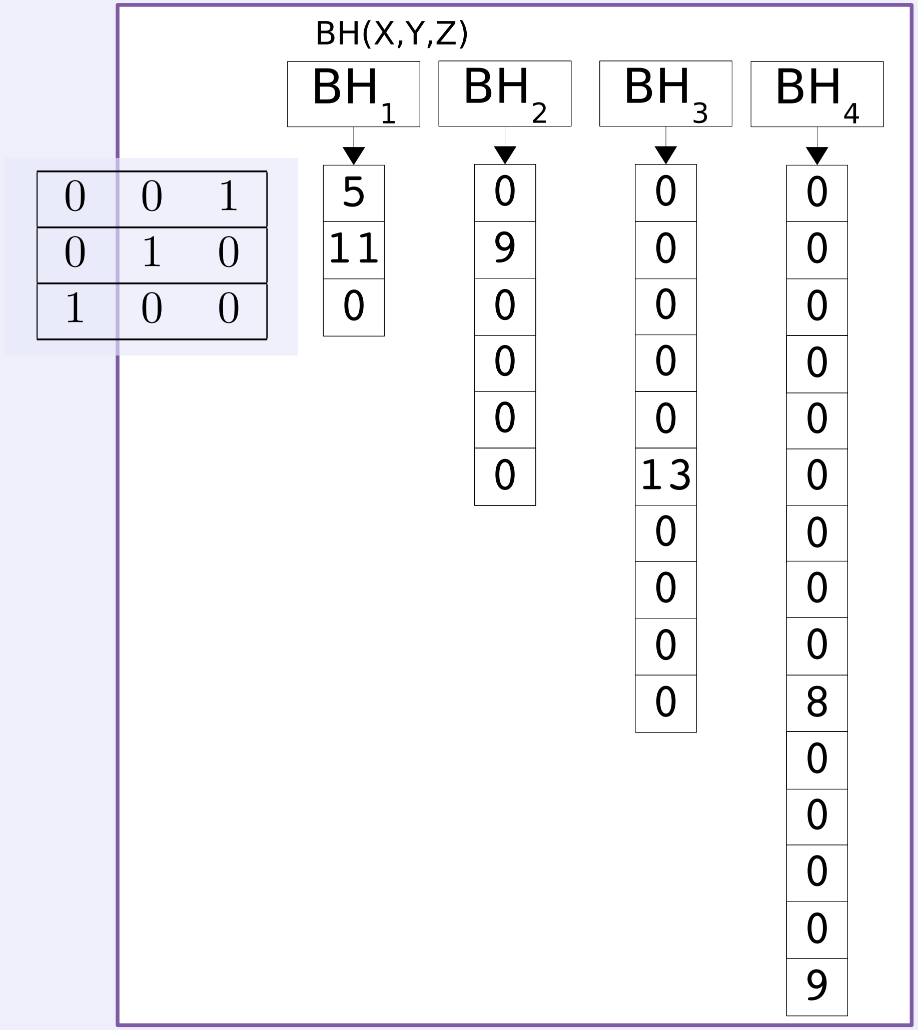
- Store only the coefficients in the vector



$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$

Homogeneous blocks 3/4

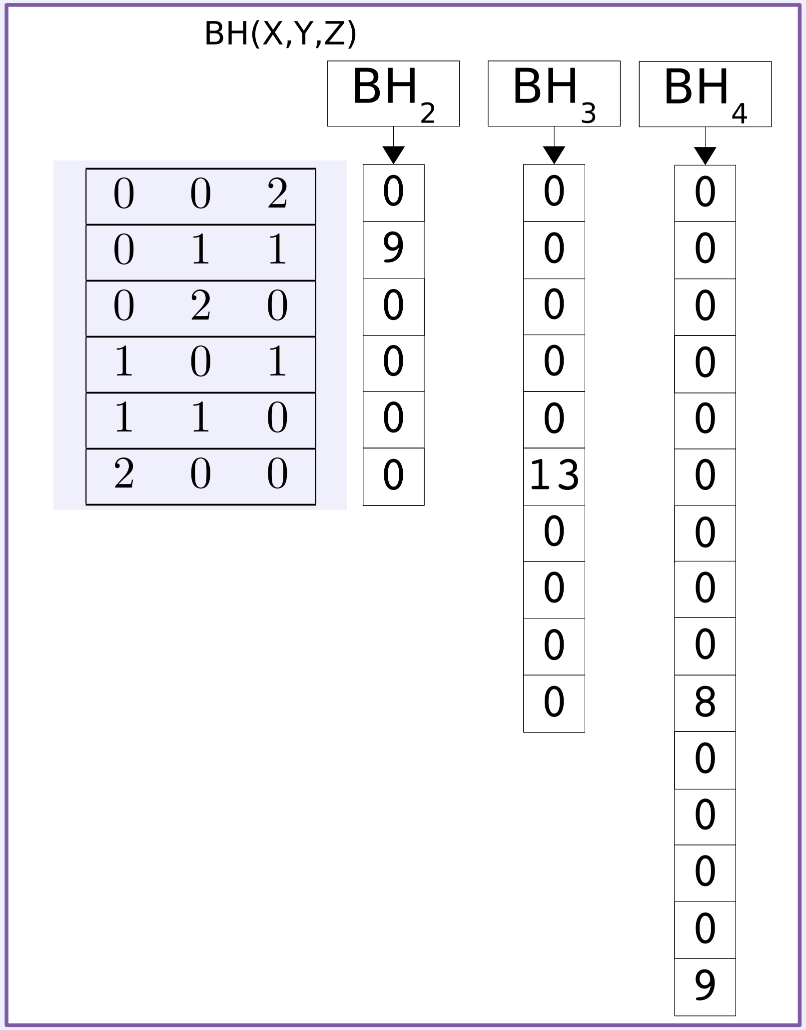
- Store only the coefficients in the vector



$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$

Homogeneous blocks 3/4

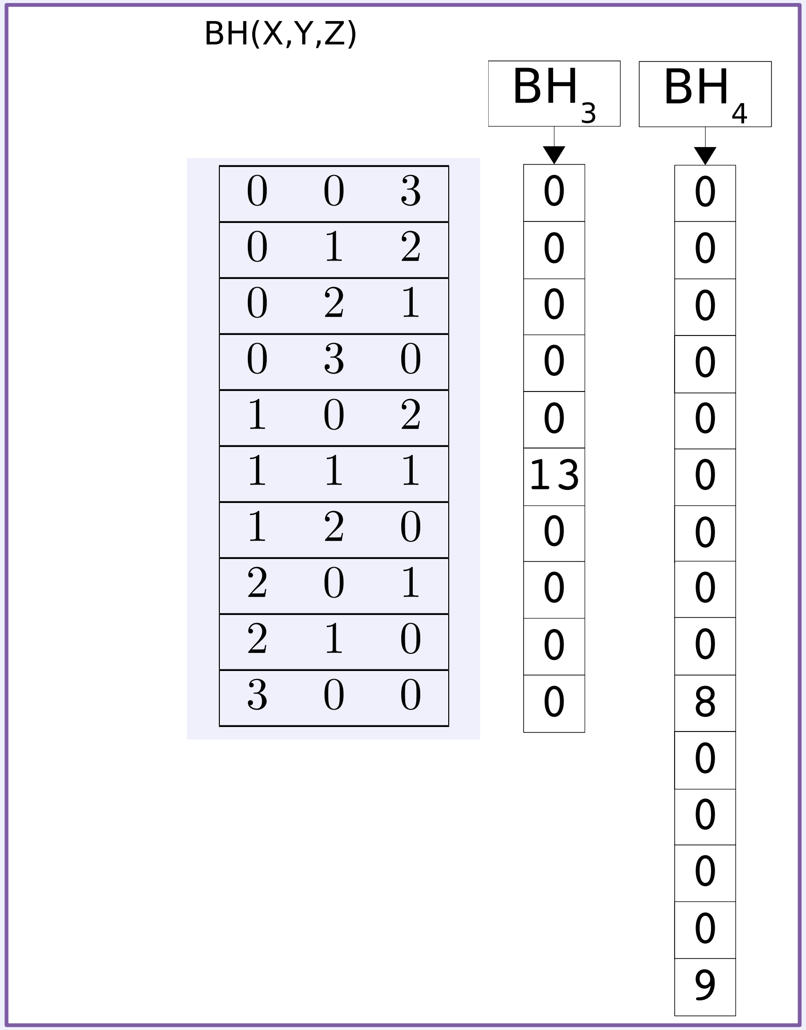
- Store only the coefficients in the vector



$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$

Homogeneous blocks 3/4

- Store only the coefficients in the vector



$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$

Homogeneous blocks 3/4

- Store only the coefficients in the vector

BH(X,Y,Z)

BH_4

0	0	4	0
0	1	3	0
0	2	2	0
0	3	1	0
0	4	0	0
1	0	3	0
1	1	2	0
1	2	1	0
1	3	0	0
2	0	2	8
2	1	1	0
2	2	0	0
3	0	1	0
3	1	0	0
4	0	0	9

$$P(x,y,z) = 3 + 5z + 7z^3 + 11y + 9yz + 13xyz + 8x^2z^2 + 9x^4$$

Function to compute index in a homogeneous block

Algorithm 1: Compute the index of a coefficient in a homogeneous block of degree δ in n variables from the exponents

Input: $expo$: array of n exponents
Output: $index$: index of the coefficient in the array of coefficients

$\delta \leftarrow \sum_{i=1}^{n} expo[n]$
$index \leftarrow 1$
$d \leftarrow \delta$
for $i \leftarrow 1$ **to** $n-1$ **do**
  $index \leftarrow index + \sum_{j\geqslant 0}^{expo[n]-1} C_{d-j+i-1}^{d-j}$
  $d \leftarrow d - expo[i]$
end
return $index$

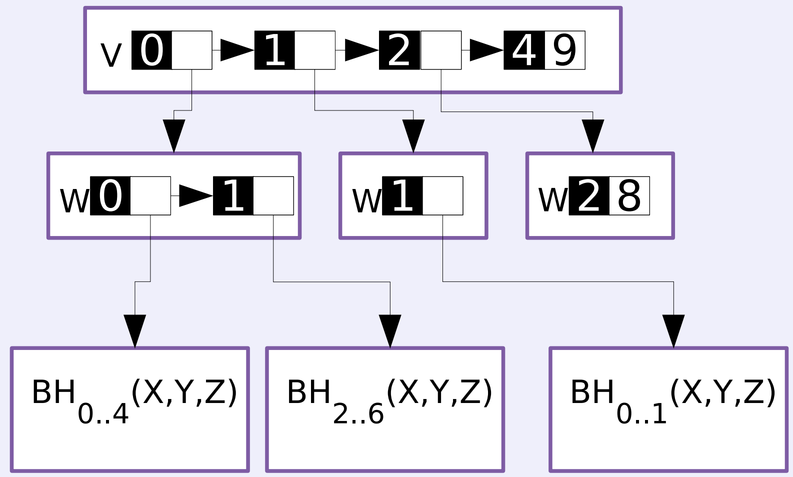
- Optimization if n fixed at compilation time

Homogeneous blocks 3/4

- Exponents :
  - Shared exponent tables :
    - All blocks in memory of same degree will reference the same exponent table
    - Used in *TRIP*
    - *Pros*: Fast access to all exponents
    - *Cons*: Require large amount of memory for high degree
  - Always computed with an internal function
    - *Pros*: memory footprint $= 0$ bytes !
    - *Cons*: Require computations to have exponents from the location

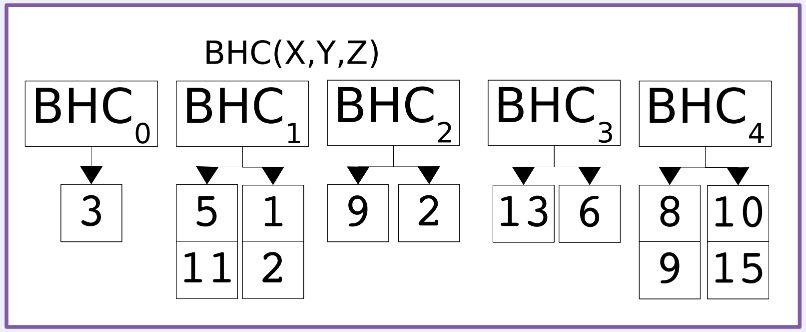
Mixing representation

- sparse structures for some variables
- distributed structures for other variables



Compacted homogeneous blocks 1/2

- For very sparse polynomials
  - remove the zeros
  - use an index vector to keep the location in the exponent table or to retrieve exponents from a computational function.
  - Insertion algorithm requires a duplication of the two vectors, but multiple insertion could work on the non-compacted block.
  - The search algorithm has the complexity of the size of the index vector.



d'Alembert block

- If combination of exponents has some properties
- Based on homogeneous block
  - use a modified function to compute the new location
  - use a specific exponent table
    - Used in *TRIP* for the d'Alembert properties
- Example : d'Alembert rules

$$P(z,\overline{z},\zeta,\overline{\zeta},z',\overline{z'},\zeta',\overline{\zeta'},\lambda,\lambda') = \sum C z^{n_i}\overline{z}^{\overline{n_i}}\zeta^{n_j}\overline{\zeta}^{\overline{n_j}}z'^{n'_i}\overline{z'}^{\overline{n'_i}}\zeta'^{n'_j}\overline{\zeta'}^{\overline{n'_j}}\exp^{i(k\lambda+k'\lambda')}$$

- $\delta = n_i + \overline{n_i} + n_j + \overline{n_j} + n'_i + \overline{n'_i} + n'_j + \overline{n'_j}$
- $c_I(T) = k + k'$
- $c_M(T) = (\overline{n_i} + \overline{n'_i} + \overline{n_j} + \overline{n'_j}) - (n_i + n'_i + n_j + n'_j).$

$c_I(T) = C_M(T)$ and $c_I(T)$, $C_M(T)$ and δ have the same parity.

d'Alembert block

Block with even parity in $(y, \overline{y})$

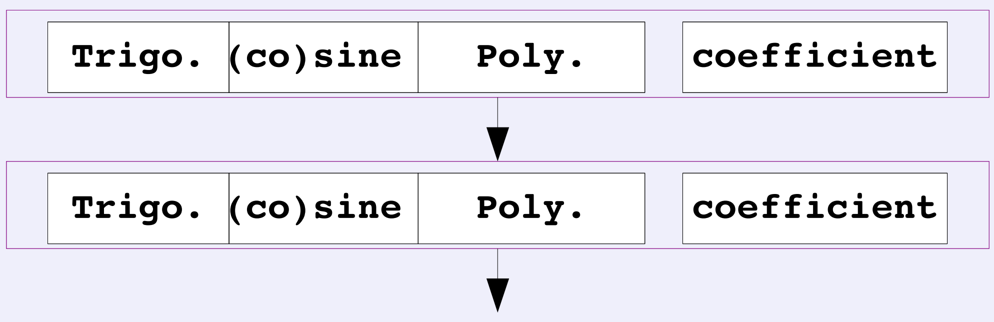
c, δ	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	–	8	–	52	–	200	–	617	–	1568	–	3536	–	7200	–
1		2	–	20	–	100	–	350	–	980	–	2352	–	5040	–	9900
2			6	–	40	–	178	–	560	–	1476	–	3360	–	6940	–
3				10	–	70	–	280	–	840	–	2100	–	4620	–	9240
4					19	–	112	–	424	–	1200	–	2895	–	6160	–
5						28	–	168	–	600	–	1650	–	3850	–	8008
6							44	–	240	–	830	–	2200	–	5014	–
7								60	–	330	–	1100	–	2860	–	6370
8									85	–	440	–	1436	–	3640	–
9										110	–	572	–	1820	–	4550
10											146	–	728	–	2282	–
11												182	–	910	–	2800
12													231	–	1120	–
13														280	–	1360
14															344	–
15																408

Table 1: Number of monomials $(x^i\overline{x}^{\overline{i}}y^j\overline{y}^{\overline{j}}x'^{i'}\overline{x'}^{\overline{i'}}y'^{j'}\overline{y'}^{\overline{j'}})$ of degree δ and with characteristic c.

Poisson series

$$S(X_1, ...X_n, \lambda_1, .., \lambda_m) = \sum_{d_1,\ldots,d_n,k_1,\ldots,k_m} c_{d,k} X_1^{d_1} ... X_n^{d_n} \begin{pmatrix} cos \\ sin \end{pmatrix} (k_1\lambda_1 + ... + k_m\lambda_m)$$

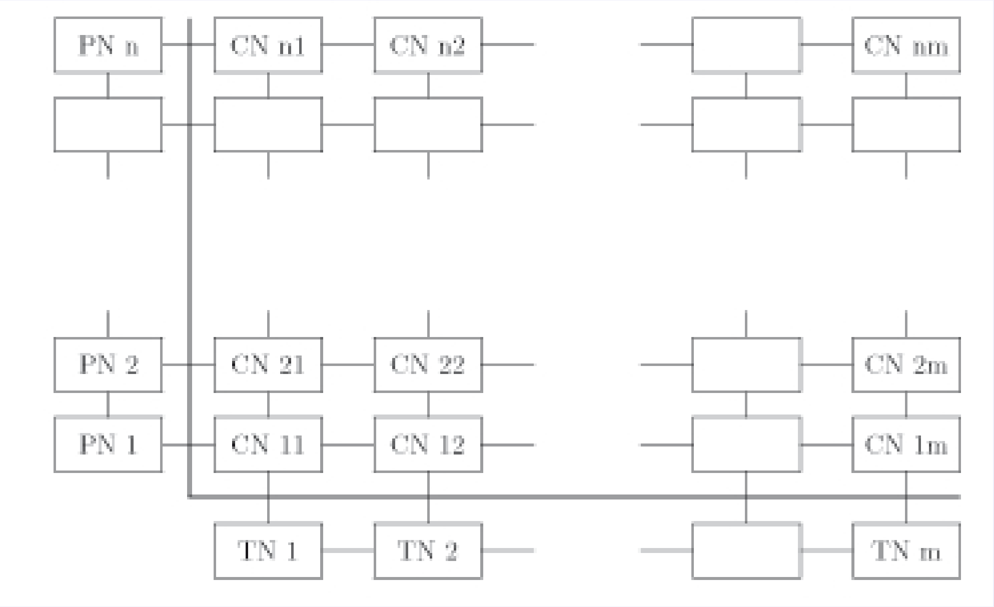
- list of terms



- circular linked-list of terms in EPSP, developed by Ivanova

Poisson series

- bidimensional structures
  - access from polynomial and trigonometric part



Poisson series

- Complex form

$$S(X_1, ...X_n, \lambda_1, .., \lambda_m) = \sum_{d_1,...,d_n,k_1,...,k_m} c_{d,k} X_1^{d_1} ...X_n^{d_n} \exp^{\imath(k_1\lambda_1+....+k_m\lambda_m)}$$

- same representation as polynomials
- new variables $\Lambda_m = \exp^{\imath(\omega_m\lambda_m+\varphi_m)}$
- $k_1, ..., k_m$
  - ➪ integers
  - ➪ literal values

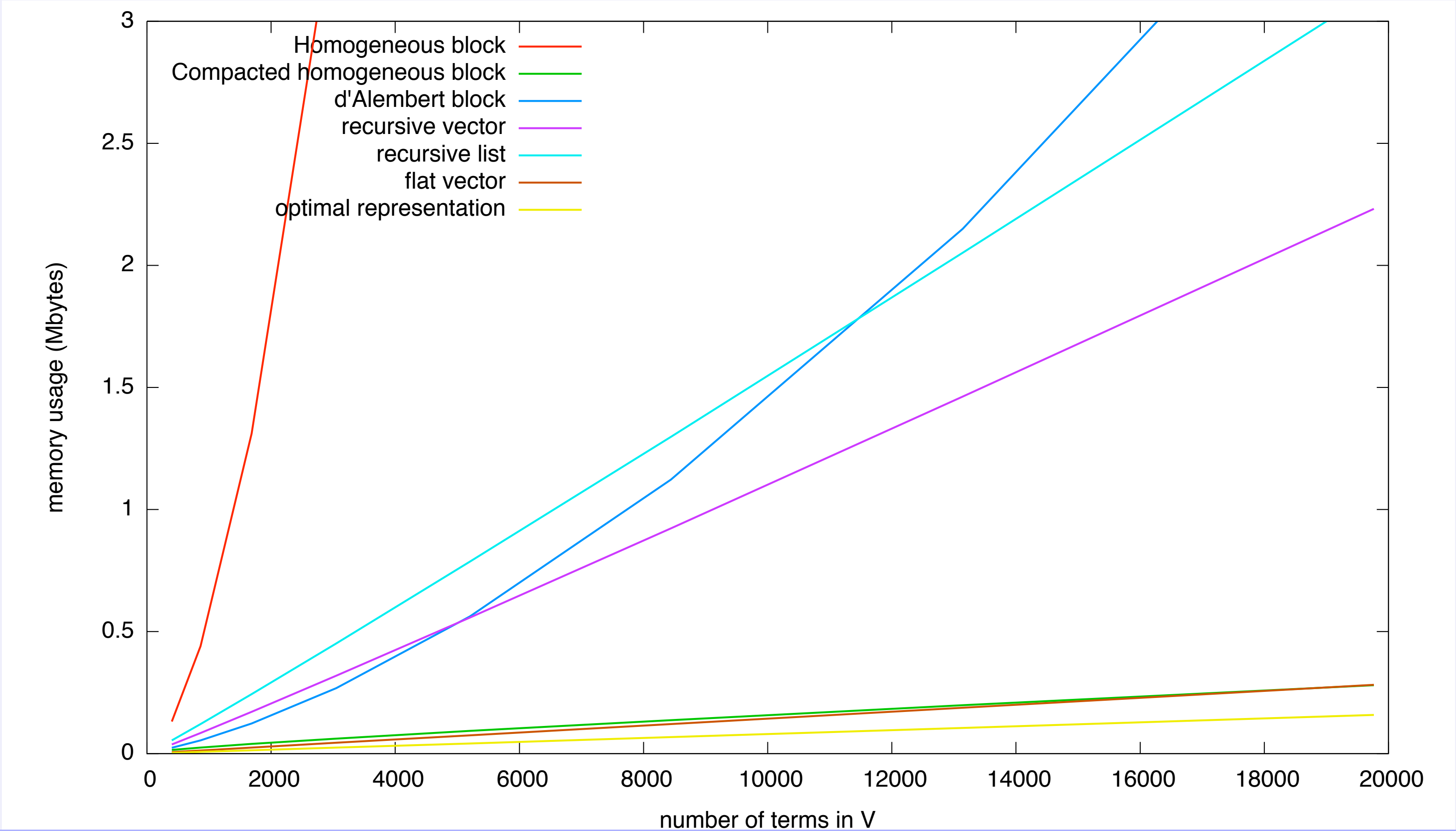
Benckmarks

- Expansion of the perturbing function in the planetary problem

$$V = \alpha^2[\frac{\rho^2}{\alpha^2} - 1] + 2\alpha[\cos(\lambda - \lambda') - \frac{\rho}{\alpha}\cos S]$$

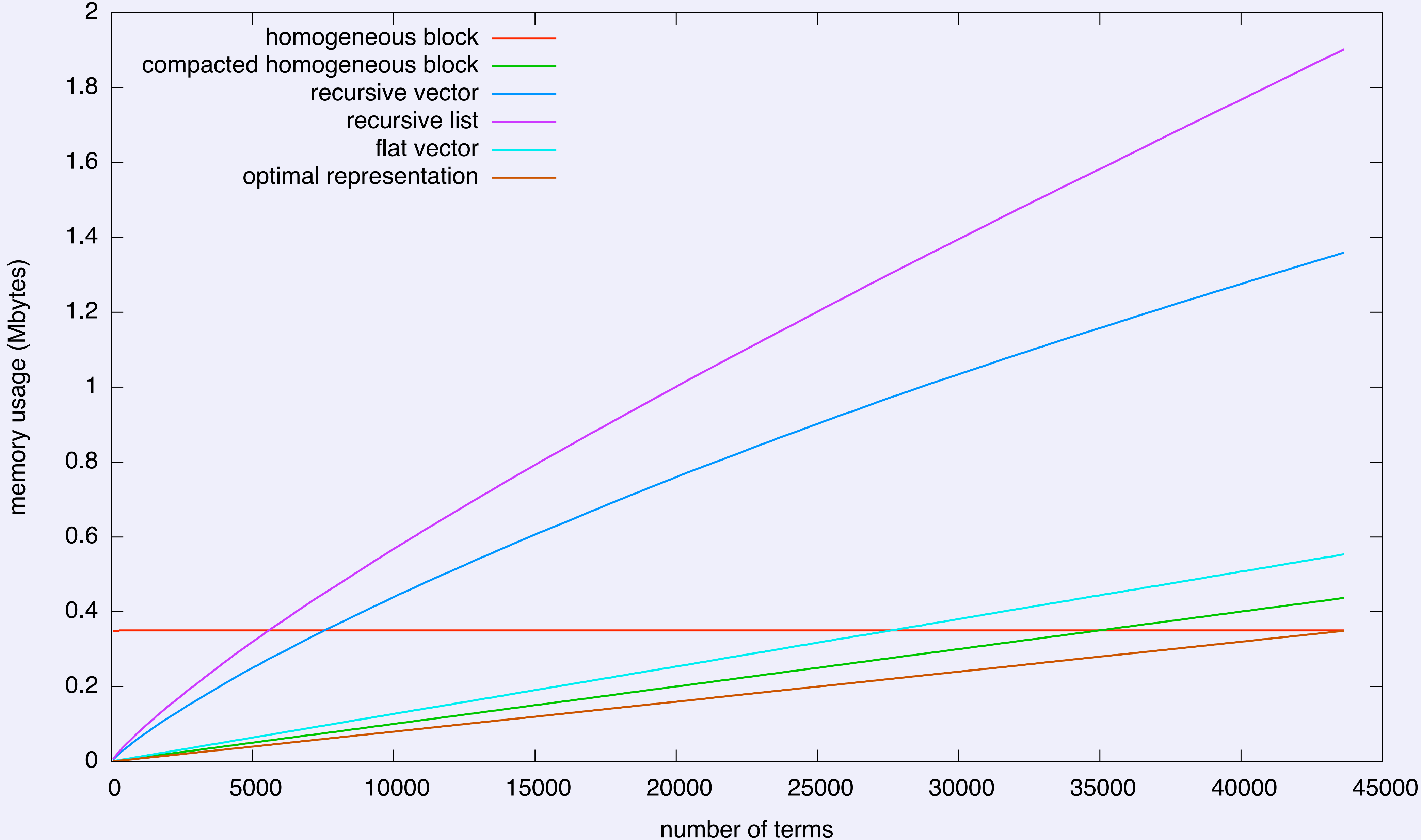
- expanded using the Poincaré's variables

$$V(\lambda, \lambda', X, \overline{X}, Y, \overline{Y}, X', \overline{X'}, Y', \overline{Y'}) = \sum X^{d_1}\overline{X}^{d_2}Y^{d_3}\overline{Y}^{d_4}, X'^{d_5}\overline{X'}^{d_6}Y'^{d_7}\overline{Y'}^{d_8}\exp^{\imath(k_1\lambda + k_2\lambda')}$$



Benchmarks - sparsity

- randomily selected terms from $P(x_1, ...x_{10}) = (1 + x_1 + ... + x_{10})^{10}$



Representations of coefficients

- Numerical coefficients
  - Fixed-size
    - hardware floating-point numbers
    - hardware or software integer and rational numbers
    - floating-point interval
  - multiple precision
    - floating-point numbers
    - integer and rational numbers
    - floating-point interval
- Literal coefficients

fixed-size floating-point numbers

- representation of floating-point numbers in IEEE 754 standard

sign	exponent	significand	
63	62 52	51 0	double precision

Format	double	extended	quadruple
Format width in bits	64	80	128
Significand in bits	52+1	64+0	112+1
Exponent width in bits	11	15	15
ulp(1)	$2^{-52} \approx 2,22.10^{-16}$	$2^{-63} \approx 1,08.10^{-19}$	$2^{-112} \approx 1,92.10^{-34}$

- double precision available on all platforms : hardware registers
- extended precision available on intel or compatible processors 32 or 64 bits
- quadruple precision is emulated with software on all platforms
  - encoded with 2 doubles on PowerPC architectures (double-double)

Time overhead

- Compilation with Intel c++ 10.0.023
- Execution on an Intel Xeon MP

Operation	$\frac{t_{extended}}{t_{double}}$	$\frac{t_{quadruple}}{t_{double}}$
$+$	1.6	7.3
$\times$	1.9	9.1
$/$	2.0	7.6
cos	2.0	76.0
sqrt	4.5	96.0

- Execution on an Intel Itanium2

Operation	$\frac{t_{extended}}{t_{double}}$	$\frac{t_{quadruple}}{t_{double}}$
$+$	2	17
$\times$	2	20
$/$	2	38
cos	2	90
sqrt	2	140

fixed-size integer and rational numbers

- use hardware registers (32 or 64 bits)
- could use guard bits to detect the overflows on addition
- rational numbers :
  - structure of 2 integers
  - use binary gcd to avoid division
  - requires to redefine all basic operations
- Used in *SINGULAR* for integers between 2^(-28) and 2^(+28)

integer and rational multiple precision

- handle numbers with million bits !!!
- require dynamic memory allocation ➪ automatic grow
  - use FMA whenever possible ➪ avoid one temporary value
    FMA(x,y,z) = x+y.z
- available libraries
  - GMP http://gmplib.org/ very portable and highly optimized
  - NTL http://www.shoup.net/ntl slower than GMP
  - CLN http://www.ginac.de/CLN/ less portable
- Timings

Operation	$\frac{t_{mpz\_t}}{t_{int64\_t}}$	$\frac{t_{mpq\_t}}{t_{double}}$
$+$	11	48
$\times$	6	26
$/$	12	85

floating-point multiple precision

- library MPFR
  - extend the significant part of the IEEE754 real numbers
  - FMA to reduce the intermediate values

- library MPC
  - based on MPFR
  - implement the complex numbers
  - provide correctly rounded arithmetic operations

floating-point interval

- Interval $\pi \approx 3.14159$ replaced by $\pi \subset [3.14158, 3.14160]$

- Interval arithmetic
  - all operations must guarantee that the exact result is in the returned interval
  - interval could become very large $[2,3] - [2,3] = [-1,1]$

- Implementation using hardware floating-point
  - use rounding-mode of the processor
  - library FILIB++ and CostLy

- Implementation using multiple precision floating-point
  - library MPFI based on GMP/MPFR

Benchmarks

$$P(x, y, z, t, u) = (1 + x + y + z + t + u)^{31}$$

Type	Size (bytes)
double precision floating-point	13739456
quadruple precision floating-point	19771328
multiple precision floating-point (mpfr_t) (significand = 53bits)	34851008
multiple precision floating-point (mpfr_t) (significand = 200bits)	58978496
double precision floating-point interval	19771328
quadruple precision floating-point interval	25803200
multiple precision integer (mpz_t)	22915568

Type	Execution time (s)
double precision floating-point	6.42
quadruple precision floating-point	36.13
multiple precision floating-point (mpfr_t) (significand = 53bits)	131.54
multiple precision floating-point (mpfr_t) (significand = 200bits)	337.04
double precision floating-point interval	323.51
quadruple precision floating-point interval	372.26
multiple precision integer (mpz_t)	19.34

Benckmarks

$$P(x,y,z,t,u) = (1/2\imath + 3/5x + y + (1+7/11\imath)z + 13/17t + (31+21/29\imath)u)^{31}$$

Type	Size (bytes)
double precision floating-point	13739456
quadruple precision floating-point	25753152
multiple precision floating-point (mpc_t) (significand = 53bits)	55787392
multiple precision floating-point (mpc_t) (significand = 200bits)	103842176
double precision floating-point interval	25753152
quadruple precision floating-point interval	37766848
multiple precision integer (mpq_t)	61389176

Type	Execution time (s)
double precision floating-point	24.33
quadruple precision floating-point	156.74
multiple precision floating-point (mpc_t) (significand = 53bits)	611.20
multiple precision floating-point (mpc_t) (significand = 200bits)	1593.44
double precision floating-point interval	975.95
quadruple precision floating-point interval	1244.21
multiple precision rational (mpq_t)	4238.61

literals

- view as variables
  - store a pointer to the name and the numerical coefficient
  - only for recursive sparse representation
- view as an opaque expression
- Rational function and small divisor

$$C(\lambda_1, ...\lambda_n) = \frac{P(\lambda_1, ...\lambda_n)}{Q(\lambda_1, ...\lambda_n)} \text{ with } P \text{ and } Q \in K[\lambda_1, ...\lambda_n]$$

  - encoding with an array of integers
  - fraction-free representation
- optimized expressions