Experiences with the Gröbner Engine in SINGULAR

H. Schönemann

http://www.singular.uni-kl.de

Universität Kaiserslautern
Fachbereich Mathematik
D-67663 Kaiserslautern

Overview of SINGULAR

- Computations in very general rings, including polynomial rings, localizations hereof at a prime ideal and tensor products of such rings. This includes, in particular, Buchberger's and Mora's algorithm as special cases.
- Many ground fields for the above rings, such as the rational numbers, finite fields Z/p, p a prime ≤ 2147483629, finite fields with $q = p^n$ elements, transcendental and algebraic extensions, floating point real numbers with arbitary (fixed) precision.
- Usual ideal theoretic operations, such as intersection, ideal quotient, elimination and saturation and more advanced algorithms based on free resolutions of finitely generated modules. Several combinatorial algorithms for computing dimensions, multiplicities, Hilbert series ....
- Library of procedures, written in the SINGULAR language, which are useful for many applications to mathematical problems.
- ... (and much more, not directly connected to Groebner bases)

Requirements for a general Groebner base engine

- different coefficient domains (integers resp. rationals, $\mathbf{Z}$/p, algebraic extensions, ...)
- different orderings
  - the standard orderings: degrevlex, lexicografic
  - eliminations orderings for variables / block orderings for parameters
  - local orderings for computations in local rings
  - extension to modules for syzygies, lifting(transformation matrix) etc.
- different multiplications: non-commutative Groebner bases
- different results: Groebner basis, minimal generating set, transformation matrix, syzygies, ...

Problems for an efficient implementation

- choosing the right algorithm (Buchbergers algorithm and variants (which?), FGLM, Gröbner walk, standard basis computation driven by Hilbert function, etc.)
- data structures
  - How should polynomials and monomial be represented and their operations be implemented?
  - What is the best way to implement coefficients?
  - How should the memory management be realized?
- flexiblity versus speed

Monomial representations

- Macaulay 3.0 (1994): encode monomial by coefficient and an integer (enumerating all monomial by the monomial ordering)
  comparing is very fast, multiplication slow, divisibility test improved by a second represention for head terms: vector of exponents
  degree bound
- PoSSo (1993-1995): encode monomial by coefficient and exponent vector and ordering vector:
  (the exponent vector multiplyed by the order matrix): only lexicographical comparison necessary (fast)
  fast monomial operations: simply add the complete vector for multiplication etc.
  but used a "lot" of memory for each monomial
- CoCoA: Hilbert-Poincaré series (1997): bit support for fast divisibility tests
- Faugéres Algorithm F_4 (1999): monomial correspond to matrix entries: a monomial is a coefficient and a (column) number

- SINGULAR 1.4: exponent vector as char/short, operations on an array of long: smaller representation, vectorized monomial operations.
- SINGULAR 2.0: exponent vector as bit fields, operations on an array of long: smaller representation, vectorized monomial operations, Geo buckets, divisibility tests by generalized bit support(in a 32 bit word)
- SINGULAR 3.0: better handling of the case of more than 32 variables in bit support (still experimental)

Monomial representations in SINGULAR

2-0 / 3-0

- bit fields for exponents
- degree of (sub-)sets of variables according to the monomial ordering

For example $9ab^2x^3y^4z \in K[a,b][x,y]$ with an degree-reverse-lex. ordering on both blocks of variables will be representetd as: $(9,((3),(1,2)),((8),(3,4,1)))$
coefficient: 9
degree for first block (a,b): 3
exponents first block: 1,2
degree for second block (x,y,z): 8
exponents second block: 3,4
used space: 4 words

Polynomial operations

- general: one API for all types of rings, all orderings, all coeffient fields
- data representation is parametrized by the monomial ordering: a vector to compare monomials, a vector to check divisiblity
- operations for coefficients are handled via function pointers
- optimize this basic idea:
  - merge the 2 vectors in a monomial
  - special variants of import subroutines for specific orderings and specific coefficient fields (automatically generated: depending on the version: 351 to 1651 generated variants for 15 routines:
    `p_Add_q, p_Copy, p_Delete, p_Merge_q`, `p_Minus_mm_Mult_qq`, `p_Mult_mm`, `p_Mult_nn`, `p_Neg`, `p_ShallowCopyDelete`, `p_kBucketSetLm`, `pp_Mult_mm_Noether`, `pp_Mult_mm`, `pp_Mult_nn`...

Memory management I

Most of SINGULAR's computations boil down to primitive polynomial operations like copying, deleting, adding, and multiplying of polynomials. For example, standard bases computations over finite fields spent (on average) 90 % of their time realizing the operation `p - m*q` where `m` is a monomial, and `p,q` are polynomials.

Size of monomials: minimum size is 3 words, average size is 4 to 6 machine words

requirements of a memory manager for SINGULAR:

(1) allocation/deallocation of (small) memory blocks must be extremely fast

(2) consecutive memory blocks in linked lists must have a high locality of reference

(3) the size overhead to maintain small blocks of memory must be small

(4) the memory manager must have a clean API and it must support debugging

(5) the memory manager must be customizable, tunable, extensible and portable

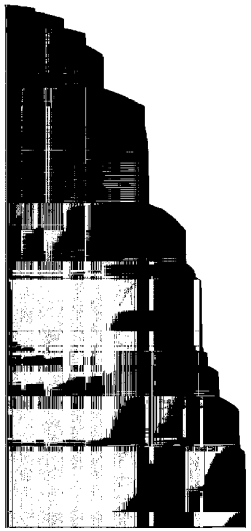
Memory management II

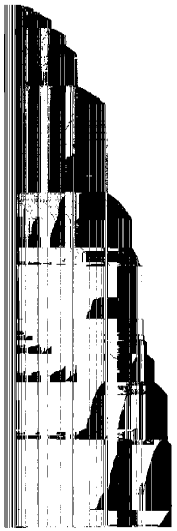
OMALLOC manages small blocks of memory on a per-page basis. That is, each used page is split up into a page-header and equally-sized memory blocks. The page-header has a size of 6 words (i.e., 24 Byte on a 32 Bit machine), and stores (among others) a pointer to the free-list and a counter of the used memory blocks of this page.

On *memory allocation*, an appropriate page (i.e. one which has a non-empty free list of the appropriate block size) is determined based on the used memory allocation mechanism and its arguments. The counter of the page is incremented, and the provided memory block is dequeued from the free-list of the page.

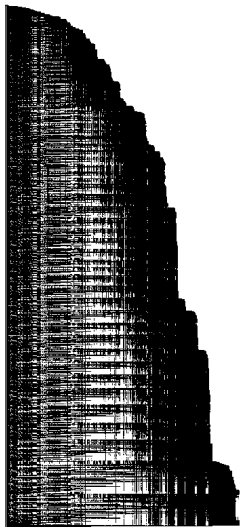
This design results in

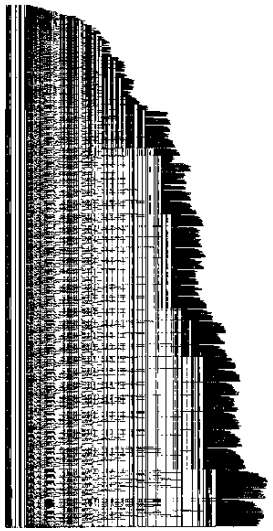
- very fast allocation/deallocation of small memory blocks
- extremely high locality of reference (may be further improved by using specific pages (i.e. specific free lists) for certain elements)
- small maintenance size overhead: 24 Bytes per page (0.6 %)
- still "usable" for larger blocks



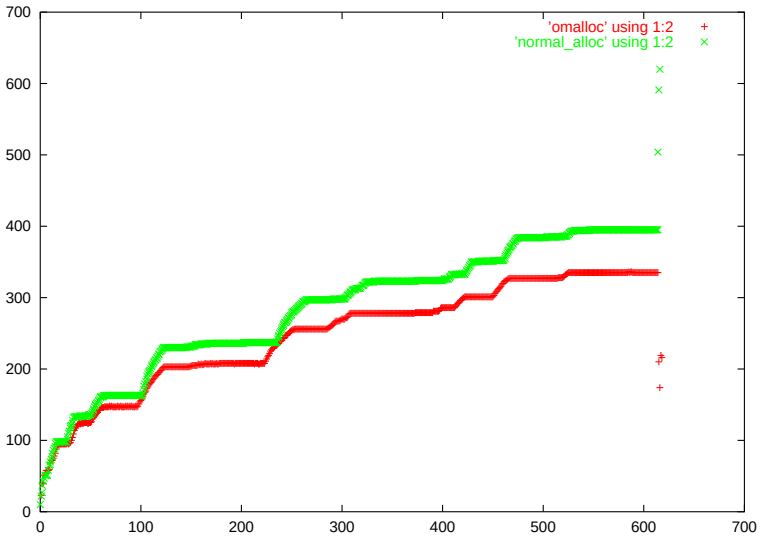


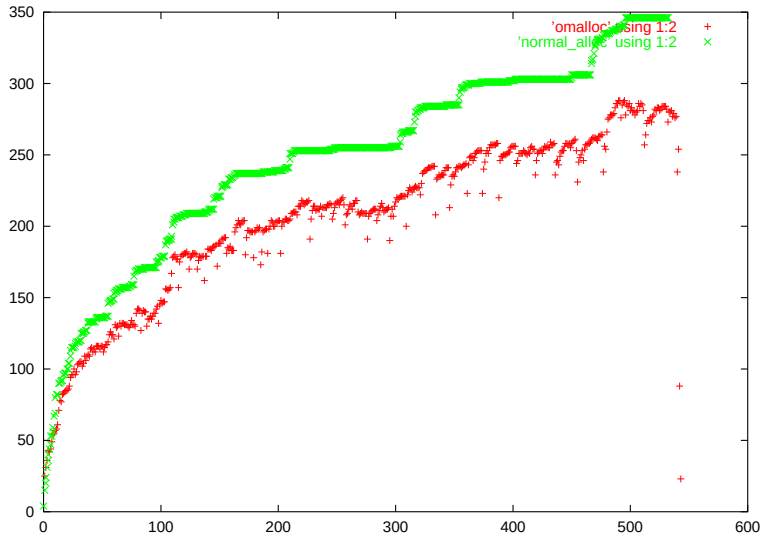
example in char p

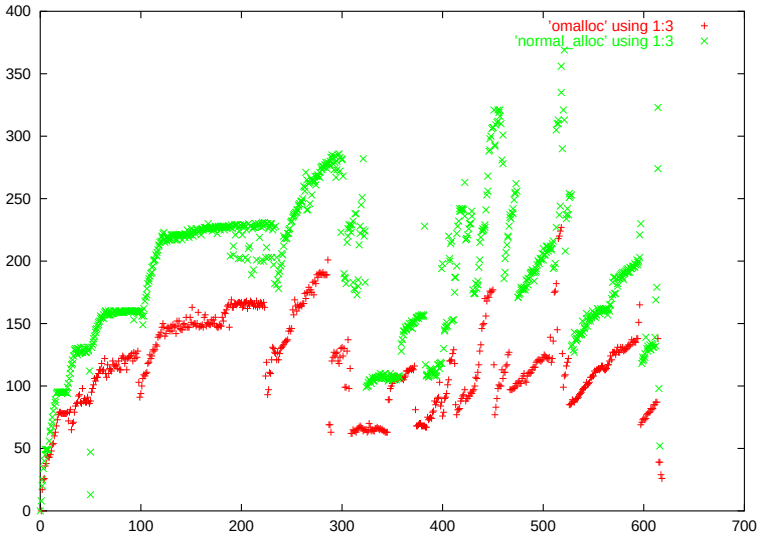


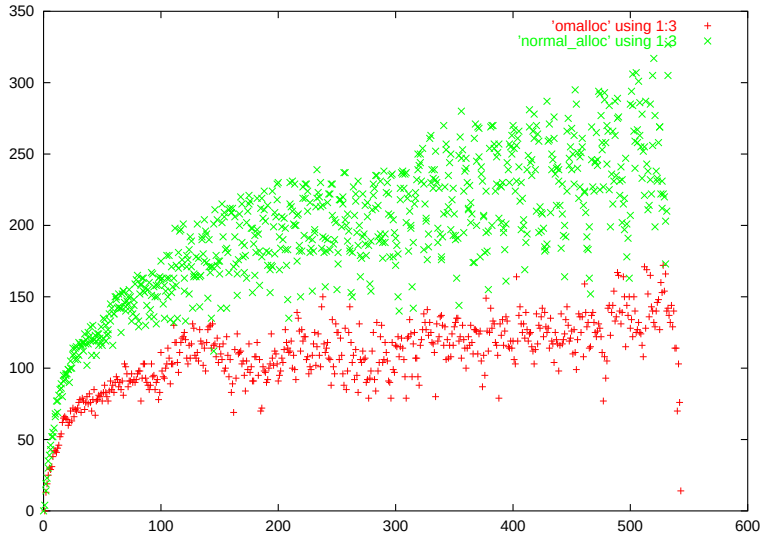


example in char 0









example in char p

example in char 0

Bit support

use a machine int (integer $\in 0..2^{31}$) for an pre-test

- > 16 variables: use 1 bit per variable:
  bit $i = 1$: exponent of x_i is non-zero
- 10..16 variables: use 2 bits per variable:
  field $i = 00$: exponent of x_i is 0
  field $i = 01$: exponent of x_i is 1
  field $i = 11$: exponent of x_i is > 1
- 9..10 variables: use 3 bits per variable
  ...
- problem: many variables
  - cover only the first 32 variables
  - 1 bit for a group of variables
  - use larger bit sets

Geo buckets

experimental implementation in Macaulay 3.0 (1998) by Yan

- lazy addition of polynomials: try to add only polynomials of the "same" length
- store polynomials as n-tupel of partial polynomials (of length k, $k^2, ..., k^n$) (experimental: best choice for k: 3 or 4)
- extract leading term from the leading terms of the partial polynomial (if needed)
- simplify a "bucket" to a normal polynomial after some operations (experimental: after 50 reductions or according to the algorithm)
- problem: content extraction/ coefficient swell

The Groebner Engine in SINGULAR

- select an algorithm according to ring, ideal properties, monomial ordering, required results
- select ordering routines for the sets (all sets are ordered, ordering can be changed (function pointer))
- some selection procedures can be changed (first appropriate element found, best found etc.)
- change of strategy during the computation is (to some extent) possible: (example: highest corner method for local zero-dimensional ideals)
- change of monomial representation (w.r.t. a degree bound)
- avoid bad choices: lazy strategy
  - during the reduction step
  - while applying the criteria

Selection strategies

- use different sets for the set of reducers (T) and the set of the Groebner basis to build (S)
- different orderings in T and S (by degree, (weighted) length, ecart, degree+ecart, ...)
- add each new element for the Groebner basis to S and T
- add "good" element occuring during the reduction to T
- exchange "bad" elements in S and T with better elements occuring during the reduction (slimgb)

"Lazy" strategies

- during the reductions: Geobuckets
- in the selection from the pairset:
  "pairset": (f,g,lm(spoly(f,g))
  (f,g,spoly(f,g))
  (f,g,partially reduced spoly(f,g))
  (postpone reduction on difficult to reduce elements, get the next one)
- try different algorithms in parallel (experimental)

Using additional information

- different criteria in the non-commutative case
- computing in R/I: a Groebner basis of a subset is known
- Hilbert function known: Hilbert driven Groebner basis computation (`stdhilb`)
- 0-dimensional ideal in a local ring: "highest corner" is known
- known syzygies: pair selection according to them (computing a free resolution) (`lres`)

Implementation in SINGULAR

- to compute a Groebner basis / standard basis: `std, slimgb, janet, frwalk, grwalk, stdhilb, fglm` are possible, summarized by `groebner`
- to compute a free resoltion of an ideal/module: `mres, nres, lres, sres` are possible, summarized by `res`
- TO DO: `groebner` should allow to pass additional information (like`std` does)
- TO DO: for internal Groebner basis computations (like `eliminate`) should use `groebner` instead of `std`
- `slimgb` and `std`