Monomial Representations for Gröbner Bases Computations*

Olaf Bachmann Hans Schönemann
Centre for Computer Algebra
Department of Mathematics
University of Kaiserslautern
Kaiserslautern, Germany
[obachman,hannes]@mathematik.uni-kl.de
http://www.mathematik.uni-kl.de/~[obachman,hannes]

Abstract

Monomial representations and operations for Gröbner bases computations are investigated from an implementation point of view. The technique of vectorized monomial operations is introduced and it is shown how it expedites computations of Gröbner bases. Furthermore, a rank-based monomial representation and comparison technique is examined and it is concluded that this technique does not yield an additional speedup over vectorized comparisons. Extensive benchmark tests with the Computer Algebra System SINGULAR are used to evaluate these concepts.

1 Introduction

The method of Gröbner bases (GB) (see, for example, [8] for an introduction) is undoubtly one of the most important and prominent success stories of the field of Computer Algebra. Starting in the 1960's, an unsolved problem has developed into an essential computational tool with a great variety of applications and more and more powerful implementations. The heart of the GB method are computations of Gröbner or Standard bases with the Buchberger algorithm or descended variants thereof (we call such computations GB computations, for short).

Unfortunately, the Buchberger algorithm and its variants have a worst case exponential time and space complexity [3]. Consequently, GB computations have limited practicality and tend to tremendously long running times and consumptions of huge amounts of memory. For these reasons, a large amount of work has been done to try to improve the (time and space) efficiency of GB computations (see, for example, [6, 7, 9]). Most of this work resulted in algorithmic improvements which led to more manageable computations for many classes of problems. Our approach to improving the efficiency of GB computations is somewhat different: instead of trying to improve algorithmic aspects of GB computations we take the algorithms more or less as they are and study efficient ways to implement them.

*Work reported herein has been supported in part by the Deutsche Forschungsgemeinschaft (DFG).

Analyzing GB computations from an implementation point of view leads to many interesting questions. For example: What is the best way to implement coefficient arithmetic? How should polynomials and monomials be represented and their operations be implemented? How should memory management be realized?

Apart from some exceptions (for example, [1]), these questions have not yet received a lot of systematic research and literature attention. Nevertheless, work in this direction can be very fruitful, as we report in this article. As a start, we concentrate on monomials – the most frequently used primitive data in GB computations – and show that only improvements in the representation of and operations on monomials can lead to significant efficiency gains of GB computations.

In Section 2 we discuss the basics of monomial operations and representations. In Section 3 we investigate vectorized monomial operations, which handle more than one exponent at a time. In section 4 we examine a rank-based representation of monomials, which encodes the exponent vector as one single integer for the purpose of expediting monomial comparisons. In both of these sections, we give extended sets of timings which show the effects of the respective concepts. Finally, we summarize our results in Section 5 and give more details about our used benchmark examples in the Appendix.

We used the Computer Algebra System (CAS) SINGULAR [10] to implement, examine and benchmark the techniques discussed in this paper. SINGULAR is CAS for polynomial computations with special emphasize on the needs of Commutative Algebra, Algebraic Geometry, and Singularity Theory. It features, among others, one of the fastest and most general implementation of GB computations and is therefore an ideal test-bed for our experiments.

2 Basic monomial operations and representations

Given an integer $n > 0$ we define the set of exponent vectors M_n by $\{\alpha = (\alpha_1, \dots, \alpha_n) | \alpha \in \mathbf{N}^n\}$. Notice that monomials usually denote terms of the form $c\,x_1^{\alpha_1} \dots x_n^{\alpha_n}$. However, in this paper we do only consider the exponent vector of a monomial and shall therefore use the words exponent vector and monomial interchangeably (i.e., we identify a monomial with its exponent vector).

We furthermore use Greek letters to denote monomials and the letter n to denote the a–priory given length of monomials (which is the number of variables in the corresponding polynomial ring).

Monomials play a central role in GB computations. In this section, we describe the basic monomial operations and discuss basic facts about monomial (resp. polynomial) representations for GB computations.

2.1 monomial operations

The basic monomial operations in GB computations are:

1. Computations of the degree (resp. weighted degree): the *degree* (resp. *weighted degree*) of a monomial α is the sum of the exponents $deg(\alpha) := \sum_{i=1}^n \alpha_i$ (resp. the weighted sum with respect to a weight vector w: $deg(\alpha) := \sum_{i=1}^n \alpha_i\, w_i$).
2. Test for divisibility: $\alpha | \beta \Leftrightarrow \forall i \in \{1..n\} : \alpha_i \leq \beta_i$.
3. Addition of two monomials: $\gamma := \alpha + \beta$ with $\forall i \in \{1..n\} : \gamma_i = \alpha_i + \beta_i$.
4. Comparison of two monomials with respect to a monomial ordering.

A *monomial ordering* $>$ (term ordering) on the set of monomials M_n is a total ordering on M_n which is compatible with the natural semigroup structure, i.e., $\alpha > \beta$ implies $\gamma + \alpha > \gamma + \beta$ for any $\gamma \in M_n$. A monomial ordering is a well-ordering if $(0, \ldots, 0)$ is the smallest monomial. We furthermore call an ordering negative if $(0, \ldots, 0)$ is the largest monomial.

Robbiano (cf.[12]) proved that any monomial ordering $>$ can be defined by a matrix $A \in GL(n, \mathbf{R})$: $\alpha > \beta \Leftrightarrow A\alpha >_{lex} A\beta$. Matrix–based descriptions of monomial orderings are very general, but have the disadvantage that their realization in an actual implementation is usually rather time–consuming. Therefore, they are not very widely used in practice.

Instead, the most frequently used descriptions of orderings have at most two defining conditions: a (possibly weighted) degree and a (normal or reverse) lexicographical comparison. We call such orderings *simple orderings.* The most important simple orderings (and their SINGULAR abbreviations) are: lexicographical (lp): used to eliminate variables and solve equations; (weighted) degree reverse lexicographical (dp): in general the most efficient one to compute a GB (as shown in [4]); (weighted) degree lexicographical (Dp); negative lexicographical ordering (ls); (weighted) negative degree reverse lexicographical (ds): also called tangent cone ordering; and (weighted) negative degree lexicographical (Ds).

For monomials $\alpha, \beta \in M_n$ let

$$\mathrm{Lex}(\alpha,\beta) = \begin{cases} 1, \text{if } \exists i : \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i > \beta_i \\ 0, \text{if } \alpha = \beta \\ -1, \text{otherwise,} \end{cases}$$

$$\mathrm{RevLex}(\alpha,\beta) = \begin{cases} 1, \text{if } \exists i : \alpha_n = \beta_n, \ldots, \alpha_{i-1} = \beta_{i-1}, \alpha_i < \beta_i \\ 0, \text{if } \alpha = \beta \\ -1, \text{otherwise,} \end{cases}$$

$$\mathrm{Deg}(\alpha,\beta) = \begin{cases} 1, \text{if } \deg(\alpha) > \deg(\beta) \\ 0, \text{if } \deg(\alpha) = \deg(\beta) \\ -1, \text{otherwise.} \end{cases}$$

Then we can define $\alpha > \beta$ for the above mentioned simple monomial orderings by:

lp:	$\mathrm{Lex}(\alpha,\beta) = 1$
ls:	$\mathrm{Lex}(\alpha,\beta) = -1$
Dp:	$\mathrm{Deg}(\alpha,\beta) = 1$, or $\mathrm{Deg}(\alpha,\beta) = 0$ and $\mathrm{Lex}(\alpha,\beta) = 1$
Ds:	$\mathrm{Deg}(\alpha,\beta) = -1$, or $\mathrm{Deg}(\alpha,\beta) = 0$ and $\mathrm{Lex}(\alpha,\beta) = 1$
dp:	$\mathrm{Deg}(\alpha,\beta) = 1$, or $\mathrm{Deg}(\alpha,\beta) = 0$ and $\mathrm{RevLex}(\alpha,\beta) = 1$
ds:	$\mathrm{Deg}(\alpha,\beta) = -1$, or $\mathrm{Deg}(\alpha,\beta) = 0$ and $\mathrm{RevLex}(\alpha,\beta) = 1$

We furthermore call a monomial ordering $>$ a *degree based monomial ordering* if $\forall\ \alpha, \beta \in M_n : \deg(\alpha) > \deg(\beta) \Rightarrow \alpha > \beta$ (e.g., Dp and dp and their weighted relatives are degree based orderings).

Due to the nature of the GB algorithm, monomial operations are by far the most frequently used primitive operations. For example, monomial comparisons are performed much more often than, and monomial additions at least as often as, arithmetic operations over the coefficient field. The number of divisibility tests depends very much on the given (input) ideal–generators, but is usually very large, as well (see also Table 1).

Nevertheless, whether or not monomial operations dominate the running time of a GB computation depends on the coefficient field of the underlying polynomial ring: monomial operations are certainly run–time dominating for finite fields with a small[1] characteristic (e.g., integers modulo a small prime number), since an arithmetic operation over these fields can usually be realized much faster than a monomial operation. However, for fields of characteristic 0 (like the rational numbers), GB computations are usually dominated by the arithmetic operations over these fields, since the time needed for these operations is proportional to the size of the coefficients which tend to grow rapidly during a GB computation.

Therefore, improvements in the efficiency of monomial operations will have less of an impact on GB computations over fields of characteristic 0.

2.2 monomial representations

As a first question, one might wonder which kind of polynomial (and, consequently, monomial) representation is best suited for GB computations. Although there are several alternatives to choose from (e.g., distributive/recursive, sparse/dense), it is generally agreed upon that efficiency considerations lead to only one real choice: a dense distributive representation. That is, a polynomial is stored as a sequence of terms where each term consists of a coefficient and an array of exponents which encodes the respective monomial. With such a representation, one has not only very efficient access to the leading monomial of a polynomial (which is the most frequently needed part of a polynomial during GB computations), but also to the single (exponent) values of a monomial.

Now, what type should the exponent values have? Efficiency considerations lead again to only one realistic choice, namely to fixed–length integers whose size is smaller or equal to the size of a machine word. While assuring the most generality, operations on and representations of arbitrary–length exponent values usually incur an intolerable slow–down of GB computations. Of course, a fixed–length exponent size restricts the range of representable exponent values. However, exponent bound restrictions are usually not very critical for GB computations: on the one hand, the (worst case) complexity of GB computations grows exponentially with the degree of the input polynomials, i.e.,

[1]say, smaller than the maximal representable machine integer, i.e. smaller than `LONG_MAX`

large exponents usually make a problem practically uncomputable. On the other hand, checks on bounds of the exponent values can be realized by degree bound checks in the outer loops of the GB computation (e.g., during the S–pair selection) which makes exponent value checks in subsequent monomial operations unnecessary.

Furthermore, the degree of a monomial is so often needed during GB computations, that, as experience has shown, it is advantageous to add an additional degree field to the monomial data structure.

As an illustration, and for later reference, we show below SINGULAR's internal Term_t data structure:

```
struct  Term_t
{
  Term_t*     next;
  void*       coef;
  long        order;
  Exponent_t  exp[1];
};
```

Following the arguments outlined above, a SINGULAR polynomial is represented as a linked list of terms, where each term consists of a coefficient (implemented as a hidden type: could be a pointer or a `long`) and a monomial. A monomial is represented by its exponent vector, together with its degree field (`order`). The data type of the exponent values (`Exponent_t`) can be set at configuration time with the restriction that it must be an integer type whose size is a multiple of the word size of the used machine[2]. The size of the `Term_t` structure is dynamically set at run–time, depending on the number of variables in the current polynomial ring.

Based on a monomial representation like SINGULAR's, the basic monomial operations are "traditionally" implemented by straightforward realizations of their definitions. Only monomial additions need also to update the `order` field of the result monomial, which is simply accomplished by adding the degree values of the operands (since the values of the `order`–field are additive).

3 Vectorized monomial operations

The main idea behind what we call *vectorized monomial operations* is the following: provided that the size of the machine word is a multiple (say, m) of the size of one exponent, we perform monomial operations on machine words, instead of directly on exponents. By doing so, we process a vector of m exponents with word operations only, thereby reducing the length of the inner loops of the monomial operations and avoiding non–aligned accesses of exponents.

The PoSSo library [11] already uses parts of this idea to speed up monomial additions, assignments, and tests for equalities, but has not (yet?) carried these ideas over to the much more time-critical monomial comparisons and divisibility tests.

The details of this technique are based on the following lemma whose proof is straightforward:

Lemma 1: Let $s_e, m \in \mathbf{N}$, and $\alpha = (\alpha_0, \dots \alpha_{m-1}), \beta = (\beta_0, \dots \beta_{m-1}) \in M_m$ with $\alpha_i, \beta_i < 2^{s_e - 1}$, $s_w = s_e\, m$ and $a, b, \widehat{a}, \widehat{b}, \gamma_0, \dots \gamma_{m-1}, \delta_0 \dots \delta_{m-1} \in \mathbf{N}$ with $\gamma_i, \delta_i < 2^{s_e}$ s.t.

$$\begin{aligned} a &= \alpha_0 + \alpha_1 2^{s_e} + \dots + \alpha_{m-1} 2^{(m-1)s_e}, \\ b &= \beta_0 + \beta_1 2^{s_e} + \dots + \beta_{m-1} 2^{(m-1)s_e} \\ \widehat{a} &= \alpha_{m-1} + \alpha_{m-2} 2^{s_e} + \dots + \alpha_0 2^{(m-1)s_e}, \\ \widehat{b} &= \beta_{m-1} + \beta_{m-2} 2^{s_e} + \dots + \beta_0 2^{(m-1)s_e}, \\ b + a \pmod{2^{s_w}} &= \gamma_0 + \gamma_1 2^{s_e} + \dots + \gamma_{m-1} 2^{(m-1)s_e}, \\ b - a \pmod{2^{s_w}} &= \delta_0 + \delta_1 2^{s_e} + \dots + \delta_{m-1} 2^{(m-1)s_e}. \end{aligned}$$

Then the following holds:

$$\begin{array}{rcll} b - a \pmod{2^{s_w}} < 2^{s_w - 1} & \text{iff} & \mathrm{Lex}(\alpha, \beta) = 1 & (1) \\ \widehat{b} - \widehat{a} \pmod{2^{s_w}} < 2^{s_w - 1} & \text{iff} & \mathrm{RevLex}(\alpha, \beta) = -1 & (2) \\ (\gamma_0, \dots, \gamma_{m-1}) & = & \alpha + \beta & (3) \\ \forall\, 0 \le i < m : \delta_i < 2^{s_e - 1} & \text{iff} & \alpha \mid \beta & (4) \end{array}$$

Lemma 1 shows how the monomial operations on the right–hand sides of (1) – (4) can be "vectorized", such that, on a 2's complement machine with word–size s_w, the checks (resp. operations) on the left hand sides of (1) – (4) can be performed in at most three single machine instructions (see the source code examples below for further details).

However, before Lemma 1 can be applied in an actual implementation, some technical difficulties have to be overcome:

Firstly, stricter bounds on the values of the single exponents have to be assured (i.e., the exponent values need to be less than $2^{s_e - 1}$ instead of 2^{s_e}).

Secondly, the condition $s_w = s_e\, m$ implies that the total length of the exponent vector has to be a multiple of the word–size which requires that $(-n) \pmod m$ "unnecessary" exponents (whose value is set to and always kept at 0) might have to be added to the exponent vector .

Thirdly, and most importantly, the order and arrangement of the exponents within the exponent vector has to be adjusted, depending on the monomial ordering and on the endianess of the used machine. On big–endian machines, the order of the exponents has to be reversed for reverse lexicographical orderings whereas on little–endian machines, the order of the exponents has to be reversed for lexicographical orderings. In practice, this fact can be hidden behind appropriate macro (or inline) definitions for accessing single exponents. In our implementation, we used a global variable called `pVarOffSet` and implemented the exponent access macro as follows:

```
#define pGetExp(p, i)   \
        p->exp[(pVarOffSet ? pVarOffSet - i : i)]
```

Provided that n is the number of variables of the current ring then we set the value of `pVarOffSet` as follows:

	type of machine	
type of ordering	big–endian	little–endian
lexicographical	0	$n-1$
reverse lexicographical	$n-1$	0

Some source code fragments can probably explain it best: Figure 1 shows (somewhat simplified) versions of our implementation of the vectorized monomial operations. Some explanatory remarks are in order:

`LexSgn` and `OrdSgn` (used in `MonComp`) are global variables which are used to appropriately manipulate the return value of the comparison routine and whose values are set as follows:

	lp	ls	Dp	Ds	dp	ds
`OrdSgn`	1	-1	1	-1	1	-1
`LexSgn`	1	-1	1	1	-1	-1

[2] By default, we declare `Exponent_t` to be of type `short` which seems to be a good alternative between flexibility and efficiency.

```
inline long MonComp(Term_t* a, Term_t* b)
{ // return   0, if a = b
  //        > 0, if a > b; < 0, if a < b
  long d = a->order - b->order; //check degree
  if (d) return d*OrdSgn;
#ifdef WORDS_BIGENDIAN        //check exponents
   for (long i = 0; i<n_w; i++)
#else
   for (long i = n_w -1; i; i--)
#endif
   {
     d=((long*)a->exp)[i]-((long*)b->exp)[i];
     if (d) return d*LexSgn;
   }
   return 0;
}
inline void MonAdd(Term_t* c, Term_t* a, Term_t* b)
{ // Set c = a + b
   for (long i=0; i<n_w; i++)
     ((long*)c->exp)[i] =
        ((long*)a->exp)[i] + ((long*)b->exp)[i];
  c->order = a->order + b->order // update order
}
inline bool MonDivBy(Term_t* a, Term_t* b)
{ // return true,  if a divides b
  //        false, otherwise
#if SIZEOF_LONG == 4
  #if SIZEOF_EXPONENT == 1
    #define DIV_MASK 0x80808080
  #else // SIZEOF_EXPONENT == 2
    #define DIV_MASK 0x80008000
  #endif
#else // now assume SIZEOF_LONG == 8
  // define DIV_MASK similarly
#endif
  for (long i=0; i<n_w; i++)
  {
    long d = ((long*) b->exp)[i]
             - ((long*) a->exp)[i];
    if (d & DIV_MASK) return false;
  }
  return true;
}
```

Figure 1: Vectorized monomial operations in SINGULAR

Together with the above described order-dependent arrangement of the exponents within the exponent vector, these variables allow us to reduce monomial comparisons for *all* simple monomial orderings to *one* single routine.

`n_w` is a global variable denoting the length of the exponent vectors in machine words (i.e., if s_w is the size of a machine word, s_e the size of an exponent, n the number of variables, then $n_w = \lceil n\, s_e/s_w \rceil$).

Notice that `MonAdd` works on three monomials and it is most often used as a "hidden" initializer (or, assignment), since monomial additions are the "natural source" of most new monomials.

Our actual implementation contains various, tediously to describe, but more or less obvious, optimizations (like loop unrolling, use of pointer arithmetic, replacement of multiplications by bit operations, etc). We apply, furthermore, the idea of "vectorized operations" to monomial assignments (i.e. copies) and equality tests, too. However, we shall not describe here the details of these routines, since their implementation is more or less obvious and they have less of an impact on the running time than the basic monomial operations.

So much for the theory, now let us look at some actual timings: Table 1 shows various timings illustrating the effects of the vectorized monomial operations described in this section. In the first column, we list the used examples — details about those can be found in the Appendix. All GB computations were done using the degree reverse lexicographical ordering (dp) over the coefficient field $\mathbf{Z}/32003$. We measured the following times (all in seconds):

$t_{1.0}$ Running time of SINGULAR version 1.0 which uses the straightforward implementation of monomial operations.

$t_{n,s}$, $t_{n,c}$ Running time of SINGULAR *without* vectorized monomial operations and exponents of type `short` (for $t_{n,s}$) and `char` (for $t_{n,c}$)[3].

t_s, t_c Running time of SINGULAR *with* vectorized monomial operations and exponents of type `short` (for t_s) and `char` (for t_c).

%mon,%comp,%div,%add Percentage of the running time (of $t_{n,s}$ runs) spent in basic monomial operations, monomial comparisons, divisibility tests, and additions, respectively (i.e., %mon = %comp + %div + %add)

Before evaluating these numbers, we should like to mention that running the same tests on different machines and/or with different simple monomial orderings leads to a similar picture (see also [2] for more timings).

Now, what do these numbers tell us?

Firstly, they support our assertion that for GB computations over finite fields of small characteristic, the running time is largely dominated by basic monomial operations (see column %mon). However, in which of the (three) basic monomial operations the most time is spent varies very much from example to example (compare, e.g., line "gonnet" with line "ecyclic 6").

Secondly, and most importantly: the impact of the vectorized monomial operations is quite substantial (see columns $t_{n,s}/t_s$, $t_{n,c}/t_c$ which show the speedup gained by vectorized operations). As expected, the larger the ratio $m = s_w/s_e$ (i.e, the number of exponents packed in one machine word), the more speedup is usually gained (compare column $t_{n,s}/t_s$ and $t_{n,c}/t_c$). However, notice that we cannot conclude a direct correlation between the percentage of time spent in monomial operations and the efficiency gains of vectorized operations. This is due to the fact that the number of inner loop iterations (and, hence, reduction of inner loop iterations) for comparisons and divisibility tests is not constant, but depends on the input monomials.

Thirdly: as we would expect, the more exponents are encoded into one machine word, the faster the GB computation is accomplished (see the t_s/t_c column). This has two main reasons: first, more exponents are handled by one machine operation; and second, less memory is used, and therefore, the memory performance is increased (e.g., the number of cache misses is reduced). However, we also need to keep in mind that the more exponent are encoded into one word, the smaller are the upper bounds on the value of a single exponents. This is especially crucial for computations with

[3]Using our previously introduced terminology, we have $s_e = 2$ for $t_{n,s}$ (resp. 4 for $t_{n,c}$); and $m = s_w/s_e = 2$ (resp. 4 for $t_{n,c}$) on a 32–bit machine (like the HP C160, or Pentium Pro) and $m = 4$ for $t_{n,s}$ (resp. 8 for $t_{n,c}$) on a 64–bit machine (like the DEC Alpha).

Example	$t_{n,s}$	%mon	%comp	%div	%add	$\frac{t_{n,s}}{t_s}$	$\frac{t_{n,c}}{t_c}$	$\frac{t_s}{t_c}$	$\frac{t_{1.0}}{t_{n,s}}$	$\frac{t_{1.0}}{t_c}$
ecyclic 6	3.9	79.4	68.9	1.6	8.8	1.7	2.0	1.3	1.3	2.8
homog cyclic 7	125.8	72.5	33.6	11.8	27.1	1.5	1.9	1.3	1.3	2.6
homog cyclic 6	0.9	69.5	30.5	13.3	25.8	1.6	1.9	1.3	1.3	2.6
katsura 7	8.8	71.5	41.5	3.7	26.3	1.6	1.9	1.2	1.3	2.6
rcyclic 15	53.2	76.2	29.6	26.0	20.6	1.6	2.0	1.4	1.2	2.5
rcyclic 14	22.9	76.0	28.1	28.5	19.4	1.5	1.9	1.3	1.2	2.5
rcyclic 16	106.4	78.2	26.5	30.7	21.0	1.6	2.0	1.4	1.2	2.5
rcyclic 19	740.0	80.3	29.5	29.9	20.9	1.5	1.9	1.4	1.2	2.5
rcyclic 18	396.5	80.0	29.4	29.2	21.4	1.5	1.9	1.4	1.2	2.4
rcyclic 13	10.2	74.8	27.6	26.6	20.7	1.5	1.8	1.3	1.2	2.4
rcyclic 12	4.3	74.7	28.4	29.8	16.5	1.6	1.9	1.2	1.2	2.4
rcyclic 11	1.6	67.4	22.9	24.7	19.8	1.6	1.8	1.2	1.2	2.4
bjork 8	4.0	67.8	32.0	13.2	22.5	1.5	1.8	1.2	1.3	2.3
rcyclic 17	210.9	78.2	26.5	30.3	21.4	1.5	1.8	1.3	1.2	2.3
gerhard 1	1.7	68.7	48.1	4.5	16.0	1.3	1.5	1.2	1.5	2.3
katsura 8	94.3	73.1	45.1	3.5	24.5	1.5	1.7	1.2	1.3	2.3
homog 2mat3	126.0	81.9	77.5	1.9	2.5	1.3	1.5	1.4	1.2	2.2
cyclic 7	190.0	66.9	31.0	11.2	24.7	1.4	1.8	1.3	1.2	2.2
rcyclic 10	0.5	53.7	16.2	20.0	17.5	1.6	1.7	1.1	1.2	2.2
cyclic 6	0.7	61.6	26.8	16.3	18.6	1.4	1.7	1.2	1.2	2.1
homog gonnet	125.4	76.8	43.0	31.5	2.2	1.5	1.7	1.2	1.1	2.1
gerhard 2	13.8	72.3	54.8	3.2	14.3	1.3	1.4	1.1	1.4	2.1
homog alex 2	12.9	71.7	58.2	4.8	8.8	1.3	1.4	1.1	1.4	2.0
schwarz 11	131.6	65.4	30.6	18.4	16.5	1.3	1.6	1.3	1.2	2.0
2mat3	123.1	80.2	75.7	1.9	2.6	1.2	1.4	1.4	1.2	2.0
schwarz 10	21.9	62.5	28.2	18.4	15.9	1.3	1.5	1.2	1.2	2.0
cohn2	9.7	64.4	29.7	4.1	30.6	1.4	1.5	1.1	1.2	1.9
gerhard 3	24.4	67.1	45.5	13.1	8.6	1.3	1.3	1.1	1.3	1.8
symmetric 6	60.6	68.0	25.6	14.5	28.0	1.4	1.4	1.1	1.2	1.8
schwarz 9	3.3	55.5	23.0	20.0	12.5	1.3	1.4	1.1	1.2	1.8
alex 2	7.1	63.6	52.1	3.3	8.2	1.3	1.4	1.1	1.2	1.8
schwarz 8	0.6	50.6	24.7	14.6	11.2	1.3	1.5	1.1	1.1	1.7
ecyclic 7	1076.4	86.4	84.2	0.3	2.0	1.2	1.4	1.2	1.1	1.6
alex 3	2.0	65.9	53.5	7.7	4.7	1.3	1.3	1.1	1.2	1.6
homog alex 3	2.0	66.2	47.2	14.8	4.1	1.2	1.2	1.1	1.3	1.6
gonnet	1.2	43.8	6.2	34.2	3.4	1.1	1.3	1.2	1.1	1.5
averages										
Pentium Pro	103.3	69.8	38.4	15.6	15.8	1.4	1.6	1.2	1.2	2.2
HP C160	125.4	69.0	41.7	17.9	9.5	1.2	1.5	1.3	1.8	2.9
DEC Alpha	221.4	69.8	37.7	16.1	16.0	1.7	1.7	1.2	1.1	2.3

Table 1: Detailed timings for vectorized monomial operations: SINGULAR compiled with gcc version 2.7.2 for a Pentium Pro 200 (32 bit, little–endian) running Linux and average of timings for a HP C160 (32 bit, big–endian) running HP–UX 10.20 and a DEC Alpha (64 bit, little endian) running Linux.

`char` exponents, since these require that an exponent may not be larger than 127. Ideally, we would like to "dynamically" switch from one exponent size to the next larger one, whenever it becomes necessary. With SINGULAR, we cannot do this yet, at the moment, but we intend to implement this feature in one of the next major upgrades.

Fourthly: we would like to point out that the only difference between the SINGULAR versions resulting in $t_{1.0}$ and $t_{n,s}$ is that monomial comparisons for the different orderings are not reduced to one inlined routine but instead are realized by function pointers (and, therefore, each monomial comparisons requires a function call). Hence, column $t_{1.0}/t_{n,s}$ shows that already the in–place realization of monomial comparisons results in a considerable efficiency gain which by far compensates the additional overhead incurred by the additional indirection for accessing one (single) exponent value (i.e., the overhead incurred by the `pGetExp` macro shown above).

Last, but not least, the combination of all these factors leads to a rather significant improvement of our newly released SINGULAR version 1.2 over the "old" SINGULAR version 1.0 (see column $t_{1.0}/t_c$).

4 Rank-based monomial representation

A different monomial representation is what we call the *rank-based representation*. This monomial representations was, to the best of our knowledge, first developed for and used in the CAS Macaulay [5]. The main idea behind this monomial representation stems from the following property of a degree based ordering $>$ on M_n:

$$\forall m \in M_n : \#\{m' \in M_n : 1 \leq m' \leq m\} < \infty \qquad (5)$$

Therefore, it is possible to bijectively enumerate the monomials in the order given by the monomial ordering and to represent an entire monomial by just a single integer, which we call the *rank* of a monomial.

The bijective maps between the monomials and the integers depend on the used monomial ordering: for the degree reverse lexicographic ordering it is given by the following lemma:

Lemma 2: Let $\alpha \in M_n$ and $r_{dp} : M_n \to \mathbf{N}$ given by

$$r_{dp}(\alpha) := \sum_{i=1}^{n} \binom{\sum_{j=1}^{i} \alpha_j + i - 1}{i}. \qquad (6)$$

Then the following holds:

$$\forall \alpha, \beta \in M_n : r_{dp}(\alpha) > r_{dp}(\beta) \Leftrightarrow \alpha >_{dp} \beta$$
$$\forall \alpha, \beta \in M_n : r_{dp}(\alpha) = r_{dp}(\beta) \Leftrightarrow \alpha = \beta.$$

A similar bijective map can be given for the ordering Dp (degree lexicographic). For negative orderings like Ds or ds the same idea can be exploited but it is necessary to change the signs since the zero monomial is, with respect to these orderings, the largest, not the smallest monomial.

Using the rank-based representation for monomials, monomial comparisons can very easily and efficiently be realized by simply comparing their corresponding ranks (we also call such monomial comparisons simply *rank-based comparisons*). Furthermore, a pure rank-based polynomial representation is very compact, i.e. requires an almost minimal amount of memory. Unfortunately, we have to pay for these advantages with the following difficulties:

(i) Monomial additions and divisibility tests can not be easily accomplished, since the bijective enumeration maps are neither additive nor do they allow any direct conclusions about monomial divisibility.

(ii) An unrestricted realization of the enumeration maps would require that arbitrary precision integers are used which would in turn result in considerable performance losses. On the other hand, restricting the range of the enumeration maps to machine integers, imposes a limit on the largest representable monomial (which we also call the *maximal integer monomial*). Clearly, this limit depends on the word–length of the machine and on the number of variables of the polynomial ring. For example, using a 32–bit integer for the realization of (6) restricts the degree of the maximal integer monomials as shown in the following table:

variables	3	4	5	7	10	15	20	25
max. deg	2340	471	186	66	31	17	12	9

(iii) Non–degree orderings (like the lexicographical or negative lexicographical ordering) do not admit a bijective enumeration map which is compatible with the monomial ordering, since they do not enjoy (5).

Problem (i) is overcome in the Macaulay system by keeping two representations of the the leading monomial of a polynomial (the integer representation and the exponent vector representation) and by computing the inverse of the enumeration map for the non–leading monomials of a polynomial.

Problem (ii) is overcome by restricting the input polynomials to being homogeneous and by Macaulay's "infamous" degree bound messages and enforcement mechanisms.

Problem (iii) is simply ignored by Macaulay — no computations with non–degree orderings are possible.

In Singular, we choose a slightly different way to overcome problems (i) – (iii): first, we always keep the exponent vector representation of a monomial. And, second, if a monomial is smaller than the maximal integer monomial, we store its rank in the negative range of the `order` field (i.e., `a->order + INT_MAX` yields the rank of the monomial `a`). If, on the other hand, a monomial is larger than the maximal integer monomial, then its `order` field contains (as before) the (positive) degree of the monomial.

Based on such a data representation, monomial comparisons can simply be accomplished as follows:

```
inline long MonComp(Term_t* a, Term_t* b)
{
  if (a->order != b->order)
    return (a->order - b->order);
  // a->order==b->order: are we already done?
  if (a->order < 0) return 0;  // monom's are equal
  // now compare using the exponent values,
  // assuming the degrees are equal ...
}
```

Notice that we do not need to distinguish between the two different interpretations of the `order` field: if we compare a monomial which is smaller than the maximal integer monomial (`order` value is negative) with one that is larger (`order` value is positive), then the different signs of the `order` fields result in an immediate and correct return of the monomial comparison. Therefore, only a sign–check (for the case of equal `order` fields) needs to be added to the previously given monomial comparison routines to have them take advantage of ranks.

Since we always keep the exponent vector representation alongside with the ranks, monomial divisibility tests and additions can be accomplished as usual, except that we need to recompute and set the `order` field at the end of the monomial addition routine. For monomials smaller than the maximal integer monomial, this basically amounts to computations of the respective ranks. We accomplish the latter by a simple implementation of (6) using a pre–computed table of binomial coefficients.

As with vectorized monomial additions, overflow or degree checks can largely be moved to the outer loops of the GB computations so that we can almost always avoid the costly checks on whether or not the sum of two monomials is smaller than the maximal integer monomial.

As a summary, we can conclude that with a relatively small effort, it is possible to extend a "traditional" exponent vector–based monomial representation so that it can fully take advantage of ranks while, at the same time, the limitations and efficiency bottlenecks of the Macaulay system (degree bounds, no support for non–degree orderings, repeated inverse rank computations) can be avoided.

But now again, let us examine some timings (Table 2) to see how much all that buys us in practice: we again used the examples of the Appendix and performed all GB computations using the degree reverse lexicographical ordering (dp) over the coefficient field $\mathbf{Z}/32003$. We obtained timings for three basic Singular configurations: `char` and `short` exponents with vectorized monomial operations, and `short` exponents with "traditional" (i.e., not vectorized) monomial operations. For each of these "conventional" configurations, we also obtained a Singular version which uses rank-based comparisons and measured the following:

$t_<, t_{r,<}$: the time spent in monomial comparisons, only ($t_<$–conventional comparisons, $t_{r,<}$–rank-based comparisons).

%R, %comp: the percentage of time spent in rank computations (%R) and in conventional monomial comparisons (%comp).

t, t_r: the overall running time for the GB computation (t–conventional configuration, t_r–rank-based configuration). For the conventional configuration, these timings are the same as those of Table 1.

And again, let us see what we can conclude from these experiments:

Firstly, rank-based comparisons are realized substantially faster than traditional and vectorized monomial comparisons (see the $t_</t_{r,<}$ columns), except in such examples, where a majority of the monomials is larger than the maximal integer monomial (examples marked with a $*$).

Secondly, the time spent for rank computations during monomial additions is significant, and cannot be neglected (see the %R columns).

Thirdly, whether or not the time saved by rank-based comparisons pays off against the additional time spent for

exponent type	char				short				short			
monomial operations	vectorized				vectorized				not vectorized			
Example	$\frac{t_<}{t_{r,<}}$	%R	%comp	$\frac{t}{t_r}$	$\frac{t_<}{t_{r,<}}$	%R	%comp	$\frac{t}{t_r}$	$\frac{t_<}{t_{r,<}}$	%R	%comp	$\frac{t}{t_r}$
2mat3	1.3	7.6	69.0	1.1	1.7	6.5	74.4	1.4	1.9	5.0	76.3	1.6
homog gonnet	1.3	6.7	34.5	1.0	1.8	4.7	38.0	1.1	2.7	4.8	43.9	1.3
homog alex 2	1.3	9.7	51.1	0.9	1.4	8.5	51.2	1.0	1.8	7.8	57.4	1.2
alex 2	1.3	7.9	52.1	0.9	1.1	7.4	49.0	1.0	1.6	7.5	56.8	1.2
alex 3	1.3	5.2	47.8	1.0	1.0	2.9	49.1	0.9	1.4	2.8	53.7	1.2
gerhard 1	1.4	15.6	41.9	0.8	1.2	12.8	41.5	0.9	2.0	13.4	48.1	1.2
homog 2mat3	1.2	7.2	70.6	1.0	1.3	4.6	75.4	1.0	1.6	4.1	78.2	1.2
katsura 8	1.4	26.3	39.6	0.8	1.5	25.0	38.5	0.9	2.2	21.6	44.6	1.2
gerhard 2	1.1	10.4	47.4	0.9	1.3	12.6	48.4	0.9	1.7	10.6	54.3	1.1
katsura 7	1.2	28.3	32.7	0.7	1.4	23.0	30.2	0.8	2.1	22.5	38.2	1.1
gerhard 3	1.3	8.0	40.0	0.9	1.5	7.6	43.4	0.9	1.7	7.6	45.4	1.1
homog alex 3	1.2	5.0	39.1	1.0	1.2	5.8	42.0	1.0	1.4	5.8	43.1	1.1
cohn2	1.4	19.6	22.2	0.8	1.3	19.8	23.8	0.9	2.1	19.6	29.9	1.1
bjork 8	1.2	23.6	21.5	0.7	1.6	23.2	21.1	0.8	2.3	21.8	30.1	1.1
cyclic 7	1.3	25.0	22.0	0.8	1.6	22.3	23.8	0.8	2.6	19.6	31.4	1.1
schwarz 11	1.2	20.1	24.4	0.8	1.6	15.9	25.4	0.9	2.2	14.7	30.1	1.1
symmetric 6	1.3	22.1	18.3	0.8	1.5	20.0	19.1	0.9	2.4	18.0	26.9	1.1
schwarz 10	1.2	19.8	20.1	0.8	1.5	15.6	21.1	0.9	2.7	15.5	28.5	1.1
homog cyclic 7	1.4	28.6	24.8	0.7	1.6	24.3	24.2	0.8	2.8	22.0	33.7	1.1
rcyclic 13	1.5	31.1	21.9	0.7	1.7	24.5	19.8	0.9	2.9	19.2	26.1	1.0
rcyclic 14	1.4	28.5	19.1	0.7	2.0	20.3	19.5	0.9	3.5	20.6	27.7	1.0
schwarz 9	1.2	21.2	15.7	0.8	1.4	15.3	18.7	0.9	2.0	16.2	22.4	1.0
rcyclic 12	2.0	25.1	24.2	0.7	1.3	20.1	16.5	0.9	3.1	20.3	24.0	1.0
rcyclic 11	1.4	26.2	21.1	0.7	1.3	23.1	19.1	0.8	3.7	18.7	30.1	1.0
rcyclic 15	1.3	26.2	20.9	0.7	1.5	19.3	20.4	0.9	2.2	17.2	30.3	1.0
homog cyclic 6	1.1	27.3	15.2	0.6	3.1	19.0	32.9	0.8	3.3	19.3	28.0	1.0
gonnet	1.4	7.9	5.6	0.9	1.3	9.3	4.0	0.9	2.3	6.6	9.2	1.0
ecyclic 7*	1.0	3.2	81.4	1.0	1.0	1.7	82.3	1.0	1.0	1.3	84.5	0.9
rcyclic 16*	1.1	17.6	19.4	0.8	1.0	11.8	18.4	0.9	1.0	9.6	26.6	0.9
rcyclic 17*	0.9	15.9	20.5	0.8	0.9	10.8	19.1	0.9	0.9	8.2	27.1	0.9
rcyclic 18*	0.9	15.9	18.2	0.8	1.0	10.0	19.6	0.9	1.0	7.8	29.5	0.9
rcyclic 19*	1.0	16.0	20.4	0.8	1.0	10.1	19.9	0.9	1.1	7.8	29.7	0.9
ecyclic 6*	0.9	18.0	55.3	0.8	1.0	10.9	51.4	0.8	1.2	8.6	66.3	0.8
					averages							
Pentium Pro	1.3	17.1	33.6	0.8	1.4	13.8	33.8	0.9	2.1	12.6	40.3	1.1
HP C160	1.4	9.1	36.6	0.8	1.6	8.4	39.0	0.9	1.7	9.1	41.6	1.1
DEC Alpha	1.8	11.8	33.1	0.8	1.5	11.7	31.5	0.8	2.4	10.2	35.5	1.0

Table 2: Detailed timings for rank-based monomial operations – same settings as in Table 1.

rank computations, depends on the relative efficiency of the "conventional" monomial comparisons and on the characteristics of the example — where the percentage of time spent in monomial comparisons seems to be the most important factor (compare the %comp and t/t_r columns).

Finally, evaluating the t/t_r columns, we can conclude good news and bad news: the good news is that rank-based comparisons, by and large, lead to an overall speedup when used in combination with traditional (i.e., not vectorized) monomial operations. That is, our results make a strong case for such a usage of the rank-based representation. The bad news is, that rank-based comparisons by and large do *not* lead to an overall speedup when used in combination with vectorized monomial operations. Our results show that the more efficient vectorized monomial operations result in smaller speedups of the rank-based comparisons, i.e., speedups which are generally too small to outweigh the additional cost of computing ranks.

5 Summary

In this paper, we examined monomial representations and operations for GB computations. In section 2 we argued that (i) the canonical polynomial representation is a linked list of terms where each term consists of a coefficient, a degree field and an array of fixed-length exponent values, and, that (ii) the canonical implementation of monomial operations is simply the straight-forward realization of their definitions.

In section 3 we introduced the idea of vectorized monomial operations: Instead of working on one exponent at a time, we work with as many exponents at a time, as fit into one machine word (say, m). Vectorized monomial operations can be applied to monomial additions, divisibility tests, assignments, and comparisons w.r.t. simple monomial orderings; and they have the following advantages:

1. The maximal number of iterations in the inner loops of the monomial operations is cut by approximately a factor of m.
2. Non–aligned accesses of single exponents are avoided.
3. Monomial comparisons for the different monomial orderings are reduced to *one* routine which does not need to explicitly distinguish the different simple monomial orderings.

We have illustrated with our timings that these advantages directly translate in significant overall speedups of GB computations (which are in the range of 1.5 to 3 in our implementation).

The concept of vectorized monomial comparisons could be extended to non-simple orderings (like elimination, block, or even matrix orderings) based on the following idea: Let A be a matrix describing an arbitrary monomial ordering. Besides representing a monomial by a vector α of exponents we keep an additional vector $A\alpha$ and accomplish divisibility tests as before, comparisons by vectorized lexicographical comparisons of the $A\alpha$ vectors and additions by vectorized additions of both, α and $A\alpha$. However, the practicality of this idea remains to be investigated.

In section 4 we examined a rank-based representations of monomials. For degree-based orderings, we can uniquely associate an order-preserving rank with each monomial, and use this rank to reduce a monomial comparisons to just one integer comparison. Unfortunately, our timings indicate that this technique does not generally lead to efficiency gains over vectorized monomial operations, since the time saved in monomial comparisons is usually not enough to make up for the additional time spent in rank computations.

In other words, we highly recommend the usage of vectorized monomial operations in GB computations, whereas we can not recommend rank-based monomial representations and comparisons.

Although parts of our conclusions are based on the timings obtained from a Singular implementation of these techniques, they are system-independent in their nature and should therefor apply in similar ways to other GB implementations.

With our results we have shown that it can be rewarding to systematically examine implementational aspects of GB computations. We hope to continue along these lines and, especially, to get more programmers of the various GB systems to join in such discussions and investigations.

6 Acknowledgments

We should like to thank all members of the Singular developer team for their support of our work and for giving us some time off from our day-to-day developer tasks. We furthermore are grateful to F. O. Schreyer and his colleagues for placing their DEC Alpha machines at our disposal.

References

[1] Attardi, G., and Flagella, T. Memory management in the PoSSo solver. *J. Symbolic Computation 21*, 3 (March 1996), 293–312.

[2] Bachmann, O., and Schönemann, H. Monomial operations for computations of Gröbner bases. In *Reports On Computer Algebra*, no. 18. Centre for Computer Algebra, University of Kaiserslautern, January 1998. Also available from `http://www.mathematik.uni-kl.de/~zca/`

[3] Bayer, D., and Mumford, D. What can be computed in algebraic geometry? Cambridge University Press, Cambridge, 1993, pp. 1–48.

[4] Bayer, D., and Stillman, M. A theorem on refining division orders by the revers lexicographic order. *Duke J. Math. 55* (1987), 321–328.

[5] Bayer, D., and Stillman, M. *Macaulay Classic*: A computer algebra system for algebraic geometry, 1993. Available via anonymous ftp from `ftp://math.harvard.edu/Macaulay`.

[6] Buchberger, B. Groebner bases: an algorithmic method in polynomial ideal theory. D. Reidel Publishing Company, 1985, pp. 184–232.

[7] Caboara, M., De Dominicis, G., and Robbiano, L. Multigraded Hilbert Functions and Buchberger Algorithm. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'96)* (Zurich, Switzerland, July 1996), ACM Press, pp. 72–78.

[8] Cox, D., Little, J., and O'Shea, D. *Ideals, Varieties, and Algorithms*, 2nd ed. Springer-Verlag, 1997.

[9] Giovini, A., Mora, T., Niesi, G., Robbiano, L., and Traverso, C. One sugar cube, please or Selection strategies in Buchberger algorithms. In *Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computations, ISSAC'91* (1991), S. Watt, Ed., ACM press, pp. 49–54.

[10] Greuel, G.-M., Pfister, G., and Schönemann, H. Singular Reference Manual. In *Reports On Computer Algebra*, no. 12. Centre for Computer Algebra, University of Kaiserslautern, May 1997. `http://www.mathematik.uni-kl.de/~zca/Singular`

[11] *PoSSo*: Polynomial System Solving, 1995. `http://posso.dm.unipi.it/`.

[12] Robbiano, L. Term Orderings on the Polynomial Ring. In *Proceedings of EUROCAL 85, Lecture Notes in Computer Science* **204** (1985), pp. 513–517.

Appendix A: Benchmark examples

Table 3 lists a summary of their properties (see [2] for a complete description and references to their origins): column *#vars* shows the number of occurring variables, column *#polys* the number of elements (polynomials), column *homog* gives the homogeneity, and Deg_s shows the maximal degree of the input sets. $\mathrm{Deg}_{\max}$ gives the maximal degree occurring during the GB computation w.r.t. the degree reverse lexicographical ordering.

Example	#vars	#polys	homog	Deg_s	$\mathrm{Deg}_{\max}$
ecyclic 7	43	7	no	7	27
ecyclic 6	31	6	no	6	17
rcyclic $10 \le i \le 19$	i	$i-1$	no	$i-1$	$\lfloor i/2 \rfloor * 2 + 4$
homog 2mat3	19	8	yes	4	13
2mat3	18	8	no	4	13
homog gonnet	18	19	yes	2	11
gonnet	17	19	no	2	11
schwarz 11	11	11	no	2	13
schwarz 10	10	10	no	2	12
katsura 8	9	9	no	2	10
katsura 7	8	8	no	2	9
bjork 8	8	9	no	8	18
homog cyclic 7	8	7	yes	7	20
cyclic 7	7	7	no	7	27
homog cyclic 6	7	6	yes	6	17
cyclic 6	6	6	no	6	17
homog alex 3	6	4	yes	14	51
alex 3	5	4	no	14	51
gerhard 1	5	3	yes	10	32
symmetric 6	5	5	yes	6	23
homog alex 2	5	3	yes	12	40
cohn2	4	4	no	6	20
alex 2	4	3	no	12	33
gerhard 2	4	3	yes	9	44
gerhard 3	4	3	yes	23	81

Table 3: Summary of properties of benchmark examples