The Projective Noether Maple Package: Computing the Dimension of a Projective Variety

MARC GIUSTI[†], KLEMENS HÄGELE[‡], GRÉGOIRE LECERF[†], JOËL MARCHAND[†], BRUNO SALVY[*]

[†]*Laboratoire GAGE, École polytechnique, F-91128 Palaiseau, France*
[‡]*Matemáticas, Universidad de Cantabria, E-39071 Santander, Spain*
[*]*Projet ALGO, INRIA Rocquencourt, F-78153 Le Chesnay Cedex, France*

(Received 21 February 2000)

Recent theoretical advances in elimination theory use straight-line programs as a data-structure to represent multivariate polynomials. We present here the *Projective Noether Package* which is a Maple implementation of one of these new algorithms, yielding as a byproduct a computation of the dimension of a projective variety. Comparative results on benchmarks for time and space of several families of multivariate polynomial equation systems are given and we point out both weaknesses and advantages of different approaches.

Keywords: Benchmark, Maple, Noether position, polynomial equation system, straight-line programs.

1. Introduction

Classical methods to study and solve systems of polynomial equations are based on numerous avatars of Gröbner (standard) basis algorithms or Riquier-Janet type methods (Ritt-Wu's algorithm). All these methods use implicitly but deeply the dense or sparse representation of multivariate polynomials, which is the computer science counterpart of the expansion of these mathematical objects on the monomial basis of the polynomial algebra. Also, all these methods can be interpreted as *rewriting* techniques.

Considerable efforts have been made in order to improve both theoretical and practical aspects of these techniques and to produce efficient algorithms and implementations. Restricting to this last aspect, all most commonly available computer algebra systems offer Gröbner basis implementations.

The knowledge of a standard basis yields as a simple byproduct the dimension of the algebraic variety defined by such systems. Actually, one can show that focusing on the simpler problem of computing the dimension of a projective algebraic variety will lead to a better worst-case complexity than the whole construction of a standard basis (Giusti, 1988). Considering the *unit cost measure* model, i.e., each arithmetic operation of the ground field is counted as one, this complexity is polynomial in the size of the intermediate expressions computed.

Having in mind the problem of determining the dimension, it seemed at that time that there was no hope to design an algorithm whose complexity is *polynomial* in the size of the *input*, since the intermediate computations are not. But this observation is only valid if we stay stuck in the dense representation context. A breakthrough was obtained by Giusti and Heintz (1993), resulting in the existence of an algorithm with polynomial behaviour w.r.t. the dense measure of the input, provided one uses a mixed representation for intermediate computations and output.

Actually the algorithm described in *loc. cit.* computes more, i.e., a change of coordinates putting the new variables in *Noether position.* Informally speaking, the variables are then separated into two subsets of different nature: the independent and dependent ones. The number of independent variables is the dimension. The key point is the introduction of a mixed data structure to represent the polynomials occurring in intermediate computations. While we use the dense representation w.r.t. the dependent variables, their coefficients (which are polynomials in the independent ones) are coded by *arithmetic circuits*, also called *straight-line programs.* This means that these latter polynomials are represented by programs evaluating them at a point (of the ground field) using only additions and multiplications (of the ground field). We will conveniently refer to this latter representation as the *evaluation data structure.*

Mixing these two data structures was successfully used in a series of theoretical papers to design a new geometric elimination algorithm (see the joint works by Giusti, Hägele, Heintz, Montaña, Morais, Morgenstern, Pardo, 1995–97 at `http://tera.medicis.polytechnique.fr`). An efficient implementation of the complete elimination algorithm will require some time to collect more practical experience with the first experimental prototypes. Some basic but important steps were made already, but there is still a lot of work left (see the works by Aldaz, Castaño, Hägele, Lecerf, Llovet, Martínez, Matera within the Tera project *loc. cit.*).

We present here modestly a Maple program called the *Projective Noether Package* derived from the algorithm of (Giusti and Heintz, 1993). (The package and its documentation are available at `http://tera.medicis.polytechnique.fr/tera/soft.html`). It turns out that a not so well-known functionality of Maple is the systematic use of the evaluation data structure. Consequently we can compare the more traditional algorithms already available within Maple with the new ones, experimenting with several possible strategies. Comparative results on benchmarks for time and space of several different families of multivariate polynomial equation systems are given and we point out both advantages and weaknesses of the different approaches. One of the encouraging results is provided by an example (see Section 4) where our Maple implementation computes an upper bound for the dimension more than fifty times as fast as the available version of Faugère's `Gb` system (Faugère 1995,1997). However, `Gb` computes much more, i.e., a full Gröbner basis, from which an upper bound on the dimension can be extracted.

The paper is organized as follows. In Section 2, we recall the main definitions and results concerning straight-line programs and Noether position and we give the theoretical algorithm from (Giusti and Heintz, 1993). Section 3 shows how straight-line programs can be handled in practice, first by exploiting the directed acyclic graphs which are fundamental to Maple, then by appealing to a mechanism called *deforestation* from theoretical computer science. In Section 3.4, we use these techniques to cast the theoretical algorithm in a different form on which the implementation is based. It turns out that the theoretical bounds which lead to a polynomial complexity of the algorithm are much too large to be of practical use. Therefore when we compare our implementation with a Gröbner basis package, we distinguish two subtasks: computing an upper bound on the dimension and proving that the bound is reached. Our implementation is very efficient for the former task, while we can only provide a heuristic answer to the latter one, in so far as a straight-line program representing a multivariate polynomial has been evaluated to 0 at a settable number of points but has not been proved to be identically 0.

Acknowledgments. This work was supported in part by the Long Term Research Project Alcom-IT (#20244) of the European Union, the French and Spanish grants: GDR CNRS 1026 MEDICIS and PB93–0472–C02–02.

2. Evaluation Data Structures and Deterministic versus Probabilistic Algorithms

2.1. Directed Acyclic Graphs and Straight-Line Programs

Let k be an infinite effective field; this means that the arithmetic operations (addition, subtraction, multiplication, division) and basic equality checking (comparison) between elements of k are realizable by algorithms. Let $x_1, \ldots, x_n$ be indeterminates over k. A polynomial of $k[x_1, \ldots, x_n]$ is usually coded as an expanded sum of monomials and each operation on such polynomials is related to operations on the vector of their coefficients. Instead, we use multivariate polynomials represented by *straight-line programs* that compute values at points of k^n.

All algorithms below will be represented by *arithmetic networks* over k i.e., directed acyclic graphs (DAGs) whose internal nodes are labelled by arithmetic operations of k, by Boolean operations corresponding to propositional logic, and by selectors associated with equality checking of elements of k. The external nodes of the graph represent the inputs and the outputs of the network. The inputs are always elements of k and the outputs may be elements of k, Boolean values, or integers of limited range (represented by vectors of Boolean values).

Particular arithmetic networks are of special interest: *arithmetic circuits* or *straight-line programs* (SLPs), without division or branching, containing neither selectors nor (propositional) Boolean operations. Generally speaking the *size* of the DAG (or the *sequential complexity* of the arithmetic network) is nothing but the number of its nodes (thus for a SLP the number of additions and multiplications involved). For details and elementary properties of straight-line programs we refer to Strassen (1972), von zur Gathen (1986), Stoß (1989) or Heintz (1989).

2.2. Zero Testing: Deterministic versus Probabilistic Algorithms

Arithmetic operations on polynomials represented by straight-line programs are immediate. The only non-trivial point when dealing with this data structure is equality checking (or zero testing). One can perform this task by evaluating the SLP at sufficiently many points. This is formalized in terms of "correct test sequences" of points with coordinates from k according to a theorem of Heintz and Schnorr (1982), which we recall for completeness:

Let D and L be two positive integers, and let us define the subset $W(D, n, L)$ of polynomials of $k[x_1, \ldots, x_n]$, of degree at most D, which can be coded by a SLP with at most L arithmetic operations. Furthermore given a subset Γ of k, a family $\gamma := \{\gamma_1, \ldots, \gamma_m\}$

(with $\gamma_i \in \Gamma$) of m points in k^n is called a *correct test sequence* for $W(D, n, L)$ if every polynomial in the latter vanishing on the points of γ is actually identically zero.

Theorem 2.1. (Heintz and Schnorr (1982), Theorem 4.4) *Let us fix a subset Γ of k of cardinality $\#\Gamma = 2L(D+1)^2$, and a cardinality $m := 6(L+n)(L+n+1)$. The subset $\tau(D, n, L, \Gamma) \in \Gamma^{nm}$ of correct test sequences for $W(D, n, L)$ satisfies:*

$$\#\tau(D, n, L, \Gamma) \geq (\#\Gamma)^{nm}(1 - (\#\Gamma)^{-\frac{m}{6}}).$$

In other words, the ratio of incorrect test sequences over all sequences of Γ^{nm} is at most the quantity:

$$\frac{1}{(2L(D+1)^2)^{(L+n)(L+n+1)}}$$

which is uniformly bounded by $\varepsilon := 1/262144$.

Although the choice of such a correct test sequence could be done by a deterministic algorithm, the cost of doing so would exceed the main complexity class we want. Therefore the algorithms we study below will be non-uniform insofar as they depend on the choice of correct test sequences. On the other hand, the theorem allows us to randomly choose correct test sequences with a probability of failure which becomes arbitrarily small as the parameters D, n and L increase. Therefore our algorithms can be uniformly randomized within the same order of (average) complexity. In doing so we encounter the following kind of probabilistic procedure which we call a *randomized algorithm.*

A randomized algorithm has error probability bounded by some $0 \leq \varepsilon < 1/2$ when accepting an input and error probability zero when rejecting it (in our case we may choose $\varepsilon = 1/262144$). As far as our algorithms compute polynomials or rational functions the correctness of the output depends only on the correctness of previous and intermediate decisions made by probabilistic procedures and can be checked randomly. In this sense, we apply also the term *randomized procedure* to the computation of polynomials or rational functions. Thus our results are valid not only in the sense of the non-uniform complexity model, but also in the sense of probabilistic (randomized) algorithms (see Balcázar *et al.* (1995), § 6.6, Giusti and Heintz (1993), § 1.2.3, § 1.3 and § 2.2 and Fitchas *et al.* (1995), § 1.3 and § 2.1 for more details).

To sum up we get the weakened following form which allows a probabilistic treatment of the theorem:

Theorem 2.2. (Fitchas *et al.* (1995), Theorem 2.1) *There exists an arithmetic network over k of size $O(Lm) = O(L(L+n)^2)$, which given any SLP (without divisions) of size at most L, checks if the n-variate polynomial it represents is identically zero. Moreover the network can be constructed by a probabilistic algorithm in sequential time $O(L(L+n)^2)$ with a probability of failure uniformly bounded by $\varepsilon := 1/262144$.*

In the sequel, we shall consider a special class of polynomials in n variables of total degree D whose complexity of evaluation L is of the same order as D (compare with $\binom{D+n}{n}$ the number of monomials of such a dense polynomial). In other words, these polynomials can be evaluated much faster than we should expect from their total degree. For this class, the above theorem yields a probabilistic zero-test of complexity polynomial in D (or L).

In practice, we encounter two difficulties when using these ideas for large values of D. First, the size L induces an important memory cost to store the SLP. Next, although polynomial, the bound $m = 6(L+n)(L+n+1)$ from Theorem 2.1 on the length of a correct test sequence is by far too large to be useful. Thus, our implementation will not be able to *certify* that such a multivariate polynomial coded as a SLP is identically zero.

However, we show below (see §3.2) how to evaluate the SLP (without storing it) at a number of points which can be made arbitrarily large.

2.3. Noether Position of Projective Varieties

Let k be an infinite effective field, and $\overline{k}$ be an algebraic closure of k. Given a set of homogeneous polynomials $f_1, \dots, f_s$ in $k[x_0, \dots, x_n]$, consider the projective variety $V = V(f_1, \dots, f_s)$ generated by the f_i in projective n-space $\mathbb{P}^n(\overline{k})$. We want to calculate the dimension of the projective variety V.

There are several approaches to this problem. We distinguish two different tasks, first the computation of an upper bound; and second the certification that an integer known to be an upper bound is actually the dimension.

Giusti and Heintz (1993) gave an algorithm to compute the dimension, which actually computes a change of coordinates putting the new variables in Noether position. The variables $x_0, \dots, x_r$ are said to be *independent* with respect to V if $(f_1, \dots, f_s) \cap k[x_0, \dots, x_r]$ is the trivial ideal (0). If moreover the canonical homomorphism

$$k[x_0, \dots, x_r] \to k[x_0, \dots, x_n]/(f_1, \dots, f_s)$$

is an integral ring extension, the variables $x_0, \dots, x_n$ are said to be in *Noether position* with respect to V. The latter condition means that the canonical images of the variables $x_{r+1}, \dots, x_n$ satisfy integral dependence relations (in other words are algebraic integers) over $k[x_0, \dots, x_r]$. As a consequence the dimension of V is nothing but r. In order to simplify the complexity considerations we shall suppose that the total degree d of the f_i's is at least n.

Theorem 2.3. (Giusti and Heintz (1993)) *Let $f_1, \dots, f_s$ be homogeneous polynomials in $k[x_0, \dots, x_n]$ of degree at most d ($\geq n$), defining a projective variety V in $\mathbb{P}^n(\overline{k})$. There exists a randomized algorithm without divisions which computes with sequential complexity $s^{O(1)}\, d^{O(n)}$ a linear change of coordinates over k such that the new variables are in Noether position with respect to V.*

Let us recall the main steps of this algorithm. It is organized around a loop, with decreasing index m starting from n. The $(n-m)$th iteration is entered with the condition: the canonical images of $x_{m+1}, \dots, x_n$ are already algebraic integers over $R^{(m)} := k[x_0, \dots, x_m]$.

Let z be a new variable; we denote by $f_i^{(m)}$ the polynomial

$$f_i(zx_0, zx_1, \dots, zx_m, x_{m+1}, \dots, x_n)$$

considered as a polynomial in $R^{(m)}[x_{m+1}, \dots, x_n, z]$. It is homogeneous of degree (with respect to the variables $x_{m+1}, \dots, x_n, z$) the total degree of f_i.

Let W be the projective variety in $\mathbb{P}^{n-m}(\overline{K})$ defined by $f_1^{(m)}, \dots, f_s^{(m)}$, where $K = K^{(m)}$ is the field of fractions $k(x_0, \dots, x_m)$ of $R^{(m)}$.
In this situation a *criterion for independence* is that W is not empty, which can be checked by computing a Gröbner basis or by applying an effective projective Nullstellensatz as follows:

- Let $N = 1+\sum_{i=1}^{s}(\deg(f_i)-1)$ (the bound of the effective projective Nullstellensatz). Create the matrix Q of the linear $R^{(m)}$-linear application $(h_1, \dots, h_s) \mapsto h_1 f_1^{(m)} + \cdots + h_s f_s^{(m)}$, h_i being an homogeneous polynomial of degree $N-\deg(f_i)$ in $R^{(m)}[x_{m+1}, \dots, x_n, z]$, on the monomial basis (in $x_{m+1}, \dots, x_n, z$); the coefficients are of course in $R^{(m)}$.
- Use Berkowitz-Mulmuley linear algebra (Berkowitz (1984), Mulmuley (1987)) to construct the DAG corresponding to the determinant of the product matrix $Q^t Q$ to check the surjectivity of Q (this determinant lives in $R^{(m)}$).
- A probabilistic test is then performed to determine whether this DAG represents zero (W is not empty) or not, in which case a point in $\mathbb{P}^m(k)$ with homogeneous coordinates $(a_0, \dots, a_m)$ $(a_m \neq 0)$ where it is non zero is found.

If the criterion is satisfied, we are in Noether position and we stop.
Otherwise, there exists a non-zero homogeneous polynomial g in $k[x_0, \dots, x_m]$ which vanishes on V and does not vanish at $(a_0, \dots, a_m)$. After the change of coordinates $x_0 \leftarrow x_0 + a_0 x_m, \dots, x_{m-1} \leftarrow x_{m-1} + a_{m-1} x_m, x_m \leftarrow a_m x_m$ the polynomial g becomes monic in x_m, hence the canonical image of x_m is an algebraic integer over $R^{(m-1)}$ and we can enter the $(n+1-m)$th iteration.

For a more complete mathematical description of this algorithm, we refer to (Giusti and Heintz, 1993, pp. 25–27) and the Userguide of (Lecerf, 1997). After showing in the next section how SLPs can be dealt with in practice, we discuss the implementation of this algorithm.

3. Evaluation Data Structure and Maple Implementation

In this section, we show how algorithms based on straight-line programs can be turned into programs. The fundamental use of directed acyclic graphs made by the computer algebra system Maple is first used to provide a straightforward implementation exhibiting the required complexity. The efficiency of this implementation is however hindered by an important consumption of memory. A method based on evaluation, and reminiscent of the technique called deforestation (Wadler, 1990) from theoretical computer science, can then be applied to reduce the memory used in intermediate steps and leads to faster programs.

3.1. Efficient Evaluation in Maple

The Maple computer algebra system is based on a systematic use of common subexpression sharing. Objects which might look like expression trees to the user are actually stored as directed acyclic graphs, where only one copy of each distinct subtree is kept. This is accomplished by maintaining a hash table of all the expressions occurring simultaneously in a session. The structure thus obtained can be viewed as a single directed acyclic graph the children of whose root correspond to all the distinct subexpressions

Table 1. Substitution and DAGs

n	30	31	32	33	34
`subs`	6sec	11sec	15sec	29sec	47sec
	46Mb	75Mb	121Mb	195Mb	316Mb
`dagsubs`	6sec	9sec	12sec	19sec	37sec
	25kb	26kb	28kb	28kb	29kb

residing simultaneously in memory. A simple consequence of this representation is that syntactic equality of expressions is reduced to checking equality of addresses and can thus be performed in constant time. This provides the basis for the efficiency of Maple's *option remember* which can be used to record the pairs (input, output) of a procedure. Conversely, this option can be used to write recursive procedures performing DAG traversals of their argument instead of tree traversals without this option. This can lead to an improved complexity of the algorithm.

For instance, it is unfortunate that up to the current version, Maple's substitution command `subs`, which is commonly used to evaluate an expression at a point, does not benefit from this nice mechanism and has a complexity related to the size of the tree instead of the size of the DAG. The use of a DAG traversal to improve the complexity can be illustrated with a simple alternate procedure:

```
dagsubs := proc(tosubs::{name = algebraic, list(name = algebraic)}, expr)
    local dosubs, i, res;
        dosubs := proc(expr) option remember;
                if nops(expr) = 1 then expr else map(procname, expr) fi end;
        if type(tosubs, name = algebraic) then dosubs(op(1, tosubs)) := op(2, tosubs)
        else for i in tosubs do dosubs(op(1, i)) := op(2, i) od fi;
        res := dosubs(expr);
        dosubs := subsop(4 = NULL, op(dosubs));
        res
    end
```

In Table 1, we give examples of the time and memory[†] required by both `subs` and this simple `dagsubs`. The test suite is the following sequence of polynomials:

$$P_0(x) = 1, \qquad P_1(x) = x, \qquad P_{n+2}(x) = xP_{n+1}(x) + P_n(x) + 1, \quad n \geq 0.$$

Of course the polynomials are not expanded and the substitution consists in replacing x by another variable y. The time difference is not very large, but `subs` is a function of Maple's compiled kernel, whereas `dagsubs` is interpreted. However, while `dagsubs` needs a very limited amount of memory, `subs` requires more than one thousand times this amount, and the ratio increases very fast with the index of the polynomials. This example clearly demonstrates that working with DAGs when possible can be crucial in terms of efficiency.

[†] The tests in this article have been performed on a PC Pentium Pro (200Mhz) with 512Mb of memory, running Linux 2.0.27 and Maple V.3. This computer forms part of the equipment of GDR Medicis: `http://medicis.polytechnique.fr`.

When the DAG is large and many evaluations of it at different values are required, for instance to prove that it is the zero polynomial with a correct test sequence, Maple also provides tools working at the DAG level (via the *option remember*) that generate optimized Fortran or C code. For instance, on the polynomial P_{10} above, Maple's `optimize` yields:

$$t_1 = x^2, \quad t_3 = x(t_1+2), \quad t_5 = x(t_3+x+1), \quad t_7 = x(t_5+t_1+3),$$

$$t_9 = x(t_7+t_3+x+2), \quad t_{11} = x(t_9+t_5+t_1+4), \quad t_{13} = x(t_{11}+t_7+t_3+x+3),$$

$$t_{18} = x(x(t_{13}+t_9+t_5+t_1+5)+t_{11}+t_7+t_3+x+4)+t_{13}+t_9+t_5+t_1+6.$$

This can then be translated into a Maple procedure (`makeproc`) or alternatively into Fortran or C code. Here is for instance the corresponding C code:

```
t1  = x*x;
t3  = x*(t1+2.0);
t5  = x*(t3+x+1.0);
t7  = x*(t5+t1+3.0);
t9  = x*(t7+t3+x+2.0);
t11 = x*(t9+t5+t1+4.0);
t13 = x*(t11+t7+t3+x+3.0);
t18 = x*(x*(t13+t9+t5+t1+5.0)+t11+t7+t3+x+4.0)+t13+t9+t5+t1+6.0;
```

All these tools performing conversions between DAGs and SLPs implement the canonical isomorphism between polynomials and polynomial functions (the ground field being infinite).

3.2. Deforestation

The algorithm described in Section 2.3 computes SLPs. The only information required about some of these SLPs is whether they represent the zero polynomial and if not, a point at which they are different from zero. This is done by evaluating the SLP at one or several points (see §2.1). The complexity of evaluating the SLP is directly related to its size, i.e. the memory it uses. The idea of *deforestation* is to perform the same evaluations *without computing the SLP first*, thus saving memory occupied by unnecessary expression trees (whence the name deforestation).

More generally the deforestation of a program is a transformation consisting in eliminating the building of intermediate structures introduced by composition of functions.

For instance, Lisp code to compute the last element of a list could be implemented by

```
(defun last (l) (car (reverse l)))
```

where `reverse` is written

```
(defun reverse (l) (reverse2 l nil))
(defun reverse2 (l1 l2) (if l1 (reverse2 (cdr l1) (cons (car l1) l2)) l2))
```

This implementation has the disadvantage of creating an intermediate list of the same length as the argument. The deforested version would read

```
(defun last (l) (last2 l nil))
(defun last2 (l1 l2) (if l1 (last2 (cdr l1) (car l1)) l2))
```

In some cases this program transformation can be performed automatically, see (Wadler, 1990).

In the case of the algorithm of Section 2.3 this deforestation in Section 3.4 will mainly consist in a simple exchange of loops. Let `pol(x1,...,xn)` be any Maple procedure using only `+, -, *, :=`, integers and loops with fixed depth. Such a function has the particularity to run with any kind of entries in a ring: if its arguments are variable names it returns a DAG in which the arguments are the leaves, if its arguments are integers it returns an integer. Thus, if $x_1, \dots, x_n$ are variable names, the code:

```
$p := \mathrm{pol}(x_1, \dots, x_n)$;
**for** point **in** list_of_points **do**
  **if** $dagsubs([\mathbf{seq}(x_i = \mathrm{point}[i], i = 1, \dots, n)], p) \neq 0$ **then** RETURN(point) **fi**
**od**;
...
```

can be written:

```
**for** point **in** list_of_points **do if** pol(**op**(point)) $\neq 0$ **then** RETURN(point) **fi od**;
...
```

The memory required by the DAG p in the first version of the code is not necessary in the second one and this has an important impact on the speed of the execution (by reducing the time spent in memory handling). In the application to the algorithm of Section 2.3, the complexity of the deforested version is that given by Theorem 2.3. Note however that in a more general context, the DAG p might have smaller number of arithmetic operations.

After a detailed example illustrating the evaluation strategies presented above, the rest of this paper describes a "manual" deforestation of the algorithm from Section 2.3. Besides the reduction of the amount of memory used in intermediate steps, further gains are possible by substituting faster algorithms operating on constants for those operating on SLPs. Then the complexity estimates based on SLPs provide an upper bound on the complexity of the actual computation.

3.3. Example: Determinant of a Matrix of Polynomials

In this section, we illustrate how the systematic exploitation of the DAG structure leads to improved algorithms computing the determinants of matrices of multivariate polynomials with integer coefficients. This task is an important building block of the theoretical algorithm from Section 2.3.

The matrices we take in our examples are square $k \times k$ matrices with polynomial entries having 3 variables, at most 6 terms, a total degree at most 5 and coefficients with 2 decimal digits. The matrices are sparse with $5k$ entries at random filled by polynomials provided by Maple's `randpoly` function. To obtain regular matrices, the diagonal is first filled with 1's. To obtain singular matrices, the columns from 1 to $k-1$ are summed into the kth one. Timings for regular and singular matrices turn out to be similar, thus we give only one table (Table 2) of results.

Table 2. Computations on matrices with polynomial entries

dimension	10	20	50	100	200
naive (maple `det`)	∞	∞	∞	∞	∞
computation of the DAG (Berkowitz)	.08	1.5	400	> 5000	
test sequence (#pts)	$5.7\,10^7$	$2.3\,10^{10}$	$8.0\,10^{13}$		
evaluation at one point of the DAG (`subs`)	53	> 5000			
optimization of the DAG (`optimize`)	.02	> 5000			
evaluation at one point, deforested version	.03	.20	23	47	1050

Naive approach

We first use Maple's `det` command, which is based on a mixture of fraction free Gaussian elimination and minor expansion. Since the resulting polynomial is always expanded, the computation is expensive because of the exponential growth of the number of multivariate monomials as the degree increases. This shows in Table 2: on all our examples, Maple returns an error message "object too large" (abbreviated ∞ in the table).

Straight-line programs and Berkowitz's algorithm

In order to overcome the exponential complexity of expanding determinants, it is natural to turn to DAGs or to straight-line programs evaluating them. Recognizing zero with this data-structure becomes an expensive operation. Thus an approach based on Gaussian elimination does not apply anymore. Several other algorithms can be applied. We use an algorithm due to Berkowitz (1984) which has the advantage of a simple description. Given an $n \times n$ matrix, it computes its characteristic polynomial (and in particular its determinant) in $O(n^4)$ arithmetic operations on the coefficients and requires neither test nor division.

On a generic square matrix of size n, the number of nodes of the DAG evaluating the expanded determinant and the one produced by Berkowitz's algorithm are indicated in Table 3. In this case, the polynomial complexity of Berkowitz's algorithm quickly yields better results than the number $n! + 1$ of monomials of the generic determinant. This is naturally reflected by the time required for the computation. For our test matrices, the results turn out to be very similar: Berkowitz's algorithm takes almost no time on matrices for which `det` cannot compute the result. This appears in the second line of Table 2.

Table 3. Sizes of different representations of the determinant of an $n \times n$ matrix

dimension	2	3	4	5	6	7	8	9
dag size (expanded)	3	7	25	121	721	5041	40321	362881
dag size (Berkowitz)	3	20	64	169	343	664	1104	1817

Evaluation and test sequences

In line 4 of Table 2, we give the time used to evaluate the DAG computed via Berkowitz's algorithm at one point. In regular cases, this is usually also the time required to prove that the matrix is regular. In the singular cases, we also show an estimate of the number of points m forming the correct test sequences for DAGs of the corresponding size. The table shows that although this approach makes it possible to deal with objects which are too large for Maple when expanded, it also rapidly produces objects with which it is impossible to proceed in a reasonable amount of time. (Part of this might be due to the limited size of the hash tables used by Maple.) Whatever the speed of evaluation, the number of points indicated on line 3 leads to the conclusion that *the theoretical bound which leads to polynomial complexity of the deterministic algorithm is much too large to be of practical use.*

Deforestation

The process outlined above consists of two steps. First a DAG is constructed via Berkowitz's algorithm, then this DAG is evaluated at one or several points. Inverting the loops as discussed in §3.2 is then a natural strategy: we first evaluate the matrix at these points and then compute the determinant. This is clearly an improvement, since the DAG for the determinant does not need to be stored in memory anymore and the determinant can be evaluated by faster algorithms. In the singular case, we can still use the bound on the number of points which follows from considering the DAG produced via Berkowitz's algorithm without actually executing this algorithm. The resulting timings appear in the last line of Table 2 and vindicate this strategy. Preliminary experiments in C using LEDA's bignums (Mehlhorn *et al.*, 1997) seem to indicate that we can only expect an improvement of a factor four to five by performing this evaluation directly in C.

3.4. Deforestation of the Algorithm

The algorithm described in Section 2.3 relies on the computation of determinants of matrices of polynomials. As shown in the previous Section 3, this operation benefits from deforestation. We can go one step further and apply deforestation to the whole algorithm.

The deforested algorithm is organized around nested loops. The outer loop is decreasing of index m and starts from n. Its $(n-m)$th iteration is entered with the condition: the canonical images of $x_{m+1}, \dots, x_n$ are already algebraic integers over $R^{(m)} := k[x_0, \dots, x_m]$. The inner loop is indexed by a list of points l_m in $\mathbb{P}^m(k)$. Each point of homogeneous coordinates $(a_0, \dots, a_m)$ of l_m satisfies $a_m \neq 0$.

Algorithm

```
For $m$ from $n$ to 0 do:

    For each point $(a_0, \dots, a_m)$ of $l_m$ do:

        – Specialize the homogeneous polynomials $f_i^{(m)}$ of $k(x_0, \dots, x_m)[x_{m+1}, \dots, x_n, z]$
          to homogeneous polynomials $f_i(a_0 z, \dots, a_m z, x_{m+1}, \dots, x_n)$ of $k[x_{m+1}, \dots, x_n, z]$;
```

- Apply a Gröbner basis algorithm with total degree ordering to the specialized $f_i^{(m)}$ to compute a standard basis; then determine whether the variety $W \subseteq \mathbb{P}^{n-m}(\overline{k})$ they define is empty or not.
    - ∗ If it is empty, break this inner loop;
    - ∗ If it is not empty, repeat this process with another point of l_m.

We exit from this inner loop in two possible cases:

- A point $(a_0, \ldots, a_m)$ of l_m has been found such that the corresponding variety W is not empty. We perform the following change of variables in the input polynomials f_i: $x_0 \leftarrow x_0 + a_0 x_m, \ldots, x_{m-1} \leftarrow x_{m-1} + a_{m-1} x_m, x_m \leftarrow a_m x_m$ (Recall that a_m is not zero). The variables x_m is now an algebraic integer over $R^{(m-1)}$, we can enter the $(n+1-m)$th step of the outer loop.
- All the points of l_m are such that the corresponding W is empty. In this case we return the integer m and the polynomials f_i.

That this algorithm is the deforestation of the one of Section 2.3 follows from the fact that the computation of a Gröbner basis in the projective case is nothing but a triangulation of the block Toeplitz matrix Q by elementary column operations.

The index m of the outer loop represents an upper bound on the dimension of the variety. As seen in the previous section, the number of points where the inner loop has to be performed when the corresponding determinant is actually zero is very large. However, this will only happen once, at the end of the algorithm, when m reaches the dimension. In practice, we shall therefore stop the inner loop after a settable number of points. We thus obtain a proved bound on the dimension and a likely value for it.

The practical complexity of the deforested algorithm inherits the good complexity behavior of the theoretical one but the non-uniform deterministic aspect is lost. It is clear that the specializations which do not lead to a correct answer are enclosed in a strict algebraic subset but we do not know any precise bound on the probability of failure. One reason is that we do not know bounds on the probability of failure of a Gröbner basis of a specialized system to be the specialization of the Gröbner basis of the system.

In the same way as the specialization of the free variables we could specialize the integers, that is, perform the computations modulo a prime number picked-up at random, but this is out of the scope of this paper.

Obviously, the computation will be fast when the dimension is large, since it is reduced to Gröbner basis computations on systems of polynomials in much less variables. Thus in cases of low dimension, the timings of our method are comparable with a direct Gröbner basis computation, while our method is best suited for cases of large dimension, as exemplified in the experimental data below.

Illustration of the Algorithm

Let us consider $f_1 = x_0 x_1$ and $f_2 = x_0 x_2$ in $k[x_0, x_1, x_2]$, we have $n = 2$. We now detail what the algorithm does:

First step: set $m = 2$; $f_1^{(2)} = x_0 x_1 z^2$, $f_2^{(2)} = x_0 x_2 z^2$ are seen as homogeneous equations in $K[z]$, where $K = k(x_0, x_1, x_2)$. The corresponding variety W is empty, the specialization $a_0 = 1$, $a_1 = 0$, $a_2 = 1$ leads to the change of variables: x_0 is replaced by $x_0 + x_2$, x_1

remains x_1 and x_2 remains x_2. The new equations are $f_1 = x_0x_1 + x_1x_2$, $f_2 = x_0x_2 + x_2^2$, the canonical image of the variable x_2 is now an algebraic integer over $k[x_0, x_1]$. We go to the second step.

Second step: set $m = 1$; $f_1^{(1)} = x_0x_1z^2 + x_1zx_2$, $f_2^{(1)} = x_2^2 + x_0zx_2$ are seen as homogeneous equations in $K[x_2, z]$, where $K = k(x_0, x_1)$. For any specialization (a_0, a_1) of (x_0, x_1) the corresponding variety W is not empty since it contains the projective point $x_2 = -a_0z$. So the algorithm returns 1 as the dimension.

4. Examples

4.1. The Behavior at Infinity of the Cyclic n-roots Systems

This system is related to the question of finiteness and structure of the corresponding set of cyclic n-roots (Björck, 1990). Let $x_1, \ldots, x_n$ be variables and M_i be the monomial $x_1 \cdots x_i$. Let σ be the cycle $(1, 2, \ldots, n)$ of the nth permutation group. We define the ith cyclic equation of the nth system as:

$$H_n^i = \sum_{k=0}^{n-1} \sigma^k(M_i) \quad \text{for} \quad 1 \le i \le n-1, \qquad H_n^n = M_n.$$

Now, `infcyclic` n defines the system $\{H_n^n = H_n^{n-1} = \ldots = H_n^1 = 0\}$. For example, when $n = 3$ the system H_3^∞ is $\{x_1x_2x_3 = x_1x_2 + x_2x_3 + x_3x_1 = x_1 + x_2 + x_3 = 0\}$.

Proposition 4.1. (in collaboration with E. Schost) *The system H_n^∞ defines in $\mathbb{P}^{n-1}(\mathbb{C})$ a projective variety of dimension at least:*

$$r_{\inf}(n) = n - \left\lceil \frac{n}{\lfloor \sqrt{n} \rfloor} \right\rceil - \lfloor \sqrt{n} \rfloor.$$

Proof. Let $k = \lfloor \sqrt{n} \rfloor$, $l = \left\lceil \frac{n}{\lfloor \sqrt{n} \rfloor} \right\rceil$. The polynomial H_n^n is reduced to a monomial containing x_1, which shows that the variety contains a subvariety defined by $x_1 = H_n^{n-1} = \cdots = H_n^1 = 0$. Modulo $x_1 = 0$, H_n^{n-1} is again reduced to a monomial. This monomial contains x_{1+k}. Proceeding in this manner using the equations $H_n^{n-1} = 0, \ldots, H_n^{n-(l-2)k-1} = 0$ shows that the variety contains a subvariety defined by $x_1 = x_{1+k} = \cdots = x_{1+(l-1)k} = H_n^{k-1} = \cdots = H_n^1 = 0$. Then applying Krull's Lemma (Eisenbud, 1995, 8.2.2) completes the proof. □

The resulting lower bounds for the dimension of the system `infcyclic` H_n^∞ are given in Table 4.

Note that it is possible to decompose by hand the system `infcyclic` and thus obtain better time/space results. This is for instance done by Maple's `gsolve` function. In our

Table 4. Lower bounds for the dimension of the system `infcyclic` H_n^∞

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$r_{\inf}(n)$	-1	-1	-1	0	0	1	1	2	3	3	4	5	5	6	7	8	8

Table 5. Bounding the dimension

system / method	pnp		grobner		Gb	
UBD	Time	Space	Time	space	Time	space
`infcyclic 6`						
2	0.2	720	0.71	1180		
1	2.5	1400	8.48	1500	1.0	1200
`infcyclic 7`						
4			1.7	1400		
3	0.2	720	2.5	1500		
2	2.5	1440	65	2230	7	2200
1	900	3200	13000	6200	950	3200
`infcyclic 8`						
5			5.8	1500		
4	0.3	720	6.18	1500		
3	3.0	1500	396	2700	3	2200
2	1400	4130	> 250000	> 18000	54000	27600
`infcyclic 9`						
6			21	1900		
5	0.3	720	22	1900		
4	5.1	1700	2293	3080	3	3400
3	1600	7000	>200000	> 8000	> 108000	> 180000

tests, we shall not allow this simplification for either our package or Gröbner basis packages. The reason for us to use the `infcyclic` system for our tests is the known lower bounds for the dimension from Table 4.

We present the tables of experimental results obtained for the two tasks of bounding and computing the dimension on these systems.

The algorithm described in Section 3.4 and denoted `pnp` is compared with Maple's `grobner` function (`grobner`) and with Faugère's `Gb` applied directly to the systems. The `Gb` computation uses the function `sugar`, with the optional parameter `"info"`, and a total degree ordering (degree reverse lexicographic). Time is measured in *seconds* and memory space in *kilobytes*. A time t preceded by the symbol $>$ means that the computation has been manually aborted after t seconds, the corresponding value in the column labelled `space` is the memory allocated until then. Empty entries in the tables are due to technical problems measuring very small quantities and/or their scaling with the rest of the entries in the same table.

In table 5, UBD stands for "upper bound on the dimension". For example the first line of Table 5 reads: the dimension of the system `infcyclic 6` has been proved to be at most 2 in 0.2 seconds with a memory space of 720kb, using our program, whereas using the Maple `grobner` function it took 0.71 seconds and 1180kb.

The first task is the determination of upper bounds on the dimension of the variety. Using the algorithm of §3.4, we know that at step m of the iteration the dimension of the variety is at most m. It is also possible to compute upper bounds on the variety during the calculation of the Maple `grobner` function: the dimension is at most the dimension of the monomial ideal generated by the leading monomials of the S-polynomials computed when performing a Buchberger algorithm, see e.g., (Cox *et al.*, 1997).

Table 6. Computing the dimension

		pnp	(one pt)	grobner	
System	Dim.	Time	Space	Time	Space
`infcyclic`					
2	-1	0.00	243	0.05	105
3	-1	0.01	448	0.04	149
4	0	0.07	2165	0.13	481
5	0	6.5	1834	4	1507
6	1	107	2948	114	2424

The computation of upper bounds on the dimension of the projective variety is the strong point of our method. Where the Gröbner basis algorithms are forced to work with the complete equation system in many variables, our recursive approach yields the correct answer a lot faster. First, it can be seen that Maple's `grobner` function is the least efficient approach. We observe then, that even using this same Maple `grobner` function for the intermediate calculations in our method, the resulting performance is already competitive with the stand-alone `Gb` program. Note that any improvements in the field of Gröbner bases computations will also yield direct improvements for our method.

The second task is to determine the exact dimension of the given variety. The correct test sequences are out of reach and thus we can only evaluate a zero-polynomial given as a SLP at a certain number of points. The dimension computed is indicated in the second column of Table 6. In the next column, we give the time necessary to evaluate the final polynomial at one point. As already discussed, this example being of low dimension, the performance of our method in this case is comparable with that of a mere Gröbner basis computation.

For the next examples, we did not try to estimate the time taken by the Gröbner basis package to bound the dimension and we only give the total time of the computation.

4.2. Generic Polynomials

We consider a polynomial system of three homogeneous equations in 11 variables of degree 2. The coefficients are randomly chosen integers between -99 and 99.

Our package takes 1.33s to prove that the dimension is at most 8. Then the final putative zero-polynomial is evaluated at random points in 2.22 sec. each in a constant memory of 1.9Mb. Maple's `grobner` computes a Gröbner basis for the total degree order in 10053 sec. and 5.3Mb. By extrapolation, 4528 points could be tested by our program.

4.3. Incomplete Determinental Ideals

Let M be a 5×3 matrix, filled with linear forms in 11 variables, with random coefficients between -10 and 10. M has ten 3×3 submatrices. Their determinants are homogeneous polynomials of degree 3 in the 11 variables. We drop one of them, in order to obtain of system with 9 polynomials.

Our package takes 1s to prove that the dimension is less than 8. As before, each random point with random coordinates between 0 and 100 is tested in 1.16s and 3.8Mb.

Maple's `grobner` with total degree order computes the basis in 7093s and 8.32Mb. By extrapolation 6114 points could be tested by our program.

M is now a 7×4 matrix filled with random linear forms in eleven variables, with coefficients between -10 and 10. M has thirty five 4×4 submatrices. Their determinants are homogeneous polynomials of degree 4. We have chosen nine of them, corresponding to the following sets of number lines :

$$[1,2,3,4],[1,2,3,5],[1,2,3,6],[1,2,3,7],[1,2,4,5],$$

$$[1,2,4,6],[1,2,4,7],[1,2,5,6],[1,2,5,7].$$

We now compare Faugère's `Gb` package with a version of our program which calls `Gb` externally for the Gröbner bases computations via `gblink` (Lecerf and Schost, 1997).

Our program takes 0.08s and 0.6Mb to prove that the dimension is less than 8. Each point tested is, as previously, a random point with coordinates between 0 and 100. Eighty-eight have been tested in 1056s and 1.2Mb. `Gb's Grobner` with its `tgrobner` function has not finished in 3 days and 451Mb. By extrapolation at least 21600 points can be tested with our program.

5. Conclusion

Algorithms using SLPs to store multivariate polynomials suffer two practical problems: first, the memory management becomes prohibitive when dealing with very big SLPs and second the use of non-uniform deterministic zero tests requires evaluations of the SLPs on a very big set of points. A direct implementation of these algorithms does not turn out to be efficient. Along the example of computing a Noether position of a projective variety, we have shown how to transform the theoretical algorithm into a practical one which does not require SLPs anymore in the intermediate computations and which is very simple to implement. The practical complexity of the new algorithm inherits the one of the theoretical one. This idea should apply in a wide class of algorithms in elimination theory. (See (Giusti *et al.*, 1999) for another step in this direction.)

References

Balcázar, J. L., Díaz, J., Gabarró, J. (1995). *Structural complexity. I.* Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, second edition.

Berkowitz, S. J. (1984). On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, **18**(3):147–150.

Björck, G. (1990). Functions of modulus 1 on Z_n whose Fourier transforms have constant modulus, and "cyclic n-roots". In *Recent advances in Fourier analysis and its applications (Il Ciocco, 1989)*, volume 315 of *NATO Advance Science Institutes Series C: Mathematical and Physical Sciences*, pages 131–140. Kluwer Academic Publishers, Dordrecht.

Cox, D., Little, J., O'Shea, D. (1997). *Ideals, varieties, and algorithms.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition. An introduction to computational algebraic geometry and commutative algebra.

Eisenbud, D. (1995). *Commutative algebra with a view toward algebraic geometry*, volume 150 of *Graduate Texts in Mathematics.* Springer-Verlag, New York.

Faugère, J.-C. (1995). *GB Reference Manual.* LITP. `http://posso.ibp.fr/GB.html`.

Faugère, J.-C. (1997). Gb: State of gb + tutorial. LITP.

Fitchas, N., Giusti, M., Smietanski, F. (1995). Sur la complexité du théorème des zéros. In *Approximation and optimization in the Caribbean, II (Havana, 1993)*, volume 8 of *Approximation and Optimization*, pages 274–329. Peter Lang Verlag, Frankfurt am Main. With the collaboration of Joos Heintz, Luis Miguel Pardo, Juan Sabia and Pablo Solernó.

Giusti, M. (1988). Combinatorial dimension theory of algebraic varieties. *Journal of Symbolic Computation*, **6**(2-3):249–265. Special issue on computational aspects of commutative algebra.

Giusti, M., Heintz, J. (1993). La détermination des points isolés et de la dimension d'une variété algébrique peut se faire en temps polynomial. In Eisenbud, D., Robbiano, L., editors, *Computational algebraic geometry and commutative algebra (Cortona, 1991)*, volume XXXIV of *Symposia Matematica*, pages 216–256. Cambridge University Press, Cambridge.

Giusti, M., Lecerf, G., Salvy, B. (1999). A Gröbner free alternative for polynomial system solving. Preprint. Available at `http://www.medicis.polytechnique.fr/gage/notes/1999.html`. Note #99-04.

Heintz, J. (1989). On the computational complexity of polynomials and bilinear mappings. A survey. In *Applied algebra, algebraic algorithms and error-correcting codes (Menorca, 1987)*, volume 356 of *Lecture Notes in Computer Science*, pages 269–300. Springer, Berlin.

Heintz, J., Schnorr, C. P. (1982). Testing polynomials which are easy to compute. In *Logic and Algorithmic (Zürich, 1980)*, volume 30 of *Monographie de l'Enseignement Mathématique*, pages 237–254.

Lecerf, G. (1997). *The Projective Noether Package, User's Manual.* Laboratoire GAGE, École polytechnique, Palaiseau, France.

Lecerf, G., Schost, E. (1997). *Maple Package: GB link.* Laboratoire GAGE, École polytechnique, Palaiseau, France. `http://tera.medicis.polytechnique.fr/tera/soft.html`.

Mehlhorn, K., Näher, S., Uhrig, C. (1997). *Library for Efficient Datastructures and Algorithms.* Max Planck Institute for Computer Science, Saarbrücken. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

Mulmuley, K. (1987). A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, **7**(1):101–104.

Stoß, H.-J. (1989). On the representation of rational functions of bounded complexity. *Theoretical Computer Science*, **64**(1):1–13.

Strassen, V. (1972). Berechnung und Programm. I, II. *Acta Informatica*, **1**(4):320–355; *ibid.* 2(1), 64–79 (1973).

von zur Gathen, J. (1986). Parallel arithmetic computations: a survey. In *Mathematical foundations of computer science, 1986 (Bratislava, 1986)*, volume 233 of *Lecture Notes in Computer Science*, pages 93–112, Berlin. Springer.

Wadler, P. (1990). Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, **73**:231–248. Special issue of selected papers from 2'nd ESOP.