Greater Easy Common Divisor and standard basis completion algorithms

ANDRÉ GALLIGO[†], LOÏC POTTIER[†], CARLO TRAVERSO[‡]

[†]Université de Nice et INRIA – [‡]Università di Pisa

INTRODUCTION

The computation of a standard basis (also called Gröbner basis) of a multivariate polynomial ideal over a field K is crucial in many applications. The problem is intrinsically time and space consuming, and many researches aim to improve the basic algorithm due to B. Buchberger [**Bu1**]. One has investigated the problem of the optimal choice of the term ordering depending from the use that has to be made of the results, [**BS**], and the systematic elimination of unnecessary reductions [**Bu2**], [**GM**], [**Po**]. We can call all these problems "combinatorial complexity problems".

The present paper considers arithmetic complexity problems; our main problem is how to limit the growth of the coefficients in the algorithms and the complexity of the field operations involved. The problem is important with every ground field, with the obvious exception of finite fields.

The ground field is often represented as the field of fractions of some explicit domain, which is usually a quotient of a finite extension of $\mathbf{Z}$, and the computations are hence reduced to these domains.

The problem of coefficient growth and complexity already appeared in the calculation of the GCD of two univariate polynomials, which is indeed a very special case of standard basis computation; the PRS algorithms of Brown and Collins operate the partial coefficient simplifications predicted by a theorem, hence succeeding in controlling this complexity.

Our approach looks for analogies with these algorithms, but a general structure theorem is missing, hence our approach relies on a limited search of coefficient simplifications. The basic idea is the following: since the GCD is usually costly, we can use in its place the "greatest between the common divisors that are easy to compute" (the GECD), this suggestion allowing different instances. A set of such instances is based on the remark that if you have elements in factorized form, then many common divisors are immediately evident. Since irreducible factorization, even assuming that it exists in our domain, is costly, we use a partial factorization basically obtained using a "lazy multiplication" technique, i.e.performing coefficient multiplications only if they are unavoidable. The resulting algorithms were tested with a "simulated" implementation on the integers, and the results suggest that a complete implementation should be very efficient, at least when the coefficient domain is a multivariate rational function field.

1. STANDARD BASIS COMPLETION ALGORITHM REVISITED

Let A be a domain, K its quotient field, and $X = (x_1, \ldots, x_n)$ a set of indeterminates. A polynomial in $K[X]$ is a sum of monomials, each one being composed of a non-zero coefficient and a *multiplicative term* (term for short), i.e. a product of indeterminates.

A *term-ordering* is a total ordering on the set of terms, making it an ordered monoid, and such that 1, identified with the empty term, is the minimum of the monoid. From now on, we assume that a term-ordering is given.

Research with the contribution of Ministero della Pubblica Istruzione (Italy) and C.N.R.S. (France).

If f is a non-zero polynomial, define $Lm(f)$, $Lt(f)$, $Lc(f)$ (*leading monomial, leading term, leading coefficient* of f) being respectively the monomial of f with the maximum term, its term and its coefficient. Similarly, define the sets $Lm(S)$, $Lt(S)$, $Lc(S)$ for any subset $S \subseteq K[X]$.

A polynomial (a subset) is called *monic* if its leading coefficient(s) is (are) 1.

A standard basis is a finite set G of generators of an ideal $I \subseteq K[X]$ such that $Lm(G)$ generates the ideal spanned by $Lm(I)$. By noetherianity of $K[X]$ any family of polynomials can be completed into a standard basis; the effective computation is not straightforward. B. Buchberger gave in [**Bu1**] the first completion procedure.

In this paper we propose an improved approach whenever K is not a finite field.

We describe here the Buchberger algorithm as an "algorithm scheme", i.e. as a series of named (but undefined) data structures and subalgorithms. Our algorithms will consist in alternative definitions for some of these structures and algorithms, keeping unchanged the overall structure of the algorithm, and undefined some of the subalgorithms (meaning that any correct algorithm available can be used).

Data structures.

The data structures for the algorithm are:

(1) a polynomial basis G of I
(2) a set S of elements of I (the simplifiers); in the original Buchberger algorithm this coincides with G.
(3) a set B of pairs of elements of G (the pairs to process)
(4) additional auxiliary structures (non existing in the original algorithm)

Subalgorithms.

The subalgorithms are defined as functions, having arguments, values and side-effects on the data structures (no side effect in the description means that side-effects are not necessary to the correctness).

A (Initialize). Arguments: a polynomial basis G. Values: none; Side-effects: initialize the data structures.
B (Selection-strategy) Arguments: none. Values: either an element σ of B; or empty ($\emptyset$). Side-effects: delete σ from B.
C (Compute-Sp) Arguments: a pair σ. Values: a polynomial.
D (Simplification-strategy). Arguments: a polynomial f. Value: either an element $g_i \in G$, and a term T such that $T \cdot Lt(g_i)$ appears as a term of f; or empty.
E (Simplify). Argument: two polynomials and a term. Value: a polynomial
F (Normalize). Argument: a polynomial. Value: a polynomial.
G (Add-polynomial). Argument: a polynomial. Value: none. Side effects: adjust all data structures.
H (Final-post-processing) Argument: none; value: a polynomial basis (the final result). This procedure may use both Simplification-strategy and Simplify.

A simplification strategy is called a Lt-simplification if we always have that $T \cdot Lt(g_i) = Lt(f)$, and a full simplification if we have $\emptyset$ as value only if no $Lt(g_i)$ divides a term of f. To simplify the descriptions, we will always assume that we have an Lt-simplification, but the generalization is straightforward.

The algorithm scheme is the following:

(1) Initialize

(2) Let σ be the value of Selection-strategy; if $\sigma = \emptyset$ then exit with the value of Final-post-processing.
(3) Let $f :=$ Compute-$Sp(\sigma)$
(4) if $f = 0$ then go to (2)
(5) Let $\gamma :=$ Simplification-strategy(f). If $\gamma = \emptyset$ then perform (Add-polynomial(Normalize(f)) and go to (2)
(6) Let $f :=$ Simplify(f, γ); go to (4)

Steps B, D and G (especially the deletion of useless pairs) are the basic strategical points of the algorithm; they decide its combinatorial complexity, and also have a strong influence on the coefficient growth. We review some points concerning them in the next section.

In this paper we are mainly interested in improvements to subalgorithms C, E and F; the aim is to control both the growth of coefficient complexity, and the cost of operations needed to perform this control.

2. On unnecessary reductions

There are at least three points concerning combinatorial strategies in standard base completion algorithm:

(1) the choice of the term–ordering, if we are allowed to choose.
(2) the choice of the order in which we choose the pairs of elements to process and the simplifications to perform. (Selection and Simplification strategies).
(3) the early identification of "useless" pairs, i.e. of pairs of elements (g_i, g_j) of the basis that are not going to contribute a new element to the basis since we eventually reach $f = 0$ at point (4). (Redundancy criteria).

A partial answer to (1) is that "generically" in the homogeneous ideal case one of the best choices is the one that approaches as well as possible the "reverse lex" ordering [**BS**].

The answer to (2) is at present unsettled; many numerical experiments are needed to discover either theorems or heuristics for this problem.

For the third point, two types of results are known: the first type corresponds to choosing a suitable set of generators of the module of relations between the leading terms of the basis; these are binomial relations, and the pairs not involved are useless, if we choose a correct evaluation order. A criterion of this type was first described in [**Bu2**] in an implementable form. See also [**GM**], [**Po**] for improvements. The second type of result says that if $GCD(Lt(g_i), Lt(g_j) = 1$ (extraneous pair) then the pair (g_i, g_j) is useless (the proof is immediate).

D. Lazard had suggested a possibility to extend the extraneous pairs criterion considering the "history" of a monomial. Namely, a "monomial with history" is a pair (m, t) where t is a term that divides the monomial m. When we multiply a monomial by a term, we multiply the history by the same term, and when we add two monomials we take the GCD of their histories. In the completion algorithms we start with empty histories.

The suggestion was that when we have suitable division properties between $Lt(g_i)$, $Lt(g_j)$ and their histories, then the pair is useless. We show a counterexample to the following conjecture, that seems the weakest possible form of this suggestion to be useful.

We state that a pair σ is *eventually useless* (with respect of a previously defined redundancy criterion) if we can omit σ in the standard basis completion algorithm, and still have a correct result; or equivalently, if we keep the redundancy criterion and modify the selection strategy delaying the treatement of σ, we have that σ becomes useless.

(FALSE) CONJECTURE. *If $GCD(Lt(g_i), Lt(g_j)$ divides the histories of g_i and of g_j, then the pair (g_i, g_j) is eventually useless (i.e. is useless if it is evaluated as the last pair).*

COUNTEREXAMPLE: Consider the following polynomials in $k[x, y, z, a, b, c, d]$ with lexicographic ordering:

$$(g_1, g_2, g_3, g_4) = (xz - a,\ x - b,\ yz - c,\ y - d)$$

Then let $Sp(g_1, g_2) \implies g_5 = zb - a$, $Sp(g_3, g_4) \implies g_6 = zd - c$, where $(f, g) \implies h$ indicates that h is obtained simplifying $Sp(f, g)$. One sees rapidly that (g_5, g_6) satisfies the hypothesis of the conjecture, and all other pairs are eliminated by the usual criteria (these have as a particular case the criterion that if $Lt(g_i)$ divides $Lt(g_j)$ and we have already processed the pair (g_i, g_j) then all other pairs (g_j, g_k) are useless). However, $Sp(g_5, g_6) \implies ad - bc$.

The suggestion of Lazard was however very fruitful, since shifting the history concept from the terms to the coefficients we obtain the representation of coefficients in explicitly factored form, that is the core of the GECD concept, and is the subject of the following sections.

3. THE GECD (GREATER EASY COMMON DIVISOR) COMPLETION ALGORITHM

In this section we look more closely at points (C), (E) and (F) of the Buchberger algorithm. As already stated, we assume a Lt-simplification strategy.

Points (C) and (E) allow a common description as pseudo–division. Pseudo–dividing f (or g_i) by g_j means computing a "smallest" monomial m, such that the leading monomial of $m \cdot f$ is multiple of $Lm(g_j)$, (say $m \cdot Lm(f) = n \cdot Lm(g_j)$), and then compute $m \cdot f - n \cdot g_j$. If we assume that all g_j are monic (this can be done if we want to use field arithmetic) then m can be taken monic, and more precisely

$$(*) \qquad \begin{aligned} m &= Lm(g_j)/GCD(Lm(f), Lm(g_j)) \\ n &= Lm(f)/GCD(Lm(f), Lm(g_j)) \ . \end{aligned}$$

The GCD can be taken without any assumption on A since $Lm(g_j)$, being a term, is an explicit product of irreducible elements (the indeterminates). This describes the (classical) points (C) and (E).

To be able to continue with the assumptions that we have made, whenever we add an element to the basis we have to make it monic; this is step (F), and this introduces denominators.

If we want to use arithmetic on A, we cannot make the polynomials monic, hence the remark above on the existence of $GCD(Lm(f), Lm(g_j))$ does not apply. At this point we have two possibilities;

- if A is a GCD–domain, the formula above can remain unchanged; we can moreover make a polynomial primitive (step F) whenever we add it to the basis. However, the result is that most of the algorithm time may be spent in GCD computations.
- If instead we do not have GCD's on A, or we don't want to use them, we have to take $GCD(Lt(f), Lt(g_j))$ instead of $GCD(Lm(f), Lm(g_j))$ in (*) (we want to interprete this as a purely combinatorial, hence easy, GCD of monomials). Moreover, we cannot make polynomials primitive (step (c) is empty). The result is that we have very rapid coefficient growth.

We have seen that systematic GCD may be very costly, and no GCD is even worse, since coefficients grow intolerably. The best thing to do, is to use, instead of GCD, "the largest common divisor that we can find with reasonable effort".

Since we want to give many variations of this concept, we want to formalize it, defining a *Greater Easy Common Divisor* (GECD) of elements of A (and of monomials in $A[X]$). We want to use GECD instead of GCD in formula (*) to describe the pseudo–division process.

More precisely, we want to modify the reduction algorithm precisely at those points where we have GCD computations, namely C, E and F. For all other points of the algorithm (in particular, strategies and useless pair elimination) any correct subalgorithm can be used. Since we need additional data structures, also subalgorithm A is involved, and also step H, for which we do not describe the obvious details, if we want the result to be a reduced standard basis.

We want to use in the GECD any knowledge that we have, hence GECD is not a map $A \times A \to A$, but a map $A \times A \times context \to A$. The formal meaning of *context* will be defined in the different specializations of the algorithm. Context–free examples are the GCD (when existent) and a constant map with value 1, and lead to the two classical versions of the algorithm described above. We always write $GECD(a, b)$ without explicitly mentioning the context.

The GECD of two monomials m_1, m_2 has the following properties:

- $GECD(m_1, m_2) = m$ is a common divisor of m_1 and m_2.
- We can compute m_1/m and m_2/m
- If $m_3 = GECD(m_1, m_2)$ and $m_i = a_i t_i$ is the coefficient–term decomposition, then $t_3 = GCD(t_1, t_2)$.

(remark that these are precisely the points needed to be able to perform the computations indicated in (*))

We can also talk of GECD of more polynomials, (we do not insist on associativity), and of LECM (least easy common multiple), $LECM(m_1, m_2) = m_1 m_2 / GECD(m_1, m_2)$

To understand the spirit of the GECD definition, consider the reduced and primitive PRS algorithms for GCD calculations; in the course of the algorithm we know at some points that all coefficients of some polynomial are multiple of some previous coefficient; we can take this as GECD (it is the easiest ever possible: no computation at all!). These systematic factors can be known "a–priori" due to the deterministic, almost straigth–line character of the Euclidean algorithm (we always know to which elements we have to apply the pseudo–division algorithm, and the only branch point is the termination check). This does not happen with Buchberger algorithm, so the same type of procedure cannot be applied. We try in different versions to simulate this knowledge with different types of bookkeeping.

4. Data structures for coefficient partial factorization Partial factorization GECD

We describe two types of structures for elements of A, that give good "contexts" for GECD calculations. For both we maintain a vector $C = (c_i)$ of elements of A, called *systematic factors*, and in both an element of A is represented as a pair (a, ν) (called generalized element of A), where $a \in A$ and $\nu = (n_i)$ is a vector of non–negative integers. The two structures differ in the element of A that they represent. We say that (a, ν)

represents α in partially factored form (p.f.f.) if

$$\text{(p.f.f.)} \qquad \alpha = a \cdot \prod_i c_i^{n_i};$$

Instead, (a, ν) represents $\alpha \in A$ with explicit divisors (e.d.f.) if

$$\text{(e.d.f.)} \qquad a = \alpha, \qquad \prod_i c_i^{n_i} \text{ divides } a.$$

The product of generalized elements is $(\times, +)$; the sum is different for the two representations;

$$\text{(p.f.f.)} \qquad (a,\nu) + (b,\mu) = (a \cdot C^{\nu - \nu\wedge\mu} + b \cdot C^{\mu - \nu\wedge\mu}, \nu \wedge \mu)\,.$$
$$\text{(e.d.f.)} \qquad (a,\nu) + (b,\mu) = (a + b, \nu \wedge \mu)$$

(where $\wedge$ denotes componentwise inf, and C^ν denotes $\prod c_i^{\nu_i}$).

We define the GECD with the following formulas:

$$\text{(p.f.f.)} \qquad \begin{aligned} GECD((a,\nu),(b,\mu)) &= (1, \nu \wedge \mu) \\ (a,\nu)/GECD((a,\nu),(b,\mu)) &= (a, \nu - \nu \wedge \mu)\ ; \end{aligned}$$

and

$$\text{(e.d.f.)} \qquad \begin{aligned} GECD((a,\nu),(b,\mu)) &= (C^{\nu\wedge\mu}, \nu \wedge \mu) \\ (a,\nu)/GECD((a,\nu),(b,\mu)) &= (a/C^{\nu\wedge\mu}, \nu - \nu \wedge \mu)\ . \end{aligned}$$

The two forms are equivalent, but not computationally: one trades many multiplications for fewer exact divisions.

A generalized monomial (a, ν) is called *almost–monic* if $a = 1$ (p.f.f.) or $a = C^\nu$ (e.d.f.). A polynomial is almost–monic, if its leading monomial is almost–monic. A polynomial is called *quasi–primitive* if the GECD of the coefficients of its monomials is 1.

Remarks on exact divisions. We do not know the existence of computable rings in which no exact division exist (but we strongly suspect that they can be constructed adapting the example of [**vdW**] of a computable ring non explicitly factorial).

We want to remark that the exact division test $a|b$ corresponds to checking ideal inclusion $bA \subseteq aA$, and computing the exact quotient a/b corresponds to finding a generator of $bA : aA$. Hence finitely generated rings allow exact division test and exact division. These however may be costly, especially the second. We want to analize the theme with some details.

- If $A = \mathbf{Z}$, $A = \mathbf{Z}[X]$, $A = \mathbf{F}_q[X]$ ($\mathbf{F}_q$ is the finite field with q elements) then exact division is easy, being reduced to linear algebra. Remark however that sparseness is not preserved.
- If $A = k[X]/J$ or $A = \mathbf{Z}[X]/J$ where J is a prime ideal, then to check and to perform exact division of elements $a \in A$ one has to consider the ideals $J_a = aA + J$. The direct computation of $J_a : J_b$ finding then d such that $J_d = J_a : J_b$ is possible but is an hard problem.

In the second case, since we are going to divide several times by the same divisor, we can improve the performance with a pre–processing of the data for any systematic divisor c.

Compute a standard basis for J_c, $(\gamma_1, \ldots, \gamma_m)$, and represent $\gamma_i = g_i \cdot c + h_i$, $h_i \in J$. This can be obtained at (almost) no extra cost from the completion algorithm for J_c.

Now, to test divisibility of a by c, we have to test $a \in J_c$, and this can be made reducing a with the γ_i, obtaining $a = \sum a_i \gamma_i + r$, where r cannot be further reduced. Then, if $r = 0$, $a = c \cdot \sum a_i g_i \mod J$, otherwise a is not divisible by c. This shows that it is convenient to store, along with the systematic factors, the lists (γ_i) and (g_i).

We can also remark that when we work with an $A = \mathbf{Z}[X]/J$ or $A = k[X]/J$ it is somtimes better to change the ring A; namely, we are interested in the computation on the quotient field K of A, so any other A' having the same quotient field will suffice. For example, if A' can be represented by $k[Y]/L$, where k is a field and L is a principal ideal, the exact divisions are usually simpler. However, we have to check that the data of our original problem do not become excessively harder.

When exact divisions are not easy, the p.f.f. of generalized monomials has to be preferred. In some versions of the algorithm we have nevertheless to perform exact divisions, when we maintain the systematic factor list.

Another advantage of the p.f.f. is that the length of the representation of a coefficient is far smaller: $(1, [100])$ is smaller than $(c^{100}, [100])$. However, when doing additions we have often to redo over and over the same multiplications of systematic factors. We still do not have enough experimental evidence to give an heuristic strategy for the choice of the representation.

5. Maintenance of systematic factors and post-processing. Partial factorization GECD completion

In this section we describe a series of ideas to design different forms of the subalgorithms F and G of the completion algorithm as described in section 1. One of these forms is in course of being implemented, and we give a more precise description in section 7.

The core of the algorithm is the maintenance of the systematic factor list, adding to it new factors appearing in the course of the algorithm. We expose only a few of the many versions that we have considered, which seem to be the best ones and seem not to require an overlong list. The different versions correspond to different properties of the ring A that we are willing to use (existence of exact division, of GCD, of factorization).

In these versions the list maintenance is done when we add a new element of the basis (at the same moment that we make its post–processing and the maintenance of the critical pair stack). More precisely, one of the meanings of the list maintenance is to allow to represent the new element as almost–monic, almost–primitive polynomial.

The list maintenance consists in adding new elements, and maybe splitting or deleting some old ones. Whenever we add new elements, no maintenance is needed in the representation of the existing monomials, but when we split or delete an element then the existing monomials have to be reviewed. Of course, when implementing the algorithm this can be delayed, e.g. replacing an element of the factor list with the list of its factors, or marking it for deletion, and adjusting the monomials when they have to be used. We describe a form of this list maintenance with delayed updating (but without deletions) in section 7.

To simplify the description, we assume that we use the partially factored form, but the algorithms apply with minor changes to the explicit divisors form.

The simplest form of the algorithm starts with the factor list coinciding with the list

of leading coefficients of the elements of the basis (deleting duplicates and invertible elements). The leading monomials of the basis are trivially partially factored, making them hence almost–monic.

Whenever we add a new element f to the basis, let (a, ν) be its leading coefficient. Then, if a is invertible, divide f by it; if a already appears in C, increase the corresponding n_i in ν; otherwise, add a to C. This makes f almost–monic (without any division, indeed). Moreover, make f almost–primitive, dividing it by the GECD of its coefficients.

We can make several modifications to the procedure. We give a non–exaustive list:

(1) We may ask that no element of C is a multiple of another; then, when adding an element, we have to check this condition, and maybe split one of the elements. We have thus to continue the checking with the new factors.

(2) Assuming that A has GCD's, we may ask that the elements of C are coprime. Then when adding an element c we have to check that its GCD with any existing element c' is 1, otherwise we have to split both c and c'. The control does not propagate, since the factors of c' are already coprime with the already existing elements. This choice is explained with many details in section 7.

(3) Assuming that A has factorizations, we may ask that the elements of C are irreducible. Hence, factor an element before adding him.

(4) When we add an element f to the basis, we ask that it is almost–primitive, i.e. we have to divide its coefficients by their GECD. But we may ask before doing that to discover the "concealed factors", i.e. we may test for division by elements of C the unfactored part of all coefficients of f (of course, only factors dividing all the preceding factors have to be tested for the next). This too is detailed in section 7.

(we refer as version (0) the basic form with no variation).

Point (4) may be tricky if elements of C may have common multiples not divisible by the product (e.g., they may have common factors) since we have to choose for which element we have to divide. So it is easier to implement point (4) when point (2) is implemented.

With feature (4) the GECD completion algorithm contains as a subcase (when $K[X]$ is a univariate polynomial ring) the Reduced and Improved PRS algorithm of Brown and Collins, [**Lo**]. Of course, the PRS algorithms are better, since they do not need any bookkeeping, but the coefficients are the same.

Deletions. When we add many elements to the factor list, and especially when we systematically split them, the list may become unmanageable. Moreover, the PRS algorithms show that the full coefficient list is useless, since we may forget the factors when we have used them, hence in this case at most a list of three factors needs to be kept.

We want to show what one has to do to delete a systematic factor from the list. The actions are different in p.f.f. and e.d.f.; namely, in e.d.f. we have simply to delete the reference to the corresponding factor (and this may be delayed, simply binding to a "deleted factor" reference the value of the deleted factor; when treating a generalized coefficient (ν, α) with such a reference, we can delete it in the ν list).

When we use the p.f.f. instead, assuming e.g. that we delete c_1, we have to transform all (ν, α) into $(\nu', c_1^{n_1}\alpha)$, where n_1 is the first element of ν, and ν' is ν with the first item deleted. This too could be delayed, but we need however to keep a list of the deleted factors; moreover, we lose the advantage of having shorter representations. So maybe, in this setting, we can simply label a factor as "deleted", not to use it in the division tests.

At present, we do not have an heuristics to delete useless factors. This themes shall be

considered only after extensive experiments.

6. Modular GECD completion algorithm

In the previous section we have described a main form of maintenance of the factor list, with three variations, and a main post–processing for the polynomial to add to the basis with a variation (n. (4) of the list). If we make an example of GCD calculation for two dense univariate polynomials of the same degree with the basis algorithm, (version (0)) we remark that the algorithm leads to absolutely nothing, since no systematic factor of the coefficients is evidenced. The basis algorithm works fairly in very sparse situations (and this is often the case in multivariate polynomials).

To recover the full strength of the reduced and improved PRS algorithm, we need to apply at least variations (1) and (4). We have remarked that (1) may be intriguing, so if GCD in A exists and is not too expensive we would rather choose (2) and (4).

We show how a modular approach can help in reducing the cost of GCD computations or of exact division tests. To simplify the discussion, we assume that A is a GCD domain, that P is a prime ideal of A and that A/P is an euclidean domain (but not a field). The most common examples are:

- $A = k[t_1, \ldots, t_n]$, P is generated by $n-1$ linear forms, hence $A/P \cong k[t]$;
- $A = \mathbf{Z}[t_1, \ldots, t_n]$, P is generated by $t_i - a_i$, hence $A/P \cong \mathbf{Z}$;
- $A = \mathbf{Z}[t_1, \ldots, t_n]$, $A/P \cong \mathbf{Z}/p\,[t]$ where p is a prime number (indeed, the generator of $P \cap \mathbf{Z}$).

We have an ideal $I \subseteq A[X]$, and we want to compute its standard basis. We assume that I reduces "well" with respect to P. This can be expressed abstractly as a flatness condition, (namely, normal flatness of a suitable presentation of the ideal), but practically it means that the polynomials that we consider in the course of the algorithm do not have leading coefficient in P. This condition is verified in the course of the algorithm, and may lead to failure of the algorithm: recovery is made choosing another P (we can use all the computations up to that point). This is possible if A has infinitely many suitable prime ideals, and this is indeed the case if A is finitely generated.

We run parallel completion algorithms for A and A/P. This means that we run on A/P the GECD completion algorithm for I/P with variations (2) and (4). At the same time we run a completion algorithm on A for I; this algorithm is lead by the algorithm on A/P, i.e. we use the selection strategy and the simplification strategy of the algorithm for A/P also for the algorithm on A. The assumption that I reduces well means that these choices are applicable and correct.

We keep the list of systematic factors for the computations on A, in parallel with the computation on A/P, and we use the GECD found on A/P as an "oracle" to look for GECD on A. Of course, it may happen that common divisors existing on A/P do not correspond to common divisors in A. If this happens we can either discard P and trying another, since for generic P the GCD reduces to the GCD, or (better) consider on A/P only those divisors that lift to A.

The advantage of this form of the algorithm is that we make many exact division tests and many GCD on A/P where they are less expensive, and when the result is negative we can avoid completely the computation on A.

One can remark that this is also made with the modular GCD computations on A. However, the idea can be generalized to the general situations (we only need for A an exact division test and procedure), using for A/P variations (1) and (4). This should be

in general the best choice; indeed, for a basis G with non–special coefficients (1) and (2), if combined with (4), should give the same result.

Indeed, if on A and A/P we have GCD's we might as well choose a mixed method between (1) and (2), using GCD only to resolve ambiguities of (1).

This algorithm can be seen as an "GECD completion procedure": the GECD in A is defined as the GECD in A/P lifted to A (modular GECD).

Remark moreover that the algorithm combines well with the modular algorithm (see [**Tr**]) if a probabilistic algorithm is sufficient. Namely (referring to the notations of the above paper), on can build a three–step *trace reconstruction algorithm*: take $P \subset M \subset A$, where M is a maximal ideal, and P a prime ideal if dimension 1; then find a Gröbner trace in A/M, perform the trace–lifting algoritm for A/P and A as above, addimg to the trace–lifting algorithm the feature of modular GECD.

Avoiding the full completion. In many applications, the completion of a family of polynomials into a standard basis is only an intermediate computation. It is hence important sometimes to find methods that short–circuits the full completion, since often partial results are sufficient. The following questions are often encountered:

(1) Prove that $1 \in I = (f_1, \ldots, f_n)$, i.e. find q_i such that $1 = \sum q_i f_i$.
(2) Given $I = (f_1, \ldots, f_n)$ find $f \in I$ depending only on the variables $(x_1, \ldots, x_d)$ (elimination). Find q_i such that $f = \sum q_i f_i$.
(3) Find a "non trivial" relation between $f_1, \ldots, f_n$, i.e. $(q_1, \ldots, q_n)(f_1, \ldots, f_n)^t = 0$.
(4) Given a matrix N with polynomial entries, find a matrix M of rank s such that $M \cdot N = 0$.

In all these problems, if a solution is suggested by an oracle, a direct check can be easily performed.

7. Implementations: present and near future.

The present state of the implementation is the following: we are concentrating on the form of the algorithm containing variations (2) and (4) (hence assuming that we can compute GCD on A with a relative ease). The predicted behaviour of the algorithm should be good if exact division and division test are easy, and GCD is not so easy. A ring of multivariate polynomials is the typical example.

We have made two implementations on $\mathbf{Z}$; the first one, in LeLISP, is an implementation of the basic (form (0)) algorithm, with a larger systematic factor list (also intermediate coefficients contribute to systematic factors). In this form of the algorithm, special strategies are needed to control the length of the systematic factor list. The second implementation is in MuLISP, and is a part of the AlPi package [**Tr1**]. The explicit factors of the coefficients are not kept, but are recomputed every times that we have to do a GECD. More precisely, every time that we have to compute $GECD(a, b)$, we first completely factor a with respect to the systematic factor list by trial exact division, then look for the same factors in b. This is obviously a meanigless procedure if you look for efficiency, but it has the advantage of being quite easy to implement just modifying a few function definitions of an existing implementation, and from the other side one can simulate the behaviour of the full algorithm; in particular, one can see the behaviour of the systematic factor list and the GECD's computed. Hence this implementation can be seen as a simulation of some combinatorial points of the algorithm. We report the results of some of the tests in section 8.

The results were encouraging: the factor list is usually of a size comparable to the size of the standard basis found, and also the timings are not bad compared with the usual algorithm (the running time is in average only 15% higher than with the usual algorithm)

We have done a complete analysis of a full implementation (always with variations (2) and (4)). The implementation is in course taking as coefficients univariate polynomials, and will require only small modifications to our current implementations. A complete implementation will be done in SCRATCHPAD–II, to be able to use the more general type of coefficients, where we hope to give statistical evidence of the good behaviour the algorithm in the multivariate polynomial case.

In the rest of the section we give a definition of the algorithms following the analysis.

We start from elements of A, considered as primitive structures; on elements of A we have equality test, arithmetic operations including division test and exact division, invertibility test and inverse, and algorithms for GCD. We define now the derived structures and algorithms.

To simplify the notations we use p.f.f., but e.d.f. is not substantially different. The implementation will contain both.

Data structures.

(1) The *current factor list* (c.f.l) is a set of coprime elements of A. The other structures are defined starting from the c.f.l..

(2) A *current generalized element* is a pair (α, a), where α (the factored part) is a list of pairs (c, n), where c is in the c.f.l. and n is a positive integer, and $a \in A$ (the unfactored part). It is updated if α does not contain two (c, n), (c, m) with the same c. A c.g.e. is *totally factored* if the unfactored part is invertible.

(3) The *systematic factor list* (s.f.l.) is the union of the c.f.l. and a list of updated, totally factored generalized elements. (old factors list).

(4) A *generalized element* (g.e.) is defined as a c.g.e., but referring to the s.f.l. instead of the c.f.l.. It is irredundant if it does not make reference to duplicate elements of the s.f.l..

One can make an element irredundant replacing for every a all the pairs (a, m_i) with a unique pair $(a, \sum m_i)$.

We have obvious coercions between the different structures. The most important is the *updating*, sending a g.e. to an updated c.g.e., done replacing every occurrence (α, n) where $\alpha = ((a_1, n_1) \ldots (a_k, n_k))$ by $((a_1, nn_1), \ldots, (a_k, nn_k))$ and making the result irredundant.

Operations.

We refer to the operations defined in section 4. The formula for product is directly applicable, but the formula for sums should be applied only to updated elements; hence we first update elements, then make the sum.

The *concealed factors retrieval* is a kind of coercion from A to generalized elements, and sends a to (α, a') such that no systematic factor divides a'. The obvious algorithm is composed of subsequent trial divisions. Concealed factors retrieval can also be applied to generalized elements.

We have to use two types of GECD: we call them the "plain" and the "concealed" (GECCD). Let $\alpha = ((a_1, n_1), \ldots, (a_k, n_k))$ be a totally factored, updated element, and (β, b) a generalized element. Then update β, obtaining β'. The plain GECD is obtained as $\alpha \wedge \beta'$.

The GECCD between a totally factored c.g.e. (α, a) and $b \in A$ is defined as the composition of concealed factors retrieval on the second factor and GECD. It can be however more efficiently computed as follows:

(1) if α is empty, then the result is 1
(2) if a_1 does not divide b, then it is equal to the GECCD of $((a_2, n_2), \ldots, (a_k, n_k))$ and b
(3) otherwise is the product of $((a_1, 1))$ and $GECCD(\alpha/((a_1, 1)), b/a_1)$.

To compute the GECCD between α and (β, b), let $\delta = GECD(\alpha, \beta)$.

Then $GECCD(\alpha, (\beta.b)) = \delta \cdot GECCD(\alpha/\delta, b/\Gamma(\delta)$, where Γ is the coercion from totally factored elements to A.

Updating the systematic factor list.

As remarked in section 3, when we normalize an element before adding it to the basis, we want to make it almost-monic and almost-primitive, hence we may have to update at the same time the systematic factor list.

The updating needs a subalgorithm (add-and-split) that, given a list L of elements of A pairwise coprime elements of A, and an element $b \in A$ produces a new list L' of pairwise coprime elements and a concealed factors retrieval for all elements of L and for b.

ALGORITHM **split-two.**

Input: two elements a, $b \in A$.

Output: a_0, b_0, $c_1, \ldots, c_n \in A$ *pairwise coprime,* $r_1, \ldots, r_n$, $s_1, \ldots, s_n$ *positive integers such that* $((a_0, 1), (c_i, r_i)) = a$, $((c_i, s_i), (b_0, 1)) = b$.

The algorithm runs as follows: consider a list L initialized at (a, b). Represent $a = ((a, 1))$, $b = ((b, 1))$

LOOP. Repeat until all adjacent elements of L are coprime:

Pick two non-coprime elements (u, v) of L (they are always adjacent)

Let $d = GCD(u, v)$; replace in L the pair (u, v) with the triple $(u/d, d, v/d)$ (remark that u/d, v/d are coprime); represent $u = ((u/d, 1), (d, 1))$, $v = ((v/d, 1), (d, 1))$ and update a, b.

(ENDLOOP)

Delete invertible elements from L, modifying suitably the representations of a, b (at the level of the invertible factor). At this point, $L = (a_0, c_1, \ldots, c_n, b_0)$.

The proof that the algorithm does what we have stated is almost immediate, considering the assertions made during the algorithm.

ALGORITHM **add-and-split.**

Input: a list L *of elements of* A*, pairwise coprime; an element* $b \in A$

Output: a list L' *of elements of* A*, pairwise coprime, and representations of* b *and the elements of* L *as totally factored elements in* L'.

The algorithm runs as follows: if b is invertible output L; if L is empty output (b); otherwise, perform the split-two algorithm for the pair (a, b) where a is the first element of L; let L'' be its output, and b' its last element. Append L'' (with last element, b', removed) and the list output of **add-and-split**(L''', b'). Update the representation of the input.

The standard basis GECD completion algorithm.

The completion algorithm is specified, as outlined in sections 1 and 3, describing the GECD at different places, and describing the operation to perform when we obtain a new element of the basis.

(1) We represent polynomials with generalized coefficients
(2) When making divisions, we use the plain GECD
(3) when we obtain a new element g of the basis, let (α, b) be its lead coefficient. Update the s.c.l. with the add-and-split algorithm (input: the old s.c.l. and b)
(4) before adding g to the basis, divide it by the GECCD of its coefficients.

The initial installation of the s.c.l. is obtained from the empty list, adding the elements of the original basis using the above recipe.

Avoid the updating. Frequent updating can be time-consuming, and useless if we are updating elements already updated. Hence it might be useful to keep for every element an "updating tag", i.e. an integer stating "when" the element was updated for the last time. One has to keep the "current date", increasing it every time that the add-and-split algorithm changes the current factor list. Elements updated in the current date do not need further updating.

8. Examples

We show here with some details one of the first examples that we have tried, and this, even if it is very short, already shows the power of the algorithm. The computation has always the same trace, for every non–zero value of seven parameters; we show the example with symbolic parameters (a, b, c, d, e, f, g), reconstructed by an example computed numerically. We consider $G = (g_0, g_1, g_2) = (ax^3 + byz + c,\ dxy + ez^2,\ fxz + gy)$ with lexicographic ordering $x > y > z$. The trace of the algorithm is the following:

$$(1,2) \longrightarrow g_3 = c_3y^2 - c_2ez^3$$

$$(1,3) \xrightarrow[c_1]{} \xrightarrow[c_2]{2} 0$$

$$(0,1) \xrightarrow{2} \xrightarrow{1} \xrightarrow[c_1]{3} g_4 = c_1^2c_2c_3c_4y + c_1c_2^2bez^4 - c_0c_3^2e^2z^3$$

$$(4,1) \xrightarrow[c_1]{} \xrightarrow[c_2]{2} \xrightarrow[c_1c_2c_3]{2} \xrightarrow[c_1c_3c_4]{4} g_5 = c_1^2c_2^4c_5z^7 - 2c_0c_1c_2^2c_3^2be^3z^6 + c_0^2c_3^4e^4z^5 - c_1^3c_2^3c_3c_4^2ez^2$$

$$(4,3) \xrightarrow[c_1c_3]{} \xrightarrow[c_1c_2]{4} \xrightarrow[c_1c_3c_4]{4} \xrightarrow[c_1^2c_2^4c_5]{5} 0$$

$$(0,2) \xrightarrow{1} \xrightarrow{2} \xrightarrow[c_1c_2]{4} \xrightarrow[c_1c_3c_4]{4} g_6 = c_1^2c_2^4c_6z^6 - 2c_1c_2^2c_3^2be^2z^5 + c_0^2c_3^4e^3z^4 - c_1^3c_2^3c_3c_4^2z$$

$$(5,6) \xrightarrow[c_1^2c_2^4]{} 0$$

$$(2,6) \xrightarrow[c_2]{2} \xrightarrow[c_2]{2} \xrightarrow[c_1^2c_2c_3]{2} \xrightarrow[c_1c_2c_3c_4]{4} \xrightarrow[c_1c_3c_4]{4} \xrightarrow[c_1^2c_2c_3c_4]{4} \xrightarrow[c_1^2c_2^4]{5} \xrightarrow[c_1^2c_2^4]{5} 0$$

The trace is interpreted in the following way: every line is the simplification of a critical pair; every arrow corresponds to a pseudo–division, the superscript, if existing, is the

index i of the dividing element g_i; otherwise the division is a coefficient division. The subscript is the divisor discovered by the GECD; if it does not exist, this divisor is 1. The elements $c_0, \ldots, c_6$ are the systematic factors following the basic algorithm (version (0)). The values of $(c_0, \ldots, c_6)$ are $(a, d, f, g, c, b^2e^2, b^2e)$ (remark that we have not used the fact that c_6 divides c_5; indeed, this would have allowed to split c_5 (as $c_6 \cdot e$) and then c_6 (as $e \cdot b^2$).

Here we show other examples, computed with the AlPi package; for each we indicate, in the order, the term-ordering (R or L for Rlex and Lex), the number of elements of the basis computed and of the reduced standard basis, the final factor list (with the algorithm described in section 7), and the number of simplifications needed for the algorithm. In the algorithm we have followed the following strategies (see [**Tr1**] for the description): for the pair selection strategy, process first the pairs such that the "guessed $Lt(Sp)$" has lowest degree, or is lower in the term-ordering; for the simplification strategy, select the shortest (with less monomials) element of the basis. When simplifying a polynomial, perform total reduction with respect to the already computed basis. Do not give priority to interreductions.

$(x+y+z+t+u,\ xy+yz+zt+tu+ux,\ xyz+yzt+ztu+tux+uxy,$
$xyzt+yztu+ztux+tuxy+uxyz,\ xyztu+1)$
$R,\ 51,\ 20,\ (2,3,5,7,11,17,23,29,37,47,1013,1367,2213),\ 1258$
(starting from the above computed standard basis)
$L,\ 47,\ 11,\ (7,2,11,5,3,229,53,673,151,136141),\ 991$
$(17x^3-23ytu-z+71u^2,\ 101xz^2+41xtu+7,\ 91xz^2+11yt-5)$
$R,\ 7,\ 6,\ (17,101,91,41,23,11,71),\ 38$
$L,\ 27,3,\ (17,101,13,7,41,11,1142,5,61,3,593),\ 260$
$(1735y^2z-734xu^2-361xu+173,\ 183y^2-3172zt-2964xu,$
$137xyzt-3812xt^2+4285y^2u+183)$
$R,7,\ 6,\ (137,1735,3,61,13,2,19,367),\ 74$

More details on the computations and other examples are found in [**GPT**].

References

Ba Bayer, D. A., *The division algorithm and the Hilbert scheme*, Ph.D. thesis, Harvard (1982).

BS Bayer, D. A., Stillman, M., *A theorem on refining division orders by the reverse lexicographic order*, Duke Math. J. **55** (1987 ;), 1–11.

BCL Buchberger, B., Collins, G. E., Loos, R., "Computer Algebra," Springer Verlag, Wien-New York, 1982.

Bu1 Buchberger, B., *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*, Aequationes Mathematicae **4** (1970), 374–383. (Ph.D. Thesis, Math. Inst. Univ. Innsbruck, 1965).

Bu2 ———, *A Criterion for Detecting Unnecessary Reductions in the Construction of Gröbner Bases*, in "EUROSAM 1979," Lecture Notes in Computer Science **72**, Springer Verlag, Berlin-Heidelberg-New York, 1979, pp. 3–21.

Ga Galligo, A., *Algorithmes de calcul de base standard*, Université de Nice (preprint) (1982).

GPT Galligo, A., Traverso, C., Pottier,L., *Appendix to: Greater Easy Common Divisor and standard basis completion algorithms* , Rapports de Recherche (1988), INRIA, centre de Sophia Antipolis.

GM Gebauer, R., Möller, H. M., *An installation of of Buchberger's algorithm*, J. of Symbolic Computation (to appear).

Lo Loos, R., *Generalized polynomial remainder sequences*, in "Computer Algebra," (**BCL**), pp. 115–137.

Po Pottier, L., *Algorithmes de completion, constructions, preuves, stratégies, applications.* (to appear) ;(submitted to "1988 IEEE Symposium on Logic in ;Computer Science")

Tr Traverso, C., *Gröbner trace algorithms*, (these proceedings).

Tr2 ———————, *AlPi: a package to test different versions of the Gröbner basis completion algorithm..* (to appear)

vdW van der Wærden, B. L., "Moderne Algebra," Springer Verlag, Berlin-Heidelberg-New York.

†Départment de Mathématiques — Parc Valrose — F–06034 NICE CEDEX

‡Dipartimento di Matematica — Via F.Buonarroti 2 — I–56100 PISA