A Modular GCD algorithm over Number Fields presented with Multiple Extensions.

Mark van Hoeij[*]
Department of Mathematics
Florida State University
Tallahassee, FL 32306-4510, USA.

Michael Monagan[†]
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada. V5A 1S6.

ABSTRACT

We consider the problem of computing the monic gcd of two polynomials over a number field $L = \mathbb{Q}(\alpha_1, \ldots, \alpha_n)$. Encarnacion, Langemyr and McCallum have already shown how Brown's modular GCD algorithm for polynomials over $\mathbb{Q}$ can be modified to work for $\mathbb{Q}(\alpha)$.

Our first contribution is an extension of Encarnacion's modular GCD algorithm to the case $n > 1$ without converting to a single field extension. Our second contribution is a proof that it is not necessary to test if p divides the discriminant. This simplifies the algorithm; it is correct without this test.

Our third contribution is the design of a data structure for representing multivariate polynomials over number fields with multiple field extensions. We have a complete implementation of the modular GCD algorithm using it. We provide details of some practical improvements.

1. INTRODUCTION

We recall the relevant details of the so called *modular GCD algorithm* first developed by Brown in [3] for polynomials over $\mathbb{Z}$ and then by Langemyr and McCallum in [7] and Encarnacion in [4] for polynomials over $L = \mathbb{Q}(\alpha)$, which we shall generalize to $L = \mathbb{Q}(\alpha_1, \ldots, \alpha_n)$. First some notation.

We denote the input polynomials by f_1 and f_2, their monic gcd by g. The *cofactors* are the polynomials f_1/g and f_2/g. If $f \in \mathbb{Q}[x]$ then let the *denominator* $\mathrm{den}(f)$ be the smallest positive integer such that $\mathrm{den}(f)f \in \mathbb{Z}[x]$. See section 2 for the definition of $\mathrm{den}(f)$ if $f \in \mathbb{Q}(\alpha_1, \ldots, \alpha_n)[x]$. The *associate* $\tilde{f}$ of f is defined as $\tilde{f} = \mathrm{den}(g)g$ where $g = \mathrm{monic}(f)$. Here $\mathrm{monic}(f)$ is defined as $\mathrm{lc}(f)^{-1}f$ where $\mathrm{lc}(f)$ is the *leading coefficient* of f. Define the *semi-associate* $\check{f}$ as rf where r is the smallest positive rational number for which $\mathrm{den}(rf) = 1$.

[*]Supported by NSF grant 0098034.

[†]Supported by NSERC of Canada and the MITACS NCE of Canada.

Computing the associate $\tilde{f}$ is useful for removing denominators, but could be expensive if $\mathrm{lc}(f)$ is a complicated algebraic number. So we preprocess the input polynomials in our algorithm by taking the semi-associate instead. If $\mathrm{lc}(f) \in \mathbb{Q}$ then the two notions are the same up to a sign:

$$\check{f} = \pm\tilde{f} \iff \mathrm{lc}(f) \in \mathbb{Q}$$

Examples: If $f = 2x - 2/3$ then $\check{f} = \tilde{f} = 3x - 1$. If $\alpha = \sqrt{2}$ and $f = -\alpha x + 1$ then $\check{f} = f$, $\mathrm{monic}(f) = x - \alpha/2$ and $\tilde{f} = 2x - \alpha$.

The *modular GCD algorithm* computes the *monic gcd* $g \in L[x]$ of f_1 and f_2. It does this by reducing f_1, f_2 modulo one or more primes and calling the *Euclidean algorithm mod p* for each of these primes p. If p is a good prime, the Euclidean algorithm mod p returns g mod p. The modular GCD algorithm reconstructs g from these modular images. Only good primes should be used during the reconstruction for it to be successful. However, not all primes are good. We distinguish the following cases:

Definition 1. *Let $f_1, f_2 \in L[x]$ and g be their monic gcd. We will distinguish four types of primes.*

- lc-bad primes. *Let $m_1, \ldots, m_n$ be the minimal polynomials of the field extensions $\alpha_1, \ldots, \alpha_n$. So $m_i(z)$ is a monic irreducible polynomial in $\mathbb{Q}(\alpha_1, \ldots, \alpha_{i-1})[z]$ and $m_i(\alpha_i) = 0$. If $\mathrm{den}(f_1)$, $\mathrm{den}(f_2)$ or any leading coefficient of $\check{f}_2, \check{m}_1, \ldots, \check{m}_n$ vanishes mod p then we call p an* lc-bad prime.
- Fail primes. *If p is not an lc-bad prime, and the Euclidean algorithm mod p returns "failed", then p is called a* fail prime.
- Unlucky primes. *If p is not an lc-bad prime nor a fail prime, and if the output of the Euclidean algorithm mod p has higher degree than g, then p is called an* unlucky prime.
- Good primes. *A prime p is called a* good prime *if the Euclidean algorithm mod p returns g mod p. Theorem 1 in section 2 says that all primes that are not lc-bad are either fail, unlucky or good.*

Remarks:

1. Our definition of lc-bad prime is not symmetric in f_1, f_2. It could be that p is lc-bad for f_1, f_2 but not lc-bad for f_2, f_1. In that case, because of how we set up the algorithm, we should either: not use p, or: interchange f_1, f_2 mod p before calling the Euclidean algorithm mod p.

2. Our definitions are not the same as the standard definitions in [3]. For example, it is possible that the Euclidean algorithm mod p fails even if the monic gcd of f_1 mod p, f_2 mod p exists and equals g mod p. We call such p a fail prime and not a good prime. This distinction is not necessary if $f_1, f_2 \in \mathbb{Q}[x]$ where there are no fail primes.

3. If $p \mid \text{den}(g)$ (in the standard definition these primes are called bad primes) then g mod p is not defined and so p can not be a good prime. According to theorem 1, p must then be either lc-bad, fail, or unlucky.

4. Minimal polynomials are monic so the leading coefficients of $\breve{m}_1, \ldots, \breve{m}_n$ are $\text{den}(m_1), \ldots, \text{den}(m_n) \in \mathbb{Z}$. However, $\text{lc}(\breve{f}_2)$ is in general not an integer but an algebraic number.

5. It is very easy to tell if a prime p is lc-bad or not, but we can not tell in advance if p is fail, unlucky, or good. So we will end up calling the Euclidean algorithm mod p with fail, unlucky, and good primes but never with lc-bad primes.

1.1 lc-bad primes

If $f_1 = 5x + 1$, $f_2 = 5x - 1$ and $p = 5$ then p satisfies our definition of an lc-bad prime as well as the definition of a good prime. However, there are good reasons not to use any lc-bad prime. Take for example $f_1 = f_2 = 5x + 1$. Also, the proof of theorem 1 requires that p not be lc-bad.

Another example is $L = \mathbb{Q}(\alpha)$, $f_1, f_2 \in L[x]$ with gcd $g = x + \alpha^3$, $p = 5$, and the minimal polynomial of α is $m = z^5 + z^4 + \frac{1}{5}z^3 - \frac{1}{5}$. Because of preprocessing, in the algorithm we work with $\breve{m} = 5z^5 + 5z^4 + z^3 - 1$. Modulo $p = 5$ this becomes $z^3 + 4$. If we used the prime $p = 5$, it is easy to give an example f_1, f_2 where the Euclidean algorithm mod p returns g mod $(5, \alpha^3 + 4)$ which is $x + 1$. But, viewing α as a variable, $g \not\equiv x + 1$ mod 5.

For our algorithm, the best solution to the above problems is: *never use an lc-bad prime.*

1.2 Fail primes

Fail primes are primes for which the Euclidean algorithm mod p tries to divide by a zero divisor, in which case it returns "failed". Take for example $f_1 = x^2 - 1$, $f_2 = ax - a$ where $a = 2^{1/5} + 5$. Denote a mod p as $\overline{a}$. The Euclidean algorithm mod p will first try to make f_2 mod p monic by multiplying it with $1/\overline{a}$. But if $N(a)$, the norm of a, vanishes mod p then $\overline{a}$ is zero or a zero-divisor, and the computation of $1/\overline{a}$ fails. In this example $N(a) = 53 \cdot 59$ so the fail primes are 53 and 59.

The reason that in our terminology 53 and 59 are called fail primes and not lc-bad primes in the example (after all, the problem was caused by $\text{lc}(f_2)$ mod p) is to indicate how these primes are discarded: We do not actively avoid these primes, instead, they "discard themselves" when the Euclidean algorithm mod p is called.

One can also construct examples where p is not lc-bad, $\text{lc}(f_2)$ is a unit mod p, but p still divides $\text{den}(g)$ (occasionally such p can be unlucky instead of fail). Take for example α with minimal polynomial $m = z^3 + 3z^2 - 46z + 1$, $f_1 = x^3 - 2x^2 + (-2\alpha^2 + 8\alpha + 2)x - \alpha^2 + 11\alpha - 1$, $f_2 = x^3 - 2x^2 - x + 1$. The monic gcd is $g = x - \frac{1}{91}\alpha^2 - \frac{23}{91}\alpha - \frac{50}{91}$. The denominator is $\text{den}(g) = 91 = 7 \cdot 13$. In this example, if $p \in \{7, 13\}$ then p is not lc-bad and the leading coefficient of f_2 (as well as of f_1) is a unit mod p. Nevertheless, p can not be a good prime because $p \mid \text{den}(g)$. In this type of example p must divide the discriminant. For this reason, Encarnacion [4] tests if the discriminant is 0 mod p and avoids such primes. However, even without the discriminant-test, the primes $p \in \{7, 13\}$ would still have been discarded at some point: The Euclidean algorithm mod p will calculate $r_3 = f_1 \bmod (p, f_2)$, try to make r_3 monic and fail because the leading coefficient of r_3, namely, $-2\alpha^2 + 8\alpha + 3$, is a zero divisor mod p.

Although one can generalize the discriminant-test to L, see section 3 in [6], our algorithm does not use it because it makes no difference for the correctness of the algorithm. For an intuitive explanation see lemma 4 and for a proof see theorem 1.

1.3 Unlucky primes

Unlucky primes are not trivially detectable like lc-bad primes and do not "discard themselves" like fail primes do, but need to be detected and discarded nevertheless. Fortunately, Brown [3] showed how to do this in a way that is efficient and easy to implement: Whenever modular gcd's do not have the same degree, keep only those of smallest degree and discard the others.

As an example, take $f_1 = x^2 + (2\sqrt{5} + 1)x + 3$, $f_2 = x^2 - x - 1$, $g = x + (\sqrt{5} - 1)/2$. Then the Euclidean algorithm mod 2 will return $x^2 + x + 1$, so $p = 2$ is an unlucky prime. But if $f_1 = x^2 + \sqrt{5}\, x + 1$, f_2 and g the same as before, then $p = 2$ is a fail prime.

1.4 Good primes

All but finitely many primes must be good. This is because if one would run the Euclidean algorithm in characteristic 0, it would be a finite computation, and so there can only be finitely many conditions on the primes and each condition only excludes finitely many primes (see lemma 5).

Of course we will not run the Euclidean algorithm in characteristic 0, so this does not tell us which primes to use. But this is not a problem because to guarantee correctness of the algorithm, just as in Brown's algorithm, all we need to do is to avoid the lc-bad primes. Experiments show that random primes are good with high probability. Hence, even if there was an oracle that quickly provided good primes, it would not noticeably improve the running time.

1.5 Motivation for the algorithm

The goal of this paper is to present an efficient modular GCD algorithm over a field L that consists of multiple extensions over $\mathbb{Q}$. Suppose the largest numerator or denominator in g is c. To reconstruct g by computing $g \bmod P = p_1 \cdots p_m$ using primes $p_1, \ldots, p_m$, if we want $\log(P) = O(\log(c))$, that is, if we want the number of primes used to be proportional to the size of the coefficients in g, then we are forced to

1. Not use a primitive element to convert to a single extension, which is expensive and can cause a blowup in the size of the coefficients. This problem is well known, e.g. see [1].

2. Not invert $\mathrm{lc}(f_2)$, which can also cause a blowup, and can also be more expensive than computing g.

3. Use rational reconstruction. Otherwise a denominator bound would be necessary, but such bounds are generally too large. The defect bound (usually the (reduced [6]) discriminant), which is part of the denominator bound, is usually also too large.

Encarnacion's paper confirms and deals with these items. As a result, Encarnacion's algorithm is the fastest algorithm for a single extension. As for item 1, his paper deals only with a single extension, but he does illustrate that modifying that extension (making α_1 an algebraic integer) is not efficient. But if modifying one extension α_1 is not efficient, then modifying n extensions (replacing it by a primitive element) is certainly not efficient.

Our goal is to generalize Encarnacion's algorithm to multiple extensions, without using a primitive element. A technical difficulty is how to generalize the discriminant test, which we did a preprint [6], where we also gave a formula for the reduced discriminant. It turned out, however, that a discriminant test is not necessary, it can be omitted.

Omitting the discriminant test simplifies the algorithm but has no noticeable impact on the running time. So for the single extension case, our algorithm is essentially the same as Encarnacion's algorithm. For the multiple extension case our algorithm provides an obvious efficiency improvement because the primitive element conversion alone can easily cost more time than our entire algorithm, especially if the gcd is small.

In this paper we only treat univariate polynomials $f_1, f_2 \in L[x]$, but our implementation handles the multivariate case as well.

2. THE EUCLIDEAN ALGORITHM OVER A RING

Let $\alpha_1, \ldots, \alpha_n$ be algebraic numbers. Let $L_i = \mathbb{Q}(\alpha_1, \ldots, \alpha_i)$ and $L = L_n$. Let d_i be the degree of α_i over L_{i-1}. The dimension of L as a $\mathbb{Q}$-vector space is $d_* := d_1 \cdots d_n$. A basis of L is:

$$M := \{\prod_{i=1}^{n} \alpha_i^{e_i} \mid 0 \le e_i < d_i\}.$$

Let $\tilde{R}$ be the set of all $\mathbb{Z}$-linear combinations of M and let $\tilde{R}_i = \tilde{R} \bigcap L_i$. Let m_i be the minimal polynomial of α_i over L_{i-1}. The degree of m_i is d_i, m_i is *monic* (the leading coefficient is $\mathrm{lc}(m_i) = 1$) and $m_i(\alpha_i) = 0$. The coefficients of m_i are in L_{i-1}. Let l_i be the smallest positive integer such that the coefficients of $l_i m_i$ are in $\tilde{R}_{i-1}$. Denote $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ and $l_* = l_1 \cdots l_n$.

In general $\tilde{R}$ is not a ring. For example, $\alpha_1 \in \tilde{R}$, but $\alpha_1^{d_1}$ is not in $\tilde{R}$ unless $l_1 = 1$. When $a, b \in \tilde{R}$, to compute the product $ab \in L$ we replace $\alpha_1, \ldots, \alpha_n$ by variables $z_1, \ldots, z_n$, then multiply a,b as polynomials, and after that take the remainder modulo the polynomials $m_1(z_1), \ldots, m_n(z_n)$. During this computation we only divide a bounded number of times by $l_1, \ldots, l_n$. Hence, if k is a sufficiently large integer, then $l_*^k ab \in \tilde{R}$ for all $a, b \in \tilde{R}$.

If $a \in L$ then define the *denominator* of a as the smallest positive integer $\mathrm{den}(a)$ such that $\mathrm{den}(a)a \in \tilde{R}$. Note that $\tilde{R}$, and hence $\mathrm{den}(a)$, depends on the choice of $\alpha_1, \ldots, \alpha_n$. For example, if $\alpha_1 = \sqrt{8}$ and $a = \frac{1}{2}\alpha_1$ then $\mathrm{den}(a) = 2$. For $a \in L$ one has $a \in \tilde{R} \iff \mathrm{den}(a) = 1$, in particular $\mathrm{den}(0) = 1$. Define

$$\begin{aligned} R_p &= \{a \in L \mid \mathrm{den}(a) \not\equiv 0 \bmod p\} && (1)\\ &= \{\frac{a}{m} \mid a \in \tilde{R},\ m \in \mathbb{Z},\ m \not\equiv 0 \bmod p\}. && (2) \end{aligned}$$

If $a, b \in L$ then $\mathrm{den}(ab)$ divides $\mathrm{den}(a)\mathrm{den}(b)l_*^k$ for some k. Hence, if $p \nmid l_*$ then R_p is a ring. We will *always assume that p does not divide l_** so that R_p is a ring (if $p \mid l_*$ then p is an lc-bad prime). Denote

$$\mathbb{Z}_{(p)} = R_p \bigcap \mathbb{Q} = \{\frac{a}{m} \mid a, m \in \mathbb{Z},\ m \not\equiv 0 \bmod p\}.$$

Then R_p is a $\mathbb{Z}_{(p)}$-module with basis M. Define

$$\overline{R} = R_p / pR_p.$$

If $a \in R_p$ then we use the notation $\overline{a}$, or also $a \bmod p$, for the image of a in $\overline{R}$. If $a \in L$, then (primes that divide l_* are always excluded)

$$\overline{a} \text{ is defined} \iff a \in R_p \iff p \nmid\ l_* \mathrm{den}(a).$$

If $\overline{a}$ is defined we will say that *a can be reduced mod p.*

Now $\overline{R}$ is a ring and also an $\mathbb{F}_p$-vector space with basis M mod p. We can do the following identifications:

$$R_p = \tilde{R} \otimes_{\mathbb{Z}} \mathbb{Z}_{(p)}, \quad L = \tilde{R} \otimes_{\mathbb{Z}} \mathbb{Q}, \quad \text{and} \quad \overline{R} = \tilde{R} \otimes_{\mathbb{Z}} \mathbb{F}_p \qquad (3)$$

If $a \in L$ then a is a *unit* in R_p if and only if both a and $1/a$ are in R_p (whenever we write $1/a$ it is implicitly assumed that $a \neq 0$). This is equivalent to $p \nmid l_* \mathrm{den}(a)\mathrm{den}(1/a)$. If $a \in L$ we will call a a *unit* mod p if $a \in R_p$ and $\overline{a}$ is a unit in $\overline{R}$. The following lemma shows that these two notions are equivalent.

LEMMA 1. *Let $a \in R_p$. Then a is a unit in R_p if and only if $\overline{a}$ is a unit in $\overline{R}$.*

Proof: If a is a unit in R_p then a and $1/a$ are in R_p, hence $\overline{a}$ and $\overline{1/a}$ are defined, and since $a \mapsto \overline{a}$ is a ring homomorphism $R_p \to \overline{R}$ one sees that $\overline{1/a}$ is the inverse of $\overline{a}$. Hence

$\overline{a}$ is a unit in $\overline{R}$.
Conversely, assume $\overline{a}$ is a unit. Then $a \neq 0$ so we can take $b := 1/a \in L$. To finish the proof we need to show that $b \in R_p$. Take the smallest integer k for which $c := bp^k \in R_p$. Since k is minimal, we have $\overline{c} \neq 0$ but then $\overline{ac}$ is the product of a unit and a nonzero element in $\overline{R}$ and hence nonzero. But $\overline{ac}$ equals $\overline{abp^k} = \overline{p^k}$ so $\overline{p^k} \neq 0$, hence $k = 0$, so $b \in R_p$ and a is invertible in R_p.

If $f \in L[x]$ then *the denominator* $\mathrm{den}(f)$ is defined as the smallest positive integer such that $\mathrm{den}(f)f \in \tilde{R}[x]$. Now $f \in R_p[x]$ if and only if $p \nmid \mathrm{den}(f)l_*$. The polynomial $\overline{f}$ is the image of f in $\overline{R}[x]$, and is defined if and only if $f \in R_p[x]$, in which case we will say that f *can be reduced* mod p. Furthermore, if f and $\overline{f}$ have the same degree (when $\mathrm{lc}(f)$ is nonzero mod p) then we will say that f *reduces properly* mod p. If p is not an lc-bad prime it means that f_1, f_2 can be reduced mod p, and that f_2 reduces properly mod p.

Let $0 \leq i \leq j \leq n$ and $a \in L_j$. Multiplication by a is an L_i-linear map $\psi : L_j \to L_j$. The *characteristic polynomial* $\mathrm{cp}_i^j(a) \in L_i[x]$ of a over the extension $L_j : L_i$ is defined as the characteristic polynomial of this linear map. The *trace* $\mathrm{Tr}_i^j(a)$ of a over $L_j : L_i$ is the trace of ψ and the *norm* $N_i^j(a)$ of a over $L_j : L_i$ is the determinant of ψ. Whenever we do not mention the extension $L_j : L_i$ it is assumed to be $L : \mathbb{Q}$ (so $i = 0$ and $j = n$) in which case we write $\mathrm{Tr}(a)$, $N(a)$, $\mathrm{cp}(a)$. Now the *integral closure of* $\mathbb{Z}$ *in* L is

$$\mathcal{O} = \{a \in L \mid \mathrm{cp}(a) \in \mathbb{Z}[x]\}.$$

This is a ring (see [5]), and the elements of $\mathcal{O}$ are called the *algebraic integers* in L. We will use the following notation for the integral closure of $\mathbb{Z}_{(p)}$ in L

$$\mathcal{O}_p = \{a \in L \mid \mathrm{cp}(a) \in \mathbb{Z}_{(p)}[x]\}.$$

Suppose $a \in L$ and $m = \mathrm{den}(\mathrm{cp}(a))$. Then by definition $a \in \mathcal{O}_p$ if and only if $m \not\equiv 0 \bmod p$. The characteristic polynomial of ma is in $\mathbb{Z}[x]$, hence $ma \in \mathcal{O}$ and hence

$$\mathcal{O}_p = \{\frac{a}{m} \mid a \in \mathcal{O},\ m \in \mathbb{Z},\ m \not\equiv 0 \bmod p\}. \qquad (4)$$

Lemma 2. *If $0 \leq i \leq j \leq n$ and $a \in \mathcal{O}_p \bigcap L_j$ then a is a unit in $\mathcal{O}_p$ if and only if $N_i^j(a)$ is a unit in $\mathcal{O}_p$. In particular, $a \in \mathcal{O}_p$ is a unit if and only if $N(a) \in \mathbb{Q}$ is a unit in $\mathbb{Z}_{(p)}$, in other words, both numerator and denominator of $N(a)$ are not divisible by p. The same is also true for R_p.*

Remark: If $p \nmid l_*$ then $R_p \subseteq \mathcal{O}_p$ and the lemma implies that if $a \in R_p$ and $1/a \in \mathcal{O}_p$ then $1/a \in R_p$.

Proof: The L_i-linear map $\psi : L_j \to L_j$ that corresponds to multiplication by a is defined over $\mathcal{O}_p$, i.e. the entries of the matrix of ψ are in $\mathcal{O}_p$. If $N_i^j(a)$, the determinant of ψ, is a unit in $\mathcal{O}_p$ then the matrix is invertible over $\mathcal{O}_p$. So then $\psi^{-1}(1) \in \mathcal{O}_p$, so $1/a \in \mathcal{O}_p$. Conversely, if a is invertible in $\mathcal{O}_p$ then ψ is an invertible linear map, so its determinant must be a unit.
Now $N(a) = N_0^n(a) \in L_0 = \mathbb{Q}$ and $\mathbb{Q} \bigcap \mathcal{O}_p = \mathbb{Z}_{(p)}$ so the second statement follows. The proof for R_p is the same, although as always p must not divide l_* so R_p is a ring.

Note that one can check if $a \in R_p$ is invertible, and if so, compute its inverse, with linear algebra over $\mathbb{Z}_{(p)}$ or over its field of fractions $\mathbb{Q}$. The matrix of the system to be solved is the matrix of ψ. The same also holds for $\overline{a} \in \overline{R}$, whenever it is invertible, its inverse can be computed with linear algebra over $\mathbb{F}_p$. But instead of solving linear equations, we will use the extended Euclidean algorithm to calculate inverses in $\overline{R}$. However, this can increase the number of fail primes because the calculation can fail even if $\overline{a}$ is invertible. This is not a serious problem because the number of fail primes will still be finite (see section 1.4).

In the following, let $\mathcal{R}$ be a commutative ring with identity $1 \neq 0$. For a univariate polynomial $f \in \mathcal{R}[x]$ define $\mathrm{monic}(f)$ as follows: If $f = 0$ then $\mathrm{monic}(f) = 0$. If $f \neq 0$ and if the leading coefficient $\mathrm{lc}(f) \in \mathcal{R}$ of f is a unit, then define $\mathrm{monic}(f) = \mathrm{lc}(f)^{-1}f$. If $f \neq 0$ and $\mathrm{lc}(f)$ is not a unit then define $\mathrm{monic}(f)$="failed".

If $f_1, f_2 \in \mathcal{R}[x]$ then the *monic gcd* is defined as a polynomial $g \in \mathcal{R}[x]$ such that $g = \mathrm{monic}(g)$ and for every polynomial h one has: $h \mid f_1$ and $h \mid f_2$ if and only if $h \mid g$. It is easy to show that if a monic gcd of f_1, f_2 exists, then it is unique. The well-known *Euclidean algorithm* over $\mathcal{R}$ works as follows.

Euclidean algorithm.
Input: a list (f_1, f_2) of two univariate polynomials with coefficients in $\mathcal{R}$.
Output: Either a message "failed" or the monic gcd.

1. Set $r_1 = f_1$, $r_2 = f_2$, $i = 2$.
2. If $r_2 = 0$ then set $r_1 = \mathrm{monic}(r_1)$. If r_1 = "failed" then return "failed".
3. If $r_i = 0$ then return r_{i-1}.
4. Set $r_i = \mathrm{monic}(r_i)$. If r_i = "failed" then return "failed".
5. Set r_{i+1} to be the remainder of r_{i-1} divided by r_i.
6. Set $i = i + 1$ and go back to Step 3.

Remark on a shortcut: Suppose that r_i in step 3 is a nonzero constant. Some implementations of the Euclidean algorithm over a field will then take a *shortcut*: stop the computation, the output is 1. Over a ring *we should not use this shortcut* because that would invalidate lemma 3 below. This plays a role because our algorithm will not test if p divides the discriminant. We may only use the shortcut if r_i is a unit. For $r_i \in \overline{R}$ we can test that efficiently by computing $N(r_i)$ mod p (see lemmas 1,2).

Denote $\mathrm{GCD}_{\mathcal{R}}(f_1, f_2)$ as the output of this algorithm. If $\mathrm{GCD}_{\mathcal{R}}(f_1, f_2) \neq$ "failed" then the sequence of polynomials $r_1, \ldots, r_m$ with $r_{m-1} \neq 0$, $r_m = 0$, is called the *monic polynomial remainder sequence* of f_1, f_2.

Lemma 3. *If $g = \mathrm{GCD}_{\mathcal{R}}(f_1, f_2)$ and $g \neq$ "failed" then the ideal $(r_{i-1}, r_i) = \mathcal{R}[x]r_{i-1} + \mathcal{R}[x]r_i$ remains the same during each step. In particular $(f_1, f_2) = (g)$ which implies:*

1. *There exist $s, t \in \mathcal{R}[x]$ such that $g = sf_1 + tf_2$.*
2. *f_1 and f_2 are divisible by g.*
3. *g is the monic gcd of f_1 and f_2.*

Proof: When we make r_i monic, we divide by a unit, which does not change the ideal. In step 6 we increase i so we must show that $(r_{i-1}, r_i) = (r_i, r_{i+1})$ which is clear because r_{i+1} is the remainder of r_{i-1} modulo r_i. Hence $(f_1, f_2) = (r_1, r_2) = (r_{m-1}, r_m) = (g, 0) = (g)$. So $g \in (f_1, f_2)$ which is part 1, $f_1, f_2 \in (g)$ which is part 2. Finally, every h that divides both f_1 and f_2 divides any element of (f_1, f_2) in particular it divides g. Since g is monic it satisfies precisely the definition of the monic gcd.

Remark: If $\mathrm{GCD}_{\mathcal{R}}(f_1, f_2) \neq$ "failed" then the *extended Euclidean algorithm*, which calculates s and t as well as g will not fail either.

Let $\mathbf{d} = \mathrm{GCD}_{\mathcal{R}}(f_1, f_2)$ be the output of the Euclidean algorithm. If all leading coefficients during the computation are units then the algorithm succeeds, the monic gcd exists and equals $\mathbf{d} = r_{m-1}$. If there is no monic gcd in $\mathcal{R}[x]$ then $\mathbf{d} =$ "failed". If a monic gcd g does exist then it is not necessarily true that the algorithm will find it; the output $\mathbf{d}$ is then either g or "failed". A situation where the output is "failed" even when a monic gcd exists is given in the following lemma.

LEMMA 4. *Suppose $p \nmid l_*$ and $f_1, f_2 \in R_p[x]$. Then $f_1, f_2 \in \mathcal{O}_p[x]$. Suppose a monic gcd $g \in \mathcal{O}_p[x]$ exists and that $g \notin R_p[x]$. Then $\mathrm{GCD}_{\mathcal{O}_p}(f_1, f_2) =$ "failed".*

Proof: If $p \nmid l_*$ then $\alpha_1, \ldots, \alpha_n \in \mathcal{O}_p$, hence $R_p \subseteq \mathcal{O}_p$ so $f_1, f_2 \in \mathcal{O}_p[x]$. Since $\mathrm{GCD}_{R_p}(f_1, f_2) =$ "failed", when we run the Euclidean algorithm over R_p we will encounter a leading coefficient in R_p that is not a unit in R_p. But according to the remark after lemma 2, if $a \in R_p$ is not a unit in R_p then it is also not a unit in $\mathcal{O}_p$ and hence the algorithm fails over $\mathcal{O}_p$ as well.

If the ring $\mathcal{R}$ in the Euclidean algorithm is a field L, then the output is never "failed", so $\mathrm{GCD}_L(f_1, f_2)$ is always the monic gcd of $f_1, f_2 \in L[x]$.

LEMMA 5. *Suppose $f_1, f_2 \in L[x]$ and $r_1, \ldots, r_m \in L[x]$ is the monic polynomial remainder sequence. Let $\mathrm{lc}_1, \ldots, \mathrm{lc}_{m-1}$ in L be the leading coefficients that we divided by in steps 2 and 4. For all but finitely many primes the following holds:*

1. *$f_1, f_2 \in R_p[x]$, and $\mathrm{lc}_1, \ldots, \mathrm{lc}_{m-1}$ are units in R_p.*
2. *$r_1, \ldots, r_m \in R_p[x]$ and $\overline{r_1}, \ldots, \overline{r_m}$ is the monic polynomial remainder sequence of $\overline{f_1}, \overline{f_2}$.*
3. *p is a* good prime *which means: The monic gcd of $\overline{f_1}, \overline{f_2}$ exists, will be found by the Euclidean algorithm, and equals $\overline{g}$ where $g \in L[x]$ is the monic gcd of f_1, f_2.*

Proof: Part 1 holds for all primes that do not divide any of the following: l_*, $\mathrm{den}(f_1)$, $\mathrm{den}(f_2)$, $\mathrm{den}(\mathrm{lc}_i)$, $\mathrm{den}(1/\mathrm{lc}_i)$ for $i < m$. Since these are finitely many integers, all nonzero, we see that part 1 holds for all but finitely many primes. The only divisions in the Euclidean algorithm are divisions by lc_i, so if the input is in $R_p[x]$ and all lc_i are units in R_p, then all polynomials in the $\mathrm{GCD}_L(f_1, f_2)$ computation are in $R_p[x]$. Induction shows that $\overline{r_1}, \ldots, \overline{r_m}$ is precisely the monic polynomial remainder sequence of $\overline{f_1}, \overline{f_2}$, so part 2 follows from part 1. Part 3 follows from part 2.

Since we will only run the Euclidean algorithm in $\overline{R}[x]$ for various primes p, and not in $L[x]$, we do not know the values of lc_i. So the lemma does not tell us which primes are good, it only says that all but finitely many primes are good. We now investigate the relation between $\mathrm{GCD}_{\overline{R}}(\overline{f_1}, \overline{f_2})$ and $\mathrm{GCD}_L(f_1, f_2)$ when p is not an lc-bad prime.

THEOREM 1. *Let $f_1, f_2 \in L[x]$ and let $g \in L[x]$ be the monic gcd. Assume $p \nmid l_* \mathrm{den}(f_1)\mathrm{den}(f_2)$, $f_2 \neq 0$ and $\mathrm{lc}(f_2) \not\equiv 0$ mod p, so p is not an lc-bad prime. Let $\mathbf{d} = \mathrm{GCD}_{\overline{R}}(\overline{f_1}, \overline{f_2})$. If $\mathbf{d} \neq$ "failed" then*

$$\deg(\mathbf{d}) \geq \deg(g).$$

Furthermore, if $\deg(\mathbf{d}) = \deg(g)$ then g reduces properly mod p and $\mathbf{d} = \overline{g}$.

Remark: The theorem says that if p is not lc-bad then p is either fail, unlucky, or good. This implies that if lc-bad primes are avoided then the modular GCD algorithm is correct.

Proof: $\mathrm{lc}(f_2) \not\equiv 0 \bmod p$, so if we assume $\mathbf{d} \neq$ "failed" then $\mathrm{lc}(f_2)$ must be a unit mod p, see step 4 in the Euclidean algorithm. There exist (see lemma 3) $s_0, t_0 \in R_p[x]$ such that

$$\overline{s_0}\,\overline{f_1} + \overline{t_0}\,\overline{f_2} = \mathbf{d}.$$

Now take a monic polynomial $\mathbf{d}_0 \in R_p[x]$ such that $\mathbf{d} = \overline{\mathbf{d}_0}$. Then we have

$$s_0 f_1 + t_0 f_2 \equiv \mathbf{d}_0 \bmod p.$$

We will apply *Hensel lifting* to increase the modulus p to a higher power of p. Define (starting with $i = 1$)

$$h_i = (s_{i-1} f_1 + t_{i-1} f_2 - \mathbf{d}_{i-1})/p^i \in R_p[x]$$

and let $\mathrm{q}_i, \mathrm{r}_i \in R_p[x]$ be the quotient and remainder of h_i divided by $\mathbf{d}_0$ (this division works because $\mathbf{d}_0$ is monic). Then define

$$\tilde{s}_i = s_{i-1} - p^i \mathrm{q}_i s_0, \quad \tilde{t}_i = t_{i-1} - p^i \mathrm{q}_i t_0, \quad \mathbf{d}_i = \mathbf{d}_{i-1} + p^i \mathrm{r}_i.$$

Then

$$\tilde{s}_i f_1 + \tilde{t}_i f_2 \equiv \mathbf{d}_i \bmod p^{i+1}.$$

Now $\tilde{s}_i, \tilde{t}_i$ can have higher degrees than s_{i-1}, t_{i-1}. To remedy this, do the following. For $j \in \{1, 2\}$ denote $f_{j,d} \in R_p[x]$ as a polynomial whose modular image equals $\overline{f_j}/\mathbf{d}$. Take $\mathrm{q}_i s_0$ mod p, and divide it by $\overline{f_{2,d}} \in \overline{R}[x]$. This division works because the leading coefficient of $\overline{f_{2,d}}$ is $\mathrm{lc}(f_2)$ mod p, which is invertible. Take $q, r \in R_p[x]$ such that $\overline{q}, \overline{r}$ are the quotient and remainder of this division. Take q, r in such a way that they have the same degree as $\overline{q}, \overline{r}$. Then define

$$s_i = s_{i-1} - p^i r, \quad \text{and} \quad t_i = t_{i-1} - p^i(\mathrm{q}_i t_0 + q f_{1,d}),$$

and we still have

$$s_i f_1 + t_i f_2 \equiv \mathbf{d}_i \bmod p^{i+1}.$$

We can now increase i and do the next Hensel step, and continue in this way. Because $\deg(r) < \deg(\overline{f_{2,d}})$ and $\deg(\mathrm{r}_i) <$

$\deg(\mathbf{d}_0)$, the degrees of s_i and $\mathbf{d}_i$ will be bounded as i increases, and hence the degree of $t_i \bmod p^{i+1}$ is bounded as well. So when $i \to \infty$, the limit $\hat{s}, \hat{t}, \hat{\mathbf{d}}$ of $s_i, t_i, \mathbf{d}_i$ exists in the ring $\hat{R}_p[x]$ defined below.

Denote $\hat{\mathbb{Z}}_p$ as the ring of p-adic integers. $\hat{\mathbb{Z}}_p$ is the completion of $\mathbb{Z}_{(p)}$ with respect to the p-adic valuation norm. Let $\hat{\mathbb{Q}}_p$ be the field of p-adic numbers, the field of fractions of $\hat{\mathbb{Z}}_p$. Denote $\hat{L}_p = R_p \otimes_{\mathbb{Z}_{(p)}} \hat{\mathbb{Q}}_p = L \otimes_{\mathbb{Q}} \hat{\mathbb{Q}}_p$. This is in general not an integral domain because minimal polynomials can become reducible when one replaces $\mathbb{Q}$ by a larger field $\hat{\mathbb{Q}}_p$. Denote $\hat{R}_p = R_p \otimes_{\mathbb{Z}_{(p)}} \hat{\mathbb{Z}}_p$. Now $\hat{R}_p$ and L can be viewed as subrings of $\hat{L}_p$ and

$$R_p = \hat{R}_p \bigcap L \qquad (5)$$

After doing infinitely many Hensel steps we find $\hat{s}, \hat{t}, \hat{\mathbf{d}} \in \hat{R}_p[x]$ such that

$$\hat{s} f_1 + \hat{t} f_2 = \hat{\mathbf{d}}.$$

Now $\hat{\mathbf{d}}$ is monic and $\deg(\hat{\mathbf{d}}) = \deg(\mathbf{d}_0) = \deg(\mathbf{d})$ because the $p^i \mathrm{r}_i$, $i = 1, 2, \ldots$, that we added to $\mathbf{d}_0$ have smaller degree than $\mathbf{d}_0$. The polynomials f_1, f_2 are elements of $L[x]g \subseteq \hat{L}_p[x]g$. Hence $\hat{s}f_1 + \hat{t}f_2$, which equals $\hat{\mathbf{d}}$, is a also an element of $\hat{L}_p[x]g$. But $\hat{\mathbf{d}} \neq 0$ so

$$\deg(\mathbf{d}) = \deg(\hat{\mathbf{d}}) \geq \deg(g).$$

If the degrees are the same then $\hat{\mathbf{d}} = g$ because g is the only monic element of $\hat{L}_p[x]g$ of that degree. Equation (5) then implies $g \in R_p[x]$ (recall that $\hat{\mathbf{d}} \in \hat{R}_p[x]$ and $g \in L[x]$). So g can be reduced mod p. Hence g reduces properly mod p because it is monic. The theorem now follows because $\mathbf{d}$ equals $\hat{\mathbf{d}}$ mod p, which equals g mod p.

3. IMPLEMENTATION

In this section we describe our implementation of the modular GCD algorithm for multivariate polynomials over L. There are several multivariate "modular" GCD algorithms over $\mathbb{Q}$ that one may consider extending to work over L. We have completed an implementation of Brown's algorithm (see [3]) which uses rational reconstruction and trial division (see [8]), and have begun work on an implementation of Zippel's algorithm (see [11]). We have encountered three bottlenecks on real problems, namely, (i) rational reconstruction, (ii) the trial divisions, and (iii) extensions of low degree. We will address (i) and (ii) in this paper. Problem (iii) is addressed in [8].

We first give details of the data structure that we use for multivariate polynomials over L. The data structure is designed to make the modular GCD algorithm fast. It supports $n \geq 0$ extensions over $\mathbb{Q}$ and $\mathbb{F}_p$. To fix notation, recall that $L = \mathbb{Q}(\alpha_1, \ldots, \alpha_n)$ where α_i is algebraic over $L_{i-1} = \mathbb{Q}(\alpha_1, \ldots, \alpha_{i-1})$, and $m_i(z_i) \in L_{i-1}[z_i]$ is the minimal polynomial for α_i over L_{i-1}. Let $R = L[x_1, \ldots, x_k]$. Let f_1, f_2 be non-zero polynomials in R and let g be their monic GCD.

3.1 A Data Structure for R

In [9], Stoutemyer asked the question "Which polynomial representation is best?" (for a general purpose computer algebra system). Based on his data, he concluded that the *recursive dense* representation was best overall, a conclusion that ran contrary to the general belief that one must use a sparse representation. In our context, we have additional reasons to choose this representation. Our input polynomials to the modular GCD algorithm are multivariate polynomials in $x_1, \ldots, x_k$ and $z_1, \ldots, z_n$. Because the modular GCD algorithm is recursive and because the extensions must also be defined recursively if they are dependent, a recursive data structure will minimize data structure overhead. Because we compute the GCD modulo machine primes $p_1, p_2, \ldots$, most of the work takes place in the last variable, i.e. in the ring $\mathbb{F}_p[z_1]$. Since the bottom level of the recursive dense representation is a dense vector of machine integers, this yields the best representation for arithmetic in $\mathbb{F}_p[z_1]$. We now describe the data structure `<poly>` using a BNF notation with some examples.

```
<poly> ::= POLYNOMIAL( <ring>, <data> )
<ring> ::= [ <char>, <vars>, <exts> ]
<char> ::= <nonnegative integer>
<data> ::= <rational number> | <immediate integer>
                             | vector(<data>)
<vars> ::= vector(<variables>)
<exts> ::= vector(<data>)
```

The characteristic of the ring is encoded by `<char>` and `<exts>` is a vector of the minimal polynomials. Thus the ring for the polynomial is encoded in the data structure. Since this information is identical for polynomials in the same ring it should be stored once so that the cost of storing the ring information is one word.

We impose the following restriction which is a key property of the recursive dense representation; a zero coefficient at any level in the data structure is represented by the immediate integer 0 (or nil pointer). This means that every algorithm must treat 0 as a special case. This exceptional case does not bother us greatly because in the implementation of most operations 0 is a special case anyway.

The bottom of the data structure is a word of storage which is either a pointer to a rational number or an immediate integer. In our Maple implementation, immediate integers are signed integers of 30 bits in length, hence, one bit is used to distinguish them from pointers. In the examples below, vectors are indicated by square brackets.

Example 1: The representation of the polynomial $z^4 - 10z^2 + 1$ in characteristic 0 and characteristic 3 is

```
POLYNOMIAL( [0,[z],[]], [1,0,-10,0,1] )
POLYNOMIAL( [3,[z],[]], [1,0,2,0,1] )
```

The empty vector `[]` indicates that there are no extensions and the data in both these examples is a vector of machine integers. Allowing one word as a header word for the POLYNOMIAL structure and for each vector, the storage requirement for both polynomials is 16 words (count one word for POLYNOMIAL and each [in the above). Since the ring information can be shared between polynomials over the same ring, a more accurate count is that 9 words are required. From now on we will not count the storage for the ring.

Example 2: The representation of the polynomial $x^3 - zx + z^2$ in $\mathbb{Q}[z][x]$ and $\mathbb{Q}[z]/(z^2 - 2)$ is

```
POLYNOMIAL([0,[x,z],[]], [[0,0,1],[0,-1],0,[1]])
POLYNOMIAL([0,[x,z],[[-2,0,1]]],[[2],[0,-1],0,[1]])
```

In the data structure, polynomials are reduced modulo the m_i on input. The storage requirement is 17 and 15 words respectively.

Example 3: The recursive dense data structure is not sparse, but neither is it truly dense. On sparse polynomials, the storage requirement is still very good. Consider the sparse polynomial $1 + 2x^n + 3y^n + 4z^n$ where $n = 3$. Our data structure for this polynomial is

```
POLYNOMIAL(R,[[[1,0,0,4],0,0,[3]],0,0,[[2]]]);
```

This is 24 words (not counting the storage for the ring). In general it is $15 + 3n$ words. One of the main sparse representations for polynomials that is used in AXIOM is a linked list of pairs where each pair is a pointer to a coefficient and a pointer to a monomial where the monomial $x^i y^j z^k$ would be stored as an exponent vector $[i, j, k]$. Thus each non-zero term of the polynomial requires $2 + 2 + 4 = 8$ words of storage. On our example this would be 35 words, allowing 3 words for the top level of the data structure. Of course, this is not truly a sparse data structure because the *monomial* representation is not sparse. Nevertheless, on this example, the recursive dense representation uses less storage for $n \leq 6$.

Example 4: Multiple extensions are handled in the obvious way. The polynomial $x + \sqrt{1/5}\ y + \sqrt{1 + \sqrt{1/5}}$ is represented by

```
POLYNOMIAL([0, [x,y,z2,z1], [[[-1,-1],0,[1]],
  [-1, 0, 5]]], [[[0,[1]], [[0,1]]], [[[1]]]]])
```

where the two minimal polynomials $m_1(z_1)$ and $m_2(z_2)$ have been replaced by $\check{m}_1$ and $\check{m}_2$. We remark that although this makes the test for whether a prime p in the modular GCD algorithm divides $\mathrm{den}(m_i)$ easy, and makes reduction of the minimal polynomials modulo p easy, after having reduced the minimal polynomials modulo p, one should make them monic over $\mathbb{F}_p$ so that we do not repeatedly invert their leading coefficients.

3.2 Rational Reconstruction

If one naively applies rational reconstruction after computing the GCD modulo each prime p, the cost of the rational reconstruction may become the bottleneck asymptotically as well as in practice. Suppose $g = x + a/b$ where a and b are integers of m digits in length. Then we need $O(m)$ primes to reconstruct g. The total cost of Chinese remaindering will be $O(m^2)$ but the total cost of rational reconstruction will be $O(m^3)$. The reason is that Chinese remaindering can be done "incrementally" after each prime in $O(m)$ time but, as far as we know, rational reconstruction cannot. We resolve this problem by attempting rational reconstruction after $1, 2, 3, 5, 8, 13, \ldots$ primes. This ensures that the cost of rational reconstruction is also $O(m^2)$.

We now state the modular GCD algorithm. As a preprocessing step, we compute $\check{f}_1$ and $\check{f}_2$, that is, we cancel any rational scalar from the input polynomials before proceeding. We do not compute $\tilde{f}_1$ or $\tilde{f}_2$ which can cause a blowup. Let $\mathrm{ic}(f)$ denote the integer content of a polynomial. This is the rational constant c such that $f/c = \check{f}$, the semi-associate of f. For example, if $f = 10/3x^2 - 15$ then $\mathrm{ic}(f) = 5/3$ and $\check{f} = 2x^2 - 9$.

Modular GCD algorithm.
Input: Non-zero $f_1, f_2 \in L[x]$.
Output: g, the monic GCD.

```
1. Set $n = 0$, $f_1 = f_1/\mathrm{ic}(f_1)$ and $f_2 = f_2/\mathrm{ic}(f_2)$.
2. Take a new prime $p$ that is not lc-bad.
3. Let $\mathbf{d}$ be the result of the Euclidean algorithm mod $p$.
   If $\mathbf{d} =$ "failed" then go back to Step 2.
4. If $\mathbf{d} = 1$ then return 1.
5. If $n = 0$ or $\deg(\mathbf{d}) < \deg(G)$ then
   set $G = \mathbf{d}, m = p, n = 1$ and go to Step 9.
6. If $\deg(\mathbf{d}) > \deg(G)$ then go back to Step 2.
7. Let $G$ be the result of Chinese remaindering on $G$ mod
   $m$ and $\mathbf{d}$ mod $p$. Set $m = mp, n = n + 1$.
8. If $n$ is not a Fibonacci number go back to Step 2.
9. Apply rational reconstruction to obtain $h \in L[x]$ from
   $G$ mod $m$. If this fails, go back to Step 2.
10. Apply the Euclidean algorithm mod one more prime $p$
    to obtain $\mathbf{d}$ and test if $h$ mod $p = \mathbf{d}$. If not, go back
    to Step 4.
11. Trial division. If $h|f_1$ and $h|f_2$ then return $h$, otherwise
    go back to Step 4.
```

The algorithm will terminate as soon as we have enough good primes to reconstruct the coefficients of g. We will (i) state the expected running time in terms of $\check{f}_1$ and $\check{f}_2$, (ii) not include the cost of the trial divisions, and (iii) assume classical (quadratic) algorithms for integer and polynomial arithmetic.

Let m be the number of good primes needed to reconstruct g. The algorithm as stated has been designed so that m is proportional to the length of largest integer in the coefficients of g. We will assume that the probability that a prime is good is high, so that m is close to the actual number of primes that were used. This assumption is true in practice. However, for theoretical completeness of the complexity estimate, we would need to determine some $B = B(f_1, f_2)$ such that if $p > B$ then the probability that p is good is greater than some constant. We did not determine such B because this issue would not have consequences for the algorithm in practice (one hardly ever encounters primes that are not good).

The other quantities appearing in the running time are (i) D, the degree of the number field L, (ii) N, the degree of the largest input polynomial, (iii) n the degree of the GCD, and (iv) M the size of the largest integer coefficient in $\check{f}_1$ and $\check{f}_2$. The average running time of the modular GCD algorithm is $O(mMND + mN^2D^2 + m^2nD))$ where the three contributions are for reducing $\check{f}_1$ and $\check{f}_2$ modulo m primes, applying the Euclidean algorithm m times, and the reconstruction time, respectively. If we consider the case where

$M = 2m$ and $N = 2n$ corresponding to a GCD problem where the cofactors are the same size as the GCD, then this simplifies to $O(MND(M + ND))$.

3.3 Trial Division

Another bottleneck of the modular GCD algorithm is the trial divisions. If h is the result of rational reconstruction then we must check that $h|f_1$ and $h|f_2$ to show that $h = g$. Because these trial divisions can be expensive, we have considered abandoning trial divisions altogether in favor of a probabilistic result, that is, check that result of rational reconstruction agrees, say, with the GCD modulo five additional primes instead of one. However, in many applications where one computes GCDs, for example, normalizing a rational function, one wants to compute also the cofactors f_1/g and f_2/g, hence, the divisions cannot be avoided.

We can use either classical division or a modular division algorithm. If g is small in size compared with f_1 and f_2 then classical division is asymptotically faster. On the other hand, if g is of similar size to its cofactors f_1/g and f_2/g then a modular division algorithm will be asymptotically faster. When dividing f_1 and f_2 by h over L using the classical division algorithm, a significant improvement (we saw a speedup of a factor of 10 on one large example) can be obtained if one avoids fractions as much as possible. Notice that the leading coefficient of $\check{h}$ in the modular GCD algorithm is an integer. If also $l_i = \mathrm{den}(m_i) = 1$, which is often the case, then the entire division algorithm can be completed using only integer arithmetic. If $l_i \neq 1$ for some i then the division algorithm can still be modified to avoid fractions. We will describe how to do this for univariate polynomials with one field extension with minimal polynomial M.

Algorithm Fraction Free Trial Division.
Input: $A, B \in \mathbb{Q}[x, z],\ M \in \mathbb{Z}[z] : B \neq 0,\ \mathrm{lc}_x B \in \mathbb{Q}$, and $\deg M \geq 1$.
Output: $Q = A/B \bmod M$ if $B|A \bmod M$; "failed" otherwise.

```
Set $m = \deg_x A$, $n = \deg_x B$ and $d = \deg_z M$.
Set $i_a = \mathrm{ic}(A)$ and $a = A/i_a$.
Set $i_b = \mathrm{ic}(B)$ and $b = B/i_b$.
Set $l_b = \mathrm{lc}_x b$ and $l_m = \mathrm{lc}_z M$.
Set $s = 1$, $r = a$, and $q = 0$.
While $r \neq 0$ and $m \geq n$ do
    Set $l_r = \mathrm{lc}_x r$. Note that $l_r \in \mathbb{Z}[z]$.
    Set $g = \mathrm{GCD}(\mathrm{ic}(l_r), l_b)$ and $l_r = l_r/g$.
    Set $s = (l_b/g) \times s$.
    Set $t = l_r \times x^{m-n}$ and $q = q + t/s$.
    Set $r = (l_b/g) \times r - t \times b$.
    Set $p = 1$.
    While $r \neq 0$ and $\deg_z r \geq d$ do
        Set $l_r = \mathrm{lc}_z r$. Note that $l_r \in \mathbb{Z}[x]$.
        Set $g = \mathrm{GCD}(\mathrm{ic}(l_r), l_m)$ and $l_r = l_r/g$.
        Set $t = l_r z^{\deg_z r - m}$ and $p = p \times (l_m/g)$.
        Set $r = (l_m/g) \times r - t \times M$.
    Set $s = s \times p$.
    Set $m = \deg_x r$.
If $r \neq 0$ then output "failed".
Set $Q = (i_a/i_b) \times q$ and output $Q$.
```

The algorithm first makes the inputs A and B primitive over $\mathbb{Z}$. We claim that each time round the outer loop r and q satisfy $a = bq + cr$ for some scalar $c \in \mathbb{Q}$ and r has integer coefficients. The outer loop reduces the degree of the remainder r in x. In the outer loop we multiply r by the smallest possible integer so that $\mathrm{lc}_x r$, a polynomial in $\mathbb{Z}[z]$, will be divisible by $\mathrm{lc}_x b$. The inner loop then reduces the remainder r modulo M. In the inner loop we multiply r by the smallest integer so that $\mathrm{lc}_z r$, a polynomial in $\mathbb{Z}[x]$, will be divisible by $\mathrm{lc}_z M$. The integers s and p are multipliers. They keep track of the integer factors of $\mathrm{lc}_x b$ and $\mathrm{lc}_z M$, respectively, that r was multiplied by so that the quotient Q may be correctly computed from q.

Acknowledgement

We acknowledge the help of Craig Pastro with parts of the implementation, and the referees for valuable comments.

4. REFERENCES

[1] J. A. Abbott, R. J. Bradford, J. H. Davenport, The Bath Algebraic Number Package, *Proceedings of SYMSAC '86*, ACM press (1986), pp. 250–253.

[2] R. J. Bradford, Some Results on the Defect, *Proceedings of ISSAC '89*, ACM press (1989), pp. 129–135.

[3] W. S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, *J. ACM* **18** (1971), pp. 476–504.

[4] M. J. Encarnacion, Computing GCDs of Polynomials over Algebraic Number Fields, *J. Symbolic Computation* **20** (1995), pp. 299–313.

[5] E. Hecke, Lectures on the Theory of Algebraic Numbers, *Springer Graduate Texts in Mathematics* **77**, (1981).

[6] M. van Hoeij, M. Monagan, A Modular GCD algorithm over Number Fields presented with Multiple Extensions, *FSU preprint 02-03*, (2002).

[7] L. Langemyr, S. McCallum, The Computation of Polynomial GCD's over an Algebraic Number Field, *J. Symbolic Computation* **8** (1989), pp. 429–448.

[8] M. B. Monagan, A. D. Wittkopf, On the Design and Implementation of Brown's Algorithm over the Integers and Number Fields, *Proceedings of ISSAC '2000* (2000), ACM Press, pp. 225–233.

[9] D. Stoutemyer, Which Polynomial Representation is Best?, *Proceedings of the 1984 Macsyma User's Conference*, 1984.

[10] P. Wang, M. J. T. Guy, J. H. Davenport, *p-adic Reconstruction of Rational Numbers*, in SIGSAM Bulletin, **16**, No 2 (1982).

[11] R. Zippel, Probabilistic algorithms for sparse polynomials, *Proceedings of EUROSAM '79*, Springer-Verlag LNCS, **2** (1979), pp. 216–226.