Computing Univariate GCDs over Number Fields.

Michael Monagan* Roger Margot†

Abstract

We compare four fast methods for univariate polynomial GCD computation over an algebraic number field. The first two are the modular method of Langemyr and McCallum (1987), and the heuristic method of Smedley et al. (1988). Because of recent improvements to the modular method by Encarnacion (1994), we expected it now to always be the better of the two in practice, provided it is implemented "properly". This turned out to be the case in our Maple implementations. We also implemented a quadratic lifting Hensel based method and a more direct method which we call the prime-power method. Like the heuristic and quadratic Hensel methods, the prime-power method is simple to implement efficiently, but it is usually better. Due to the large effort required to implement the modular method efficiently, we recommend the prime-power method as a practical alternative.

1 Introduction

For 10 years the algorithm Maple used to compute polynomial greatest common divisors (GCDs) over the integers was the heuristic method GCDHEU of Gonnet et al [1]. Given two polynomials $a, b \in \mathbf{Z}[x]$, this method computes the GCD of $a(x)$ and $b(x)$ by computing the integer GCD of $a(n)$ and $b(n)$ for a suitably chosen integer n, then reconstructs the polynomial GCD from this integer GCD. This method reduces a polynomial GCD computation to one large integer GCD computation. Because the integer GCD operation was carefully implemented in the Maple kernel (in compiled C), this method had an advantage, though not asymptotic, over other GCD methods. The same is the case with other computer algebra systems.

There are two competing methods that one should consider if one is going to implement a fast algorithm for GCDs in $\mathbf{Z}[x]$. Throughout this paper we will call these two methods the modular method and the Hensel methods, though both are "modular methods". Chapter 6 and 8 in Geddes et al., [2], give descriptions of all three methods.

The modular method computes the $\mathrm{GCD}(a, b)$ modulo suitably chosen primes, usually machine primes, then combines these modular GCDs using Chinese remaindering. This method was never seriously considered in Maple because Maple was simply not efficient enough at computing GCDs modulo small primes to make it competitive.

The Hensel method begins by computing the GCD modulo a single prime p. Hensel's lemma is then applied to *lift* this image GCD and its cofactor to a power of p which is sufficiently large to enable reconstruction of the GCD over $\mathbf{Z}$. Linear and quadratic versions of Hensel lifting exist. The linear Hensel method was implemented in Maple but it was not competitive with the heuristic method because the supporting polynomial arithmetic modulo small primes was not efficient.

Let us look at the theoretical complexity of the methods. Let $g = \mathrm{GCD}(a, b)$ where $a, b \in \mathbf{Z}[x]$. Consider a GCD problem where the degree of the GCD g is n, the largest coefficient of g is m digits in length, and the cofactors $\bar{a} = a/g$ and $\bar{b} = b/g$ are the same size as the GCD. We call this the balanced GCD problem. Assuming classical arithmetic is used for integer and polynomial arithmetic, which is usually the case in practice, the cost of the modular method and the linear Hensel lifting method is $O(n^2m + nm^2)$.[1] In comparison, the cost of the heuristic method and the quadratic Hensel lifting method is $O(n^2m^2)$. Thus if one is comparing the modular and heuristic methods, the modular method *should* be the method of choice.

To implement the modular method effectively, however, one needs to compute GCDs modulo primes efficiently. Specifically, one computes GCDs in the ring $\mathbf{Z}_p[x]$ using the Euclidean algorithm where one chooses machine primes so that the modular arithmetic is done using the hardware integer arithmetic instructions. If every modular arithmetic operation resulted in a storage allocation the efficiency loss would be severe.

*Centre for Experimental and Constructive Mathematics, Simon Fraser University, Burnaby, British Columbia, Canada V5A 1S6. monagan@cecm.sfu.ca

†Institut für Werkzeugmaschinen und Fertigungstechnik, ETH Zentrum, CH-8092 Zurich, Switzerland. margot@iwf.bepr.ethz.ch

[1] An analysis of the modular and linear Hensel lifting methods showing them both to be $O(n^2m + nm^2)$ is presented by Miola and Yun in [5].

Furthermore, to be competitive with the heuristic method where the integer GCD operation will be implemented "in-place", i.e., the total storage allocated is linear in the size of the inputs and it is carefully recycled during the GCD computation to eliminate costly storage management overhead, then the GCDs computed modulo machine primes must likewise be computed in-place. Even if each polynomial remainder operation in the Euclidean algorithm resulted in a storage allocation, considerable efficiency is lost – an overall factor of between 2 and 3 in our experience. Thus storage must be pre-allocated and recycled as needed. This means that in CA systems like Maple and Axiom, one must write in the systems implementation language, i.e., in C or Lisp or assembler. Similar implementation difficulties arise if one wants to implement linear Hensel lifting.

Implementing the modular method or the linear Hensel lifting method properly is a major implementation task. Note, in the context of a CA system, there is no point in doing this if the polynomial multiplication and exact division algorithms do not also use correspondingly fast algorithms because they are used more frequently. In fact, in most CA systems, polynomial multiplication and exact division are quadratic, i.e., have complexity $O(n^2m^2)$.

For a long time we did not attempt this in Maple because the heuristic GCD algorithm performed satisfactorily. However, to support the factorization of polynomials over the integers, Monagan in [6] implemented the *modp1* package. This package makes available in Maple a new data structure for polynomial arithmetic in $\mathbf{Z}_n[x]$, including routines for multiplication, quotient and remainder, GCD and resultant, evaluation and interpolation. For the machine prime case, the data structure is essentially an array of machine integers and all mentioned polynomial operations execute in-place. Using this package Monagan coded the modular GCD method and found that the modular method was almost always better than the heuristic method, hence, it is now the default method in Maple.

1.1 Over Algebraic Number Fields. The problem being addressed in this paper is how to compute univariate polynomial GCDs over algebraic number fields. Let $\mathbf{Q}(\alpha)$ be an algebraic number field with minimal polynomial $M(y)$ of degree d[2]. It is possible to apply the same ideas for computing GCDs over $\mathbf{Z}$ to $\mathbf{Q}(\alpha)$. Again, one can design a modular method, a heuristic method, and Hensel methods. Comparing the algorithms is more difficult than in the integer case because the complexity depends on $M(y)$. We consider $g(x)$ a GCD of degree n in x with coefficients polynomials of degree $d-1$ in y whose coefficients are integers of size m digits in length. To compare algorithms we again consider a balanced GCD problem where the cofactors are the same size as the GCD.

Langemyr and McCallum studied the modular method for $\mathbf{Q}(\alpha)$ in [4]. The modular method has the same implementation disadvantage as before, namely, in order to obtain good performance, the implementation requires a lot of careful coding at the system level. One must compute GCDs over the residue ring $\mathbf{Z}_{p_i}[y]/(M)$ for suitably chosen (machine) primes p_i efficiently. The modular method of Langemyr and McCallum had a technical difficulty that required that α be an algebraic integer. This was resolved by Encarnacion in [3]. Encarnacion also used Wang's rational reconstruction [8] so that a small GCD can be detected early. Using Encarnacion's improvements, we show in Appendix A that the theoretical running time for our balanced GCD problem is $\mathrm{O}(dm^2n + d^2mn^2)$.

Smedley et al in [7] extended the heuristic method to $\mathbf{Q}(\alpha)$. Their approach is to map the algebraic number α into $\mathbf{Z}$. One chooses a suitable integer n, replaces α by n and computes the GCD modulo the integer $M(n)$. If the integer $M(n)$ has small integer factors, the polynomial GCD computation is likely to fail. This can be avoided by dividing $M(n)$ by small factors of $M(n)$ or by choosing a different integer n. This method has the advantage of reducing the computation to a single univariate GCD modulo a large integer $M(n)$, thus requiring only a good implementation of long integer arithmetic. The method has time complexity $\mathrm{O}(n^2m^2d^2)$ for our balanced GCD problem.

Therefore it seemed to us that a good implementation of the modular method would also be the fastest method for computing GCDs over $\mathbf{Q}(\alpha)$. This turned out to be the case. The implementation of the modular method is discussed in the Section 2. Our main contribution to this method is a discussion of the implementation details needed to make the method competitive.

We also implemented a quadratic Hensel lifting method for computing univariate GCDs over $\mathbf{Q}(\alpha)$. The algorithm is essentially the same as that for over $\mathbf{Z}$. A description can be found in Chapter 6 of Geddes et al [2]. The running time is also $O(n^2m^2d^2)$ for the balanced GCD problem. We realized that in the case of univariate GCD computation, going from the GCD mod p^{2^k} to mod $p^{2^{k+1}}$ can more easily be done by computing the GCD mod $p^{2^{k+1}}$ directly instead of constructing it from the GCD mod p^{2^k} using Hensel's lemma. Furthermore, the problem of an "unlucky"

[2] In this paper we consider only the case of a simple algebraic extension over $\mathbf{Q}$.

prime in the Hensel methods[3] can be neatly resolved if we simply use a different prime at each step, i.e., we compute the GCD mod $p_k^{2^k}$ for $k = 0, 1, 2, ...$ We call this simple method the prime-power method. The algorithm is described in detail in Section 3. It is the simplest of the four algorithms to implement. We show in Appendix A that it also has time complexity $O(m^2d^2n^2)$, the same as the heuristic and quadratic Hensel lifting methods.

Section 4 compares the four methods for two classes of GCD problems. All methods were implemented on top of the modp1 package where the core routines are implemented carefully in compiled C code. The timing comparisons show that in practice the modular method is indeed the fastest method as expected from the theoretical complexity estimates. They also show that the that prime-power method is competitive with the heuristic method, always better than the quadratic Hensel lifting method by a factor of 2 to 3, and competitive with the modular method for modest coefficients. Since the modular method is difficult to implement well, and since there is really no point in implementing it unless one does so for polynomial multiplication and division as well, we recommend the prime-power method as a good alternative to the system implementor. This is the practical conclusion of the paper. The other conclusion is that there is no point in using the quadratic Hensel lifting for univariate GCDs – the prime-power method has the same asymptotic complexity but should always be faster by a constant factor and it easier to implement. The reader may now study the details of the modular method in Section 2, of the prime-power method in Section 3, and the timing comparisons made in Section 4.

Throughout the paper we use these notations, definitions, and assumptions.

Let $K = \mathbf{Q}(\alpha)$ be an algebraic number field over $\mathbf{Q}$.
Let $M(y) \in \mathbf{Z}[y]$ be the minimal polynomial for α with degree $d > 0$.
Let $lc(a(x))$ denote the leading coefficient of the polynomial $a(x)$.

Normally $M(y)$ would be monic over $\mathbf{Q}$. To simplify the presentation of algorithms, we multiply the minimal polynomial by the least common multiple of all denominators appearing in the coefficients. Consequently, the primes chosen in the algorithms must not divide $lc(M(y))$.

Let Δ be the discriminant of $M(y)$.
Let $\mathbf{Z}_n$ denote the integers modulo n.

[3]The problem is that the Hensel methods will not detect an unlucky prime until one lifts to a bound, which may be expensive.

Let $R_p^d = \mathbf{Z}_p[y]/(M(y))$ be the finite ring of polynomials modulo $M(y)$ modulo p, a prime.

Let $a(x), b(x) \in K[x]$. We wish to compute $g(x) = GCD(a, b)$. We denote the cofactors of $a(x)$ and $b(x)$ by $\bar{a}(x) = a(x)/g(x)$ and $\bar{b}(x) = b(x)/g(x)$. In all our algorithms we assume that the input polynomials are scaled to have integer coefficients so that we may view them as polynomials in $\mathbf{Z}[y][x]$.

2 The Modular Method

In rough outline, the modular method [3] with rational reconstruction [8] computes the GCD(a, b) as follows:

Step 1 Choose primes p_i which do not divide Δ, $lc(M)$, $lc(a(x))$, and $lc(b(x))$. Let $m = \Pi_i p_i$.

Step 2 Compute g_i = GCD(a mod p_i,b mod p_i) and make the g_i monic. If the GCD computation fails modulo p_i then use a different prime. If $g_i = 1$ then output 1 and stop.

Step 3 Apply the Chinese remainder theorem to obtain $f \in \mathbf{Z}_m[y][x]$ such that $f \equiv g_i \bmod p_i$.

Step 4 Apply rational reconstruction to the integers in f. If rational reconstruction fails then go to step 2. Otherwise we obtain $h \in \mathbf{Q}[y][x]$ such that $h \equiv f \bmod m$.

Step 5 View $h(x)$ as a polynomial in $K[x] = \mathbf{Q}[y][x]/(M(y))$. If $h|a$ and $h|b$ then output h and stop. Otherwise, go back to step 1 and try a different set of primes.

The conditions on the choice of the primes in step 1 are the usual ones. The condition that p_i not divide $lc(M)$ guarantees that M does not vanish modulo p_i, and it is necessary for reconstruction of g. The other conditions are sufficient to ensure that $\deg(g_i) \geq \deg(g)$ so that the algorithm does not mistakenly return a divisor of g.

The GCD computation in step 2 is performed over the finite ring $\mathbf{Z}_{p_i}[y]/(M)$. Arithmetic in this ring is implemented as polynomial arithmetic. If this ring is not a field then the GCD computation may "fail" in step 2 if an inverse does not exist. If p_i is chosen suitably large, the probability that this happens is low. If it happens, one simply chooses another prime.

Rational reconstruction in step 4 can "fail" because not all integers in $\mathbf{Z}_m$ correspond to rationals of a fixed size. If this happens, it just means that more image GCDs are needed in step 2.

The trial division in step 5 is to detect *unlucky* primes. The prime p_i is *unlucky* if $\text{res}_x(\bar{a}, \bar{b}) \equiv 0 \bmod p_i$, equivalently, $\deg(g_i) > \deg(g)$. This happens rarely. But if it happens, g_i cannot be used. Note, if we find that $\deg(g_j) > \deg(g_k)$ then p_j must be an unlucky prime so g_j can be discarded.

How many primes? The algorithm as sketched is incomplete because it does not say how many primes are needed. One needs enough primes to reconstruct the fractions in the GCD but one does not know in advance how large those fractions are. Langemyr and McCallum in [4] use a bound on the size of the coefficients in the GCD to establish how many primes to use. This is inefficient when the GCD is small. One could try steps 3 — 5 after each modular GCD is computed until eventually one has enough primes to reconstruct the GCD. However, this is inefficient as too much work is done in step 4.

The approach taken by Smedley et al in [7] for the heuristic method is to assume that either the GCD or a cofactor has coefficients of size at most half the size of the coefficients in the input.

Another approach that we tried was to estimate the size of the coefficients in the GCD after doing one image GCD computation from the following information: (i) the degree of $a(x)$ and $b(x)$, (ii) the size of the largest integer coefficients in $a(x)$ and $b(x)$, and (iii) the degree of $g_1(x)$. One might assume that if the degree of g_1 is small, then the size of the coefficients of the GCD will be proportionately small and vice versa.

However, none of these approaches is good if the GCD has small coefficients. Instead we use the following simple scheme which ensures early detection of a GCD with small coefficients. We start with one prime then we double the number of primes used each time in steps 1,2 before we attempt steps 3,4,5. This ensures that we do not execute step 4 too often and that we do not overestimate the size of the fractions in the GCD by more than a factor of two. Here is a complete description of the algorithm.

2.1 Algorithm ModularGCD.

Input Polynomials $a, b \in \mathbf{Z}[y][x]$ and minimal polynomial $M \in \mathbf{Z}[y]$.

Output $g \in \mathbf{Q}[y][x]$ where $g = \text{GCD}(a, b)$ over $\mathbf{Q}[y][x]/(M(y))$.

Step 0 (Initialization)
Set $k = 0$ – the number of the current prime
Set $m = 1$ – the product of primes
Set $D = \min(\deg(a), \deg(\text{b}))$ – upper bound on the degree of the GCD
Set $g = 0$ – the combined image GCDs

Step 1 (Choose next prime)
Set $k = k + 1$.
Choose the next prime p_k such that $\Delta \not\equiv 0 \bmod p_k$, $lc(M) \not\equiv 0 \bmod p_k$, $lc(a) \not\equiv 0 \bmod p_k$, and $lc(b) \not\equiv 0 \bmod p_k$.

Step 2 (Image computation)
Compute $g_k = \text{GCD}(a \bmod p_k, b \bmod p_k)$.
If the GCD computation fails then set $k = k - 1$ and go to step 1.
If $g_k = 1$ then output 1 and stop.
If $\deg(g_k) > D$ then set $k = k - 1$ and go to step 1 (unlucky prime)
If $\deg(g_k) < D$ then (all previous primes were unlucky) then

> Set $g_1 = g_k, m = 1, k = 1, g = 0$ and $D = \deg(g_k)$.

Step 3 (Chinese remaindering)
Update g so that $g \equiv g_k \bmod p_k$ and $g \equiv g \bmod m$.
Set $m = p_k \times m$.
If $k < 2^n$ for some n then go to step 1.

Step 4 (Rational reconstruction)
Apply rational reconstruction to the integers in g to obtain $h \in \mathbf{Q}[y][x]$ such that $h \equiv g \bmod m$. If rational reconstruction fails then go to step 2.

Step 5 (Trial division)
View h as a polynomial in $K[x] = \mathbf{Q}[y][x]/(M(y))$.
If $h|a$ and $h|b$ then output h and stop.
Otherwise, go to step 1.

The theoretical cost of the algorithm for the balanced GCD problem is determined in Appendix A to be $O(dm^2n + d^2mn^2)$.

A more complicated approach with the same asymptotic complexity which reduces the number of modular image GCDs to can be obtained from the following idea. After computing each image GCD try to reconstruct one rational coefficient. If unsuccessful, or the rational is different from that constructed in the previous iteration, compute another image GCD and try to reconstruct the rational coefficient again. If successful, and the rational is the same as that computed in the previous step, assume that the rational is correctly determined and proceed to reconstruct the next coefficient in the polynomial in the same manner. Keep doing this until every rational coefficient in the GCD is so obtained and then attempt step 5. This approach will require only one more image GCD computation than what is necessary.

3 The Prime Power Method

We first consider the trial division problem. This is important because it is often the bottleneck in a GCD computation. Given $a, b \in K[x]$ we want to test if $a|b$ over K, and if it does, to return the quotient. A fast way to show that a does not divide b is to choose a (machine) prime p which does not divide $lc(a)$ and does not divide $lc(M)$ and do the division modulo p. If the division is successful, and the remainder mod p is not zero, then we can stop. We have proven that a does not divide b. The division fails if and only if the $lc(a)$ is not invertible modulo p. This may happen if $\mathbf{Z}_p[y]/(M(y))$ is not field. If the primes are large enough, this failure will happen with low probability. Should it happen, we choose another prime and try again. To compute the quotient q, we try to reconstruct it from the quotient modulo $p_k^{2^k}$ for $k = 0, 1, 2, \ldots$.

Input Polynomials $a, b \in \mathbf{Z}[y][x]$ and minimal polynomial $M(y)$.

Output Either $q \in \mathbf{Q}[y][x]$, the quotient such that $b - aq = 0$ over $\mathbf{Q}(\alpha)$, or false, meaning a does not divide b with 0 remainder.

Step 0 (Initialization)
Set $k = 0$.

Step 1 (Choose new prime)
Choose the next (machine) prime p_k such that $lc(M(y)) \not\equiv 0 \bmod p_k$ and $lc(a(x))$ is invertible mod $M(y)$ mod p_k.

Step 2 (Compute quotient)
Compute the quotient q and remainder r of b divided by a modulo $p_k^{2^k}$. If the remainder r is not zero then output false and stop.

Step 3 (Rational Reconstruction)
Apply rational reconstruction to the coefficients of q modulo $p_k^{2^k}$. If rational reconstruction fails, set $k = k + 1$ and go to step 1.

Step 4 (Test quotient)
Compute $error = a - qb$ over $\mathbf{Q}[y][x]$.
If $M(y)$ divides $a - qb$ over $\mathbf{Q}[y][x]$, output q and stop. Otherwise set $k = k + 1$ and go to step 2.

This algorithm terminates because it must eventually choose a prime p_k such that $lc(a)$ is invertible and $p_k^{2^k}$ is large enough for reconstruction of the fractions in the quotient.

The implementation of step 4 should be done coefficient by coefficient so that if $p_k^{2^k}$ is too small, this can be detected early.

One could try to estimate the size of the rational coefficients in the quotient and use an appropriate power of p_k directly. This, however, is non-trivial as one must estimate how big the coefficients in $1/lc(a(x))$ are without computing it.

3.1 Algorithm PrimePowerGDC.

The prime-power algorithm for computing the GCD(a, b) is essentially the same as that for the trial division algorithm. The differences are in the conditions on the primes, the handling of unlucky primes, and the early detection of a GCD which is 1.

Input Polynomials $a, b \in \mathbf{Z}[y][x]$ and minimal polynomial $M \in \mathbf{Z}[y]$.

Output $g \in \mathbf{Q}[y][x]$ where $g = \text{GCD}(a, b)$ over $\mathbf{Q}[y][x]/(M(y))$.

Step 0 (Initialization)
Set $k = 0$, $D = \min(\deg(a), \deg(\text{b}))$.

Step 1 (Choose next prime)
Set $k = k + 1$.
Choose the next (machine) prime p_k such that $\Delta \not\equiv 0 \bmod p_k$, $lc(M) \not\equiv 0 \bmod p_k$, $lc(a) \not\equiv 0 \bmod p_k$, and $lc(b) \not\equiv 0 \bmod p_k$.

Step 2 (Image computation)
Compute $g_k = \text{GCD}(a \bmod p_k^{2^k}, b \bmod p_k^{2^k})$.
If the GCD computation mod $p_k^{2^k}$ fails then set $k = k - 1$ and go to step 1.
If $g_k = 1$ then output 1 and stop.
If $\deg(g_k) > D$ then set $k = k - 1$ and go to step 1 (unlucky prime)
If $\deg(g_k) < D$ then (All previous primes were unlucky)

Set $g_1 = g_k, k = 1$ and $D = \deg(g_k)$.

Step 3 (Rational reconstruction)
Apply rational reconstruction to the integers in g_k mod $p_k^{2^k}$.
If rational reconstruction fails then go to step 2.
Otherwise we have $h \in \mathbf{Q}[y][x]$.

Step 4 (Trial division)
View h as a polynomial in $K[x] = \mathbf{Q}[y][x]/(M(y))$.
If $h|a$ and $h|b$ then output h and stop.
Otherwise go to step 1.

The correctness of the prime-power algorithm is established by arguing that it must eventually choose a prime power $p_k^{2^k}$ such that the prime p_k is not unlucky,

the GCD computed in step 2 does not fail, and $p_k^{2^k}$ is large enough to enable reconstruction of the rational coefficients in the GCD $g(x)$.

The prime-power algorithm for GCD computation may compute up to twice the number of image GCDs needed. This can be reduced to at most 50% more than needed by combining the last two images g_{k-1} mod $p_{k-1}^{2^{k-1}}$ with g_k mod $p_k^{2^k}$ using Chinese remaindering.

4 Implementation Details and Timings Results

The comparison of the four methods presented in this section was done on a Sun 10 using Maple V Release 4. This version of Maple uses classical (i.e., quadratic) algorithms for integer and polynomial multiplication and division. To make a consistent and fair comparison, the main operations of all four GCD algorithms are implemented using the modp1 package in Maple. The modp1 package implements arithmetic for $\mathbf{Z}_n[x]$ in the Maple kernel in compiled C. There are two separate representations, one for moduli which fit in a machine integer, and one for moduli which do not. In both cases, the basic arithmetic operations of addition, multiplication, quotient and remainder, utility routines such as Chinese remaindering and conversions from Maple's general purpose representation to the modp1 representations, are implemented in the Maple kernel (in compiled C) using "in-place" and "on-line" algorithms. These algorithms allow sums of products to accumulate in the polynomial multiplication and division algorithms before reducing modulo n. Thus the core routines for arithmetic in $\mathbf{Z}_n[x]$ are efficiently implemented. See [6] for details.

Since the heuristic method maps the computation in $\mathbf{Q}(\alpha)[x]$ to $\mathbf{Z}_n[x]$, the GCD computation that it does uses the modp1 code directly. The other three methods do arithmetic in $R_p^d[x]$. Generic Maple code is used for the polynomial operations and the modp1 package is used for the coefficient arithmetic in R_p^d. For $M(y)$ of *low* degree, for example degree $d = 2,3$, or 4, the overhead of the Maple interpreter and the storage allocations that take place for each arithmetic operation in R_p^d may be significant. This overhead drops quickly to zero as d increases.

The timing data presented includes the time spent doing the two trial divisions required by all four methods. We also monitored the time spent doing Chinese remaindering, rational reconstruction, and the modular GCDs. We found that the time spent doing Chinese remaindering and the rational reconstruction never dominated the overall time.

Balanced Case. The data in Table I below was generated as follows. All timings are in CPU seconds. The parameter m specifies the size of the integer coefficients in the GCD $g(x)$, and the parameter d the degree of the minimal polynomial $M(y)$. $M(y)$ is a random, dense, monic, irreducible polynomial of degree d with random coefficients of size $d/2$ digits in length. $g(x)$ and the cofactors $\bar{a}(x)$ and $\bar{b}(x)$ are random dense polynomials of degree $n = 4$ in x with coefficients random dense polynomials of degree $d-1$ in α whose coefficients are random integers of length m digits. Thus the input polynomials $a(x)$ and $b(x)$ are dense of degree 8.

Recall that the theoretical analysis for the complexity of the four methods for this GCD problem for the modular method was $O(dm^2n+d^2mn^2)$ and $O(d^2m^2n^2)$ for the other methods. Hence, the modular method should be the fastest method for large m. That the increase in timings is not smooth is due to the *overshoot* present in the methods, i.e., the doubling of the number of primes in the modular method and the doubling of the size of the modulus in the other methods overshooting the size of the largest integer in the GCD.

Small GCDs. An important case is when the GCD is small. This is the worst case for the Euclidean algorithm but usually the best case for the modular algorithms. All of the algorithms except the heuristic algorithm should do better in this case because. The Maple implementation of the heuristic algorithm begins with a medium sized evaluation point. Hence, we expect to find the modular, prime-power and Hensel methods improving relative to the heuristic method when compared with the balanced case.

The timings in Table II are for GCDs of degree 2 with coefficients random integers of size $1 + m/8$ digits. The cofactors are degree 6 and the cofactors have coefficients of size $2m$ digits. The GCDs and cofactors are dense. Thus the input polynomials $a(x)$ and $b(x)$ here have the same size as those in the balanced case but the GCD is relatively small.

Notice that the cost of division is very significant compared with the time to compute the GCD – in fact, it dominates the cost of the faster algorithms. This is because the trial divisions, which are required by all four methods, must construct the cofactors which are relatively large.

References

[1] B. W. Char, K. O. Geddes, G. H. Gonnet, *GCDHEU: Heuristic Polynomial GCD Algorithm based on Integer GCD Computation*, Proceedings of EUROSAM 84, Springer-Verlag LNCS **174** (1984), pp. 285–296.

[2] K. O. Geddes, S. R. Czapor, G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[3] M. J. Encarnacion, *Computing GCDs of Polynomials*

m	d	div	mod	pow	heu	hen
4	2	.10	.45	.25	.08	1.13
8	2	.06	.46	.40	.50	.91
16	2	.06	1.01	.90	.21	2.25
32	2	.13	2.51	1.31	.56	4.18
64	2	.41	5.45	3.20	1.93	8.26
128	2	1.10	6.61	9.80	7.30	18.85
256	2	3.50	18.25	31.00	27.20	50.96
512	2	12.26	55.80	113.86	103.30	167.36
1024	2	47.25	168.81	424.43	411.11	598.48
4	4	.10	.36	.53	.21	1.65
8	4	.11	.85	.60	.35	2.11
16	4	.20	1.56	1.56	.81	4.31
32	4	.38	3.63	2.91	2.53	8.05
64	4	1.13	7.53	7.95	8.71	17.71
128	4	4.21	11.48	22.81	27.73	43.53
256	4	14.28	36.36	76.08	107.15	127.63
512	4	48.13	117.16	273.80	404.78	413.03
4	8	.58	.76	.96	1.28	3.28
8	8	.55	1.36	1.58	2.08	4.35
16	8	1.01	2.56	2.93	4.15	9.16
32	8	2.31	5.60	7.11	11.91	18.31
64	8	5.96	14.43	18.96	31.88	41.68
128	8	17.01	27.16	55.60	100.98	102.48
4	16	13.35	13.65	13.86	40.43	21.35
8	16	14.31	16.15	17.26	47.61	25.38
16	16	18.21	20.73	31.18	78.18	52.55
32	16	34.10	42.46	52.78	144.46	102.60
64	16	67.61	85.30	113.15	295.13	208.25

Table I: Balanced Case.

div	=	trial divisions
mod	=	modular algorithm
pow	=	prime-power algorithm
heu	=	heuristic algorithm
hen	=	Hensel algorithm

Key

m	d	div	mod	pow	heu	hen
8	2	.08	.36	.28	.13	1.43
16	2	.08	.51	.28	.15	1.78
32	2	.10	.68	.80	.28	1.91
64	2	.18	.96	.68	.78	2.06
128	2	.66	2.10	1.61	2.58	4.68
256	2	1.83	5.25	3.75	9.16	10.26
512	2	5.00	12.90	10.35	28.53	24.80
1024	2	18.88	28.81	28.40	99.13	67.88
2048	2	72.68	101.21	105.40	468.26	254.56
8	4	.15	.40	.76	.35	2.68
16	4	.25	.60	.81	.60	3.70
32	4	.28	1.33	1.43	1.58	3.76
64	4	.51	1.55	1.50	3.18	3.98
128	4	1.95	3.58	4.11	10.31	11.65
256	4	8.03	12.51	12.78	37.16	26.75
512	4	24.05	34.76	35.63	123.96	67.23
1024	4	91.10	105.08	126.23	480.90	205.95
8	8	1.15	1.33	1.55	2.58	6.18
16	8	1.21	1.76	1.98	3.91	8.83
32	8	1.96	3.11	3.75	10.80	9.68
64	8	4.26	5.45	5.86	16.60	12.11
128	8	10.13	12.58	16.95	48.56	34.80
256	8	42.23	49.06	53.93	164.01	90.60
8	16	23.36	29.71	30.83	88.90	47.05
16	16	29.28	28.35	33.00	103.38	60.06
32	16	41.83	44.23	47.06	140.86	57.88
64	16	52.38	55.83	57.01	201.98	94.30
128	16	122.05	127.40	138.03	422.91	184.85

Table II: Small GCD.

over Algebraic Number Fields, J. Symbolic Computation, **20** (1995), pp. 299–313.

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

[5] A. Miola, D. Y. Y. Yun, *Computational Aspects of Hensel-type Univariate Polynomial Greatest Common Divisor Algorithms*, Proceedings of EUROSAM '74 (1974), pp. 46–54.

[6] M. B. Monagan, *In-Place Arithmetic for Polynomials over* $\mathbf{Z}_n$, Proceedings of the DISCO'92, Springer-Verlag LNCS **721** (1993), pp. 81–94.

[7] K. O. Geddes, G. H. Gonnet, T. J. Smedley, *Heuristic Methods for Operations with Algebraic Numbers*, Proceedings of ISSAC'88, Springer-Verlag LNCS **358** (1988), pp. 475–480.

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

Appendix A: Complexity Analysis

We give a running time analysis for the cost of computing the GCD of $a, b \in K[x]$ using the modular and prime-power algorithms. The running time will depend on the degree of the GCD $g(x)$, the degree of the input polynomials $a(x), b(x)$, the size of their coefficients, the degree d of the minimal polynomial $M(y)$, and the size of its coefficients.

We consider the case of dense polynomials throughout, that is, $M(y)$, $g(y,x)$, $a(y,x)$, $b(y,x)$ are dense in y and x. We consider the balanced case where the GCD g and the cofactors are polynomials of the same size; of degree n and with integer coefficients of size m digits. Thus the degree of the input polynomials $a(x)$ and $b(x)$ is $2n$. To further simplify the analysis we consider cases where $M(y)$ is monic and its integer coefficients are relatively small in size compared with m, thus, the size of the integer coefficients in $a(x)$ and $b(x)$ will be approximately $2m$ digits in length. We remark that problems in practice often admit these assumptions.

In our complexity estimates, we assume that integer and polynomial arithmetic use classical algorithms for multiplication and division with remainder, i.e., have quadratic complexity. We assume also that the cost of trial division has the same complexity or lower than the cost of computing the GCD. This is reasonable since the same modular, heuristic, or prime-power method can be used for division.

The modular GCD algorithm. The modular algorithm does the following computations.

1 Reduction (in Step 2): Map $a, b \in \mathbf{Z}[y][x]$ to $a, b \in \mathbf{Z}_p[y][x]$. This requires taking $2(n+1)d$ coefficients of length O(m) digits modulo p. The cost of taking one coefficient mod p is $O(m)$ since we are assuming p is a machine prime, i.e., arithmetic in $\mathbf{Z}_p$ is $O(1)$. Hence the cost of a single reduction is $O(dnm)$

2 GCD computation (Step 2): Computing a GCD in $\mathbf{Z}_p[y][x]/(M(y))$. Since our polynomials are dense of degree $2n$, and the GCD computed has degree n, the Euclidean algorithm does $O(n^2)$ arithmetic operations in R_p^d. Each arithmetic operation in R_p^d requires $O(d^2)$ operations in $\mathbf{Z}_p$. Hence the cost of a single GCD is $O(n^2d^2)$.

3 Chinese remaindering (Step 3): Combining m images requires applying the Chinese remainder algorithm to each set of coefficients. There are at most nd integers to reconstruct (the leading coefficient is monic hence does not require reconstruction), each of which costs $O(m^2)$. Hence the cost of Chinese remaindering is $O(ndm^2)$.

4 Rational reconstruction (Step 4): Rational reconstruction of a single fraction uses the Euclidean algorithm. The cost for reconstructing a fraction from an integer modulo a modulus of length m digits is quadratic, i.e., $O(m^2)$. Since there are nd coefficients to reconstruct, the cost of rational reconstruction is $O(ndm^2)$.

Since unlucky primes are rare, and we need $O(m)$ primes to reconstruct the GCD, steps 1 and 2 are done $O(m)$ times. Since the number of primes used before reconstruction is attempted, is doubling at each step, the cost of step 3 and step 4 is dominated by the last time it is executed. This yields a total cost of $O(dnm^2)$ + $O(mn^2d^2)$ + $O(ndm^2)$ + $O(ndm^2)$ for steps 1,2,3,4 respectively, that is $O(dnm^2 + d^2n^2m)$ overall.

The prime-power GCD algorithm. Since it will be shown that the cost of the prime-power GCD algorithm is quadratic in the parameters n, m, d, and the algorithm is doubling the size of the modulus at each step, the last step – where p^{2^n} enables reconstruction of the GCD, will dominate the cost. This will occur when the length of p^{2^n} is $O(m)$. The prime power GCD algorithm does the following computations.

1 Reduction (in Step 2): Map $a, b \in \mathbf{Z}[y][x]$ to $a, b \in \mathbf{Z}_{p_k^{2^k}}[y][x]$. This requires taking $2(n+1)d$ coefficients of length O(m) digits modulo $p_k^{2^k}$ a prime of length O(m) digits. The cost of taking one integer coefficient mod $p_k^{2^k}$ is $O(m^2)$. Hence the cost of a reduction is $O(dnm^2)$

2 GCD computation (Step 2): Computing a GCD in $\mathbf{Z}_{p_k^{2^k}}[y][x]/(M(y))$. Since our polynomials are dense of degree $2n$, and the GCD computed has degree n, the Euclidean algorithm does is $O(n^2)$ arithmetic operations in $\mathbf{Z}_{p_k^{2^k}}[y]$. Each arithmetic operation requires $O(d^2)$ operations in $\mathbf{Z}_{p_k^{2^k}}$ where arithmetic operations cost $O(m^2)$. Hence the cost of a single GCD is $O(n^2d^2m^2)$.

3 Rational reconstruction (Step 3): $O(ndm^2)$ - same as for the modular method.

Hence, the overall cost is $O(d^2m^2n^2)$. It is dominated by the last GCD in step 2.