Huelse, D., & Hemmer, M. (2009). Generic implementation of a modular gcd over algebraic extension fields. Paper presented at 25th European Workshop on Computational Geometry, Brussels, Belgium.

Peer reviewed version

Generic implementation of a modular gcd over Algebraic Extension Fields

Michael Hemmer * Dominik Hülse †

Abstract

We report on several generic implementations for univariate polynomial gcd computation over the integers and, in particular, over algebraic extensions. Our benchmarks show that the generic implementation compares favorably to well established libraries. Even for the integer case our implementation is competitive to the one provided by the NTL, which does not support algebraic extensions. Our software is part of the new Polynomial package of CGAL release 3.4.

1 Introduction

For exact computation with non-linear geometric objects, such as semi-algebraic curves and surfaces, it is evident that the computation of the gcd of polynomials is one of the fundamental tools. This is the case for polynomials defined over rational coefficients as well as over algebraic extensions [6, 2]. It is known that methods based on modular arithmetic are indispensable for an efficient implementation [7, 5]. To the best of our knowledge there was no *generic* open source code available that supports algebraic extensions. Our software is *generic* in the sense that it uses C++'s template techniques [1] such as traits classes, and function objects. In particular, our code is independent from the coefficient type, even though we only report on algebraic extensions of degree 2 here. The presented implementation is part of the new Polynomial package of CGAL[1] release 3.4.

The paper is structured as follows: Section 2 provides an overview of the investigated algorithms. Section 3 presents the comparison of the different implementations including a comparison with the NTL[2] for integer polynomials and a comparison with Singular[3] for algebraic extensions of degree 2. Section 4 concludes the paper.

2 The Modular Methods

We now recall the principal ideas of modular gcd algorithms and the most fundamental modular methods of interest. In particular, we refer to Brown [3], who gave a solution for polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$. This was extended by Langemyr and McCallum [9] to polynomials over algebraic extensions using results from [14] in order to bound appearing denominators. Encarnacion [4] proposed a variant which uses rational reconstruction by Wang [12] in order to deal with denominators. We also present a hybrid approach that combines the ideas of both algorithms. All implementations are output sensitive, that is, the number of primes used within the computation depends on the size of the coefficients of the computed gcd and not on bounds based on the input polynomials.

2.1 Fundamentals

We restrict the presentation to univariate polynomials with integer coefficients. For more details study [3, sec. 4.3]. The principal idea is to compute the gcd with respect to several primes and to recover the original gcd in $\mathbb{Z}[x]$ or $\mathbb{Z}(\alpha)[x]$ using the Chinese Remainder theorem, e.g. see Knuth [7]. This avoids the exponential growth of coefficients in intermediate steps, whereas the actual gcd in practice has moderate coefficient size. Of course, it is important that the modular methods are output sensitive and do not rely on worst case bounds for the coefficient size in the final gcd, which is exponential [3].

For a given prime $p \in \mathbb{Z}$, let $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ be the Galois field with p elements and $\phi_p : \mathbb{Z} \to \mathbb{F}_p$ the field homomorphism defined by $\phi_p : x \mapsto (x \mod p)$. The homomorphism from $\mathbb{Z}[x]$ to $\mathbb{F}_p[x]$ induced by ϕ_p will also be denoted by ϕ_p. The image of ϕ_p will also be denoted as the *modular image* of some entity.

Let F be some polynomial in $\mathbb{Z}[x]$, we will use the following notation: $deg(F)$ - the degree of F; $lc(F)$ - the leading coefficient of F; $cont(F)$ - the content of F, that is, the gcd of all coefficients; $pp(F) = F/cont(F) \in \mathbb{Z}[x]$ - the primitive part of F; $monic(F) = F/lc(F) \in \mathbb{Q}[x]$ - the monic associate to F; $disc(F)$ - the discriminant of F.

2.2 GCD over the Integers

In this section we outline Brown's algorithm for polynomials with integer coefficients. Given $F_1', F_2' \in \mathbb{Z}[x]$, the algorithm computes $G' = gcd(F_1', F_2') \in \mathbb{Z}[x]$.

The core part of the algorithm is a while loop (step 5-13) that computes the gcd with respect to several primes until it is possible to recover the gcd using the Chinese Remainder Theorem.

*MPII Saarbrücken, hemmer@mpi-inf.mpg.de
†JoGU Mainz, dominik.huelse@gmx.de
[1] http://www.cgal.org/
[2] http://www.shoup.net/ntl/
[3] http://www.singular.uni-kl.de/

Algorithm 1: (Brown's algorithm) Given the polynomials $F_1', F_2' \in \mathbb{Z}[x]$ with $deg(F_1), deg(F_2) \geq 1$. Compute $G' \in \mathbb{Z}[x]$ the greatest common divisor of F_1' and F_2'.

(1) Set $c_1 = cont(F_1')$, $c_2 = cont(F_2'), c = gcd(c_1, c_2)$.
(2) Set $F_1 = F_1'/c_1$, $F_2 = F_2'/c_2$.
(3) Set $f_1 = lc(F_1)$, $f_2 = lc(F_2)$, $\overline{g} = gcd(f_1, f_2)$.
(4) Set n = 0, e = min(deg(F_1),deg(F_2)).
(5) Let p be a new odd prime not dividing $\overline{g}$
(6) Set $\tilde{g} = \phi_p(\overline{g})$, $\tilde{F_1} = \phi_p(F_1)$, $\tilde{F_2} = \phi_p(F_2)$.
(7) Invoke the Euclidean algorithm to compute $\tilde{G} = \tilde{g} \cdot\ gcd(\tilde{F_1}, \tilde{F_2})$, over $\mathbb{F}_p[x]$.
(8) If deg($\tilde{G}$) = 0: set G = 1 and goto (15).
If deg($\tilde{G}$) > e: (p is an unlucky prime) goto (5).
If deg($\tilde{G}$) < e: (the former primes were unlucky)
Set $n = 0$, $e = deg(\tilde{G})$.
(9) Set n = n+1.
(10) If $n = 1$: set $(q, G^\star) = (p, \tilde{G})$ and goto (5).
(11) Use Chinese Remainder to update $(q, G^\star)$:
$(q, G^\star) := chinese\_remainder((q, G^\star), (p, \tilde{G}))$.
(12) If the coefficients of $G^\star$ have changed goto (5).
(13) If $G^\star \nmid \overline{g} \cdot F_1$ or $G^\star \nmid \overline{g} \cdot F_2$ goto (5).
(14) Set G = pp($G^\star$).
(15) Output G' := cG;

For some (unlucky) primes it happens that the gcd loses a non trivial factor, which implies that the prime divides $lc(F_1)$ and $lc(F_2)$. The algorithm discards such primes in step 5. For other (unlucky) primes it happens that the gcd in $\mathbb{F}_p[x]$ contains additional factors. Therefore, the algorithm keeps track of $deg(\tilde{G})$, that is, it incorporates only those primes for which $deg(\tilde{G})$ is minimal (step 8).

Algorithm 1 deviates from the one of Brown [3] in the sense that it is output sensitive. Instead of computing as many primes as needed to guarantee a correct recovery of the gcd, it checks whether the recovered polynomial $G^\star$ becomes stable (step 12). If this is the case, $G^\star$ is in all probability the desired polynomial, which is verified in step 13. An idea which can, for instance, be found in Langemyr and McCallum [9].

Note that the Chinese Remainder can only recover polynomials in $\mathbb{Z}[x]$, that is, the algorithm must ensure that $G^\star$ is the image of a polynomial in $\mathbb{Z}[x]$. Therefore, $\tilde{G}$ is multiplied by $\tilde{g} = \phi_p(gcd(lc(F_1), lc(F2)))$ (step 7). Otherwise, $\tilde{G}$ as well as $G^\star$ would represent $monic(G)$ which is (in general) a polynomial in $\mathbb{Q}[x]$.

2.3 GCD over Algebraic Extension Fields

Given an algebraic number α and two polynomials F_1, $F_2 \in \mathbb{Z}(\alpha)[x]$, the following algorithms compute the greatest common divisor $G \in \mathbb{Z}(\alpha)[x]$ of F_1 and F_2 up to some constant factor. Note that it makes no sense to care about constant factors since $\mathbb{Z}(\alpha)$ does not support a gcd [5]

In principal, all algorithms have the same layout as Algorithm 1. However, a first fundamental difference is that the gcd in step 7 is computed over $\mathbf{R}_p = \mathbb{F}_p[t]/M_p$, where $M_p \in \mathbb{F}_p[x]$ is the modular image of the minimal polynomial M of α. In general M_p is not irreducible, thus in general $\mathbf{R}_p$ is not a field. Therefore, the computation in step 7 can fail, namely in that case that it needs to invert a zero divisor. If this happens, p is also considered as an unlucky prime and discarded.

We continue with details about the algorithm by Langemyr and McCallum [9], Encarnacion [4], and our hybrid approach combining the advantages of both algorithms.

The Algorithm of Langemyr and McCallum: In contrast to Algorithm 1, the main point is that the algorithm must take additional steps in order to ensure that the polynomial which is supposed to be recovered by the Chinese Remainder contains no denominators. In a first step, the input polynomials are normalized which removes superfluous constant factors and ensures that the leading coefficients are in $\mathbb{Z}$. This allows the computation of $\tilde{g}$ as in Algorithm 1 (step 7). However, in the presence of algebraic extensions, the multiplication with $\tilde{g}$ may not be enough to remove all denominators [14]. Therefore, $\tilde{G}$ is also multiplied by a multiplicative bound for these remaining denominators, namely $D = disc(M)$, the discriminant of the minimal polynomial of α. This finally ensures that $\tilde{G}$ is the modular image of a polynomial in $\mathbb{Z}(\alpha)[x]$, which can be recovered by the Chinese Remainder. For more details we refer to [11, 14, 8].

The Algorithm of Encarnacion: The algorithm does not multiply $\tilde{G}$ by any constant at all and the Chinese Remainder indeed tries to recover $monic(G) \in \mathbb{Q}(\alpha)[x]$ which is not possible. Instead, $G^\star$ is a polynomial in $\mathbb{Z}(\alpha)[x]$, where each coefficient is just in the same residue class as the corresponding coefficient of $monic(G)$. Therefore, the algorithm has an additional step that applies Wang's rational reconstruction [12, 13] to each coefficient in order to obtain $monic(G) \in \mathbb{Q}[x]$. Once the polynomial obtained in this step is stable, the verification (step 13) is applied.

The hybrid approach: For Langemyr and McCallum the weak point is that $gcd(f_1, f_2)D$ can be a very loose upper bound for the denominator of the gcd which causes the use of additional superfluous primes. Encarnacion's algorithm tries to avoid this, but has the overhead due to the additional rational reconstruction step which is performed in each round. In our benchmarks, see also Section 3, we observed that $gcd(f_1, f_2)$ is a good denominator bound in practice. Indeed, within all our examples the additional factor D was needed only once.

Our hybrid approach incorporates these observa-

tions in the sense that it modifies the algorithm by Langemyr and McCallum by using $gcd(f_1, f_2)$ as the denominator bound. This has the effect that the algorithm saves $O(log(D))$ rounds in almost all cases. However, for the unlucky case that D is indeed needed the algorithm would not terminate. Therefore, it uses the rational reconstruction as a fall back. More precisely, it calls Wang's algorithm if the fiftieth part of the accumulated time spent within the Chinese Remainder exceeds the time spent in the last call of Wang's algorithm. Hence, in practice our hybrid is as output sensitive as Encarnacion's algorithm but, de facto, without the extra costs of the rational reconstruction.

3 Benchmarks

Since our code is implemented within CGAL, we follow the *generic programming* paradigm using C++ templates and programming concepts such as traits classes and iterators. We introduced several traits classes to provide the functionality needed by the algorithms, for instance, providing the denominator bound required by Langemyr-McCallum. This abstracts from the actual coefficient type in use.

For the benchmarks we generated various families of 50 pairs of polynomials with fixed degree. Each pair is composed of three factors, the gcd and the two cofactors. All polynomials are random in the sense that their diced scalar coefficients have the desired bitsize. Within each family we always varied only one parameter, for instance, the bitsize of coefficients, or the degree of the gcd.

The benchmarks were measured on a Pentium(R) M processor 1.7 GHz with 512 KB cache under Linux and the GNU C++ compiler v3.4.6 with optimizations (`-O3`) and disabled assertions (`-DNDEBUG`). The used number type was `CORE::BigInt`.

3.1 Polynomials over the Integers

First we study the impact of a modification of the gcd and cofactor bitsize. For this purpose we generated polynomial pairs of degree 25 with gcd degree 1 and increasing bitsize of gcd and cofactors.

Generally we can say that Brown's algorithm performs far better than the old, non-modular implementation, hence we don't compare these two approaches. For a better quantification of the results we measured the same polynomials with the Computer Algebra Systems NTL and SINGULAR. Figure 1 shows, that the NTL implementation performs about twice as well as our generic implementation of Brown. Note, that there is another curve that also covers the time to convert our polynomials (i.e. the coefficients) to NTL polynomials and back. This curve is included for comparison with Singular, which uses NTL as well,

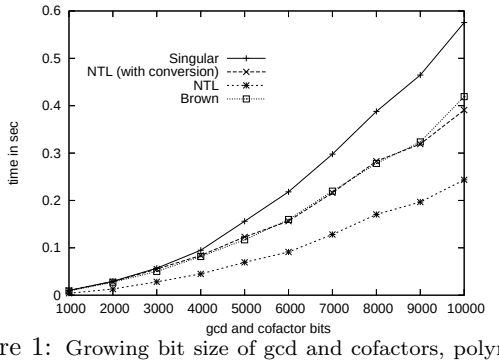

Figure 1: Growing bit size of gcd and cofactors, polynomial degree 25, gcd degree 1.

but who's conversion costs are apparently more expensive.

Furthermore, we generated polynomial pairs of degree 50 with 500 gcd bits and 5000 cofactor bits. The gcd degree ranges from 1 to 49. Figure 2 reveals a strange discontinuous behavior of the NTL: The first and the last 8 gcd degrees are computed faster than the others. Other test series show the same behavior. It seems that NTL uses two different approaches for the gcd computation. This has the consequence that our approach is even faster for moderate gcd degrees.

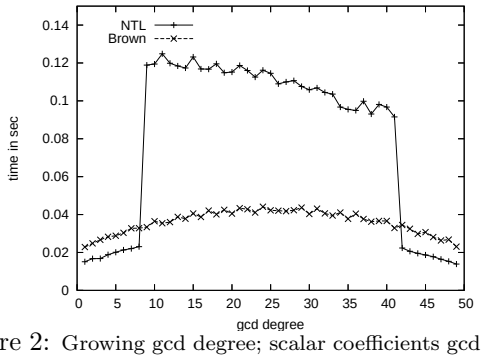

Figure 2: Growing gcd degree; scalar coefficients gcd 500 bit; cofactors 5000 bit; polynomial degree 50.

The behavior of Brown's algorithm can be explained as follows. Since degree and bit size of the polynomials are fixed, the time for the modular image is constant. With increasing degree of the gcd, the time spent in the Euclidean Algorithm decreases as it needs less steps whereas the Chinese Remainder has to recover more coefficients. These effects cancel out. The bow like form is due to the trial-division which is most expensive for moderate gcd degree.

3.2 Polynomials over Algebraic Extension Fields

To study the impact of a modification of the gcd bitsize we generated polynomial pairs of degree 10 with gcd degree 1 and 2000 cofactor bits. The Polynomials were defined over an algebraic extension of degree 2.

First of all, we can say that the hybrid approach performed far better than the non-modular implemen-

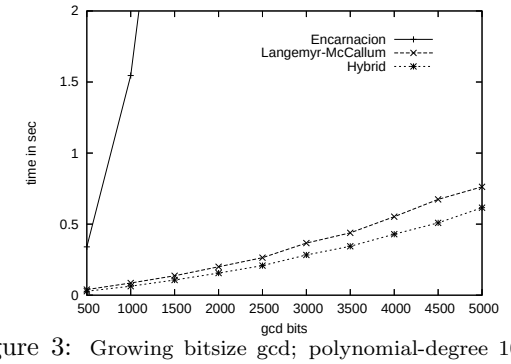

Figure 3: Growing bitsize gcd; polynomial-degree 10; gcd-degree 1; scalar coefficients cofactors 2000 bit.

tation. Hence, we don't consider the non-modular implementation.

Figure 3 shows that our hybrid approach performs better than the algorithms of Langemyr-McCallum (LM) and Encarnacion. Encarnacion's algorithm is not competitive, and for a sufficiently large bit size even slower than the old, non-modular implementation. This is due to the considerable runtime of Wang's rational reconstruction algorithm. For a higher bit size, the disadvantage of the LM approach due to the multiplication with the superfluous denominator bound becomes evident as well. The SINGULAR algorithm is the least successful, for a 500 bit gcd it needs already 128 sec. Hence, we refrained from including it in Figure 3.

A detailed decomposition of the total time spent in the hybrid algorithm is given in Figure 4. With growing gcd bits the algorithm needs more primes to reconstruct the coefficients. Additionally every call of the modular image, the Chinese remainder and the test division gets more expensive. The time for normalization and computing the denominator bound is slightly increasing, too.

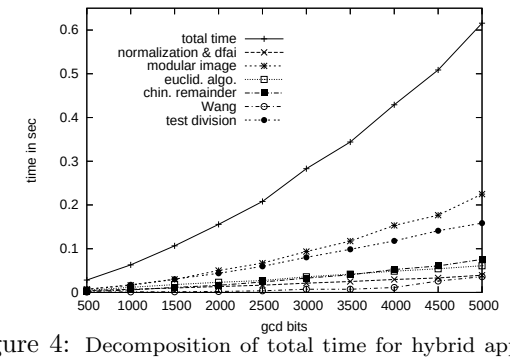

Figure 4: Decomposition of total time for hybrid approach: Growing bitsize gcd; polynomial-degree 10; gcd-degree 1; scalar coefficients cofactors 2000 bit.

4 Conclusions and Further Work

We have presented an open source implementation and comparison of several variants of gcd algorithms for algebraic extensions. Our benchmarks indicate that the hybrid approach has considerable advantages compared to the other implementations.

It is obvious that we should aim for a multivariate gcd in the spirit of [3]. As this is independent from the coefficient type, this should be straight forward.

We also expect some minor improvements for Encarnacion's algorithm, since the current implementation of Wang's algorithm does not take advantage of the known multiplicative denominator bound, as it is indicated in [10]. The algorithms are implement in CGAL, and part of the new Polynomial package of CGAL release 3.4.

References

[1] M. H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[2] E. Berberich, M. Caroli, and N. Wolpert. Exact computation of arrangements of rotated conics. In *Proc. EWCG'07*, pages 231–234. Technische Universität Graz, 2007.

[3] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18(4):478–504, 1971.

[4] M. J. Encarnacin. Computing gcds of polynomials over algebraic number fields. *J. on Symbolic Computation*, 20(3):299–313, 1995.

[5] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[6] M. Hemmer. *Exact Computation of the Adjacency Graph of an Arrangement of Quadrics*. Ph.D. thesis, Johannes Gutenberg-Universität Mainz, 2007.

[7] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 2nd edition, 1981.

[8] S. Landau. Factoring polynomials over algebraic number fields. *J. on Computing*, 14:184–195, 1985.

[9] L. Langemyr and S. McCallum. The computation of polynomial greatest common divisors over an algebraic number field. *J. on Symbolic Computation*, 8(5):429–448, 1989.

[10] M. Monagan. An almost optimal algorithm for rational reconstruction. In *Proc. ISSAC'04*, pages 243–249. ACM Press, 2004.

[11] P. S. Wang. Factoring multivariate polynomials over algebraic number fields. *Math. Comb.*, 30:1215–1231, 1978.

[12] P. S. Wang. A p-adic algorithm for univariate partial fractions. *Proc. SYMSAC '81*, pages 212–217, 1981.

[13] P. S. Wang, M. Guy, and J. Davenport. P-adic reconstruction of rational numbers. *SIGSAM Bulletin*, pages 2–3, 1982.

[14] P. J. Weinberger and L. P. Rothschild. Factoring polynomials over algebraic number fields. *J. Transactions on Mathematical Software*, 2(4):335–350, 1976.