In-place Arithmetic for Univariate Polynomials over an Algebraic Number Field

Seyed Mohammad Mahdi Javadi[1*], Michael Monagan[2*]

[1] School of Computing Science, Simon Fraser University,
Burnaby, B.C. Canada.
`sjavadi@cs.sfu.ca`

[2] Department of Mathematics, Simon Fraser University,
Burnaby, B.C. Canada.
`mmonagan@cecm.sfu.ca`

Abstract

We present a C library of *in-place* subroutines for univariate polynomial multiplication, division and GCD over L_p where L_p is an algebraic number field L with multiple field extensions reduced modulo a machine prime p. We assume elements of L_p and L are represented using a recursive dense representation. The main feature of our algorithms is that we eliminate the storage management overhead which is significant compared to the cost of arithmetic in $\mathbb{Z}_p$ by pre-allocating the exact amount of storage needed for both the output and working storage. We give an analysis for the working storage needed for each in-place algorithm and provide benchmarks demonstrating the efficiency of our library. This work improves the performance of polynomial GCD computation over algebraic number fields.

1 Introduction

In 2002, van Hoeij and Monagan in [12] presented an algorithm for computing the monic GCD $g(x)$ of two polynomials $f_1(x)$ and $f_2(x)$ in $L[x]$ where $L = \mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_k)$ is an algebraic number field. The algorithm is a *modular* GCD algorithm. It computes the GCD of f_1 and f_2 modulo a sequence of primes $p_1, p_2, \ldots, p_l$ using the monic Euclidean algorithm in $L_p[x]$ and it reconstructs the rational numbers in $g(x)$ using Chinese remaindering and rational number reconstruction. The algorithm is a generalization of earlier work of Langymyr and MaCallum [5], and Encarnación [2] to treat the case where L has multiple extensions ($k > 1$). It can be generalized to multivariate polynomials in $L[x_1, x_2, \ldots, x_n]$ using evaluation and interpolation (see [13, 4]).

Monagan implemented the algorithm in Maple in 2001 and in Magma in 2003 using the *recursive dense* polynomial representation to represent elements of L, L_p, $L[x_1, \ldots, x_n]$ and $L_p[x_1, \ldots, x_n]$. For Maple, Monagan developed a Maple package called RECDEN for doing polynomial arithmetic in $L[x_1, \ldots, x_n]$ and $L_p[x_1, \ldots, x_n]$ using this representation. This package was subsequently implemented in C in the Maple kernel in 2004. For Magma, Monagan used the `UnivariatePolynomial` and `quo` constructors to build a recursive dense representation.

The recursive dense representation is used in the PARI (see [10]) computer algebra system as the default representation for multivariate polynomials. It was chosen because it is known to be generally more efficient than the distributed and recursive representations for sparse polynomials. See for example the comparison by Fateman in [3]. And since efficiency

[*]Correspondence to: CECM Lab, Simon Fraser University, 8888 University Drive, Burnaby, BC, Canada. Tel: +1.778.782.5617

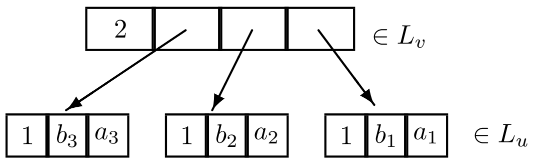
in the recursive dense representation improves for dense polynomials, and elements of L are often dense, it should be a good choice for implementing arithmetic in L and also L_p. However, we observed that arithmetic in L_p was very slow when α_1 has low degree. Since this often occurs in practical applications, and since over 90% of a GCD computation in $L[x]$ is typically spent in the Euclidean algorithm in $L_p[x]$, we sought to improve the efficiency of the arithmetic in L_p. One of the reasons why this happens is because the cost of storage management in the recursive dense representation can be higher than the cost of the arithmetic being done in $\mathbb{Z}_p$. We explain why this is the case with an example.

Example 1. Let $L = \mathbb{Q}(\alpha_1,\ \alpha_2)$ where $\alpha_1 = \sqrt{2}$ and $\alpha_2 = \sqrt[3]{1/5 + \alpha_1}$. L is an algebraic number field of degree $d = 6$ over $\mathbb{Q}$. We represent elements of L as polynomials in $\mathbb{Q}[u][v]$ and we do arithmetic in L modulo the ideal $I = \langle m_1(u), m_2(v,u)\rangle$ where $m_1(u) = u^2 - 2$ and $m_2(v,u) = v^3 - u - 1/5$ are the minimal polynomials for α_1 and, respectively, α_2.

To implement the modular GCD algorithm one uses *machine primes*, that is, the largest available primes that fit in the word of the computer so that arithmetic in $\mathbb{Z}_p$ can be done by the computer's hardware. After choosing the next machine prime p, we build the ring $L_p[x]$ where $L_p = L \bmod p$, iteratively, as follows; first we build the residue ring $L_u = \mathbb{Z}_p[u]/\langle u^2 - 2 \bmod p\rangle$. We use a dense array of machine integers to represent elements of L_u. Then we build $L_v = L_u[v]/\langle v^3 - u - 1/5 \bmod p\rangle$ and finally the polynomial ring $L_p[x]$. In the recursive dense representation we represent elements of L_v as dense arrays of pointers to elements of L_u. So a general element of L_v, which looks like

$$(a_1u + b_1)v^2 + (a_2u + b_2)v + (a_3u + b_3),$$

would be stored as follows where the degree of each element is explicitly stored.



When the monic Euclidean algorithm is executed in $L_p[x]$, it will do many multiplications and additions of elements in L_v, each of which will do many in L_u. This results in many calls to the storage manager to allocate small arrays for intermediate and final results in L_u and L_v and rapidly produces a lot of small pieces of garbage. Consider one such multiplication in L_u

$$(au + b)(cu + d) \bmod u^2 - 2.$$

The algorithms compute the product $P = acu^2 + (ad + bc)u + bd$ and then divide P by $u^2 - 2$ to get the remainder $R = (ad + bc)u + (bd + 2ac)$. They allocate arrays to store the polynomials P and R. We have observed that, even though the storage manager is not inefficient, the cost of these storage allocations and the other overhead for arithmetic in $\mathbb{Z}_p[u]/\langle u^2 - 2\rangle$ overwhelms the cost of the actual integer arithmetic in $\mathbb{Z}_p$ needed to compute $(ad + bc) \bmod p$ and $(bd + 2ac) \bmod p$.

Our main contribution is a library of *in-place* algorithms for arithmetic in L_p and $L_p[x]$ where L_p has one or more extensions. The main idea is to eliminate all calls to the storage manager by pre-allocating one large piece of working storage, and re-using parts of it in a computation.

In Section 2 we describe the recursive dense polynomial representation for elements of $L_p[x]$. In Section 3 we present algorithms for multiplication and inversion in L_p and multiplication, division with remainder and GCD in $L_p[x]$ which are given one array of storage in which to write the output and one additional array W of working storage for intermediate results. In Section 4 we give formulae for determining the size of W needed for each algorithm. In each case the amount of working storage is linear in d the degree of L. We have implemented our algorithms in the C language in a library which includes also algorithms for addition, subtraction, and other utility routines. The library is available at `http://www.cecm.sfu.ca/~sjavadi/inplace_web.c`. In Section 5 we present benchmarks demonstrating its efficiency by comparing our algorithms with the Magma ([1]) computer algebra system and we explain how to avoid most of the integer divisions by p when doing arithmetic in $\mathbb{Z}_p$ because this significantly affects overall performance.

1.1 Related Work

We have also developed an interface to Maple so that we can implement the dense GCD algorithm of van Hoeij and Monagan [13] and the sparse algorithm of Javadi and Monagan in [4] efficiently. These algorithms compute GCDs of polynomials in $K[x_1, x_2, \ldots, x_n]$ over an algebraic function field K in parameters $t_1, t_2, \ldots, t_k$ by evaluating first the parameters then all variables except x_1 and using rational function interpolation to recover the GCD. This results in many (hundreds) of GCD computations in $L_p[x_1]$. In many applications, K has field extensions of low degree, often quadratic or cubic.

In [6], Xin, Moreno Maza and Schost develop asymptotically fast algorithms for multiplication in L_p based on the FFT and use their algorithms to implement the Euclidean algorithm in $L_p[x]$ for comparison with Magma and Maple. The authors obtain a speedup for L of sufficiently large degree d. Our results here are complementary. Our benchmarks demonstrate greatest improvement when L has low degree. – cases occurring frequently in practice.

An in-place algorithm for long integer multiplication using Karatsuba's algorithm was developed by Maeder in [7]. In-place algorithms for polynomial arithmetic were developed by Monagan in [8] for computation in the ring $\mathbb{Z}_m[x]$ where $m = p^k$ is a multi-precision integer to improve the performance of quadratic Hensel lifting for polynomial factorization in $\mathbb{Z}[x]$.

2 Polynomial Representation

Let $\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_r)$ be our number field L. We build L as follows. For $1 \le i \le r$, let $m_i(z_1, \ldots, z_i) \in \mathbb{Q}[z_1, \ldots, z_i]$ be the minimal polynomial for α_i, monic and irreducible over $\mathbb{Q}[z_1, \ldots, z_{i-1}]/\langle m_1, \ldots, m_{i-1}\rangle$. Let $d_i = \deg_{z_i}(m_i)$. We assume $d_i \ge 2$. Let $L = \mathbb{Q}[z_1, \ldots, z_r]/\langle m_1, \ldots, m_r\rangle$. So L is an algebraic number field of degree $d = \prod d_i$ over $\mathbb{Q}$. For a prime p for which the rational coefficients of m_i exist modulo p, let $R_i = \mathbb{Z}_p[z_1, \ldots, z_i]/\langle \bar{m}_1, \ldots, \bar{m}_i\rangle$ where $\bar{m}_i = m_i \bmod p$ and let $R = R_r = L \bmod p$. We use the following recursive dense representation for elements of R and polynomials in $R[x]$ for our algorithms. We view an element of R_{i+1} as a polynomial with degree at most $d_{i+1} - 1$ with coefficients in R_i.

To represent a non-zero element $\beta_1 = a_0 + a_1 z_1 + \cdots + a_{d_1-1} z_1^{d_1-1} \in R_1$ we use an array A_1 of size $S_1 = d_1 + 1$ indexed from 0 to d_1, of integers (modulo p) to store β_1. We store

$A_1[0] = \deg_{z_1}(\alpha_1)$ and, for $0 \le i < d_1 : A_1[i+1] = a_i$. Note that if $\deg_{z_1}(\alpha_1) = \bar{d} < d_1 - 1$ then for $\bar{d}+1 < j \le d_1$, $A_1[j] = 0$. To represent the zero element of R_1 we use $A[0] = -1$.

Now suppose we want to represent an element $\beta_2 = b_0 + b_1 z_2 + \cdots + b_{d_2-1} z_2^{d_2-1} \in R_2$ where $b_i \in R_1$ using an array A_2 of size $S_2 = d_2 S_1 + 1 = d_2(d_1+1)+1$. We store $A_2[0] = \deg_{z_2}(\beta_2)$ and for $0 \le i < d_2$

$$A_2[i(d_1+1)+1 \ldots (i+1)(d_1+1)] = B_i[0 \ldots d_1]$$

where B_i is the array which represents $b_i \in R_1$. Again if $\beta_2 = 0$ we store $A_2[0] = -1$.

Similarly, we recursively represent $\beta_r = c_0 + c_1 z_r + \cdots + c_{d_r-1} z_r^{d_r-1} \in R_r$ based on the representation of $c_i \in R_{r-1}$. Let $S_r = d_r S_{r-1} + 1$ and suppose A_r is an array of size S_r such that $A_r[0] = \deg_{z_r}(\beta_r)$ and for $0 \le i < d_r$

$$A_r[i(d_{r-1})+1 \ldots (i+1)(d_{r-1}+1)] = C_i[0 \ldots S_{r-1}-1].$$

Remark 2. We store the degrees of the elements of R_i in $A_i[0]$ simply to avoid re-computing them.

We have

$$\prod_{i=1}^{r} d_i < S_r < \prod_{i=1}^{r} (d_i+1), S_r \in O(\prod_{i=1}^{r} d_i).$$

Now suppose we use the array C to represent a polynomial $f \in R_i[x]$ of degree d_x in the same way. Each coefficient of f in x is an element of R_i which needs an array of size S_i, hence C must be of size

$$P(d_x, R_i) = (d_x+1)S_i + 1.$$

Example 3. Let $r = 2$ and $p = 17$. Let

$$\begin{aligned}
&\bar{m}_1 = z_1^3 + 3,\\
&\bar{m}_2 = z_2^2 + 5z_1z_2 + 4z_2 + 7z_1^2 + 3z_1 + 6, \quad \text{and}\\
&\quad f = 3 + 4z_1 + (5+6z_1)z_2 + (7 + 8z_1 + 9z_1^2 + (10z_1 + 11z_1^2)z_2)x + 12x^2.
\end{aligned}$$

The representation for f is

$$C = \begin{array}{|c|}\hline 2\\\hline\end{array}\underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 1&1&3&4&0&1&5&6&0\\\hline\end{array}}_{3+4z_1+(5+6z_1)z_2}\begin{array}{|c|c|c|c|c|}\hline 1&2&7&8&9\\\hline\end{array}\underbrace{\begin{array}{|c|c|c|c|}\hline 2&0&10&11\\\hline\end{array}}_{10z_1+11z_1^2}\begin{array}{|c|c|c|c|c|c|c|c|c|}\hline 0&0&12&0&0&-1&0&0&0\\\hline\end{array}$$

Here $d_x = 2, d_1 = 3, d_2 = 2, S_1 = d_1 + 1 = 4, S_2 = d_2 S_1 + 1 = 9$ and the size of the array A is $P(d_x, R_2) = (d_x+1)S_2 + 1 = 28$.

We also need to represent the minimal polynomial $\bar{m}_i$. Let $\bar{m}_i = a_0 + a_1 z_i + \ldots a_{d_i} z_i^{d_i}$ where $a_j \in R_{i-1}$. We need an array of size S_{i-1} to represent a_j so to represent $\bar{m}_i$ in the same way we described above, we need an array of size $\bar{S}_i = 1 + (d_i+1)S_{i-1} = d_i S_{i-1} + 1 + S_{i-1} = S_i + S_{i-1}$. We define $S_0 = 1$.

We represent the set of minimal polynomials $\{\bar{m}_1, \ldots, \bar{m}_r\}$ as an Array E of size $\sum_{i=1}^{r} \bar{S}_i = \sum_{i=1}^{r} (S_i + S_{i-1}) = 1 + S_r + 2\sum_{i=1}^{r-1} S_i$ such that $E[M_i \ldots M_{i+1} - 1]$ represents m_{r-i} where $M_0 = 0$ and $M_i = \sum_{i=r-i+1}^{r} \bar{S}_i$. The minimal polynomials in Example 3 will be represented in the following figure where $E[0 \ldots 12]$ represents $\bar{m}_2$ and $E[13 \ldots 17]$ represents $\bar{m}_1$.

$$E = \underbrace{\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 2&2&6&3&7&1&4&5&0&0&1&0&0\\\hline\end{array}}_{\bar{m}_2}\underbrace{\begin{array}{|c|c|c|c|c|}\hline 3&3&0&0&1\\\hline\end{array}}_{\bar{m}_1}$$

3 In-place Algorithms

In this section we design efficient in-place algorithms for multiplication, division and GCD computation of two univariate polynomials over R. We will also give an in-place algorithm for computing the inverse of an element $\alpha \in R$, if it exists. This is needed for making a polynomial monic for the monic Euclidean algorithm in $R[x]$. We assume the following utility operations are implemented.

- IP_ADD(N, A, B) and IP_SUB(N, A, B) are used for in-place addition and subtraction of two polynomials $a, b \in R_N[x]$ represented in arrays A and B.
- IP_MUL_NO_EXT is used for multiplication of two polynomials over $\mathbb{Z}_p$. A description of this algorithm is given in Section 5.1.
- IP_REM_NO_EXT is used for computing the quotient and the remainder of dividing two polynomials over $\mathbb{Z}_p$.
- IP_INV_NO_EXT is used for computing the inverse of an element in $\mathbb{Z}_p[z]$ modulo a minimal polynomial $m \in \mathbb{Z}_p[z]$.
- IP_GCD_NO_EXT is used for computing the GCD of two univariate polynomials over $\mathbb{Z}_p$ (the Euclidean algorithm, See [8]).

3.1 In-place Multiplication

Suppose we have $a, b \in R[x]$ where $R = R_{r-1}[z_r]/\langle m_r(z_r)\rangle$. Let $a = \sum_{i=0}^{d_a} a_i x^i$ and $b = \sum_{i=0}^{d_b} b_i x^i$ where $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$ and Let $c = a \times b = \sum_{i=0}^{d_c} c_i x^i$ where $d_c = \deg_x(c) = d_a + d_b$. To reduce the number of divisions by $m_r(z_r)$ when multiplying $a \times b$, we use the Cauchy product rule to compute c_k as suggested in [8], that is,

$$c_k = \left[\sum_{i=\max(0,k-d_b)}^{\min(k,d_a)} a_i \times b_{k-i} \right] \mod m_r(z_r).$$

Thus the number of multiplications in $R_{r-1}[z_r]$ (in line 16) is $(d_a + 1) \times (d_b + 1)$ and the number of divisions in $R_{r-1}[z_r]$ (in line 20) is $d_a + d_b + 1$. Asymptotically, this saves about half the work.

Algorithm IP_MUL: In-place Multiplication

Input:
- N the number of field extensions.
- Arrays $A[0 \dots \bar{a}]$ and $B[0 \dots \bar{b}]$ representing univariate polynomials $a, b \in R_N[x]$ ($R_N = \mathbb{Z}_p[z_1, \dots, z_N]/\langle \bar{m}_1, \dots, \bar{m}_N \rangle$). Note that $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$ where $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$.
- Array $C[0 \dots \bar{c}]$: Space needed for storing $c = a \times b = \sum_{i=0}^{d_c} c_i x^i$ where $\bar{c} = P(\deg_x(a) + \deg_x(b), R_N) - 1$.
- $E[0 \dots e_N]$: representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
- $W[0 \dots w_N]$: *the working storage* for the intermediate operations.

Output: For $0 \leq k \leq d_c$, c_k will be computed and stored in $C[k]$.

```
1: Set $d_a := A[0]$ and $d_b := B[0]$.
2: if $d_a = -1$ or $d_b = -1$ then
3:     Set $C[0] := -1$.
4:     return
5: end if
6: if $N = 0$ then
7:     Call IP_MUL_NO_EXT on inputs $A$, $B$ and $C$ and return.
8: end if
9: Let $M = E[0 \ldots \bar{S}_N - 1]$ and $E' = E[\bar{S}_N \ldots e_N]$ ($M$ points to $\bar{m}_N$ in $E[0 \ldots e_N]$).
10: Let $T_1 = W[0 \ldots t-1]$ and $T_2 = W[t \ldots 2t-1]$ and $W' = W[2t \ldots w_N]$ where $t = P(2d_N - 2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
11: Set $d_c := d_a + d_b$ and $s_c := 1$.
12: for $k$ from 0 to $d_c$ do
13:     Set $s_a := 1 + iS_N$ and $s_b := 1 + (k-i)S_N$.
14:     Set $T_1[0] := -1$ ($T_1 = 0$).
15:     for $i$ from $\max(0, k - d_b)$ to $\min(k, d_a)$ do
16:         Call IP_MUL($N-1$, $A[s_a \ldots \bar{a}]$, $B[s_b \ldots \bar{b}], T_2, E', W'$).
17:         Call IP_ADD($N-1, T_1, T_2$) ($T_1 := T_1 + T_2$)
18:         Set $s_a := s_a + S_N$ and $s_b := s_b - S_N$.
19:     end for
20:     Call IP_REM($N-1, T_1, M, E', W'$). (Reduce $T_1$ modulo $M = \bar{m}_N$).
21:     Copy $C[sc \ldots \bar{c}]$ into $T_1$.
22: end for
23: Determine $\deg_x(a \times b)$: (There might be zero-divisors).
24: Set $i := d_c$ and $s_c := s_c - S_N$.
25: while $i \geq 0$ and $C[sc] = -1$ do
26:     Set $i := i - 1$ and $s_c := s_c - S_N$.
27: end while
28: Set $C[0] := i$.
```

The temporary variables T_1 and T_2 must be big enough to store the product of two coefficients in $a, b \in R_N[x]$. Coefficients of a and b are in $R_{N-1}[z_N]$ with degree (in z_N) at most $d_N - 1$. Hence these temporaries must be of size $P(d_N - 1 + d_N - 1, R_{N-1}) = P(2d_N - 2, R_{N-1})$.

3.2 In-place Division

The following algorithm divides a polynomial $a \in R_N[x]$ by a *monic* polynomial $b \in R_N[x]$. The remainder and the quotient of a divided by b will be stored in the array representing a hence a is destroyed by the algorithm. The division algorithm is organized differently from the normal long division algorithm which does $d_b \times (d_a - d_b + 1)$ multiplications and divisions in $R_{N-1}[z_r]$. The number of divisions by M in $R_{N-1}[z_r]$ in line 20 is reduced to $d_a + 1$ (see line 10). Asymptotically this saves half the work.

Algorithm IP_REM: In-place Remainder

Input:
- N the number of field extensions.
- Arrays $A[0 \ldots \bar{a}]$ and $B[0 \ldots \bar{b}]$ representing univariate polynomials $a, b \neq 0 \in R_N[x]$ ($R_N = \mathbb{Z}_p[z_1, \ldots, z_N] / \langle \bar{m}_1, \ldots, \bar{m}_N \rangle$) where $d_a = \deg_x(a) \geq d_a = \deg_x(b)$. Note b must be monic and $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$.
- $E[0 \ldots e_N]$: representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.

- $W[0 \dots w_N]$: *the working storage* for the intermediate operations.

```
Output: The remainder $\bar{R}$ of $a$ divided by $b$ will be stored in $A[0 \dots \bar{r}]$ where $\bar{r} = P(D, R_N) - 1$
    and $D = \deg_x(\bar{R}) \leq d_b - 1$. Also let $Q$ represent the quotient $\bar{Q}$ of $a$ divided by $b$. $Q[1 \dots \bar{q}]$
    will be stored in $A[1 + d_b S_N \dots \bar{a}]$ where $\bar{q} = P(d_a - d_b, R_N) - 1$. Note that we will no longer
    have the representation for $a$.
1: Set $d_a := A[0]$ and $d_b := B[0]$.
2: if $d_a < d_b$ then return.
3: if $N = 0$ then
4:     Call IP_REM_NO_EXT on inputs $A$ and $B$ and return.
5: end if
6: Set $D_q := d_a - d_b$ and $D_r := d_b - 1$.
7: Let $M = E[0 \dots \bar{S}_N - 1]$ and $E' = E[\bar{S}_N \dots e_N]$ ($M$ points to $\bar{m}_N$ in $E[0 \dots e_N]$).
8: Let $T_1 = W[0 \dots t-1]$ and $T_2 = W[t \dots 2t-1]$ and $W' = W[2t \dots w_N]$ where $t = P(2d_N -$
    $2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
9: Set $s_c := 1 + d_a S_N$
10: for $k = d_a$ to 0 by $-1$ do
11:     Copy $C[sc \dots \bar{c}]$ into $T_1$.
12:     Set $i := \max(0, k - D_q)$.
13:     Set $s_b := 1 + i S_N$
14:     Set $s_a := 1 + (k - i + d_b) S_N$
15:     while $i \leq \min(D_r, k)$ do
16:         Call IP_MUL($N - 1, A[s_a \dots \bar{a}]$, $B[s_b \dots \bar{b}]$, $T_2$, $E', W'$).
17:         Call IP_SUB($N - 1, T_1, T_2$) ($T_1 := T_1 - T_2$).
18:         Set $s_b := s_b + S_N$ and $s_a := s_a - S_N$.
19:     end while
20:     Call IP_REM($N - 1, T_1$, $M$, $E', W'$) (Reduce $T_1$ modulo $M = \bar{m}_N$).
21:     Copy $A[s_c \dots \bar{c}]$ into $T_1$.
22:     Set $s_c := s_c - S_N$.
23: end for
24: Set $i := D_r$ and $s_c := 1 + D_r S_N$.
25: while $i \geq 0$ and $A[s_c] = -1$ do
26:     Set $i := i - 1$ and $s_c := s_c - S_N$.
27: end while
28: Set $A[0] := i$.
```

Let arrays A and B represent polynomials a and b respectively. Let $d_a = \deg_x(a)$ and $d_b = \deg_x(b)$. Array A has enough space to store $d_a + 1$ *coefficients* in R_N plus one unit of storage to store d_a. Hence the total storage is $(d_a + 1)S_N + 1$. The remainder $\bar{R}$ is of degree at most $d_b - 1$ in x, i.e. $\bar{R}$ needs storage for d_b coefficients in R_N and one unit for the degree. Similarly the quotient $\bar{Q}$ is of degree $d_a - d_b$, hence needs storage for $d_a - d_b + 1$ coefficients and one unit for the degree. This the remainder and the quotient together need $d_b S_N + 1 + (d_a - d_b + 1)S_N + 1 = (d_a + 1)S_N + 2$. This means we are one unit of storage short if we want to store both $\bar{R}$ and $\bar{Q}$ in A. This is because this time we are storing two degrees for $\bar{Q}$ and $\bar{R}$. Our solution is that we will not store the degree of $\bar{Q}$. Any algorithm that calls IP_REM and needs both the quotient and the remainder must use $\deg_x(a) - \deg_x(b)$ for the degree of $\bar{Q}$.

After applying this algorithm the remainder $\bar{R}$ will be stored in $A[0 \dots d_b S_N]$ and the quotient $\bar{Q}$ minus the degree will be stored in $A[d_b S_N \dots (d_a + 1)S_N]$. Similar to IP_MUL, the remainder operation in line 20 has been moved to outside of the main loop to let the values accumulate in T_1.

3.3 Computing (In-place) the inverse of an element in R_N

In this algorithm we assume the following in-place functions:

- IP_SCAL_MUL(N, A, C, E, W): This is used for multiplying a polynomial $a \in R_N[x]$ (represented by array A) by a scalar $c \in R_N$ (represented by array C). The algorithm will multiply every coefficient of a in x by c and reduce the result modulo the minimal polynomials. It can easily be implemented using IP_MUL and IP_REM

- IP_LIN(N, C, A, B, E, W): On inputs $a, b, c \in R_N[x]$ (represented with arrays A, B and C respectively), the algorithm will compute (*in-place*) $c := a - bc$.

The algorithm computes the inverse of an element a in R_N. If the element is not invertible, then the Euclidean algorithm will compute a proper divisor of some minimal polynomial $m_i(z_i)$, a zero divisor in R_i. The algorithm will store that zero-divisor in the space provided for the inverse and return the index i of the minimal polynomial which is reducible and has caused the zero-divisor.

Algorithm IP_INV: In-place inverse of an element in R_N

Input:
- N the number of field extensions.
- Array $A[0 \dots \bar{a}]$ representing the univariate polynomial $a \in R_N$ Note that $N \geq 1$ and $\bar{a} = S_N - 1$.
- Array $I[0 \dots \bar{i}]$: Space needed for storing the inverse $a^{-1} \in R_N$. Note that $\bar{i} = S_N - 1$.
- $E[0 \dots e_N]$: representing the set of minimal polynomials. Note that $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
- $W[0 \dots w_N]$: *the working storage* for the intermediate operations.

Output: The inverse of a (or a zero-divisor, if there exists one) will be computed and stored in I. If there is a zero-divisor, the algorithm will return the index k where $\bar{m}_k$ is the reducible minimal polynomial, otherwise it will return 0.

```
1: Let $M = E[0 \dots \bar{S}_N - 1]$ and $E' = E[\bar{S}_N \dots e_N]$ ($M = \bar{m}_N$).
2: **if** $N = 1$ **then**
3:     Call IP_INV_NO_EXT on inputs $A$, $I, E, M$ and $W$ and **return.**
4: **end if**
5: **if** $A[i] = 0$, for all $0 \leq i < N$ and $A[N] = 1$ ( *Test if* $a = 1$) **then**
6:     Copy $A$ into $I$ and **return 0.**
7: **end if**
8: Let $r_1 = W[0 \dots t-1], r_2 = W[t \dots 2t-1], s_1 = I, s_2 = W[2t \dots 3t-1], T = W[3t \dots 4t-1]$
   and $W' = W[4t \dots w_N]$ where $t = P(d_N, R_{N-1}) - 1 = \bar{S}_N - 1$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
9: Copy $A$ and $M$ into $r_1$ and $r_2$ respectively.
10: Set $s_2[0] := -1$ ($s_2$ *represents 0*).
11: Let $Z \in \mathbb{Z}$ := IP_INV($N-1, A[D_a S_{N-1} + 1 \dots \bar{a}], T, E', W'$) where $D_a = A[0] = \deg_{z_N}(a)$.
    ($A[D_a S_{N-1} + 1 \dots \bar{a}]$ *represents* $l = lc_{z_N}(a)$ *and* $T$ *represents* $l^{-1}$*, the inverse of the leading coefficient*).
12: **if** $Z > 0$ **then**
13:     Copy $T$ into $I$. ($I$ *will contain the zero-divisor*).
14:     **return** $Z$ ($\bar{m}_Z$ *is reducible and there is a zero-divisor*).
15: **end if**
16: Copy $T$ into $s_1$.
17: Call IP_SCAL_MUL($N, r_1, T, E', W'$) ($r_1$ *is made monic*).
18: **while** $r_2[0] \neq -1$ **do**
19:     Let $Z \in \mathbb{Z}$ := IP_INV($N-1, r_2[D_{r_2} S_{N-1} + 1 \dots \bar{a}], T, E', W'$) where $D_{r_2} = r_2[0] = \deg_{z_N}(r_2)$.
```

```
20:     if $Z > 0$ then
21:         Copy $T$ into $I$. ($I$ will contain the zero-divisor).
22:         return $Z$ ($\bar{m}_Z$ is reducible and there is a zero-divisor).
23:     end if
24:     Call IP_SCAL_MUL($N, r_2, T, E', W'$) ($r_2$ is made monic).
25:     Call IP_SCAL_MUL($N, s_2, T, E', W'$).
26:     Set $D_q := r_1[0] - r_2[0]$. If $D_q < 0$ then set $D_q := -1$.
27:     Call IP_REM($N, r_1, r_2, E', W'$).
28:     Swap the arrays $r_1$ and $r_2$. (Interchange only the pointers).
29:     Set $t_1 := r_2[r_1[0]S_{N-1}]$.
30:     Set $r_2[r_1[0]S_{N-1}] := D_q$.
31:     Call IP_LIN($N, s_1, q, s_2, E', W'$) where $q = r_2[r_1[0]S_{N-1} \dots \bar{a}]$. ($s_1 := s_1 - qs_2$.)
32:     Set $r_2[r_1[0]S_{N-1}] := t_1$.
33:     Swap the arrays $s_1$ and $s_2$. (Interchange only the pointers).
34: end while
35: if $r_1[i] = 0$ for all $0 \le i < N$ and $r_1[N] = 1$ then
36:     Copy $s_1$ into $I$. ($r_1 = 1$ and $s_1$ is the inverse).
37:     return 0.
38: else
39:     Copy $r_1$ into $R$ ($r_1 \neq 1$ is the zero-divisor).
40:     return $N - 1$ ($\bar{m}_{N-1}$ is reducible).
41: end if
```

As discussed in Section 3.2, IP_REM will not store the degree of the quotient of a divided by b hence in line 30 we explicitly compute and set the degree of the quotient before passing it to the function IP_LIN as an argument. Here $r_2[r_1[0]S_{N-1} \dots \bar{a}]$ is the quotient of dividing r_1 by r_2 in line 27.

3.4 In-place GCD Computation

In the following algorithm we compute the GCD of $a, b \in R_N[x]$ using the monic Euclidean algorithm. This is the main subroutine used to compute univariate images of a GCD in $L[x]$ for the algorithm in [12] and images of a multivariate GCD over an algebraic function field for our algorithm in [4]. Note, since $m_i(z_i)$ may be reducible modulo p, R_N is is not necessarily a field, and therefore, the monic Euclidean algorithm may encounter a zero-divisor in R_N when calling subroutine IP_INV.

Algorithm IP_GCD: In-place GCD Computation

Input:
- N the number of field extensions.
- Arrays $A[0 \dots \bar{a}]$ and $B[0 \dots \bar{b}]$ representing univariate polynomials $a, b \neq 0 \in R_N[x]$ ($R_N = \mathbb{Z}_p[z_1, \dots, z_N] / \langle \bar{m}_1, \dots, \bar{m}_N \rangle$) where $d_a = \deg_x(a) \ge d_a = \deg_x(b)$ and $A, B \neq 0$. Note that b is monic and $\bar{a} = P(d_a, R_N) - 1$ and $\bar{b} = P(d_b, R_N) - 1$.
- $E[0 \dots e_N]$: representing the set of minimal polynomials where $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
- $W[0 \dots w_N]$: *the working storage* for the intermediate operations.

Output: If there exist a zero-divisor, it will be stored in A and the index of the reducible minimal polynomial will be returned. Otherwise the monic GCD $g = \gcd(a, b)$ will be stored in A and 0 will be returned.

```
1: if $N = 0$ then
2:     CALL IP_GCD_NO_EXT on inputs $A$ and $B$ and return 0.
3: end if
4: Set $d_a := A[0]$ and $d_b := B[0]$.
5: Let $r_1$ and $r_2$ point to $A$ and $B$ respectively.
```

```
 6: Let $I = W[0 \dots t-1]$ and $W' = W[t \dots w_N]$ where $t = \bar{S}_N - 1 = S_N + S_{N-1} - 1$.
 7: Let $Z$ be the output of IP_INV($N, r_1[1 + r_1[0]S_N \dots \bar{a}], I, E, W'$).
 8: if $Z > 0$ then
 9:     Copy $I$ into $A$. (A will contain the zero-divisor).
10:     return $Z$ ($\bar{m}_Z$ is reducible and there is a zero-divisor).
11: end if
12: Call IP_SCAL_MUL($N, r_1, I, E, W'$).
13: while $r_2[0] \neq -1$ do
14:     Let $Z$ be the output of IP_INV($N, r_2[1 + r_2[0]S_N \dots \bar{b}], I, E, W'$).
15:     if $Z > 0$ then
16:         Copy $I$ into $A$. (A will contain the zero-divisor).
17:         return $Z$ ($\bar{m}_Z$ is reducible and there is a zero-divisor).
18:     end if
19:     Call IP_SCAL_MUL($N, r_2, I, E, W'$).
20:     Call IP_REM($N, r_1, r_2, E, W'$).
21:     Swap $r_1$ and $r_2$ (interchange pointers).
22: end while
23: Copy $r_1$ into $A$.
24: return 0.
```

Similar to the algorithm IP_INV, if there exists a zero-divisor, i.e. the leading coefficient of one of the polynomials in the polynomial remainder sequence is not invertible, the algorithm will store the zero-divisor in the space provided for a. It will also return the index of the minimal polynomial which is reducible and has caused the zero-divisor.

4 Working Space

In this section we will determine recurrences for the exact amount of working storage w_N needed for each operation introduced in the previous section. Recall that $d_i = \deg_{z_i}(\bar{m}_i)$ is the degree of the ith minimal polynomial which we may assume is at least 2. Also S_i is the space needed to store an element in R_i and we have $S_{i+1} = d_{i+1}S_i + 1$ and $S_1 = d_1 + 1$.

Lemma 4. *$S_N > 2S_{N-1}$ for $N > 1$.*

Proof. We have $S_N = d_N S_{N-1} + 1$ where $d_N = \deg_{z_N}(\bar{m}_N)$. Since $d_N \geq 2$ we have $S_N \geq 2S_{N-1} + 1 \Rightarrow S_N > 2S_{N-1}$. □

Lemma 5. *$\sum_{i=1}^{N-1} S_i < S_N$ for $N > 1$.*

Proof. (by induction on N). For $N = 2$ we have $\sum_{i=1}^{1} S_i = S_1 < S_2$. For $N = k+1 \geq 2$ we have $\sum_{i=1}^{k} S_i = S_k + \sum_{i=1}^{k-1} S_i$. By induction we have $\sum_{i=1}^{k-1} S_i < S_k$ hence $\sum_{i=1}^{k} S_i < S_k + S_k = 2S_k$. Using Lemma 4 we have $2S_k < S_{k+1}$ hence $\sum_{i=1}^{k} S_i < 2S_k < S_{k+1}$ and the proof is complete. □

Corollary 6. *$\sum_{i=1}^{N} S_i < 2S_N$ for $N > 1$.*

Lemma 7. *$P(2d_N - 2, R_{N-1}) = 2S_N - S_{N-1} - 1$ for $N > 1$.*

Proof. We have $P(2d_N - 2, R_{N-1}) = (2d_N - 1)S_{N-1} + 1 = 2d_N S_{N-1} - S_{N-1} + 1 = 2(d_N S_{N-1} + 1) - S_{N-1} - 1 = 2S_N - S_{N-1} - 1$. □

4.1 Multiplication and Division Algorithms

Let $M(N)$ be the amount of working storage needed to multiply $a, b \in R_N[x]$ using the algorithm IP_MUL. Similarly let $Q(N)$ be the amount of working storage needed to divide a by b using the algorithm IP_REM. The working storage used in lines 10,16 and 20 of algorithm IP_MUL and lines 8,16 and 20 of algorithm IP_REM is

$$M(N) = 2P(2d_N - 2, R_{N-1}) + \max(M(N-1), Q(N-1)) \text{ and } \tag{1}$$

$$Q(N) = 2P(2d_N - 2, R_{N-1}) + \max(M(N-1), Q(N-1)). \tag{2}$$

Comparing equations (1) and (2) we see that $M(N) = Q(N)$ for any $N \geq 1$. Hence

$$M(N) = 2P(2d_N - 2, R_{N-1}) + M(N-1). \tag{3}$$

Simplifying (3) gives $M(N) = 2S_N - 2N + 2\sum_{i=1}^{N} S_i$. Using Corollary 6 we have the following:

Theorem 8. $M(N) = Q(N) = 2S_N - 2N + 2\sum_{i=1}^{N} S_i < 6S_N$.

Remark 9. When calling the algorithm IP_MUL to compute $c = a \times b$ where $a, b \in R[x]$, we should use a working storage array $W[0 \dots w_n]$ such that $w_n \geq M(N)$. Since $M(N) < 6S_N$, the working storage must be big enough to store only six coefficients in L_p. This is very small.

Let $C(N)$ and $L(N)$ denote the amount of working storage needed for operations IP_SCAL_MUL and IP_LIN. It is easy to show that $C(N) = M(N-1) + P(2d_N - 2, R_{N-1}) < M(N)$. Also we have $L(N) = M(N)$.

4.2 Inversion

Let $I(N)$ denote the amount of working storage needed to invert $c \in R_N$. In lines 8, 11, 17, 19, 24, 25, 27 and 31 of algorithm IP_INV we use the working storage. We have

$$I(N) = 4P(d_N, R_{N-1}) + \max(I(N-1), M(N-1), L(N-1), Q(N-1)). \tag{4}$$

But we have $M(N-1) = L(N-1) = Q(N-1)$, hence

$$I(N) = 4P(d_N, R_{N-1}) + \max(I(N-1), M(N-1)). \tag{5}$$

Lemma 10. For $N \geq 1$, we have $M(N) < I(N)$.

Proof. (by contradiction) Assume $M(N) \geq I(N)$. Using (5) we have

$$I(N) = 4P(d_N, R_{N-1}) + M(N-1).$$

On the other hand using (3) we have

$$M(N) = 2P(2d_N - 2, R_{N-1}) + M(N-1).$$

We assumed $I(N) \leq M(N)$ hence we have $4P(d_N, R_{N-1}) + M(N-1) \leq 2P(2d_N - 2, R_{N-1}) + M(N-1)$ thus $2P(d_N, R_{N-1}) \leq P(2d_N - 2, R_{N-1}) \Rightarrow 2S_N + 2S_{N-1} \leq 2S_N - S_{N-1} - 1$ which is a contradiction. Thus $I(N) > M(N)$. $\square$

Using Equation (4) and Lemma 10 we conclude that $I(N) = 4P(d_N, R_{N-1}) + I(N-1)$. Simplifying this yields:

Theorem 11.

$$I(N) = 4\sum_{i=1}^{N} P(d_i, R_{i-1}) = 4\sum_{i=1}^{N} S_i + S_{i-1} = 4S_N + 8\sum_{i=1}^{N-1} S_i.$$

Using Lemma 4 an upper bound for $I(N)$ is $I(N) < 4S_N + 8S_N = 12S_N$.

4.3 GCD Computation

Let $G(N)$ denote the amount of working storage needed to compute the GCD of $a, b \in R_N[x]$. In lines 6,7,12,14,19 and 20 of algorithm IP_GCD we use the working storage. We have

$$G(N) = \bar{S}_N + \max(I(N), C(N), Q(N)). \tag{6}$$

Lemma 10 states that $I(N) > M(N) = C(N) = Q(N)$ hence

$$G(N) = \bar{S}_N + I(N) = S_N + S_{N-1} + 4S_N + 8\sum_{i=1}^{N-1} S_i = 9S_N + S_{N-1} + 8\sum_{i=1}^{N-1} S_i.$$

Since $I(N) < 12S_N$, we have an upper bound on $G(N)$:

Theorem 12. $G(N) = S_N + S_{N-1} + I(N) < S_N + S_{N-1} + 12S_N < 14S_N$.

Remark 13. The constants 6, 12 and 14 appearing in Theorems 8, 11 and 12 respectively, are not the best possible. For example, one can reduce the constant 6 for algorithm IP_MUL if one also uses the space in the output array C for working storage. We did not do this because it complicates the description of the algorithm and yields no significant performance gain.

5 Benchmarks

We have compared our C library with the Magma (see [1]) computer algebra system. The results are reported in Table 1. For our benchmarks we used $p = 3037000453$, two field extensions with minimal polynomials $\bar{m}_1$ and $\bar{m}_2$ of varying degrees d_1 and d_2 but with $d = d_1 \times d_2 = 60$ constant so that we may compare the overhead for varying d_1. We choose three polynomials a, b, g of the same degree d_x in x with coefficients chosen from R at random. The data in the fifth and sixth columns are the times (in CPU seconds) for computing both $f_1 = a \times g$ and $f_2 = b \times g$ using IP_MUL and Magma version 2.15 respectively. Similarly, the data in the seventh and eighth columns are the times for computing both $\text{quo}(f_1, g)$ and $\text{quo}(f_2, g)$ using IP_REM and Magma respectively. Finally the data in the ninth and tenth columns are the times for computing $\gcd(f_1, f_2)$ using IP_GCD and Magma respectively. The data in the column labeled $\#f_i$ is the number of terms in f_1 and f_2.

The timings in Table 1 for *in-place* routines show that as the degree d_x doubles from 40 to 80, the time consistently goes up by a factor of 4 indicating that the underlying algorithms are all quadratic in d_x. This is not the case for Magma. The reason is that Magma uses a different algorithm for multiplication (and for division) which is not quadratic. We describe the algorithm used by Magma ([11]) briefly. Suppose we want to multiply two polynomials $a, b \in L_p[x]$. Magma first multiplies a and b as polynomials in $\mathbb{Z}[x, z_1, \ldots, z_r]$. It then reduces their product modulo the ideal $\langle m_1, \ldots, m_r, p\rangle$. To multiply in $\mathbb{Z}[x, z_1, \ldots, z_r]$, Magma evaluates each variable successively, beginning with z_r then ending with x, at integers $k_r, \ldots, k_1, k_0$ which are powers of the base of the integer representation which are sufficiently large so that that the product of the two polynomials $a(x, z_1, \ldots, z_r) \times b(x, z_1, \ldots, z_r)$ can be recovered from the product of the two (very) large integers $a(k_0, k_1, \ldots, k_r) \times b(k_0, k_1, \ldots, k_r)$. The reason to evaluate at a power of the integer base is so that evaluation and recovery can be done in linear time. In this way polynomial multiplication in $\mathbb{Z}[x, z_r, \ldots, z_1]$ is reduced to a single (very) large integer multiplication which is done using the FFT. This, note, may not be efficient if the polynomials $a(x, z_1, \ldots, z_r)$ and $b(x, z_1, \ldots, z_r)$ are sparse.

Table 1: First benchmark. Timings (in CPU seconds)

d_1	d_2	d_x	$\#f_i$	IP_MUL	MAG_MUL	IP_REM	MAG_REM	IP_GCD	MAG_GCD
2	30	40	2460	0.124	0.050	0.123	0.090	0.384	3.550
3	20	40	2460	0.108	0.058	0.106	0.100	0.340	3.860
4	15	40	2460	0.106	0.058	0.106	0.090	0.327	3.910
6	10	40	2460	0.106	0.118	0.105	0.130	0.328	7.820
10	6	40	2460	0.100	0.095	0.100	0.370	0.303	10.310
15	4	40	2460	0.097	0.053	0.095	0.150	0.283	4.580
20	3	40	2460	0.092	0.045	0.091	0.130	0.267	3.760
30	2	40	2460	0.087	0.037	0.087	0.100	0.242	3.050
2	30	80	4860	0.477	0.115	0.478	0.260	1.449	15.270
3	20	80	4860	0.407	0.130	0.409	0.270	1.304	16.690
4	15	80	4860	0.404	0.132	0.406	0.260	1.253	16.810
6	10	80	4860	0.398	0.247	0.400	0.340	1.234	33.650
10	6	80	4860	0.380	0.203	0.381	0.850	1.151	46.880
15	4	80	4860	0.365	0.119	0.364	0.390	1.081	19.920
20	3	80	4860	0.353	0.104	0.353	0.320	1.030	16.200
30	2	80	4860	0.336	0.086	0.337	0.250	0.932	12.530

The timings in Table 1 show that our in-place GCD algorithm is a factor of 10 or more times faster than Magma's GCD algorithm. Since both algorithms are using the Euclidean algorithm, this shows that our in-place algorithms for arithmetic in L_p are efficient. This is the gain we sought to achieve.

The reader can observe that as d_1 increases, the timings for IP_MUL decrease which shows that there is still some overhead for α_1 of low degree.

In Table 2 below, we repeated the first benchmark with $d_x = 80$ with the following change. This time the coefficients of a, b and g in x^{2k+1} for $k = 0 \ldots 39$ are 0, i.e. these polynomials only have terms with even degree in x. This is representative of real problems which are not completely dense.

Tables 2 and 1 show that IP_MUL is on average four times faster when a, b and g have half the number of terms demonstrating that the "recursive dense" representation is efficient for sparse problems. However this is not true for MAG_MUL which is not using the classic multiplication algorithm as discussed previously.

Table 2: Second benchmark. Timings (in CPU seconds)

d_1	d_2	d_x	IP_MUL	MAG_MUL	IP_REM	MAG_REM	IP_GCD	MAG_GCD
2	30	80	0.125	0.070	0.123	0.090	0.387	4.850
3	10	80	0.107	0.076	0.108	0.100	0.341	4.930
4	15	80	0.106	0.079	0.107	0.100	0.329	5.200
6	10	80	0.105	0.138	0.105	0.130	0.330	10.010
10	6	80	0.100	0.114	0.099	0.260	0.305	12.880
15	4	80	0.097	0.074	0.096	0.120	0.283	5.810
20	3	80	0.093	0.069	0.092	0.100	0.269	4.950
30	2	80	0.088	0.054	0.086	0.080	0.242	3.900

5.1 Optimizations in the implementation

In modular algorithms, multiplication in $\mathbb{Z}_p$ needs to be coded carefully. This is because hardware integer division (`%p` in C) is much slower than hardware integer multiplication. One can use Peter Montgomery's trick (see [9]) to replace all divisions by p by several cheaper operations for an overall gain of typically a factor of 2. Instead, we use the following scheme which replaces most divisions by p in the multiplication subroutine for $\mathbb{Z}_p[x]$ by at most one subtraction. We use a similar scheme for the division in $\mathbb{Z}_p[x]$. This makes GCD computation in $L_p[x]$ more efficient as well. We observed a gain of a factor of 5 on average for the GCD computations in our benchmarks.

The following C code explains the idea. Suppose we have two polynomials $a, b \in \mathbb{Z}_p[x]$ where $a = \sum_{i=0}^{d_a} a_i x^i$ and $b = \sum_{j=0}^{d_b} b_j x^j$ where $a_i, b_j \in \mathbb{Z}_p$. Suppose the coefficients a_i and b_i are stored in two Arrays A and B indexed from 0 to d_a and 0 to d_b respectively. We assume that p is a machine prime and elements of $\mathbb{Z}_p$ are stored as signed integers such that $-p^2 < a_i, b_i < p^2$ fits in a machine word. The following computes $c = a \times b = \sum_{k=0}^{d_a+d_b} c_k x^k$.

```
M = p*p;
d_c = d_a+d_b;
for( k=0; k<=d_c; k++ ) {
   t = 0;
   for( i=max(0,k-d_b); i <= min(k,d_a); i++ )
   {
      if( t<0 ); else t = t-M;
      t = t+A[i]*B[k-i];
   }
   t = t % p;
   if( t<0 ) t = t+p;
   C[k] = t;
}
```

The trick here is to put t in the range $-p^2 < t \leq 0$ by subtracting p^2 from it when it is positive so that we can add the product of two integers $0 \leq a_i, b_{k-i} < p$ to t without overflow. Thus the number of divisions by p is linear in d_c, the degree of the product. One can further reduce the number of divisions by p. In our implementation, when multiplying elements $a, b \in \mathbb{Z}_p[z][x]/\langle m(z)\rangle$ we multiply $a, b \in \mathbb{Z}_p[z][x]$ without division by p before dividing by $m(z)$.

Note that the statement `if( t<0 ); else t = t-M;` is done this way rather than the more obvious `if( t>0 ) t = t-M;` because it is faster. The reason is that $t < 0$ holds about 75% of the time and the code generated by the newer compilers is optimized for the case the condition of an if statement is true. If one codes the if statement using `if( t>0 ) t = t-M;` instead, we observe a loss of a factor of 2.6 on an Intel Core i7, 2.3 on an Intel Core 2 duo, and 2.2 on an AMD Opteron for the above code.

References

[1] Wieb Bosma, John J. Cannon, and Catherine Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3–4):235–266, 1997.

[2] Mark J. Encarnación. Computing gcds of polynomials over algebraic number fields. *J. Symb. Comp.*, 20(3):299–313, 1995.

[3] Richard Fateman. Comparing the speed of programs for sparse polynomial multiplication. *SIGSAM Bull.*, 37(1):4–15, 2003.

[4] Seyed Mohammad Mahdi Javadi and Michael Monagan. A sparse modular gcd algorithm for polynomials over algebraic function fields. In *Proceedings of ISSAC '07*, pages 187–194. ACM, 2007.

[5] Lars Langemyr and Scott McCallum. The computation of polynomial greatest common divisors over an algebraic number field. *J. Symb. Comp.*, 8(5):429–448, 1989.

[6] Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: from theory to practice. In *Proceedings of ISSAC '07*, pages 269–276. ACM, 2007.

[7] Roman Maeder. Storage allocation for the karatsuba integer multipliation algorithm. In *Proceedings of Disco '93*, pages 59–65, London, UK, 1993. Springer-Verlag.

[8] Michael B. Monagan. In-place arithmetic for polynominals over $\mathbb{Z}_n$. In *Proceedings of DISCO '92*, pages 22–34. Springer-Verlag, 1993.

[9] Peter Montgomery. Modular multiplication without trial division. *Math. Comp.*, 44(70):519–521, 1985.

[10] *PARI/GP, version 2.3.4*. Bordeaux, 2008. `http://pari.math.u-bordeaux.fr/`.

[11] Allan Steel. Multiplication in $L_p[x]$ in Magma. private communication, 2009.

[12] Mark van Hoeij and Michael Monagan. A modular gcd algorithm over number fields presented with multiple extensions. In *Proceedings of ISSAC '02*, pages 109–116. ACM Press, 2002.

[13] Mark van Hoeij and Michael Monagan. Algorithms for polynomial gcd computation over algebraic function fields. In *Proceedings of ISSAC '04*, pages 297–304. ACM Press: New York, NY, 2004.

Appendix

In this section we give the Magma code used for benchmarks in Section 5. First we need to generate irreducible minimal polynomials $\bar{m}_1$ and $\bar{m}_2$ with degrees d_1 and d_2 in z_1 and z_2 respectively. To do this we use the following Maple code.

```
p := modp1(Prime(1));
m1 := Randprime(d1, z1) mod p:

alias(alpha1 = RootOf(m1, z1)):
m2 := Randprime(d2, z2, alpha1) mod p:

fd := fopen("m1.dat", WRITE);
fprintf(fd,"dm1 := %d;\ndm2 := %d;\nm1 := %a;\n", d1, d2, m1);
fclose(fd);

fd := fopen("m2.dat", WRITE);
fprintf(fd,"m2 := %a;\n", m2);
fclose(fd);
```

This code will generate two files which include the minimal polynomials. These files will be loaded by the following Magma code.

```
p := 3037000453;
Zp := FiniteField(p);

gen_Zp := function()
        return Random(Zp);
end function;

Zp<z1> := PolynomialRing(Zp);

load "m1.dat";

R1<alpha1> := quo<Zp | m1>;
F1<z2> := PolynomialRing(R1);

gen_alpha1 := function()
         return &+[gen_Zp()*alpha1^i : i in [0..dm1 - 1]];
end function;

load "m2.dat";

R2<alpha2> := quo<F1 | m2>;
F<x> := PolynomialRing(R2);

gen_coeff := function()
        return &+[gen_alpha1()*alpha2^i : i in [0..dm2 - 1]];
end function;

gen_poly := function(d)
        return &+[gen_coeff()*x^i: i in [0..d]];
end function;
```

```
for d in [40,80] do

a := gen_poly(d);
b := gen_poly(d);
g := gen_poly(d);

t1 := Cputime();
for i in [1..20] do f1 := a*g; f2 := b*g; h1 := Hash(f1); h2 := Hash(f2); end for;
t2 := Cputime(t1);
tmul := t2 / 20.0;

t2 := Cputime();
g := Gcd(f1,f2); h := Hash(g);
t3 := Cputime(t2);

t3 := Cputime();
r := f1 mod  g; r:= Hash(r);
t4 := Cputime(t3);

printf "MAG_MUL = %o\t\t MAG_REM = %o\t\t MAG_GCD = %o\n",tmul,t4,t3;
end for;
```

Remark 14. In Magma, one can also create the ring $\mathbb{Z}_p$ using `Zp := ResidueClassRing(p);` instead of `Zp := FiniteField(p);` We observed that all Magma timings are about twice as slow using `Zp := ResidueClassRing(p);`