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., V5A 1S6, Canada.
sjavadi@cs.sfu.ca

[2] Department of Mathematics, Simon Fraser University,
Burnaby, B.C., V5A 1S6, 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 [10] 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 [4, 11]).

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 $L_p[x_1, \ldots, x_n]$. This representation is generally more efficient than the distributed and recursive sparse representations for sparse polynomials. See for example the comparison by Fateman in [3]. And since efficiency 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 have observed that arithmetic in L_p is very slow when α_1 has low degree. Since this case 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 reason why this happens is because the cost

*Correspondence to: CECM, Simon Fraser University, Burnaby, BC, Canada. Tel: +1.778.782.5617

of storage management, allocating small arrays for storing intermediate polynomials of low degree can be much higher than the cost of the actual arithmetic being done in $\mathbb{Z}_p$.

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. 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 also significantly affects overall performance.

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 $0 \le i < d_r$

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

Note, 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 1. Let $r = 2$ and $p = 17$. Let $\bar{m}_1 = z_1^3 + 3$, $\bar{m}_2 = z_2^2 + 5z_1z_2 + 4z_2 + 7z_1^2 + 3z_1 + 6$, and $f = 3 + 4z_1 + (5 + 6z_1)z_2 + (7 + 8z_1 + 9z_1^2 + (10z_1 + 11z_1^2)z_2)x + 12x^2$.
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 $d_x = 2, d_1 = 3, d_2 = 2, S_1 = d_1 + 1 = 4, S_2 = d_2S_1 + 1 = 9$ 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 $\bar{m}_i = a_0 + a_1z_i + \ldots a_{d_i}z_i^{d_i}$ 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 $\bar{S}_i = 1 + (d_i + 1)S_{i-1} = d_iS_{i-1} + 1 + S_{i-1} = S_i + S_{i-1}$. 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 1 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 [7]).

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 [7], 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 11) is $(d_a + 1) \times (d_b + 1)$ and the number of divisions in $R_{r-1}[z_r]$ (in line 15) 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 $0 \le k \le d_c$, c_k will be computed and stored in $C[kS_N + 1 \dots (k+1)S_N]$.

1: Set $d_a := A[0]$ and $d_b := B[0]$.
2: **if** $d_a = -1$ or $d_b = -1$ **then** Set $C[0] := -1$ and **return.**
3: **if** $N = 0$ **then** Call IP_MUL_NO_EXT on inputs A, B and C and **return.**
4: 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 $E[0 \dots e_N]$*).
5: 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 $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
6: Set $d_c := d_a + d_b$ and $s_c := 1$.
7: **for** k from 0 to d_c **do**
8:   Set $s_a := 1 + iS_N$ and $s_b := 1 + (k-i)S_N$.
9:   Set $T_1[0] := -1$ ($T_1 = 0$).
10:   **for** i from $\max(0, k - d_b)$ to $\min(k, d_a)$ **do**
11:     Call IP_MUL($N-1$, $A[s_a \dots \bar{a}]$, $B[s_b \dots \bar{b}], T_2, E', W'$).
12:     Call IP_ADD($N-1, T_1, T_2$) ($T_1 := T_1 + T_2$)
13:     Set $s_a := s_a + S_N$ and $s_b := s_b - S_N$.
14:   **end for**
15:   Call IP_REM($N-1, T_1, M, E', W'$). (*Reduce T_1 modulo $M = \bar{m}_N$*).
16:   Copy $T_1[0 \dots S_N - 1]$ into $C[s_c \dots s_c + S_N - 1]$.
17:   Set $s_c := s_c + S_N$.
18: **end for**
19: Determine $\deg_x(a \times b)$: (*There might be zero-divisors*).
20: Set $s_c := s_c - S_N$.
21: **while** $d_c \ge 0$ and $C[sc] = -1$ **do** Set $d_c := d_c - 1$ and $s_c := s_c - S_N$.
22: Set $C[0] := d_c$.

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 16 is reduced to $d_a + 1$ (see line 8). 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 $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
- $W[0 \ldots w_N]$: *the working storage* for the intermediate operations.

Output: The remainder $\bar{R}$ of a divided by b will be stored in $A[0 \ldots \bar{r}]$ where $\bar{r} = P(D_r, R_N) - 1$ and $D_r = \deg_x(\bar{R}) \leq d_b - 1$. Also let Q represent the quotient $\bar{Q}$ of a divided by b. $Q[1 \ldots \bar{q}]$ will be stored in $A[1 + d_b S_N \ldots \bar{a}]$ where $\bar{q} = P(d_a - d_b, R_N) - 1$.

1: Set $d_a := A[0]$ and $d_b := B[0]$.
2: **if** $d_a < d_b$ **then return**.
3: **if** $N = 0$ **then** Call IP_REM_NO_EXT on inputs A and B and **return.**
4: Set $D_q := d_a - d_b$ and $D_r := d_b - 1$.
5: 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 $E[0 \ldots e_N]$*).
6: 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 $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
7: Set $s_c := 1 + d_a S_N$
8: **for** $k = d_a$ to 0 by -1 **do**
9:    Copy $A[s_c \ldots s_c + S_N - 1]$ into $T_1[0 \ldots S_N - 1]$.
10:    Set $i := \max(0, k - D_q)$, $s_b := 1 + iS_N$ and $s_a := 1 + (k - i + d_b)S_N$.
11:    **while** $i \leq \min(D_r, k)$ **do**
12:       Call IP_MUL($N-1, A[s_a \ldots \bar{a}], B[s_b \ldots \bar{b}], T_2, E', W'$).
13:       Call IP_SUB($N-1, T_1, T_2$) ($T_1 := T_1 - T_2$).
14:       Set $s_b := s_b + S_N$ and $s_a := s_a - S_N$.
15:    **end while**
16:    Call IP_REM($N-1, T_1, M, E', W'$) (*Reduce T_1 modulo $M = \bar{m}_N$*).
17:    Copy $T_1[0 \ldots S_N - 1]$ into $A[s_c \ldots s_c + S_N - 1]$.
18:    Set $s_c := s_c - S_N$.
19: **end for**
20: Set $s_c := 1 + D_r S_N$.
21: **while** $D_r \geq 0$ and $A[s_c] = -1$ **do** Set $D_r := D_r - 1$ and $s_c := s_c - S_N$.
22: Set $A[0] := D_r$.

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. Thus 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 16 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 function:

- 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.

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:
- $N \geq 1$ the number of field extensions.
- Array $A[0 \dots S_N - 1]$ representing the univariate polynomial $a \in R_N$.
- Array $I[0 \dots S_N - 1]$: Space needed for storing the inverse $a^{-1} \in R_N$.
- $E[0 \dots e_N]$ representing the set of minimal polynomials. Note $e_N = S_N + 2\sum_{i=1}^{N-1} S_i$.
- $W[0 \dots w_N]$: *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** Call IP_INV_NO_EXT on inputs A, I, E, M and W and **return.**
3: **if** $A[i] = 0$ for all $0 \leq i < N$ and $A[N] = 1$ (*Test if* $a = 1$) **then**
4:    Copy A into I and **return 0.**
5: **end if**
6: Let $r_1 = W[0 \dots t-1], r_2 = W[t \dots 2t-1], s_1 = I, s_2 = W[2t \dots 3t-1], T = W[3t \dots 4t-1]$, $T' = W[4t \dots 4t + t' - 1]$ and $W' = W[4t + t' \dots w_N]$ where $t = P(d_N, R_{N-1}) - 1 = \bar{S}_N - 1$, $t' = P(2d_N - 2, R_{N-1})$ and $d_N = M[0] = \deg_{z_N}(\bar{m}_N)$.
7: Copy A and M into r_1 and r_2 respectively.
8: Set $s_2[0] := -1$ (s_2 *represents 0*).
9: Let $Z \in \mathbb{Z} :=$ IP_INV$(N-1, A + D_a\, S_{N-1} + 1, T, E', W')$ where $D_a = A[0] = \deg_{z_N}(a)$. ($A[D_a S_{N-1} + 1 \dots S_N - 1]$ *represents* $l = lc_{z_N}(a)$ *and* T *represents* l^{-1}.)
10: **if** $Z > 0$ **then** Copy T into I and **return** Z.
11: Copy T into s_1.
12: Call IP_SCAL_MUL(N, r_1, T, E', W') (r_1 *is made monic*).
13: **while** $r_2[0] \neq -1$ **do**
14:    Set $Z =$ IP_INV$(N-1, r_2 + D_{r_2}\, S_{N-1} + 1, T, E', W')$ where $D_{r_2} = r_2[0] = \deg_{z_N}(r_2)$.
15:    **if** $Z > 0$ **then** Copy T into I and **return** Z.
16:    Call IP_SCAL_MUL(N, r_2, T, E', W') (r_2 *is made monic*).
17:    Call IP_SCAL_MUL(N, s_2, T, E', W').
18:    Set $D_q := \max(-1, r_1[0] - r_2[0])$.

19: Call IP_REM(N, r_1, r_2, E', W').
20: Swap the arrays r_1 and r_2. (*Interchange only the pointers*).
21: Set $t_1 := r_2[r_1[0]S_{N-1}]$ and set $r_2[r_1[0]S_{N-1}] := D_q$.
22: Call IP_MUL($N-1, r_2[r_1[0]S_{N-1} \dots S_N - 1], s_2, T', E', W'$).
23: Call IP_REM($N-1, T', M, E', W'$) and then IP_SUB($N-1, s_1, T'$). ($s_1 := s_1 - qs_2$.)
24: Set $r_2[r_1[0]S_{N-1}] := t_1$.
25: Swap the arrays s_1 and s_2. (*Interchange only the pointers*).
26: **end while**
27: **if** $r_1[i] = 0$ for all $0 \le i < N$ and $r_1[N] = 1$ **then**
28: Copy s_1 into I ($r_1 = 1$ *and* s_1 *is the inverse*) and **return** 0.
29: **else**
30: Copy r_1 into I ($r_1 \neq 1$ *is the zero-divisor*) and **return** $N-1$ ($\bar{m}_{N-1}$ *is reducible*).
31: **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 21 we explicitly compute and set the degree of the quotient before using it to compute $s_1 := s_1 - qs_2$ in lines 22 and 23. Here $r_2[r_1[0]S_{N-1} \dots S_N - 1]$ is the quotient of $r_1 \div r_2$ in line 19.

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. 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 a zero-divisor is encountered, 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. Also, B is destroyed.

if $N = 0$ **then** CALL IP_GCD_NO_EXT on inputs A and B and **return** 0.
Set $d_a := A[0]$ and $d_b := B[0]$.
Let r_1 and r_2 point to A and B respectively.
Let $I = W[0 \dots t-1]$ and $W' = W[t \dots w_N]$ where $t = \bar{S}_N - 1 = S_N + S_{N-1} - 1$.
Let Z be the output of IP_INV($N, r_1 + r_1[0]\, S_N + 1, I, E, W'$).
if $Z > 0$ **then** Copy I into A and **return** Z.
Call IP_SCAL_MUL(N, r_1, I, E, W').
while $r_2[0] \neq -1$ **do**
Let Z be the output of IP_INV($N, r_2 + r_2[0]\, S_N + 1,\ I,\ E, W'$).
if $Z > 0$ **then** Copy I into A and **return** Z.
Call IP_SCAL_MUL(N, r_2, I, E, W') (*make* r_2 *monic*).
Call IP_REM(N, r_1, r_2, E, W') (*the remainder of* $r_1 \div r_2$ *is in* r_1).
Swap r_1 and r_2 (*interchange pointers*).
end while
Copy r_1 into A.
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, in steps 6 and 10 the algorithm stores the zero-divisor in the space provided for a and returns Z 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 2. $S_N > 2S_{N-1}$ 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 3. $\sum_{i=1}^{N-1} S_i < S_N$ 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 2 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 4. $\sum_{i=1}^{N} S_i < 2S_N$ for $N > 1$.

Lemma 5. $P(2d_N - 2, R_{N-1}) = 2S_N - S_{N-1} - 1$ 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 5,11 and 15 of algorithm IP_MUL and lines 6,12 and 16 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 4 we have

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

Remark 7. 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 $W[0 \ldots w_n]$ 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.

Let $C(N)$ denote the working storage needed for the operation IP_SCAL_MUL. It is easy to show that $C(N) = M(N-1) + P(2d_N - 2, R_{N-1}) < M(N)$.

4.2 Inversion

Let $I(N)$ denote the amount of working storage needed to invert $c \in R_N$. In lines 6, 9, 12, 14, 16, 17, 19 , 22 and 23 of algorithm IP_INV we use the working storage. We have

$$I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + \max(I(N-1), M(N-1), Q(N-1)). \quad (4)$$

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

$$I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + \max(I(N-1), M(N-1)). \quad (5)$$

Lemma 8. For $N \geq 1$, we have $M(N) < I(N)$.

Proof. (by contradiction) Assume $M(N) \geq I(N)$. Using (5) we have $I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + M(N-1)$. 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 $4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + M(N-1) \leq 2P(2d_N - 2, R_{N-1}) + M(N-1)$ thus $4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) \leq 2P(2d_N - 2, R_{N-1}) \Rightarrow 6S_N + 3S_{N-1} - 1 \leq 4S_N - 2S_{N-1} - 2$ which is a contradiction. Thus $I(N) > M(N)$. □

Using Equation (4) and Lemma 8 we conclude that $I(N) = 4P(d_N, R_{N-1}) + P(2d_N - 2, R_{N-1}) + I(N-1)$. Simplifying this yields:

Theorem 9. $I(N) = 4\sum_{i=1}^{N} P(d_i, R_{i-1}) + \sum_{i=1}^{N} P(2d_i - 2, R_{i-1}) = 4\sum_{i=1}^{N} (S_i + S_{i-1}) + \sum_{i=1}^{N} (2S_i - S_{i-1} - 1) = 6S_N + 9\sum_{i=1}^{N-1} S_i - N$.

Using Lemma 2 an upper bound for $I(N)$ is $I(N) < 6S_N + 9S_N = 15S_N$.

4.3 GCD Computation

Let $G(N)$ denote the working storage needed to compute the GCD of $a, b \in R_N[x]$. In lines 4,5,7,9,11 and 12 of algorithm IP_GCD we use the working storage. We have $G(N) = \bar{S}_N + \max(I(N), C(N), Q(N))$. Lemma 8 states that $I(N) > M(N) = Q(N) > C(N)$ hence

$$G(N) = \bar{S}_N + I(N) = S_N + S_{N-1} + 6S_N + 9\sum_{i=1}^{N-1} S_i - N = 7S_N + S_{N-1} + 9\sum_{i=1}^{N-1} S_i - N.$$

Since $I(N) < 15S_N$, we have an upper bound on $G(N)$:

Theorem 10. $G(N) = S_N + S_{N-1} + I(N) < S_N + S_{N-1} + 15S_N < 17S_N$.

Remark 11. The constants 6, 15 and 17 appearing in Theorems 6, 9 and 10 respectively, are not the best possible. 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 quo(f_1, g) and 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.

Table 1: Timings in CPU seconds on an AMD Opteron 254 CPU running at 2.8 GHz

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	0.09	0.384	2.26
3	20	40	2460	0.108	0.054	0.106	0.11	0.340	2.35
4	15	40	2460	0.106	0.056	0.106	0.10	0.327	2.39
6	10	40	2460	0.106	0.121	0.105	0.14	0.328	5.44
10	6	40	2460	0.100	0.093	0.100	0.37	0.303	7.84
15	4	40	2460	0.097	0.055	0.095	0.17	0.283	3.27
20	3	40	2460	0.092	0.046	0.091	0.14	0.267	2.54
30	2	40	2460	0.087	0.038	0.087	0.10	0.242	1.85
2	30	80	4860	0.477	0.115	0.478	0.27	1.449	9.41
3	20	80	4860	0.407	0.127	0.409	0.27	1.304	9.68
4	15	80	4860	0.404	0.132	0.406	0.28	1.253	9.98
6	10	80	4860	0.398	0.253	0.400	0.35	1.234	22.01
10	6	80	4860	0.380	0.197	0.381	0.86	1.151	31.57
15	4	80	4860	0.365	0.127	0.364	0.40	1.081	13.49
20	3	80	4860	0.353	0.109	0.353	0.33	1.030	10.59
30	2	80	4860	0.336	0.086	0.337	0.26	0.932	7.83

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 because Magma is using a sub-quadratic algorithm for multiplication. We describe the algorithm used by Magma ([9]) briefly. 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 shows that our in-place GCD algorithm is a factor of 6 to 27 times faster than Magma's GCD algorithm. Since both algorithms use 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 there is still some overhead for α_1 of low degree.

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 [8]) 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 elements of $\mathbb{Z}_p$ are stored as signed integers and an integer x in the range $-p^2 < x < p^2$ 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.

6 Concluding Remarks

Our C library of in-place routines has been integrated into Maple 14 for use in the GCD algorithms in [11] and [4]. 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 the parameters and variables except x_1 and using rational function interpolation to recover the GCD. This results in many GCD computations in $L_p[x_1]$. In many applications, K has field extensions of low degree, often quadratic or cubic.

Our C library is available on our website at

`http://www.cecm.sfu.ca/CAG/code/ASCM09/inplace.c`

The code we used to generate the Magma timings in Section 5 is available in the file

`http://www.cecm.sfu.ca/CAG/code/ASCM09/magma.txt`

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 in this paper are complementary in that we sought to improve arithmetic when L has relatively low degree.

Acknowledgments

This work was supported by the MITACS NCE of Canada.

References and Notes

[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 Bulletin*, **37**(1):4–15, ACM Press, 2003.

[4] S. M. M. Javadi and M. B. Monagan. A sparse modular gcd algorithm for polynomials over algebraic function fields. *Proceedings of ISSAC '07*, pp. 187–194, ACM Press, 2007.

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

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

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

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

[9] Allan Steel. Multiplication in $L_p[x]$ in Magma. Private communication, 2009.

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

[11] Mark van Hoeij and Michael Monagan. Algorithms for polynomial gcd computation over algebraic function fields. *Proceedings of ISSAC '04*, ACM Press, pp. 297–304, 2004.