Computational schemes for subresultant chains

Mohammadali Asadi[1], Alexander Brandt[2], and Marc Moreno Maza[3]

Department of Computer Science, University of Western Ontario

[1] masadi4@uwo.ca, [2] abrandt5@uwo.ca, [3] moreno@csd.uwo.ca

Abstract. Subresultants are one of the most fundamental tools in computer algebra. They are at the core of numerous algorithms including, but not limited to, polynomial GCD computations, polynomial system solving, and symbolic integration. When the subresultant chain of two polynomials is involved in a client procedure, not all polynomials of the chain, or not all coefficients of a given subresultant, may be needed. Based on that observation, this paper discusses different practical schemes, and their implementation, for efficiently computing subresultants. Extensive experimentation supports our findings.

Keywords: resultant, subresultant chain, modular arithmetic, polynomial system solving, GCDs

1 Introduction

The goal of this paper is to investigate how several optimization techniques for subresultant chain computations benefit polynomial system solving in practice. These optimizations rely on ideas which have appeared in previous works, but without the support of successful experimental studies. Therefore, this paper aims at filling this gap.

The first of these optimizations takes advantage of the *Half-GCD* algorithm for computing GCDs of univariate polynomials over a field $\mathbf{k}$. For input polynomials of degree (at most) n, this algorithm runs within $O(\mathrm{M}(n) \log n)$ operations in $\mathbf{k}$, where $\mathrm{M}(n)$ is a polynomial multiplication time, as defined in [25, Chapter 8]. The *Half-GCD* algorithm originated in the ideas of [15], [13] and [22], while a robust implementation was a challenge for many years. One of the earliest correct designs was introduced in [24].

The idea of speeding up subresultant chain computations by means of the *Half-GCD* algorithm takes various forms in the literature. In [21], Reischert proposes a fraction-free adaptation of the *Half-GCD* algorithm, which can be executed over an effective integral domain $\mathbb{B}$, within $O(\mathrm{M}(n) \log n)$ operations in $\mathbb{B}$. We are not aware of any implementation of Reischert's algorithm.

In [17], Lickteig and Roy introduced a fast divide and conquer variant of the subresultant algorithm which avoids coefficient growth in defective cases. In [14], Lecerf analyses the complexity of their algorithm, when run over an effective ring endowed with the partially defined division routine, yielding a running time

estimate similar to that of Reischert. Lecerf realized an implementation of that algorithm, but observed in [14] that computations of subresultant chains based Ducos algorithm [9] or evaluation-interpolation strategies were faster.

In [25, Chapter 11], von zur Gathen and Gerhard show how the nominal coefficients (see Section 2 for this term) of the subresultant chain of two univariate polynomials a, b over a field can be computed within $O(\mathrm{M}(n) \log n)$ operations in $\mathbf{k}$, by means of an adaptation of the Half-GCD algorithm. In this paper, we extend their approach to compute any pair of consecutive non-zero subresultants of a, b within the same time bound. The details are presented in Section 3.

The second of these optimizations for subresultant chain computations relies on the observation that not all non-zero subresultants of a given subresultant chain may be needed. To illustrate this fact, consider two commutative rings $\mathbb{A}$ and $\mathbb{B}$, two non-constant univariate polynomials a, b in $\mathbb{A}[y]$ and a ring homomorphism Ψ from $\mathbb{A}$ to $\mathbb{B}$ so that $\Psi(\mathrm{lc}(a)) \neq 0$ and $\Psi(\mathrm{lc}(b)) \neq 0$. Then, the specialization property of subresultants (see the precise statement in Section 2) tells us that the subresultant chain of $\Psi(a), \Psi(b)$ is the image of the subresultant chain of a, b via Ψ.

This property has at least two important practical applications. When $\mathbb{B}$ is polynomial ring over a field, say $\mathbb{B}$ is $\mathbb{Z}/p\mathbb{Z}[x]$ and $\mathbb{A}$ is $\mathbb{Z}/p\mathbb{Z}$, then one can compute a GCD of $\Psi(a), \Psi(b)$ via evaluation and interpolation techniques. Similarly, when $\mathbb{B}$ is a field extension, say $\mathbb{B}$ is $\mathbb{Q}[x]/\langle m(x) \rangle$, where $m(x)$ is a square-free polynomial, and $\mathbb{A}$ is $\mathbb{Q}[x]$, then one can compute a GCD of $\Psi(a), \Psi(b)$ using the celebrated D5 Principle [7]. More generally, if $\mathbb{B}$ is $\mathbb{Q}[x_1, \ldots, x_n]/\langle T \rangle$, where $T = (t_1(x_1), \ldots, t_n(x_1, \ldots, x_n))$ is a zero-dimensional regular chain (generating a radical ideal), and $\mathbb{A}$ is $\mathbb{Q}[x_1, \ldots, x_n]$, then one can compute a so-called regular GCD of a and b modulo $\langle T \rangle$, see [4]. The principle of that calculation generalizes the D5 Principle as follows:

1. if the resultant of a, b is invertible modulo $\langle T \rangle$ then 1 is a regular GCD of a and b modulo $\langle T \rangle$;
2. if the nominal leading coefficients $s_o, s_1, \ldots, s_k$ are all zero modulo $\langle T \rangle$, for some k, and the nominal leading coefficient s_{k+1} is invertible modulo $\langle T \rangle$, the subresultant S_k of index k of a, b is a regular GCD of a and b modulo $\langle T \rangle$; and
3. one can always reduce to one of the above two cases by splitting T, when a zero-divisor of $\mathbb{B}$ is encountered.

In practice, in the above procedure, k is often zero, which can be seen as a consequence of the celebrated *Shape Lemma* [3]. This suggests to compute the subresultant chain of a, b in $\mathbb{A}[y]$ speculatively. To be precise, and taking advantage of the Half-GCD algorithm, it is desirable to compute the subresultants of index 0 and 1, delaying the computation of subresultants of higher index until proven necessary.

We discuss that idea of computing subresultants speculatively in Section 3. Making that approach successful, in comparison to non-speculative approaches, requires to overcome several obstacles:

1. computing efficiently the subresultants S_0 and S_1, via the Half-GCD; and
2. developing an effective "recovery" strategy in case of "misprediction", that is, when subresultants of index higher than 1 turn out to be needed.

To address the first obstacle, our implementation, combines various implementation schemes of the Half-GCD, inspired by the work done in NTL [23]. To address the second obstacle, when we compute the subresultants of index 0 and 1, via the Half-GCD, we record or *cache* the sequence of quotients (associated with the Euclidean remainders) so as to easily obtain subresultants of index higher than 1, if needed.

The extensive experimentation results in Section 5 indicate that the performance of our univariate polynomials over finite fields (based on FFT) are closely comparable with their counterparts in NTL. In addition, we have aggressively tuned our subresultant schemes based on evaluation-interpolation techniques. Our modular subresultant chain algorithms are up to 10× and 400× faster than non-modular counterparts (mainly Ducos' subresultant chain algorithm) in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively. Further, utilizing the Half-GCD algorithm to compute subresultants yields an additional speed-up factor of 7× and 2× for polynomials in $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$, respectively.

Further still, we optimize Ducos' subresultant chain algorithm in terms of memory access patterns, see Section 4. We have implemented both the original Ducos algorithm [9] and our optimized version over both arbitrary-precision integers and rational numbers. For univariate polynomials of degree as large as 2000, these implementations, respectively, use 11× and 3× less memory than the original Ducos algorithm implemented in Maple.

All of our code, providing also univariate and multivariate polynomial arithmetic, is open source and part of the Basic Polynomial Algebra Subprograms (BPAS) library available at www.bpaslib.org. Our many subresultant schemes have been integrated, tested and utilized in the multithreaded BPAS solver [2].

2 Review of subresultant theory

In this review of subresultant theory, we follow the presentations of [8] and [12]. Let $\mathbb{B}$ be a commutative ring with identity and let $m \le n$ be positive integers. Let M be a $m \times n$ matrix with coefficients in $\mathbb{B}$. Let M_i be the square submatrix of M consisting of the first $m-1$ columns of M and the i-th column of M, for $m \le i \le n$; let $\det(M_i)$ be the determinant of M_i. The *determinantal polynomial* of M denoted by $\mathrm{dpol}(M)$ is a polynomial in $\mathbb{B}[y]$, given by

$$\mathrm{dpol}(M) = \det(M_m)y^{n-m} + \det(M_{m+1})y^{n-m-1} + \cdots + \det(M_n).$$

Note that, if $\mathrm{dpol}(M)$ is not zero, then its degree is at most $n-m$. Let $f_1, \ldots, f_m$ be polynomials of $\mathbb{B}[y]$ of degree less than n. We denote by $\mathrm{mat}(f_1, \ldots, f_m)$ the $m \times n$ matrix whose i-th row contains the coefficients of f_i, sorted in order of decreasing degree, and such that f_i is treated as a polynomial of degree $n-1$. We denote by $\mathrm{dpol}(f_1, \ldots, f_m)$ the determinantal polynomial of $\mathrm{mat}(f_1, \ldots, f_m)$.

Let $a, b \in \mathbb{B}[y]$ be non-constant polynomials of respective degrees $m = \deg(a)$, $n = \deg(b)$ with $m \geq n$. The leading coefficient of a w.r.t. y is denoted by $\mathrm{lc}(a)$. Let k be an integer with $0 \leq k < n$. Then, the k-th *subresultant* of a and b (also known as the *subresultant of index* k of a and b), denoted by $S_k(a, b)$, is

$$S_k(a, b) = \mathrm{dpol}(y^{n-k-1}a, y^{n-k-2}a, \ldots, a, y^{m-k-1}b, \ldots, b).$$

This is a polynomial which belongs to the ideal generated by a and b in $\mathbb{B}[y]$. In particular, $S_0(a, b)$ is the resultant of a and b denoted by $\mathrm{res}(a, b)$. Observe that if $S_k(a, b)$ is not zero then its degree is at most k. If $S_k(a, b)$ has degree k, then $S_k(a, b)$ is said to be *non-defective* or *regular*; if $S_k(a, b) \neq 0$ and $\deg(S_k(a, b)) < k$, then $S_k(a, b)$ is said to be *defective*. We call k-th *nominal leading coefficient*, demoted by s_k, the coefficient of $S_k(a, b)$ in y^k. Observe that if $S_k(a, b)$ is defective, then we have $s_k = 0$. For convenience, we extend the definition to the n-th subresultant as follows:

$$S_n(a, b) = \begin{cases} \gamma(b)b, & \text{if } m > n \text{ or } \mathrm{lc}(b) \in \mathbb{B} \text{ is regular} \\ \text{undefined}, & \textit{otherwise} \end{cases}$$

where $\gamma(b) = \mathrm{lc}(b)^{m-n-1}$. In the above, *regular* means *not a zero-divisor*. Note that when m equals n and $\mathrm{lc}(b)$ is a regular element in $\mathbb{B}$, then $S_n(a, b) = \mathrm{lc}(b)^{-1}b$ is in fact a polynomial over the total fraction ring of $\mathbb{B}$. We call *specialization property of subresultants* the following property. Let $\mathbb{A}$ be another commutative ring with identity and Ψ a ring homomorphism from $\mathbb{B}$ to $\mathbb{A}$ such that we have $\Psi(\mathrm{lc}(a)) \neq 0$ and $\Psi(\mathrm{lc}(b)) \neq 0$. Then, for $0 \leq k \leq n$, we have:

$$S_k(\Psi(a), \Psi(b)) = \Psi(S_k(a, b)).$$

From now on, we assume that the ring $\mathbb{B}$ is an integral domain. Writing $\delta = \deg(a) - \deg(b)$, there exists a unique pair (q, r) of polynomials in $\mathbb{B}[y]$ satisfying $ha = qb + r$, where $h = \mathrm{lc}(b)^{\delta+1}$, and either $r = 0$ or $\deg(r) < \deg(b)$; the polynomials q and r, denoted respectively $\mathrm{pquo}(a, b)$ and $\mathrm{prem}(a, b)$, are the *pseudo-quotient* and *pseudo-reminder* of a by b. The *subresultant chain* of a and b, defined as

$$\mathrm{subres}(a, b) = (S_n(a, b), S_{n-1}(a, b), S_{n-2}(a, b), \ldots, S_0(a, b)),$$

satisfies relations which induce a Euclidean-like algorithm for computing the entire subresultant chain: $\mathrm{subres}(a, b)$. This algorithm runs within $O(n^2)$ operations in $\mathbb{B}$, when $m = n$, see [8]. For convenience, we simply write S_k instead of $S_k(a, b)$ for each k. We write $a \sim b$, for $a, b \in \mathbb{B}[y]$, whenever a, b are associate elements in $\mathrm{frac}(\mathbb{B})[y]$, the field of fractions of $\mathbb{B}$. Then for $1 \leq k < n$, we have:

(*i*) $S_{n-1} = \mathrm{prem}(a, -b)$; if S_{n-1} is non-zero, defining $e := \deg(S_{n-1})$, then we have

$$S_{e-1} = \frac{\mathrm{prem}(b, -S_{n-1})}{\mathrm{lc}(b)^{(m-n)(n-e)+1}},$$

(*ii*) if $S_{k-1} \neq 0$, defining $e := \deg(S_{k-1})$ and assuming $e < k-1$ (thus assuming S_{k-1} defective), then we have:

(*a*) $\deg(S_k) = k$, thus S_k is non-defective,

(*b*) $S_{k-1} \sim S_e$ and $\mathrm{lc}(S_{k-1})^{k-e-1} S_{k-1} = {s_k}^{k-e-1} S_e$, thus S_e is non-defective,

(*c*) $S_{k-2} = S_{k-3} = \cdots = S_{e+1} = 0$,

(*iii*) if both S_k and S_{k-1} are non-zero, with respective degrees k and e then we have:

$$S_{e-1} = \frac{\mathrm{prem}(S_k, -S_{k-1})}{\mathrm{lc}(S_k)^{k-e+1}}.$$

Algorithm 1 SUBRESULTANT (a, b, y)

Input: $a, b \in \mathbb{B}[y]$ with $m = \deg(a) \geq n = \deg(b)$ and $\mathbb{B}$ is an integral domain
Output: the non-zero subresultants from $(S_n, S_{n-1}, S_{n-2}, \ldots, S_0)$

1: **if** $m > n$ **then**
2: $\quad S := (\mathrm{lc}(b)^{m-n-1} b)$
3: **else** $S := ()$
4: $s := \mathrm{lc}(b)^{m-n}$
5: $A := b$; $B := \mathrm{prem}(a, -b)$
6: **while** true **do**
7: $\quad d := \deg(A)$; $e := \deg(B)$
8: $\quad$ **if** $B = 0$ **then return** S
9: $\quad S := (B) \cup S$; $\delta := d - e$
10: $\quad$ **if** $\delta > 1$ **then**
11: $\quad\quad C := \frac{\mathrm{lc}(B)^{\delta-1} B}{s^{\delta-1}}$
12: $\quad\quad S := (C) \cup S$
13: $\quad$ **else** $C := B$
14: $\quad$ **if** $e = 0$ **then return** S
15: $\quad B := \frac{\mathrm{prem}(A, -B)}{s^{\delta} \mathrm{lc}(A)}$
16: $\quad A := C$; $s := \mathrm{lc}(A)$
17: **end while**

Algorithm 1 from [9] is a known version of this procedure that computes all non-zero subresultants $a, b \in \mathbb{B}[y]$. Note that the core of this algorithm is the while-loop in which the computation of the subresultants S_e and S_{e-1}, with the notations of the above points (*ii*) and (*iii*), are carried out.

3 Computing subresultant chains via the Half-GCD, speculatively

As discussed in the introduction, when the ring $\mathbb{B}$ is a field $\mathbf{k}$, the computation of the subresultant chain of the polynomials $a, b \in \mathbb{B}[y]$ can take advantage of asymptotically fast algorithms for computing $\gcd(a, b)$. After recalling its

specifications, we explain how we take advantage of the Half-GCD algorithm in order to compute the subresultants in $\mathrm{subres}(a, b)$ speculatively.

Consider two non-zero univariate polynomials $a, b \in \mathbf{k}[y]$ with $n_0 := \deg(a)$, $n_1 := \deg(b)$ with $n_0 \geq n_1$. The extended Euclidean algorithm (EEA) computes the successive remainders $(r_0 := a, r_1 := b, r_2, \ldots, r_\ell = \gcd(a, b))$ with degree sequence $(n_0, n_1, n_2 := \deg(r_2) \ldots, n_\ell := \deg(r_\ell))$ and the corresponding quotients $(q_1, q_2, \ldots, q_\ell)$ defined by

(i) $r_{i+1} = \mathrm{rem}(r_i, r_{i-1}) = r_{i-1} - q_i r_i$, for $1 \leq i \leq \ell$,
(ii) $q_i = \mathrm{quo}(r_i, r_{i-1})$ for $1 \leq i \leq \ell$,
(iii) $n_{i+1} < n_i$, for $1 \leq i < \ell$, and
(iv) $r_{\ell+1} = 0$ with $\deg(r_{l+1}) = -\infty$.

This computation requires $O(n^2)$ operations in $\mathbf{k}$. We denote by Q_i, the *quotient matrices*, defined, for $1 \leq i \leq \ell$, by

$$Q_i = \begin{bmatrix} 0 & 1 \\ 1 & -q_i \end{bmatrix}.$$

so that, for $1 \leq i < \ell$, we have

$$\begin{bmatrix} r_i \\ r_{i+1} \end{bmatrix} = Q_i \begin{bmatrix} r_{i-1} \\ r_i \end{bmatrix} = Q_i \ldots Q_1 \begin{bmatrix} r_0 \\ r_1 \end{bmatrix}. \quad (1)$$

We define $m_i := \deg(q_i)$, so that we have $m_i = n_{i-1} - n_i$ for $1 \leq i \leq \ell$. The degree sequence $(n_0, \ldots, n_l)$ is said to be *normal* if $n_{i+1} = n_i - 1$ holds, for $1 \leq i < \ell$, or, equivalently if $\deg(q_i) = 1$ holds, for $1 \leq i \leq \ell$.

Using the remainder and degree sequences of non-zero polynomials $a, b \in \mathbf{k}[y]$, Proposition 1, known as the *fundamental theorem on subresultants*, introduces a procedure to compute the nominal leading coefficients of polynomials in the subresultant chain.

Proposition 1 *For $k = 0, \ldots, n_1$, the nominal leading coefficient of the k-th subresultant of (a, b) is either 0 or s_k if there exists $i \leq \ell$ such that $k = \deg(r_i)$,*

$$s_k = (-1)^{\tau_i} \prod_{1 \leq j < i} \mathrm{lc}(r_j)^{n_{j-1} - n_{j+1}} \mathrm{lc}(r_i)^{n_{i-1} - n_i},$$

where $\tau_i = \sum_{1 \leq j < i} (n_{j-1} - n_i)(n_j - n_i)$ [25].

The *Half-GCD*, also known as the *fast extended Euclidean algorithm*, is a divide-and-conquer algorithm for computing a single row of the EEA, say the last one. This can be interpreted as the computation of a 2×2 matrix Q over $\mathbf{k}[y]$ so that we have:

$$\begin{bmatrix} \gcd(a, b) \\ 0 \end{bmatrix} = Q \begin{bmatrix} a \\ b \end{bmatrix}.$$

The major difference between the classical EEA and the Half-GCD algorithm is that, while the EEA computes all the remainders $r_0, r_1, \ldots, r_\ell = \gcd(r_0, r_1)$,

the Half-GCD computes only two consecutive remainders, which are derived from the Q_i quotient matrices, which in turn are obtained from a sequence of "truncated remainders", instead of the original r_i remainders.

Here, we take advantage of the Half-GCD algorithm presented in [25, Chapter 11]. For a non-negative $k \le n_0$, this algorithm computes the quotients $q_1, \dots, q_{h_k}$ where h_k is defined as

$$h_k = \max\Big\{0 \le j \le \ell \mid \sum_{i=1}^{j} m_i \le k\Big\}, \tag{2}$$

the maximum $j \in \mathbb{N}$ so that $\sum_{1 \le i \le j} \deg(q_i) \le k$. This is done within $(22\mathrm{M}(k) + O(k)) \log k$ operations in $\mathbf{k}$. From Equation 2, $h_k \le \min(k, \ell)$, and

$$\sum_{i=1}^{h_k} m_i = \sum_{i=1}^{h_k} (n_{i-1} - n_i) = n_0 - n_{h_k} \le k < \sum_{i=1}^{h_k+1} m_i = n_0 - n_{h_k+1}. \tag{3}$$

Thus, $n_{h_k+1} < n_0 - k \le n_{h_k}$, and so h_k can be uniquely determined; see Algorithm 11.6 in [25] for more details.

Due to the deep relation between subresultants and the remainders of the EEA, the Half-GCD technique can support computing subresultants. This approach is studied in [25]. The Half-GCD algorithm is used to compute the nominal leading coefficient of subresultants up to s_ρ for $\rho = n_{h_k}$ by computing the quotients $q_1, \dots, q_{h_k}$, calculating the $\mathrm{lc}(r_i) = \mathrm{lc}(r_{i-1})/\mathrm{lc}(q_i)$ from $\mathrm{lc}(r_0)$ for $1 \le i \le h_k$, and applying Proposition 1. The resulting procedure runs within the same complexity as the Half-GCD algorithm.

However, for the purpose of computing two successive subresultants $S_{n_v}, S_{n_{v+1}}$ given $0 \le \rho < n_1$, for $0 \le v < \ell$ so that $n_{v+1} \le \rho < n_v$, we need to compute quotients $q_1, \dots, q_{h_\rho}$ where h_ρ is defined as

$$h_\rho = \max\Big\{0 \le j < \ell \mid n_j > \rho\Big\}, \tag{4}$$

using Half-GCD. Let $k = n_0 - \rho$, Equations 3 and 4 deduce $n_{h_\rho+1} \le n_0 - k < n_{h_\rho}$, and $h_\rho \le h_k$. So, to compute the array of quotients $q_1, \dots, q_{h_\rho}$, we can utilize an adaptation of the Half-GCD algorithm of [25]. Algorithm 2 is this adaptation and runs within the same complexity as the algorithm of [25].

Algorithm 2 receives as input two polynomials $r_0 \coloneqq a, r_1 \coloneqq b$ in $\mathbf{k}[y]$, with $n_0 \ge n_1$, $0 \le k \in \mathbb{N}$, $\rho \le n_0$ where ρ, by default, is $n_0 - k$, and the array $\mathcal{A}$ of the leading coefficients of the remainders that have been computed so far. This array should be initialized to size $n_0 + 1$ with $\mathcal{A}[n_0] = \mathrm{lc}(r_0)$ and $\mathcal{A}[i] = 0$ for $0 \le i < n_0$. $\mathcal{A}$ is updated in-place as necessary. The algorithm returns the array of quotients $\mathcal{Q} \coloneqq (q_1, \dots, q_{h_\rho})$ and matrix $M \coloneqq Q_{h_\rho} \cdots Q_1$.

Algorithm 2 and *the fundamental theorem on subresultants* yield Algorithm 3. This algorithm is a Half-GCD-based subresultant algorithm to calculate two successive subresultants without computing others in the chain. Moreover, this algorithm returns intermediate data that has been computed by the Half-GCD algorithm—the array $\mathcal{R}$ of the remainders, the array $\mathcal{Q}$ of the quotients and the

Algorithm 2 $\textsc{AdaptedHGCD}(r_0, r_1, k, \rho, \mathcal{A})$

Input: $r_0, r_1 \in \mathbf{k}[y]$ with $n_0 = \deg(r_0) \geq n_1 = \deg(r_1)$, $0 \leq k \leq n_0$, $0 \leq \rho \leq n_0$ is an upper bound for the degree of the last computed remainder that, by default, is $n_0 - k$ and is fixed in recursive calls (See Algorithm 3), the array $\mathcal{A}$ of the leading coefficients of the remainders (in the Euclidean sequence) which have been computed so far

Output: $h_\rho \in \mathbb{N}$ so that $h_\rho = \max\{j \mid n_j > \rho\}$, the array $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ of the first h_ρ quotients associated with the remainder in the Euclidean sequence and the matrix $M := Q_{h_\rho} \cdots Q_1$; the array $\mathcal{A}$ of leading coefficients is updated in-place

1: **if** $r_1 = 0$ **or** $\rho \geq n_1$ **then return** $\left(0, (), \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$
2: **if** $k = 0$ **and** $n_0 = n_1$ **then**
3: **return** $\left(1, (\mathrm{lc}(r_0)/\mathrm{lc}(r_1)), \begin{bmatrix} 0 & 1 \\ 1 & -\mathrm{lc}(r_0)/\mathrm{lc}(r_1) \end{bmatrix}\right)$
4: $m_1 := \lceil \frac{k}{2} \rceil$; $\delta_1 := \max(\deg(r_0) - 2\ (m_1 - 1), 0)$; $\lambda := \max(\deg(r_0) - 2k, 0)$
5: $\left(h', (q_1, \ldots, q_{h'}), R\right) := \textsc{AdaptedHGCD}(\mathrm{quo}(r_0, y^{\delta_1}), \mathrm{quo}(r_1, y^{\delta_1}), m_1 - 1, \rho, \mathcal{A})$
6: $\begin{bmatrix} c \\ d \end{bmatrix} := R \begin{bmatrix} \mathrm{quo}(r_0, y^\lambda) \\ \mathrm{quo}(r_1, y^\lambda) \end{bmatrix}$ where $R := \begin{bmatrix} R_{00} & R_{01} \\ R_{10} & R_{11} \end{bmatrix}$
7: $m_2 := \deg(c) + \deg(R_{11}) - k$
8: **if** $d = 0$ **or** $m_2 > \deg(d)$ **then return** $\left(h', (q_1, \ldots, q_{h'}), R\right)$
9: $r := \mathrm{rem}(c, d)$; $q := \mathrm{quo}(c, d)$; $Q := \begin{bmatrix} 0 & 1 \\ 1 & -q \end{bmatrix}$
10: $n_{h'+1} := n_{h'} - \deg(q)$
11: **if** $n_{h'+1} \leq \rho$ **then return** $\left(h', (q_1, \ldots, q_{h'}, q), R\right)$
12: $\mathcal{A}[n_{h'+1}] := \mathcal{A}[n_{h'}]/\mathrm{lc}(q)$
13: $\delta_2 := \max(2m_2 - \deg(d), 0)$
14: $\left(h^*, (q_{h'+2}, \ldots, q_{h'+h^*+1}), S\right) :=$
$\textsc{AdaptedHGCD}(\mathrm{quo}(d, y^{\delta_2}), \mathrm{quo}(r, y^{\delta_2}), \deg(d) - m_2, \rho, \mathcal{A})$
15: **return** $\left(h_\rho := h' + h^* + 1, \mathcal{Q} := (q_1, \ldots, q_{h_\rho}), M := SQR\right)$

array $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence—to later calculate higher subresultants in the chain without calling Half-GCD again. This *caching* scheme is shown in Algorithm 4.

Let us explain this technique with an example. For non-zero polynomials $a, b \in \mathbf{k}[y]$ with $n_0 = \deg(a), n_1 = \deg(b)$, so that we have $n_0 \geq n_1$. The subresultant call $\textsc{Subresultant}(a, b, 0)$ returns $S_0(a, b), S_1(a, b)$ without computing $(S_{n_1}, S_{n_1-1}, S_{n_1-2}, \ldots, S_2)$, arrays $\mathcal{Q} = (q_1, \ldots, q_\ell)$, $\mathcal{R} = (r_\ell, r_{\ell-1})$, and $\mathcal{A}$. Therefore, any attempt to compute subresultants with higher indices can be addressed by utilizing the arrays $\mathcal{Q}, \mathcal{R}, \mathcal{A}$ instead of calling Half-GCD again. In the *Triangularize* algorithm for solving systems of polynomial equations by triangular decomposition, the *RegularGCD* subroutine relies on this technique for improved performance; see [2] and [4] for more details and algorithms.

Algorithm 3 SUBRESULTANT(a, b, ρ)

Input: $a, b \in \mathbf{k}[x] \smallsetminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$
Output: Subresultants $S_{n_v}(a,b)$, $S_{n_{v+1}}(a,b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$, the array $\mathcal{R}$ of the remainders, the array $\mathcal{Q}$ of the quotients and the array $\mathcal{A}$ of the leading coefficients of the remainders (in the Euclidean sequence) that have been computed so far

1: $\mathcal{A} := (0, \ldots, 0, \mathrm{lc}(a))$ where $\mathcal{A}[n_0] = \mathrm{lc}(a)$ and $\mathcal{A}[i] = 0$ for $0 \leq i < n_0$
2: **if** $\rho \geq n_1$ **then**
3: $\quad \mathcal{A}[n_1] = \mathrm{lc}(b)$
4: $\quad$ **return** $\Big((a, \mathrm{lc}(b)^{m-n-1} b), (), (), \mathcal{A}\Big)$
5: $(v, \mathcal{Q}, M) :=$ ADAPTEDHGCD$(a, b, n_0 - \rho, \rho, \mathcal{A})$
6: *deduce* $\Big(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_v = \deg(r_v)\Big)$ *from* a, b and $\mathcal{Q}$.
7: $\begin{bmatrix} r_v \\ r_{v+1} \end{bmatrix} := M \begin{bmatrix} a \\ b \end{bmatrix}$; $\mathcal{R} := (r_v, r_{v+1})$; $n_{v+1} := \deg(r_{v+1})$
8: $\tau_v := 0$; $\tau_{v+1} := 0$; $\alpha := 1$
9: **for** j **from** 1 **to** $v - 1$ **do**
10: $\quad \tau_v := \tau_v + (n_{j-1} - n_v)(n_j - n_v)$
11: $\quad \tau_{v+1} := \tau_{v+1} + (n_{j-1} - n_{v+1})(n_j - n_{v+1})$
12: $\quad \alpha := \alpha \ \mathcal{A}[n_j]^{n_{j-1} - n_{j+1}}$
13: $\tau_{v+1} := \tau_{v+1} + (n_{v-1} - n_{v+1})(n_v - n_{v+1})$
14: $S_{n_v} := (-1)^{\tau_v} \alpha \ r_v$
15: $S_{n_{v+1}} := (-1)^{\tau_{v+1}} \alpha \ \mathcal{A}[n_v]^{n_{v-1} - n_{v+1}} \ r_{v+1}$
16: **return** $\Big((S_{n_v}, S_{n_{v+1}}), \mathcal{Q}, \mathcal{R}, \mathcal{A}\Big)$

Algorithm 4 SUBRESULTANT$(a, b, \rho, \mathcal{Q}, \mathcal{R}, \mathcal{A})$

Input: $a, b \in \mathbf{k}[x] \smallsetminus \{0\}$ with $n_0 = \deg(a) \geq n_1 = \deg(b)$, $0 \leq \rho \leq n_0$, the list $\mathcal{Q}$ of all the quotients in the Euclidean sequence, the list $\mathcal{R}$ of the remainders that have been computed so far; we assume that $\mathcal{R}$ contains at least $r_\mu, \ldots r_{\ell-1}, r_\ell$ with $0 \leq \mu \leq \ell - 1$, and the list $\mathcal{A}$ of the leading coefficients of the remainders in the Euclidean sequence
Output: Subresultants $S_{n_v}(a,b)$, $S_{n_{v+1}}(a,b)$ for such $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$; the list $\mathcal{R}$ of the remainders is updated in-place

1: *deduce* $\Big(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_\ell = \deg(r_\ell)\Big)$ *from* a, b and $\mathcal{Q}$
2: **if** $n_\ell \leq \rho$ **then** $\ v := \ell$
3: **else** *find* $0 \leq v < \ell$ *such that* $n_{v+1} \leq \rho < n_v$.
4: **if** $v = 0$ **then**
5: $\quad$ **return** $\Big(a, \mathrm{lc}(b)^{m-n-1} b\Big)$
6: **for** i **from** $\max(v, \mu + 1)$ **down to** v **do**
7: $\quad r_i := r_{i+1} q_{i+1} + r_{i+2}$; $\mathcal{R} := \mathcal{R} \cup (r_i)$
8: *compute* $S_{n_v}, S_{n_{v+1}}$ *using Proposition* 1 *from* r_v, r_{v+1}
9: **return** $\Big(S_{n_v}, S_{n_{v+1}}\Big)$

For polynomials $a, b \in \mathbb{Z}[y]$ with integer coefficients, a modular algorithm can be achieved by utilizing the *Chinese remainder theorem* (CRT). In this approach,

we use Algorithms 2 and 3 for a prime field $\mathbf{k}$. We define $\mathbb{Z}_p[y]$ as the ring of univariate polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, for some prime p. Further, we use an iterative and probabilistic approach to CRT from [19]. We iteratively calculate subresultants modulo different primes $p_0, p_1, \ldots$, continuing to add modular images to the CRT direct product $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_i}$ for $i \in \mathbb{N}$ until the reconstruction *stabilizes*. That is to say, the reconstruction does not change from $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_{i-1}}$ to $\mathbb{Z}_{p_0} \otimes \cdots \otimes \mathbb{Z}_{p_i}$.

We further exploit this technique to compute subresultants of bivariate polynomials over prime fields and the integers. Let $a, b \in \mathbb{B}[y]$ be polynomials with coefficients in $\mathbb{B} = \mathbb{Z}_p[x]$, thus $\mathbb{B}[y] = \mathbb{Z}_p[x, y]$, where the main variable is y and $p \in \mathbb{N}$ is an odd prime. A desirable subresultant algorithm then uses an evaluation-interpolation scheme and the aforementioned univariate routines to compute subresultants of univariate images of a, b over $\mathbb{Z}_p[y]$ and then interpolates back to obtain subresultants over $\mathbb{Z}_p[x, y]$. This approach is well-studied in [19] to compute the resultant of bivariate polynomials. We can use the same technique to compute the entire subresultant chain, or even particular subresultants via Half-GCD through Algorithms 2 and 3.

We begin with choosing a set of evaluation points of size N and evaluate each coefficient of $a, b \in \mathbb{Z}_p[x, y]$ with respect to the main variable (y). Then, we call the subresultant algorithm to compute subresultants images over $\mathbb{Z}_p[y]$. Finally, we can retrieve the bivariate subresultants by interpolating each coefficient of each subresultant from the images. The number of evaluation points is determined from an upper-bound on the degree of subresultants and resultants with respect to x. From [25], the following inequality holds,

$$N \geq \deg(b, y) \deg(a, x) + \deg(a, y) \deg(b, x) + 1.$$

For bivariate polynomials with integer coefficients, we can use the CRT algorithm in a similar manner to that which has already been reviewed for univariate polynomials over $\mathbb{Z}$. Figure 1 demonstrates this procedure for two polynomials $a, b \in \mathbb{Z}[x, y]$. In this commutative diagram, $\bar{a}, \bar{b}$ represent the modular images of the polynomials a, b modulo prime p_i for $0 \leq i \leq e$.

$$\begin{array}{ccc}
a, b \in \mathbb{Z}[x, y] & \xrightarrow{\text{Algorithm 1}} & \mathrm{subres}(a, b, y) \in \mathbb{Z}[x, y] \\
\Big\downarrow \text{modulo } p_0, p_1, \ldots, p_i & & CRT \Big\uparrow \\
\bar{a}, \bar{b} \in \mathbb{Z}_{p_i}[x, y] & & \mathrm{subres}(\bar{a}, \bar{b}, y) \in \mathbb{Z}_{p_i}[x, y] \\
\Big\downarrow \text{Evaluate at } t_0, \ldots, t_N & & \text{Interpolate at } x \Big\uparrow \\
\bar{a}(x, y)|_{x=t_i}, \bar{b}(x, y)|_{x=t_i} \in \mathbb{Z}_{p_i}[y] & \xrightarrow{\text{Algorithm 3}} & \mathrm{subres}(\bar{a}(x, y)|_{x=t_i}, \bar{b}(x, y)|_{x=t_i}, y) \in \mathbb{Z}_{p_i}[y]
\end{array}$$

Fig. 1: Computing the subresultant chain of $a, b \in \mathbb{Z}[x, y]$ using modular arithmetic, evaluation-interpolation and CRT algorithms where $(t_0, \ldots, t_N)$ is the list of evaluation points, $(p_0, \ldots, p_i,)$ is the list of distinct primes, and $\bar{a} = a \bmod p_i$ and $\bar{b} = b \bmod p_i$.

4 Optimized Ducos' Subresultant Chain

In [9], L. Ducos examines optimizations to the classic subresultant chain algorithm seen previously in Algorithm 1. The first optimization, attributed to D. Lazard, shows that it is possible to avoid the expensive exponentiations and their division on Line 11 of Algorithm 1. This second optimization considers the pseudo-remainder equation of Line 15. Applying both improvements to Algorithm 1 yields an efficient subresultant chain procedure that is known as Ducos' algorithm.

Algorithm 5 Ducos Optimization (S_d, S_{d-1}, S_e, s_d)

Input: Given $S_d, S_{d-1}, S_e \in \mathbb{B}[y]$ and $s_d \in \mathbb{B}$
Output: S_{e-1}, the next subresultant in the subresultant chain of $\text{subres}(a, b)$
1: $(d, e) := (\deg(S_d), \deg(S_{d-1}))$
2: $(c_{d-1}, s_e) := (\text{lc}(S_{d-1}), \text{lc}(S_e))$
3: **for** $j = 0, \ldots, e-1$ **do**
4: $\quad H_j := s_e y^j$
5: $H_e := s_e y^e - S_e$
6: **for** $j = e+1, \ldots, d-1$ **do**
7: $\quad H_j := yH_{j-1} - \frac{\text{coeff}(yH_{j-1}, e) S_{d-1}}{c_{d-1}}$
8: $D := \frac{\sum_{j=0}^{d-1} \text{coeff}(S_d, j) H_j}{\text{lc}(S_d)}$
9: **return** $(-1)^{d-e+1} \frac{c_{d-1}(yH_{d-1}+D) - \text{coeff}(yH_{d-1}, e) S_{d-1}}{s_d}$

The Ducos optimization that is presented in Algorithm 5, and borrowed from [9], is a well-known improvement of Algorithm 1 to compute subresultant S_{e-1} on Line 15. This optimization provides a faster procedure to compute the pseudo-division of two successive subresultants, namely $S_d, S_{d-1} \in \mathbb{B}[y]$, and one division by a power of $\text{lc}(S_d)$. The main part of this algorithm is for-loops to compute:

$$D := \frac{\sum_{j=0}^{d-1} \text{coeff}(S_d, j) H_j}{\text{lc}(S_d)},$$

where $\text{coeff}(S_d, j)$ is the coefficient of S_d in y^j.

We now introduce a new algorithmic optimization for this algorithm to make better use of memory resources through in-place arithmetic. This is shown in Algorithm 6. In this algorithm we also extract the leading coefficient and the tail, that is, the reductum of a polynomial with respect to its main variable and denoted as tail, of S_d, S_{d-1}, and S_e in-place with a procedure named InplaceTail. This operation is essentially a coefficient shift. In this way, we reuse existing memory allocations for the tails of polynomials S_d, S_{d-1}, and S_e.

Algorithm 6 *Cache Friendly* **Ducos Optimization** (S_d, S_{d-1}, S_e, s_d)

Input: $S_d, S_{d-1}, S_e \in \mathbb{B}[y]$ and $s_d \in \mathbb{B}$
Output: S_{e-1}, the next subresultant in the subresultant chain of subres(a, b)
1: *Convert* p to a recursive representation format in-place
2: $(p, c_d) := \text{InplaceTail}(S_d)$
3: $(q, c_{d-1}) := \text{InplaceTail}(S_{d-1})$
4: $(h, s_e) := \text{InplaceTail}(S_e)$
5: $h := -h$; $a := \text{coeff}(p, e)\ h$
6: **for** $i = e+1, \ldots, d-1$ **do**
7:     **if** $\deg(h) = e - 1$ **then**
8:         $h := y\ \text{tail}(h) - \text{ExactQuotient}(\text{lc}(h)\ q, c_{d-1})$
9:     **else** $h := y\ \text{tail}(h)$
10:     $a := a + \text{lc}(\text{coeff}(p, i))\ h$
11: $a := a + s_e\ \sum_{i=0}^{e-1} \text{coeff}(p, i) y^i$
12: $a := \text{ExactQuotient}(a, c_d)$
13: **if** $\deg(h) = e - 1$ **then**
14:     $a := c_{d-1}\ (y\ \text{tail}(h) + a) - \text{lc}(h)\ q$
15: **else** $a := c_{d-1}\ (y\ h + a)$
16: **return** $(-1)^{d-e+1}\ \text{ExactQuotient}(a, s_d)$

Furthermore, we reduce the cost of calculating $\sum_{j=e}^{d-1} \text{coeff}(S_d, j) H_j$ with computing the summation iteratively and in-place in the same for-loop that is used to update polynomial h. (lines 6-10 in Algorithm 6). We also update the value of h depending on its degree with respect to y as $\deg(h) \le e - 1$ for all $e + 1 \le i < d$. We utilize an optimized exact division algorithm denoted by ExactQuotient to compute quotients rather a classical Euclidean algorithm.

5 Implementation and Experimentation

In this section, we discuss the implementation of core routines and subresultant schemes implemented in the Basic Polynomial Algebra Subprograms (BPAS) library [1] and compare the performance of these routines with the NTL library [23] and Maple 2020 [18]. Throughout this section, we collect our benchmarks on a machine running Ubuntu 18.04.4 and NTL 11.4.3 with an Intel Xeon X5650 processor running at 2.67GHz, with 12×4GB DDR3 memory at 1.33 GHz.

For basic arithmetic over a prime field $\mathbb{Z}_p$ where p is an odd prime, Montgomery multiplication, originally presented in [20], is used to speed up multiplication. This method avoids division by the modulus without any effect on the performance of addition, and so, yields faster modular inverse and division algorithms.

We have developed a dense representation of univariate polynomials which take advantage of Montgomery arithmetic (following the implementation in [5]) for prime fields with $p < 2^{64}$. Figures 2, 3, 4, and 5 examine the performance of univariate polynomial arithmetic over Z_p for two randomly generated dense

polynomials a, b and 64-bit prime $p = 4179340454199820289$. These plots compare the various implementation within BPAS against NTL.

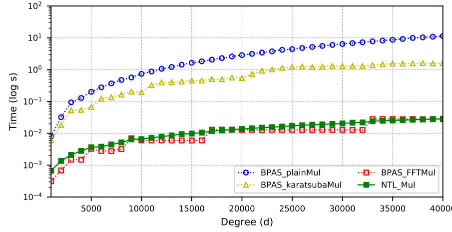

Fig. 2: Comparing plain (`BPAS_plainMul`), Karatsuba (`BPAS_karatsubaMul`), and FFT-based (`BPAS_FFTMul`) multiplication algorithms in BPAS with the wrapper **`mul`** method in NTL to compute ab for polynomials $a, b \in \mathbb{Z}_{4179340454199820289}[y]$ with $\deg(a) = \deg(b) + 1 = d$.

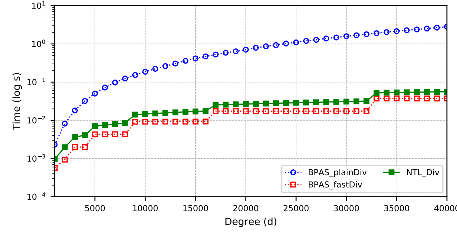

Fig. 3: Comparing Euclidean (`BPAS_plainDiv`) and Power Series division (`BPAS_fastDiv`) algorithms in BPAS with the division wrapper method in NTL to compute $\text{rem}(a, b)$ and $\text{quo}(a, b)$ for polynomials $a, b \in \mathbb{Z}_{4179340454199820289}[y]$ with $\deg(a) = 2(\deg(b) - 1) = d$.

Multiplication over $\mathbb{Z}_p[y]$ dynamically chooses the appropriate algorithm based on the input polynomials: plain and Karatsuba algorithms (following the routines in [25, Chapter 8]), or multiplication based on fast Fourier transform (FFT). The implementation of the FFT itself follows that which was introduced in [6]. Figure 2 shows the performance of these routines in BPAS against a similar "wrapper" multiplication routine in NTL for two randomly generated dense polynomials in $\mathbb{Z}_p[y]$.

For polynomials a, b over $\mathbb{Z}_p[y]$ in BPAS, plain and Karatsuba algorithms are performed when $s := \min(\deg(a), \deg(b)) < 200$. The Karatsuba algorithm is also called when $s \geq 200$ and $p < 2^{63}$. For 64-bit primes ($p > 2^{63}$) plain and Karatsuba algorithms are called for $s < 10$ and $s < 40$ respectively, otherwise FFT-based multiplication is performed. See [6] for more details about the FFT algorithm and supported primes.

The division operation is again a wrapper function, dynamically choosing between Euclidean (plain) and fast division algorithms. The fast algorithm is an optimized power series inversion procedure that is firstly implemented in **`Aldor`** [10] using the middle-product trick. Figure 3 shows the performance of these two algorithms in comparison with the NTL division over $\mathbb{Z}_p[y]$. For polynomials a, b over $\mathbb{Z}_p[y]$ in BPAS, the plain division is called for primes $p < 2^{63}$ and $\deg(b) < 1000$. However, for 64-bit primes and thus FFT support, the plain algorithm is only performed when $\deg(b) < 100$.

Our GCD operation over $\mathbb{Z}_p[y]$ derives from both the classical EEA and Half-GCD (fast EEA) algorithms following the pseudo-codes in [25, Chapter 11] and the implementation in the NTL library [23]. Figure 4 shows the performance of these two approaches named `BPAS_plainGCD` and `BPAS_fastGCD`, respectively,

in comparison with the NTL GCD algorithm for polynomials $a, b \in \mathbb{Z}_p[y]$ where $\gcd(a, b) = 1$.

To Analyze the performance of subresultant schemes of two polynomials $a, b \in \mathbb{Z}_p[y]$, we compare the naïve EEA algorithm with the modular subresultant chain and the speculative subresultant algorithm using Half-GCD technique for $\rho = 0, 2$ in Figure 5. As this figure shows, using Half-GCD algorithm to compute two successive subresultants S_1, S_0 for $\rho = 0$ is approximately $5\times$ faster than computing the entire chain, while calculating other subresultants, e.g. S_3, S_2 for $\rho = 2$ using the cached information, is nearly instantaneous.

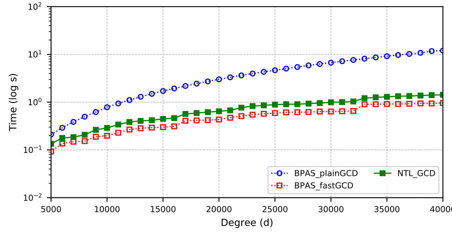

Fig. 4: Comparing Euclidean based GCD (`BPAS_plainGCD`) and Half-GCD based GCD (`BPAS_fastGCD`) algorithms in BPAS with the GCD algorithm in NTL to compute $\gcd(a, b) = 1$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$.

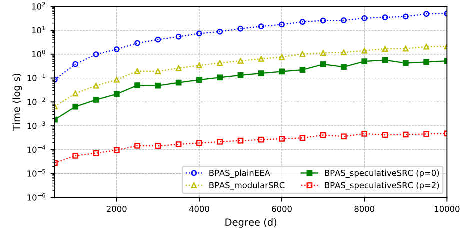

Fig. 5: Comparing EEA (`BPAS_plainEEA`), modular subresultant (`BPAS_modularSRC`), and Half-GCD based subresultant (`BPAS_speculativeSRC`) for $\rho = 0, 2$, algorithms in BPAS for dense polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$.

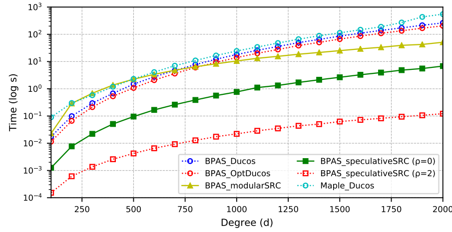

Fig. 6: Comparing (optimized) Ducos' subresultant chain algorithm, modular subresultant chain (`BPAS_modularSRC`), and Half-GCD based subresultant (`BPAS_speculativeSRC`), computing two successive subresultants for $\rho = 0, 2$, algorithms in BPAS with Ducos' subresultant chain algorithm in MAPLE for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$.

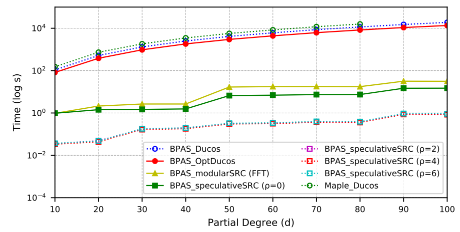

Fig. 7: Comparing (optimized) Ducos' subresultant chain, `BPAS_modularSRC` using FFT for evaluation-interpolation, and `BPAS_speculativeSRC` for $\rho = 0, 2, 4, 6$, algorithms in BPAS with Ducos' subresultant chain algorithm in MAPLE for dense polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = 50$ and $\deg(a, x) = \deg(b, x) + 1 = d$.

To develop a dense representation of univariate polynomials over arbitrary-precision integers, we use the low-level procedures of the GNU Multiple Precision Arithmetic library (GMP) [11]. Basic dense arithmetic operations, like addition, multiplication, and division, follows [25]. Moreover, we develop a family of subresultant routines including: `BPAS_modularSRC`, that computes the entire subresultant chain using Proposition 1 and the CRT algorithm, and `BPAS_speculativeSRC` that refers to Algorithms 3 and 4 to compute two successive subresultants using Half-GCD and caching techniques.

Figure 6 compares the running time of different subresultant schemes in the BPAS library and MAPLE. In this figure, `BPAS_Ducos` and `BPAS_OptDucos` denote the implementation of Algorithm 1 using Ducos' optimization (Algorithm 5) and our memory-efficient optimization (Algorithm 6), respectively. The modular approach is up to 5× faster than the optimized Ducos' algorithm. Using speculative algorithms to compute only two successive subresultants yields a speedup factor of 7 for $d = 2000$.

We further compare our routines with the Ducos subresultant chain algorithm in MAPLE, which is implemented as part of the *RegularChains* library [16]. Table 1 shows the memory usage for computing the entire subresultant chain of polynomials $a, b \in \mathbb{Z}[y]$, with $\deg(a) = \deg(b) + 1 = d$. The table presents `BPAS_Ducos` (based on Algorithm 5), `BPAS_OptDucos` (based on Algorithm 6), and `Maple_Ducos`. For $d = 2000$, results show that the optimized Ducos subresultant chain algorithms are using approximately 11× and 3× less memory than the original Ducos algorithm in MAPLE.

Degree	`BPAS_Ducos`	`BPAS_OptDucos`	`Maple_Ducos`
1000	1.088	0.320	3.762
1100	1.450	0.430	5.080
1200	1.888	0.563	6.597
1300	2.398	0.717	8.541
1400	2.968	0.902	10.645
1500	3.655	1.121	12.997
1600	4.443	1.364	15.924
1700	5.341	1.645	19.188
1800	6.325	1.958	23.041
1900	7.474	2.332	27.353
2000	8.752	2.721	31.793

Table 1: Comparing memory usage (GB) of Ducos' subresultant chain algorithms for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b) + 1 = d$ in Figure 6 over $\mathbb{Z}[y]$.

Bivariate polynomials have also been implemented using a dense representation. A bivariate polynomial $a \in \mathbb{Z}[x, y]$ (or $\mathbb{Z}_p[x, y]$ for a prime p) is stored as a list of coefficients of a, possibly including zeros. Basic arithmetic follows [25] and

subresultant algorithms follow the techniques of [19] and the previous discussion in Section 3.

Figure 7 provides a favourable comparison between the family of subresultant schemes in BPAS and the subresultant algorithm in MAPLE for dense bivariate polynomials $a, b \in \mathbb{Z}[x, y]$ where the main degree is fixed to 50, i.e. $\deg(a, y) = \deg(b, y) + 1 = 50$, and $\deg(a, x) = \deg(b, x) + 1 = d$ for $d \in \{10, 20, \ldots, 100\}$. Note that the BPAS_speculativeSRC algorithm for $\rho = 0, 2, 4, 6$ is caching the information for the next call with taking advantage of Algorithm 4.

We next compare the two main ways of computing an subresultant chain: a naïve approach using Algorithm 1, and a modular approach using evaluation-interpolation and CRT (Figure 1). Figure 8 shows the performance of the naïve approach (the top surface), with calling Optimized Ducos' algorithm, in comparison with the modular approach (the bottom surface). Note that, in this figure, interpolation may be based on Lagrange interpolation or FFT algorithms depending on the degrees of the input polynomials. Figure 9 compares the running time of the FFT-based modular algorithm (the top surface) against the speculative subresultant scheme based on the Half-GCD technique (the bottom surface).

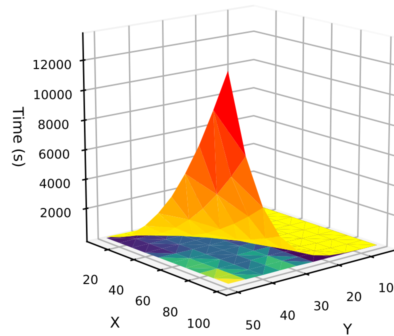

Fig. 8: Comparing Opt. Ducos' algorithm (the top surface) and modular subresultant chain (the bottom surface) to compute the entire chain for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$.

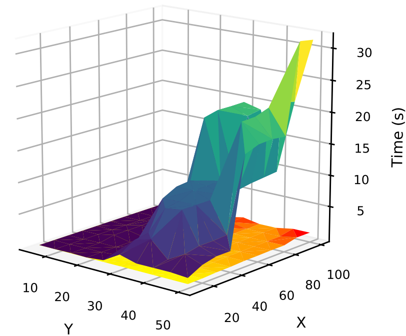

Fig. 9: Comparing modular subresultant chain with using FFT (the top surface), and HGCD-based subresultant ($\rho = 0$) (the bottom surface) for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$.

Tables 2, 3, and 4 investigate the performance of `BPAS_modularSRC` and `BPAS_speculativeSRC` schemes and the caching technique on the BPAS polynomial system solver; see [2, 4]. Table 2 shows the running time of well-known and challenging bivariate systems, forcing the solver to use only one particular subresultant scheme. Among those systems, the caching ratio ($\mathrm{SpecSRC_{naïve}}/\mathrm{SpecSRC_{cached}}$) of `vert_lines`, `L6_circles`, `ten_circles`, and `SA_2_4_eps` are $24.5, 21.6, 19.8, 9.2$, while the speculative ratio ($\mathrm{ModSRC_{cached}}/\mathrm{SpecSRC_{cached}}$) of `tryme`, `mignotte_xy`, and `vert_lines` are $1.5, 1.2$, and 1.2, respectively.

Tables 3 and 4 examine the performance of the polynomial system solver on constructed systems which aim to exploit the maximum speed-up of these new schemes. Listing 1.1 provides the Maple code to construct these input systems. For those systems, we get 3× speed-up through caching the intermediate speculative data rather than repeatedly calling the Half-GCD algorithm for each subresultant call. We got about a 1.5× speed-up with using `BPAS_speculativeSRC` instead of `BPAS_modularSRC` algorithm. Another family of constructed examples (created by Listing 1.2) is evaluated in Table 4. Here, we get up to 3× speed-up with the use of cached data, and up to 2× speed-up with using `BPAS_speculativeSRC` instead of `BPAS_modularSRC`.

n	SpecSRC$_{\text{naïve}}$	ModSRC$_{\text{cached}}$	SpecSRC$_{\text{cached}}$	deg(src[idx])	Indexes	FFTBlockSize
10	1.054	0.022	0.033	(0,5,10,15)	(0,6,11,15)	128
20	3.044	0.169	0.290	(0,10,20,30)	(0,11,21,30)	256
30	9.479	0.854	1.719	(0,15,30,45)	(0,16,31,45)	512
40	14.602	3.612	3.086	(0,20,40,60)	(0,21,41,60)	512
50	25.025	9.382	6.295	(0,25,50,75)	(0,26,51,75)	512
60	82.668	22.807	23.380	(0,30,60,90)	(0,31,61,90)	1024
70	105.253	23.593	30.477	(0,35,70,105)	(0,36,71,105)	1024
80	156.008	36.658	47.008	(0,40,80,120)	(0,41,81,120)	1024
90	236.960	133.500	73.552	(0,45,90,135)	(0,46,91,135)	1024
100	272.939	171.213	83.966	(0,50,100,150)	(0,51,101,150)	1024
110	370.628	280.952	117.106	(0,55,110,165)	(0,56,111,165)	1024
120	1035.810	491.853	331.601	(0,60,120,180)	(0,61,121,180)	2048
130	1119.720	542.905	362.631	(0,65,130,195)	(0,66,131,195)	2048
140	1445.000	804.982	470.649	(0,70,140,210)	(0,71,141,210)	2048
150	1963.920	1250.700	639.031	(0,75,150,225)	(0,76,151,225)	2048

Table 3: Comparing the execution time (in seconds) of subresultant schemes on the BPAS *Triangularize* system solver for constructed bivariate systems in Listing 1.1 to exploit the brand-new schemes. We call optimized modular subresultant chain algorithms (FFT and Lagrange) in the ModSRC$_{\text{cached}}$ mode, and Half-GCD based subresultant algorithms in the SpecSRC$_{\text{naïve}}$ and SpecSRC$_{\text{cached}}$ modes. We do cache subresultant information for further calls in the ModSRC$_{\text{cached}}$ and SpecSRC$_{\text{cached}}$ modes; deg(src[idx]) shows a list of the minimum main degrees of computed subresultants in each subresultant call and `Indexes` indicates a list of requested subresultant indexes. In addition, `FFTBlockSize` is the size of blocks used in the FFT-based evaluation and interpolation algorithms.

SysName	OptDucos	$\texttt{SpecSRC}_{\text{naïve}}$	$\texttt{ModSRC}_{\text{cached}}$	$\texttt{SpecSRC}_{\text{cached}}$	deg(src[idx])	Indexes
13_sings_9	3.417	3.465	3.416	3.408	(1)	(0)
compact_surf	10.258	26.702	11.257	10.26	(0,2,4,6)	(0,3,5,6)
curve24	4.912	4.924	4.992	4.911	(0,0,1)	(0,0,0)
curve_issac	2.528	2.541	2.554	2.531	(0,0,1)	(0,0,0)
cusps_and_flexes	4.488	8.374	4.656	4.656	(0,. . . ,2)	(0,. . . ,2)
degree_6_surf	344.564	224.215	81.887	79.394	(0,2,4,4)	(0,2,4,4)
hard_one	175.847	197.283	48.359	47.213	(0,. . . ,2)	(0,. . . ,2)
huge_cusp	23.406	33.501	23.406	23.41	(0,2,2)	(0,2,2)
L6_circles	32.347	721.49	32.906	33.422	(0,. . . ,6)	(0,. . . ,6)
large_curves	366.432	64.07	65.353	63.018	(0,0,1,1)	(0,0,0,0)
mignotte_xy	462.432	288.214	348.406	287.248	(1)	(0)
SA_2_4_eps	4.123	37.937	4.141	4.122	(0,. . . ,6)	(0,. . . ,6)
SA_4_4_eps	197.816	584.318	222.825	216.065	(0,. . . ,3)	(0,. . . ,6)
spider	293.543	294.121	293.701	295.198	(0,0,1,1)	(0,0,0,0)
spiral29_24	643.414	643.88	647.469	644.379	(1)	(0)
ten_circles	2.116	56.655	3.255	2.862	(0,. . . ,4)	(0,. . . ,4)
tryme	4893.04	4038.539	3728.085	2415.28	(0,2)	(0,2)
vert_lines	1.021	24.956	1.217	1.02	(0,. . . ,6)	(0,. . . ,6)

Table 2: Comparing the execution time (in seconds) of subresultant schemes on the BPAS *Triangularize* solver for well-known bivariate systems in the literature. We call optimized Ducos' subresultant chain algorithm in the `OptDucos` mode, modular subresultant chain algorithms (FFT and Lagrange) in the $\texttt{ModSRC}_{\text{cached}}$ mode, and Half-GCD based subresultant algorithms in the $\texttt{SpecSRC}_{\text{naïve}}$ and $\texttt{SpecSRC}_{\text{cached}}$ modes. We do cache subresultant information for further calls in the $\texttt{ModSRC}_{\text{cached}}$ and $\texttt{SpecSRC}_{\text{cached}}$ modes; deg(src[idx]) shows a list of minimum main degrees of the computed subresultants in each subresultant call and `Indexes` indicates a list of requested subresultant indexes.

n	$\texttt{SpecSRC}_{\text{naïve}}$	$\texttt{ModSRC}_{\text{cached}}$	$\texttt{SpecSRC}_{\text{cached}}$	deg(src[idx])	Indexes	FFTBlockSize
10	0.031	0.016	0.016	(0,2,2)	(0,2,2)	64
20	1.544	0.909	0.898	(0,2,2)	(0,2,2)	128
30	14.437	12.343	9.182	(0,2,2)	(0,2,2)	256
40	114.730	39.091	32.920	(0,2,2)	(0,2,2)	256
50	171.517	76.726	52.758	(0,2,2)	(0,2,2)	256
60	239.964	103.391	71.991	(0,2,2)	(0,2,2)	256
70	410.158	245.385	126.265	(0,2,2)	(0,2,2)	512
80	651.649	361.670	206.261	(0,2,2)	(0,2,2)	512
90	977.646	625.580	348.674	(0,2,2)	(0,2,2)	512
100	1467.510	894.139	474.241	(0,2,2)	(0,2,2)	512
110	2076.920	1259.850	675.806	(0,2,2)	(0,2,2)	512
120	2757.390	1807.060	963.547	(0,2,2)	(0,2,2)	512
130	4311.990	2897.150	1505.080	(0,2,2)	(0,2,2)	1024
140	5881.640	4314.300	2134.190	(0,2,2)	(0,2,2)	1024
150	7869.700	5177.410	2609.170	(0,2,2)	(0,2,2)	1024

Table 4: Comparing the execution time (in seconds) of subresultant schemes on the BPAS *Triangularize* system solver for constructed bivariate systems in Listing 1.2 to exploit the brand-new schemes. We call optimized modular subresultant chain algorithms (FFT and Lagrange) in the $\texttt{ModSRC}_{\text{cached}}$ mode, and Half-GCD based subresultant algorithms in the $\texttt{SpecSRC}_{\text{naïve}}$ and $\texttt{SpecSRC}_{\text{cached}}$ modes. We do cache subresultant information for further calls in the $\texttt{ModSRC}_{\text{cached}}$ and $\texttt{SpecSRC}_{\text{cached}}$ modes; deg(src[idx]) shows a list of the minimum main degrees of computed subresultants in each subresultant call and `Indexes` indicates a list of requested subresultant indexes. In addition, `FFTBlockSize` is the size of blocks used in the FFT-based evaluation and interpolation algorithms

```
SystemGenerator1 := proc(n)
  local R := PolynomialRing([x,y]);
  local J := PolynomialIdeals:-Intersect(<x^2+1,xy+2>,
        <x^2+3,xy^floor(n/2)+floor(n/2)+1>);
  J := PolynomialIdeals:-Intersect(J, <x^2+3,xy^n+n+1>);
  local dec := Triangularize(Generators(J),R);
  dec := map(NormalizeRegularChain,dec,R);
  dec := EquiprojectableDecomposition([%[1][1],%[2][1]],R);
  return map(expand, Equations(op(dec),R));
end proc:
```

Listing 1.1: Maple code of constructed polynomials in Table 3.

```
SystemGenerator2 := proc(n)
  local R := PolynomialRing([x,y]);
  local f := randpoly([x],dense,coeffs=rand(-1..1),degree=n);
  local J := <f,xy+2>;
  J := PolynomialIdeals:-Intersect(J,<x^2+2,(x^2+3x+1)y^2+3>)
  local dec := Triangularize(Generators(J),R);
  dec := map(NormalizeRegularChain,dec,R);
  dec := EquiprojectableDecomposition([%[1][1],%[2][1]],R);
  return map(expand,Equations(op(dec),R));
end proc:
```

Listing 1.2: Maple code of constructed polynomials in Table 4.

Acknowledgements

The authors would like to thank Robert H. C. Moir and NSERC of Canada (award CGSD3-535362-2019).

References

[1] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, Lin-Xiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS), 2021. `http://www.bpaslib.org`.

[2] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. On the parallelization of triangular decompositions. In Ioannis Z. Emiris and Lihong Zhi, editors, *ISSAC '20: International Symposium on Symbolic and Algebraic Computation, Kalamata, Greece, July 20-23, 2020*, pages 22–29. ACM, 2020.

[3] Eberhard Becker, Teo Mora, Maria Grazia Marinari, and Carlo Traverso. The shape of the shape lemma. In Malcolm A. H. MacCallum, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC '94, Oxford, UK, July 20-22, 1994*, pages 129–133. ACM, 1994.

[4] Changbo Chen and Marc Moreno Maza. Algorithms for computing triangular decomposition of polynomial systems. *J. Symb. Comput.*, 47(6):610–642, 2012.

[5] Svyatoslav Covanov, Davood Mohajerani, Marc Moreno Maza, and Linxiao Wang. Big prime field fft on multi-core processors. In *Proceedings of the 2019 on International Symposium on Symbolic and Algebraic Computation*, pages 106–113, 2019.

[6] Svyatoslav Covanov and Marc Moreno Maza. Putting Fürer algorithm into practice. Technical report, 2014. `http://www.csd.uwo.ca/~moreno//Publications/Svyatoslav-Covanov-Rapport-de-Stage-Recherche-2014.pdf`.

[7] Jean Della Dora, Claire Dicrescenzo, and Dominique Duval. About a new method for computing in algebraic number fields. In B. F. Caviness, editor, *EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 2: Research Contributions*, volume 204 of *Lecture Notes in Computer Science*, pages 289–290. Springer, 1985.

[8] Lionel Ducos. Algorithme de bareiss, algorithme des sous-résultants. *ITA*, 30(4):319–347, 1996.

[9] Lionel Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145(2):149–163, 2000.

[10] Akpodigha Filatei, Xin Li, Marc Moreno Maza, and Eric Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proceedings of the 2006 international symposium on Symbolic and algebraic computation*, pages 93–100, 2006.

[11] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.2.1 edition, 2020. `http://gmplib.org/`.

[12] M'hammed El Kahoui. An elementary approach to subresultants theory. *Journal of Symbolic Computation*, 35(3):281–292, 2003.

[13] Donald E. Knuth. The analysis of algorithms. *The Actes du Congrés International des Mathématiciens*, 3:269274, 1970.

[14] Grégoire Lecerf. On the complexity of the lickteig-roy subresultant algorithm. *J. Symb. Comput.*, 92:243–268, 2019.

[15] Derrick H. Lehmer. Euclid's algorithm for large numbers. *The American Mathematical Monthly*, 45(4):227–233, 1938.

[16] F. Lemaire, M. Moreno Maza, and Y. Xie. The regularchains library in MAPLE. *ACM SIGSAM Bulletin*, 39(3):96–97, 2005.

[17] Thomas Lickteig and Marie-Françoise Roy. Semi-algebraic complexity of quotients and sign determination of remainders. *J. Complex.*, 12(4):545–571, 1996.

[18] Maplesoft, a division of Waterloo Maple Inc. Maple 2020. `www.maplesoft.com/`.

[19] Michael B. Monagan. Probabilistic algorithms for computing resultants. In *ISSAC*, pages 245–252. ACM, 2005.

[20] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[21] Daniel Reischert. Asymptotically fast computation of subresultants. In Bruce W. Char, Paul S. Wang, and Wolfgang Küchlin, editors, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, ISSAC '97, Maui, Hawaii, USA, July 21-23, 1997*, pages 233–240. ACM, 1997.

[22] Arnold Schönhage. Schnelle berechnung von kettenbruchentwicklungen. *Acta Informatica 1*, page 139144, 1971.

[23] Victor Shoup et al. NTL: A library for doing number theory, 2021. `www.shoup.net/ntl/`.

[24] Klaus Thull and Chee K Yap. A uni ed approach to hgcd algorithms for polynomials and integers. 1990.

[25] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, NY, USA, 3 edition, 2013.