Computational Schemes for Subresultant Chains

Mohammadali Asadi(✉), Alexander Brandt, and Marc Moreno Maza

Department of Computer Science, University of Western Ontario, London, Canada
{masadi4,abrandt5}@uwo.ca, 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 [12, Chapter 8]. The *Half-GCD* algorithm originated in the ideas of [16,18] and [26], while a robust implementation was a challenge for many years. One of the earliest correct designs was introduced in [28].

The idea of speeding up subresultant chain computations by means of the *Half-GCD* algorithm takes various forms in the literature. In [25], 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 [20], Lickteig and Roy propose a "divide and conquer" algorithm for computing subresultant chains, the objective of which is to control coefficient growth

F. Boulier et al. (Eds.): CASC 2021, LNCS 12865, pp. 21–41, 2021.
https://doi.org/10.1007/978-3-030-85165-1_3

in defective cases. Lecerf in [17] introduces extensions and a complexity analysis of the algorithm of Lickteig and Roy, with a particular focus on bivariate polynomials. When run over an effective ring endowed with the partially defined division routine, the algorithm yields a running time estimate similar to that of Reischert's. Lecerf realized an implementation of that algorithm, but observed that computations of subresultant chains based on Ducos' algorithm [10], or on evaluation-interpolation strategies, were faster in practice.

In [12, Chapter 11], von zur Gathen and Gerhard show how the nominal leading coefficients (see Sect. 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 Sect. 3.

Our next optimization 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 $\Psi(\mathrm{lc}(b)) \neq 0$ both hold. Then, the specialization property of subresultants (see the precise statement in Sect. 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, say $\mathbb{B}$ is $\mathbb{Q}[x]/\langle m(x)\rangle$, where $m(x)$ is a square-free polynomial, then $\mathbb{B}$ is a product of fields then, letting $\mathbb{A}$ be $\mathbb{Q}[x]$, one can compute a GCD of $\Psi(a), \Psi(b)$ using the celebrated D5 Principle [8]. 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 [5]. 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, for some k, the nominal leading coefficients $s_0, \ldots, s_{k-1}$ are all zero modulo $\langle T\rangle$, and s_k is invertible modulo $\langle T\rangle$, then 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* [4]. 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 Sect. 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 schemes for the Half-GCD, inspired by the work done in NTL [27]. 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.

There are subresultant algorithms in almost all computer algebra software. Most notably, the *RegularChains* library [19] in MAPLE provides three different algorithms to compute the entire chain based on Ducos' optimization [9], Bézout matrix [1], or evaluation-interpolation based on FFT. Each one is well-suited for a particular type of input polynomials w.r.t the number of variables and the coefficients ring; see the MAPLE help page for `SubresultantChain` command. Similarly, the ALGEBRAMIX library in MATHEMAGIX [14] implements different subresultant algorithms, including routines based on evaluation-interpolation, Ducos' algorithm, and an enhanced version of Lickteig-Roy' s algorithm [17].

The extensive experimentation results in Sect. 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 present a third optimization for subresultant chain computations through a simple improvement of Ducos' subresultant chain algorithm. In particular, we consider memory usage and data locality to improve practical performance; see Sect. 4. We have implemented both the original Ducos algorithm [10] and our optimized version over arbitrary-precision integers. For univariate polynomials of degree as large as 2000, the optimized algorithm uses 3.2× and 11.7× less memory, respectively, than our implementation of the original Ducos' algorithm and the implementation of Ducos' algorithm 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 polynomial system solver [3].

This paper is organized as follows. Section 2 presents a review of subresultant theory following the presentations of [9] and [15]. Our modular method to compute

subresultants speculatively via Half-GCD is discussed in Sect. 3. Section 4 examines practical memory optimizations for Ducos' subresultant chain algorithm. Lastly, implementation details and experimental results are presented in Sect. 5.

2 Review of Subresultant Theory

In this review of subresultant theory, we follow the presentations of [9] and [15]. Let $\mathbb{B}$ be a commutative ring with identity and let $m \leq n$ 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 $m \leq i \leq n$; 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 [9]. 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 $(S_n, S_{n-1}, S_{n-2}, \ldots, S_0)$

if $m > n$ **then**
  $S := (\mathrm{lc}(b)^{m-n-1} b)$
else $S := ()$
$s := \mathrm{lc}(b)^{m-n}$
$A := b$; $B := \mathrm{prem}(a, -b)$
while true **do**
  $d := \deg(A)$; $e := \deg(B)$
  if $B = 0$ **then return** S
  $S := (B) \cup S$; $\delta := d - e$
  if $\delta > 1$ **then**
    $C := \frac{\mathrm{lc}(B)^{\delta-1} B}{s^{\delta-1}}$
    $S := (C) \cup S$
  else $C := B$
  if $e = 0$ **then return** S
  $B := \frac{\mathrm{prem}(A, -B)}{s^{\delta} \mathrm{lc}(A)}$
  $A := C$; $s := \mathrm{lc}(A)$
end while

Algorithm 1 from [10] 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 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 $r_{i+1} = \mathrm{rem}(r_i, r_{i-1}) = r_{i-1} - q_i r_i$, for $1 \leq i \leq \ell$, $q_i = \mathrm{quo}(r_i, r_{i-1})$ for $1 \leq i \leq \ell$, $n_{i+1} < n_i$, for $1 \leq i < \ell$, and $r_{\ell+1} = 0$ 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

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

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 $\tau_i = \sum_{1 \leq j < i} (n_{j-1} - n_i)(n_j - n_i)$ [12, Theorem 11.16].

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 [12, Chapter 11]. For a non-negative $k \leq n_0$, this algorithm computes the quotients $q_1, \ldots, q_{h_k}$ where h_k is defined as

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

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

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

Thus, $n_{h_k+1} < n_0 - k \leq n_{h_k}$, and so h_k can be uniquely determined; see Algorithm 11.6 in [12] 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 [12]. 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 $q_1, \ldots, q_{h_k}$, calculating the $\mathrm{lc}(r_i) = \mathrm{lc}(r_{i-1})/\mathrm{lc}(q_i)$ from $\mathrm{lc}(r_0)$ for $1 \leq i \leq h_k$, 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 $0 \leq \rho < n_1$, for $0 \leq v < \ell$ so that $n_{v+1} \leq \rho < n_v$, we need to compute quotients $q_1, \ldots, q_{h_\rho}$ where h_ρ is defined as

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

using Half-GCD. Let $k = n_0 - \rho$, Eqs. 3 and 4 deduce $n_{h_\rho+1} \leq n_0 - k < n_{h_\rho}$, and $h_\rho \leq h_k$. So, to compute the array of quotients $q_1, \ldots, q_{h_\rho}$, we can utilize an adaptation of the Half-GCD algorithm of [12]. Algorithm 2 is this adaptation and runs within the same complexity as the algorithm of [12].

Algorithm 2 receives as input two polynomials $r_0 := a, r_1 := b$ in $\mathbf{k}[y]$, with $n_0 \geq n_1$, $0 \leq k \in \mathbb{N}$, $\rho \leq n_0$ 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 $0 \leq i < n_0$. $\mathcal{A}$ is updated in-place as necessary. The algorithm returns the array of quotients $\mathcal{Q} := (q_1, \ldots, q_{h_\rho})$ and matrix $M := Q_{h_\rho} \cdots Q_1$.

Algorithm 2 and *the fundamental theorem on subresultants* yield Algorithm 3. This algorithm is a *speculative* subresultant algorithm based on Half-GCD to

Algorithm 2. 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 remainders 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: $\left(h', (q_1, \ldots, q_{h'}), R\right) := \text{AdaptedHGCD}(\mathrm{quo}(r_0, y^{\delta_1}), \mathrm{quo}(r_1, y^{\delta_1}), m_1 - 1, \rho, \mathcal{A})$

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: $\left(h^*, (q_{h'+2}, \ldots, q_{h'+h^*+1}), S\right) :=$
AdaptedHGCD$(\mathrm{quo}(d, y^{\delta_2}), \mathrm{quo}(r, y^{\delta_2}), \deg(d) - m_2, \rho, \mathcal{A})$

15: **return** $\left(h_\rho := h' + h^* + 1, \mathcal{Q} := (q_1, \ldots, q_{h_\rho}), M := SQR\right)$

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 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 Subresultant$(a, b, 0)$ returns $S_0(a, b), S_1(a, b)$ speculatively 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

Algorithm 3. SUBRESULTANT(a, b, ρ)

Input: $a, b \in \mathbf{k}[x] \setminus \{0\}$ 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 $\mathcal{Q}$ of the quotients, the array $\mathcal{R}$ of the remainders, 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** $\Big((S_{n_v}, S_{n_{v+1}}), \mathcal{Q}, \mathcal{R}, \mathcal{A}\Big)$

equations by triangular decomposition, the *RegularGCD* subroutine relies on this technique for improved performance; see [3,5] for more details and algorithms.

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 [22]. 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 [22] to compute the resultant of bivariate polynomials. We can use the same

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

Input: $a, b \in \mathbf{k}[x] \setminus \{0\}$ 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* $\left(n_0 = \deg(a), n_1 = \deg(b), \ldots, n_\ell = \deg(r_\ell)\right)$ *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: **return** $\left(a, \mathrm{lc}(b)^{m-n-1} b\right)$
6: **for** i **from** $\max(v, \mu + 1)$ **down to** v **do**
7: $r_i := r_{i+1} q_{i+1} + r_{i+2}$; $\mathcal{R} := \mathcal{R} \cup (r_i)$
8: *compute* $S_{n_v}, S_{n_{v+1}}$ *using Proposition* 1 *from* r_v, r_{v+1}
9: **return** $\left(S_{n_v}, S_{n_{v+1}}\right)$

technique to compute the entire subresultant chain, or even particular subresultants speculatively through Algorithms 2 and 3.

We begin with choosing a set of evaluation points of size $N \in \mathbb{N}$ 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 [12], 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 $0 \leq i \leq e$.

In practice, as the number of variables increases, the use of dense evaluation-interpolation schemes become less effective, since degree bound estimates become less sharp. In fact, sparse evaluation-interpolation schemes become more attractive [23,29], and we will consider them in future works.

4 Optimized Ducos' Subresultant Chain

In [10], Ducos proposes two optimizations for Algorithm 1. The first one, attributed to Lazard, deals with the potentially expensive exponentiations and division at Line 11 of Algorithm 1. The second optimizations considers the potentially expensive exact division (of a pseudo-remainder by an element from the

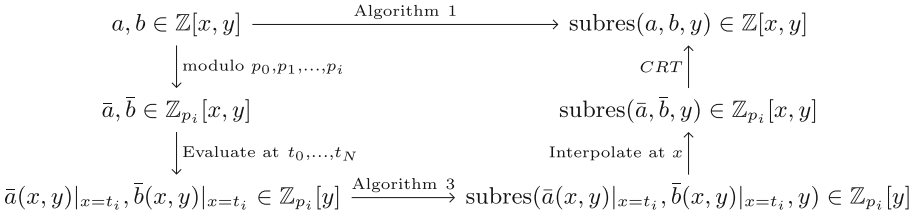

Fig. 1. Computing the subresultant chain of $a, b \in \mathbb{Z}[x, y]$ using modular arithmetic, evaluation-interpolation and CRT algorithms where $(t_0, \ldots, t_N)$ is the list of evaluation points, $(p_0, \ldots, p_i,)$ is the list of distinct primes, $\bar{a} = a \bmod p_i$, and $\bar{b} = b \bmod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

coefficient ring) at Line 15 of this algorithm. 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 $\mathrm{subres}(a, b)$

1: $(d, e) := (\deg(S_d), \deg(S_{d-1}))$
2: $(c_{d-1}, s_e) := (\mathrm{lc}(S_{d-1}), \mathrm{lc}(S_e))$
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: $\quad H_j := yH_{j-1} - \frac{\mathrm{coeff}(yH_{j-1}, e) S_{d-1}}{c_{d-1}}$
8: $D := \frac{\sum_{j=0}^{d-1} \mathrm{coeff}(S_d, j) H_j}{\mathrm{lc}(S_d)}$
9: **return** $(-1)^{d-e+1} \frac{c_{d-1}(yH_{d-1}+D) - \mathrm{coeff}(yH_{d-1}, e) S_{d-1}}{s_d}$

The Ducos optimization that is presented in Algorithm 5, and borrowed from [10], is a well-known improvement of Algorithm 1 to compute the subresultant S_{e-1} (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 a division by a power of $\mathrm{lc}(S_d)$. The main part of this algorithm is for-loops to compute:

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

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

We now introduce a new 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 use a procedure named InplaceTail to compute the tail (the reductum of a polynomial with respect to its main variable) of a polynomial, and its leading coefficient, in-place. 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. *memory-efficient* **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: $(p, c_d) :=$ InplaceTail(S_d)
2: $(q, c_{d-1}) :=$ InplaceTail(S_{d-1})
3: $(h, s_e) :=$ InplaceTail(S_e)
4: *Convert* p to a recursive representation format in-place
5: $h := -h$; $a :=$ coeff(p, e) h
6: **for** $i = e + 1, \ldots, d - 1$ **do**
7: **if** $\deg(h) = e - 1$ **then**
8: $h := y$ tail$(h) -$ ExactQuotient$(\text{lc}(h)\ q, c_{d-1})$
9: **else** $h := y$ 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: $a :=$ ExactQuotient(a, c_d)
13: **if** $\deg(h) = e - 1$ **then**
14: $a := c_{d-1}\ (y$ tail$(h) + a) - \text{lc}(h)\ q$
15: **else** $a := c_{d-1}\ (y\ h + a)$
16: **return** $(-1)^{d-e+1}$ 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). This greatly improves data locality. We also update the value of h depending on its degree with respect to y as $\deg(h) \leq e - 1$ for all $e + 1 \leq i < d$. 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 and performance of our various subresultant algorithms and their underlying core routines. Our methods are implemented as part of the Basic Polynomial Algebra Subprograms (BPAS) library [2] and we compare their performance against the NTL library [27] and Maple 2020 [21]. Throughout this section, our benchmarks were collected on a machine running Ubuntu 18.04.4, BPAS v1.791, GMP 6.1.2, and NTL 11.4.3, with an Intel Xeon X5650 processor running at 2.67 GHz, with 12×4GB DDR3 memory at 1.33 GHz.

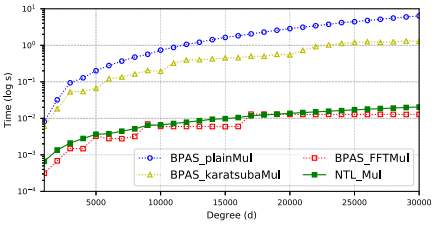

Fig. 2. Comparing plain, Karatsuba, and FFT-based multiplication in BPAS with the wrapper `mul` method in NTL to compute ab for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$

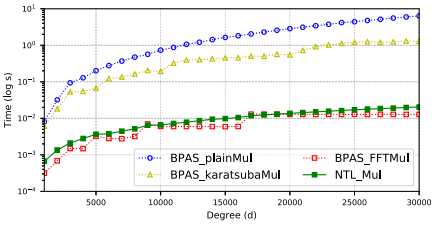

Fig. 3. Comparing Euclidean and fast division algorithms in BPAS with the division method in NTL to compute $\text{rem}(a, b)$ and $\text{quo}(a, b)$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = 2(\deg(b) - 1) = d$

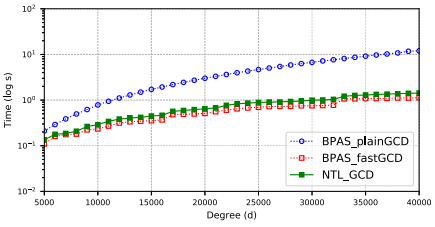

Fig. 4. Comparing Euclidean-based GCD and Half-GCD-based GCD algorithms in BPAS with the GCD algorithm in NTL to compute $\gcd(a, b) = 1$ for polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$

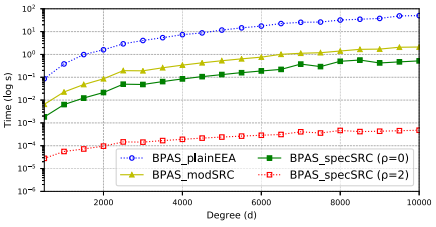

Fig. 5. Comparing EEA, modular subresultant, and Half-GCD-based subresultant (`BPAS_specSRC`, $\rho = 0, 2$), in BPAS for dense polynomials $a, b \in \mathbb{Z}_p[y]$ with $\deg(a) = \deg(b) + 1 = d$

5.1 Routines over $\mathbb{Z}_p[y]$

We begin with foundational routines for arithmetic in finite fields and polynomials over finite fields. For basic arithmetic over a prime field $\mathbb{Z}_p$ where p is an odd prime, Montgomery multiplication, originally presented in [24], 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 [6]) for prime fields with $p < 2^{64}$. Throughout this section we examine the performance of each operation for two randomly generated dense polynomials $a, b \in \mathbb{Z}_p$ with a 64-bit prime $p = 4179340454199820289$. Figures 2, 3, 4 and 5 examine, respectively, multiplication, division, GCD, and subresultant chain operations. These plots compare the various implementations within BPAS against NTL.

Our multiplication over $\mathbb{Z}_p[y]$ dynamically chooses the appropriate algorithm based on the input polynomials: plain or Karatsuba algorithms (following the

routines in [12, Chapter 8]), or multiplication based on fast Fourier transform (FFT). The implementation of FFT itself follows that which was introduced in [7]. Figure 2 shows the performance of these routines in BPAS against a similar "wrapper" multiplication routine in NTL. From empirical data, our wrapper multiplication function calls the appropriate implementation of multiplication as follows. For polynomials a, b over $\mathbb{Z}_p[y]$, with $p < 2^{63}$, the plain algorithm is called when $s := \min(\deg(a), \deg(b)) < 200$ and the Karatsuba algorithm is called when $s \geq 200$. For 64-bit primes ($p > 2^{63}$), plain and Karatsuba algorithms are called when $s < 10$ and $s < 40$, respectively, otherwise FFT-based multiplication is performed.

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` [11] using the so-called 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 $\mathbb{Z}_p[y]$, b the divisor, empirical data again guides the choice of appropriate implementation. Plain division is called for primes $p < 2^{63}$ and $\deg(b) < 1000$. However, for 64-bit primes, the plain algorithm is used when $\deg(b) < 100$, otherwise fast division supported by FFT is used.

Our GCD operation over $\mathbb{Z}_p[y]$ had two implementations: the classical extended Euclidean algorithm (EEA) and the Half-GCD (fast EEA) algorithm, respectively following the pseudo-codes in [12, Chapter 11] and the implementation in the NTL library [27]. 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 our subresultant schemes, we compare the naïve EEA algorithm with the modular subresultant chain and the speculative subresultant algorithm for $\rho = 0, 2$ in Fig. 5. As this figure shows, using the 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 $\rho = 2$ with taking advantage of the *cached* information from the first call (for $\rho = 0$), is nearly instantaneous.

5.2 Subresultants over $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$

We have developed a dense representation of univariate and bivariate polynomials over arbitrary-precision integers, using low-level procedures of the GNU Multiple Precision Arithmetic library (GMP) [13]. Basic dense arithmetic operations, like addition, multiplication, and division, follows [12]. The representation of a dense bivariate polynomial $a \in \mathbb{Z}[x, y]$ (or $\mathbb{Z}_p[x, y]$ for a prime p) is stored as a dense array of coefficients (polynomials in $\mathbb{Z}[x]$), possibly including zeros.

Following our previous discussion of various schemes for subresultants, we have implemented several subresultant algorithms over $\mathbb{Z}[y]$ and $\mathbb{Z}[x, y]$. We have four families of implementations:

(i) `BPAS_modSRC`, that computes the entire subresultant chain using Proposition 1 and the CRT algorithm (and evaluation-interpolation over $\mathbb{Z}[x, y]$);

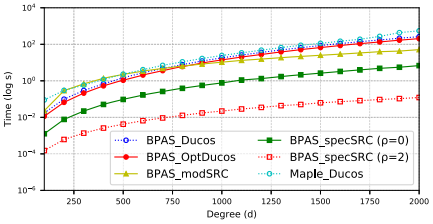

Fig. 6. Comparing (optimized) Ducos' subresultant chain algorithm, modular subresultant chain, and speculative subresultant for $\rho = 0, 2$, algorithms in BPAS with Ducos' subresultant chain algorithm in MAPLE for polynomials $a, b \in \mathbb{Z}[y]$ with $\deg(a) = \deg(b)+1 = d$

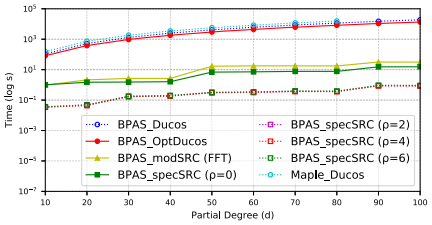

Fig. 7. Comparing (optimized) Ducos' subresultant chain, modular subresultant chain, and speculative subresultant for $\rho = 0, 2, 4, 6$, in BPAS with Ducos' algorithm in MAPLE for dense polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = 50$ and $\deg(a, x) = \deg(b, x) + 1 = d$

(*ii*) `BPAS_specSRC`, that refers to Algorithms 3 and 4 to compute two successive subresultants using Half-GCD and caching techniques;
(*iii*) `BPAS_Ducos`, for Ducos' algorithm, based on Algorithm 5; and
(*iv*) `BPAS_OptDucos`, for Ducos' algorithm based on Algorithm 6.

Figure 6 compares the running time of those subresultant schemes over $\mathbb{Z}[y]$ in the BPAS library and MAPLE. 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 $d = 2000$. 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_specSRC` algorithm for $\rho = 0, 2, 4, 6$ is caching the information for the next call with taking advantage of Algorithm 4.

We further compare our routines with the Ducos subresultant chain algorithm in MAPLE, which is implemented as part of the *RegularChains* library [19]. 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`, `BPAS_OptDucos`, and `Maple_Ducos`. For $d = 2000$, Table 1 shows that the optimized algorithm uses approximately 3× and 11× less memory than our original implementation and the Ducos' algorithm in MAPLE, respectively.

We next compare more closely the two main ways of computing an entire subresultant chain: the direct approach following Algorithm 1, and a modular approach using evaluation-interpolation and CRT (see Fig. 1). Figure 8 shows the performance of the direct approach (the top surface), calling our memory-optimized Ducos' algorithm `BPAS_OptDucos`, in comparison with the modular approach (the bottom surface), calling `BPAS_modSRC`. Note that, in this figure, interpolation may be based on Lagrange interpolation or FFT algorithms depending on the degrees of the input polynomials.

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 Fig. 6 over $\mathbb{Z}[y]$

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

Next, Fig. 9 highlights the benefit of our speculative approach to compute the resultant and subresultant of index 1 compared to computing the entire. The FFT-based modular algorithm is presented as the top surface, while the speculative subresultant algorithm based on the Half-GCD is the bottom surface.

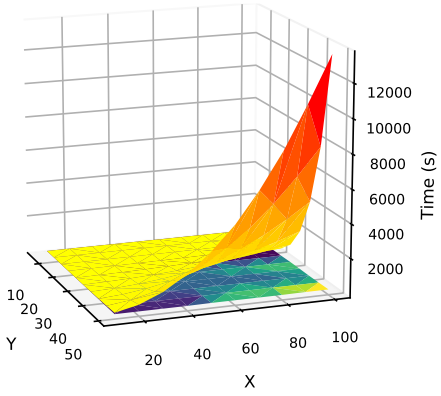

Fig. 8. Comparing Opt. Ducos' algorithm (the top surface) and modular subresultant chain (the bottom surface) to compute the entire chain for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y) + 1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$

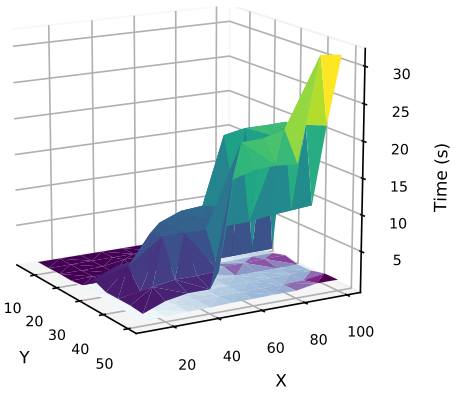

Fig. 9. Comparing modular subresultant chain with using FFT (the top surface), and speculative subresultant ($\rho = 0$) (the bottom surface) for polynomials $a, b \in \mathbb{Z}[x < y]$ with $\deg(a, y) = \deg(b, y)+1 = Y$ and $\deg(a, x) = \deg(b, x) + 1 = X$

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 `ModSRC` 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 `ModSRC` 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.

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

Lastly, we investigate the effects of different subresultant algorithms on the performance of the BPAS polynomial system solved based on triangular decomposition and regular chains; see [3,5]. Subresultants play a crucial role in computing regular GCDs (see Sect. 1) and thus in solving systems via triangular decomposition. Tables 2, 3, and 4 investigate the performance of `BPAS_modSRC`, and `BPAS_specSRC` and the caching technique, for system solving.

Table 2 shows the running time of well-known and challenging bivariate systems, where we have forced the solver to use only one particular subresultant scheme. In $\texttt{SpecSRC}_{\text{naïve}}$, `BPAS_specSRC` does not cache data and thus does not reuse the sequence of quotients computed from previous calls. Among those systems, the caching ratio ($\texttt{SpecSRC}_{\text{naïve}}$/$\texttt{SpecSRC}_{\text{cached}}$) of `vert_lines`, `L6_circles`, `ten_circles`, and `SA_2_4_eps` are 24.5, 21.6, 19.8, 9.2, respectively, while the speculative ratio ($\texttt{ModSRC}$/$\texttt{SpecSRC}_{\text{cached}}$) 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 and 1.2 in Appendix A provide the MAPLE code to construct these input systems. For those systems created by Listing 1.1, we get 3× speed-up through caching the intermediate speculative data rather than repeatedly calling the Half-GCD algorithm for each subresultant call. Using `BPAS_specSRC` provides a 1.5× speed-up over using the `BPAS_modSRC` 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 over the modular method.

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 speculative scheme. Column headings follow Table 2, and `FFTBlockSize` is block size used in the FFT-based evaluation and interpolation algorithms.

n	`ModSRC`	`SpecSRC`$_{\text{naïve}}$	`SpecSRC`$_{\text{cached}}$	deg(src[idx])	`Indexes`	`FFTBlockSize`
50	9.382	25.025	6.295	(0, 25, 50, 75)	(0, 26, 51, 75)	512
60	22.807	82.668	23.380	(0, 30, 60, 90)	(0, 31, 61, 90)	1024
70	23.593	105.253	30.477	(0, 35, 70, 105)	(0, 36, 71, 105)	1024
80	36.658	156.008	47.008	(0, 40, 80, 120)	(0,41,81,120)	1024
100	171.213	272.939	83.966	(0, 50, 100, 150)	(0, 51, 101, 150)	1024
110	280.952	370.628	117.106	(0, 55, 110, 165)	(0, 56, 111, 165)	1024
120	491.853	1035.810	331.601	(0, 60, 120, 180)	(0, 61, 121, 180)	2048
130	542.905	1119.720	362.631	(0, 65, 130, 195)	(0, 66, 131, 195)	2048
140	804.982	1445.000	470.649	(0, 70, 140, 210)	(0, 71, 141, 210)	2048
150	1250.700	1963.920	639.031	(0, 75, 150, 225)	(0, 76, 151, 225)	2048

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 speculative scheme. Column headings follow Table 3.

n	`ModSRC`	`SpecSRC`$_{\text{naïve}}$	`SpecSRC`$_{\text{cached}}$	deg(src[idx])	`Indexes`	`FFTBlockSize`
100	894.139	1467.510	474.241	(0, 2, 2)	(0, 2, 2)	512
110	1259.850	2076.920	675.806	(0, 2, 2)	(0, 2, 2)	512
120	1807.060	2757.390	963.547	(0, 2, 2)	(0, 2, 2)	512
130	2897.150	4311.990	1505.080	(0, 2, 2)	(0, 2, 2)	1024
140	4314.300	5881.640	2134.190	(0, 2, 2)	(0, 2, 2)	1024
150	5177.410	7869.700	2609.170	(0, 2, 2)	(0, 2, 2)	1024

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

A MAPLE code for Polynomial Systems

```
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.

References

1. Abdeljaoued, J., Diaz-Toca, G.M., González-Vega, L.: Bezout matrices, subresultant polynomials and parameters. Appl. Math. Comput. **214**(2), 588–594 (2009)
2. Asadi, M., et al.: Basic Polynomial Algebra Subprograms (BPAS) (version 1.791) (2021). http://www.bpaslib.org
3. Asadi, M., Brandt, A., Moir, R.H.C., Moreno Maza, M., Xie, Y.: Parallelization of triangular decompositions: techniques and implementation. J. Symb. Comput. (2021, to appear)
4. Becker, E., Mora, T., Grazia Marinari, M., Traverso, C.: The shape of the shape lemma. In: Proceedings of ISSAC 1994, pp. 129–133. ACM (1994)
5. Chen, C., Moreno Maza, M.: Algorithms for computing triangular decomposition of polynomial systems. J. Symb. Comput. **47**(6), 610–642 (2012)
6. Covanov, S., Mohajerani, D., Moreno Maza, M., Wang, L.: Big prime field FFT on multi-core processors. In: Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation (ISSAC), pp. 106–113. ACM (2019)
7. Covanov, S., Moreno Maza, M.: Putting Fürer algorithm into practice. Technical report (2014). http://www.csd.uwo.ca/~moreno//Publications/Svyatoslav-Covanov-Rapport-de-Stage-Recherche-2014.pdf

8. Della Dora, J., Dicrescenzo, C., Duval, D.: About a new method for computing in algebraic number fields. In: Caviness, B.F. (ed.) EUROCAL 1985. LNCS, vol. 204, pp. 289–290. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-15984-3_279
9. Ducos, L.: Algorithme de Bareiss, algorithme des sous-résultants. Informatique Théorique et Applications **30**(4), 319–347 (1996)
10. Ducos, L.: Optimizations of the subresultant algorithm. J. Pure Appl. Algebra **145**(2), 149–163 (2000)
11. Filatei, A., Li, X., Moreno Maza, M., Schost, E.: Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In: Proceedings of ISSAC, pp. 93–100 (2006)
12. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra, 3rd edn. Cambridge University Press, Cambridge (2013)
13. Granlund, T.: The GMP Development Team: GNU MP: the GNU multiple precision arithmetic library (version 6.1.2) (2020). http://gmplib.org
14. van der Hoeven, J., Lecerf, G., Mourrain, B.: Mathemagix (from 2002). http://www.mathemagix.org
15. Kahoui, M.E.: An elementary approach to subresultants theory. J. Symb. Comput. **35**(3), 281–292 (2003)
16. Knuth, D.E.: The analysis of algorithms. Actes du congres international des Mathématiciens **3**, 269–274 (1970)
17. Lecerf, G.: On the complexity of the Lickteig-Roy subresultant algorithm. J. Symb. Comput. **92**, 243–268 (2019)
18. Lehmer, D.H.: Euclid's algorithm for large numbers. Am. Math. Mon. **45**(4), 227–233 (1938)
19. Lemaire, F., Moreno Maza, M., Xie, Y.: The RegularChains library in MAPLE. In: Maple Conference, vol. 5, pp. 355–368 (2005)
20. Lickteig, T., Roy, M.F.: Semi-algebraic complexity of quotients and sign determination of remainders. J. Complex. **12**(4), 545–571 (1996)
21. Maplesoft, a division of Waterloo Maple Inc.: Maple (2020). www.maplesoft.com
22. Monagan, M.: Probabilistic algorithms for computing resultants. In: Proceedings of ISSAC 2005, pp. 245–252. ACM (2005)
23. Monagan, M., Tuncer, B.: Factoring multivariate polynomials with many factors and huge coefficients. In: Gerdt, V.P., Koepf, W., Seiler, W.M., Vorozhtsov, E.V. (eds.) CASC 2018. LNCS, vol. 11077, pp. 319–334. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99639-4_22
24. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**(170), 519–521 (1985)
25. Reischert, D.: Asymptotically fast computation of subresultants. In: Proceedings of ISSAC 1997, pp. 233–240. ACM (1997)
26. Schönhage, A.: Schnelle Berechnung von Kettenbruchentwicklungen. Acta Informatica **1**, 139–144 (1971)
27. Shoup, V., et al.: NTL: a library for doing number theory (version 11.4.3) (2021). www.shoup.net/ntl

28. Thull, K., Yap, C.: A unified approach to HGCD algorithms for polynomials and integers. Manuscript (1990)
29. Zippel, R.: Probabilistic algorithms for sparse polynomials. In: Ng, E.W. (ed.) Symbolic and Algebraic Computation. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979). https://doi.org/10.1007/3-540-09519-5_73