Subresultant Chains Using Bézout Matrices

Mohammadali Asadi, Alexander Brandt, David J. Jeffrey, and Marc Moreno Maza(✉)

ORCCA, The University of Western Ontario, London, Canada
{masadi4,abrandt5,djeffrey}@uwo.ca, moreno@csd.uwo.ca

Abstract. Subresultant chains over rings of multivariate polynomials are calculated using a speculative approach based on the Bézout matrix. Our experimental results yield significant speedup factors for the proposed approach against comparable methods. The determinant computations are based on fraction-free Gaussian elimination using various pivoting strategies.

Keywords: Subresultant chain · Speculative algorithm · Multithreaded algorithm · Bézout matrix

1 Introduction

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 required in a procedure, not all polynomials of the chain, or not all coefficients of a given subresultant, may be needed. Based on that observation, the authors of [5] studied different practical schemes, and their implementation, for efficiently computing subresultants.

The main objective of [5] is, given two univariate polynomials $a, b \in \mathcal{A}[y]$ over some commutative ring $\mathcal{A}$, to compute the subresultant chain of $a, b \in \mathcal{A}[y]$ *speculatively*. To be precise, the objective is to compute the subresultants of index 0 and 1, delaying the computation of subresultants of higher index until it is proven necessary. The practical importance of this objective, as well as related works, are discussed extensively in [5].

Taking advantage of the Half-GCD algorithm and evaluation-interpolation methods, the authors of [5] consider the cases in which the coefficient ring $\mathcal{A}$ is a polynomial ring with one or two variables, and with coefficients in a field, $\mathbb{Q}$ or $\mathbb{Z}/p\mathbb{Z}$, for a prime number p. The reported experimentation demonstrates the benefits of computing subresultant chains speculatively in the context of polynomial system solving.

That strategy, however, based on the Half-GCD algorithm, cannot scale to situations in which the coefficient ring $\mathcal{A}$ is a polynomial ring in many variables, say 5 or more. The reason is that, for the Half-GCD algorithm to bring benefits,

F. Boulier et al. (Eds.): CASC 2022, LNCS 13366, pp. 29–50, 2022.
https://doi.org/10.1007/978-3-031-14788-3_3

the degree in y of the polynomials a, b must be in the 100's, implying that the resultant of a, b is likely to have very large degrees in the variables of $\mathcal{A}$, thus making computations not feasible in practice when $\mathcal{A}$ has many variables.

Therefore, for this latter situation, one should consider an alternative approach in order to compute subresultant chains speculatively, which is the objective of the present paper. To this end, we consider subresultant chain computations using Bézout matrices. Most notably, [1] introduced an algorithm to compute the nominal coefficients of subresultants by calculating the determinants of sub-matrices of a modified version of the Bézout matrix. Later, [15] generalized this approach to compute all subresultants instead of only the nominal coefficients. Although the approach is theoretically slower than Ducos' subresultant chain algorithm [10], early experimental results in Maple, collected during the development of the `SubresultantChain` method in the *RegularChains* library [16], indicate that approaches based on the Bézout matrix are particularly well-suited for sparse polynomials with many variables.

In this paper, we report on further work following this approach. In Sect. 2, we discuss how to compute the necessary determinants of the sub-matrices of the Bézout matrix. We modify and optimize the fraction-free LU decomposition (FFLU) of a matrix over a polynomial ring presented in [14]. We demonstrate the efficacy of the proposed methods using implementations in Maple and the Basic Polynomial Algebra Subprograms (BPAS) library [3]. Our optimization techniques include smart-pivoting and using the BPAS multithreaded interface to parallelize the row elimination step. All of our code, is open source and part of the BPAS library available at www.bpaslib.org.

In Sect. 3, we focus on the computation of subresultants using the Bézout matrix. In Sect. 3.1, we review the definitions of the Bézout matrix and a modified version of it, known as the Hybrid Bézout matrix. Then, we introduce a speculative approach for computing subresultants by modifying the fraction-free LU factorization and utilizing the Hybrid Bézout matrices in Sect. 3.2. We have implemented these computational schemes for subresultant chains and our experimental results, presented in Sect. 3.3, illustrate the benefits of the proposed methods.

2 Fraction-Free LU Decomposition

A standard way to compute the determinant of a matrix A is to reduce it to a triangular form and then take the product of the resulting diagonal elements [18]. One such triangular form is given by an LU matrix decomposition. When the input matrix A has elements in a polynomial ring, standard LU decomposition algorithms lead to matrices with rational functions as elements. In order to keep the elements in the ring of polynomials, while controlling expression swell, one can use a *fraction-free* LU decomposition (FFLU), taking advantage of Bareiss' algorithm [6], which was originally developed for integer matrices. Although in an FFLU decomposition the matrices contain only elements from the ring of polynomials, the intermediate computations do require exact divisions. Reducing the cost of these divisions is a practical challenge, one which we discuss in this

section. The main algorithm on which we rely has been described in [12, Ch. 9] and [14]. The main theorem is the following.

Theorem 1. *A rectangular matrix A with elements from an integral domain $\mathbb{B}$, having dimensions $m \times n$ and rank r, may be factored into matrices containing only elements from $\mathbb{B}$ in the form,*

$$A = P_r L D^{-1} U P_c = P_r \begin{pmatrix} \mathcal{L} \\ \mathcal{M} \end{pmatrix} D^{-1} \begin{pmatrix} \mathcal{U} & \mathcal{V} \end{pmatrix} P_c,$$

where the permutation matrix P_r is $m \times m$; the permutation matrix P_c is $n \times n$; $\mathcal{L}$ is $r \times r$, lower triangular and has full rank:

$$\mathcal{L} = \begin{pmatrix} p_1 & 0 & \dots & 0 \\ l_{21} & p_2 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{r1} & l_{r2} & \dots & p_r \end{pmatrix},$$

where the $p_i \neq 0$ are the pivots in a Gaussian elimination; $\mathcal{M}$ is $(m-r) \times r$ and is null when $m = n$ holds; D is $r \times r$ and diagonal:

$$D = diag(p_1, p_1 p_2, p_2 p_3, \cdots, p_{r-2} p_{r-1}, p_{r-1} p_r),$$

$\mathcal{U}$ is $r \times r$ and upper triangular, while $\mathcal{V}$ is $r \times (n-r)$ and is null when $m = n$ holds:

$$\mathcal{U} = \begin{pmatrix} p_1 & u_{12} & \dots & u_{1r} \\ 0 & p_2 & \dots & u_{2r} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & p_r \end{pmatrix}.$$

Proof [14, Theorem 2]. Note that the elements of the matrix D belong to $\mathbb{B}$, but the matrix D^{-1}, if explicitly calculated, lies in the quotient field.

Algorithm 3 implements Theorem 1 while Algorithm 2 utilizes Theorem 1 for computing the determinant of A, when A is square. Both Algorithm 3 and Algorithm 2 rely on Algorithm 1, which is a helper-function. This latter algorithm updates the input matrix A in-place, to record the upper triangular matrix U; it also computes the "denominator" d, the rank r of the matrix A and the column permutation of the input matrix. This is sufficient information to calculate the determinant of a square matrix.

In Algorithm 2, the routine CHECK-PARITY calculates the parity of the given permutation modulo 2. Note that in both Algorithms 1 and 3, we only consider row-operations to find the pivot and store the row permutation patterns in the list P_r of size m. Column-permutations, and the corresponding list P_c, are used in Sect. 2.1.

To optimize the FFLU algorithm, we use a *smart-pivoting* strategy, discussed in Sect. 2.1. The idea is to find a "best" pivot by searching through the matrix

to pick a non-zero coefficient (actually a polynomial) with the minimum number of terms in each iteration. The goal of this technique is to reduce the cost of the exact divisions in Bareiss' algorithm; see Sect. 2.1 for the details.

In addition, we discuss the parallel opportunities of this algorithm in Sect. 2.2, taking advantage of the BPAS multithreaded interface. Finally, Sect. 2.3 highlights the performance of these algorithms in the BPAS library, utilizing *sparse* multivariate polynomial arithmetic.

Algorithm 1. FFLU-HELPER(A)

Input: an $m \times n$ matrix $A = (a_{i,j})_{0 \le i < m,\ 0 \le j < n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).
Output: r, d, P_r where r is the rank of A, d is the "denominator", so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular). $s \in (-1, 1)$ is decided by the parity of row permutations (encoded by P_r).

```
 1: k := 0; d := 1; k := 0; c := 0; P_r := [0, 1, ..., m − 1]
 2: while k < m and c < n do
 3:     if a_{k,c} = 0 then
 4:         i := k + 1
 5:         while i < m do
 6:             if a_{i,c} ≠ 0 then
 7:                 SWAP i-th and k-th rows of A
 8:                 P_r[i], P_r[k] := P_r[k], P_r[i]
 9:                 break
10:             i := i + 1
11:         if m ≤ i then
12:             c := c + 1
13:             continue
14:     r := r + 1
15:     for i = k + 1, ..., m − 1 do
16:         for j = c + 1, ..., n − 1 do
17:             a_{i,j} := a_{i,c} a_{k,j} − a_{i,j} a_{k,c}
18:             if k = 0 then a_{i,j} := −a_{i,j}
19:             else a_{i,j} := ExactQuotient(a_{i,j}, d)
20:     d := −a_{k,c}; k := k + 1; c := c + 1
21: return r, −d, P_r
```

Algorithm 2. DET(A)

Input: a $n \times n$ matrix A over $\mathbb{B}$
Output: det(A), the determinant of A

```
1: r, d, P_r := FFLU-HELPER(A)
2: if r < n then return 0
3: p := CHECK-PARITY(P_r)
4: if p ≠ 0 then d := −d
5: return d
```

Algorithm 3. FFLU(A)

Input: an $m \times n$ matrix $A = (a_{i,j})_{0 \le i < m,\ 0 \le j < n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).

Output: r, d, P, L, U where r is the rank of A, d is the "denominator", so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular) and $s \in (-1, 1)$ is decided by the parity of the row permutations (encoded by the matrix P) performed on A, L is the lower triangular matrix, and U is the upper triangular matrix s.t. $PA = LDU$.

1: $U := A$; $i = 0$; $j = 0$; $k = 0$
2: $r, d, P_r :=$ FFLU-HELPER(U)
3: INITIALIZE P to the null square matrix of order m
4: LET $P[i, j] := 1$ iff $Pr[i] = j$ for all $0 \le i, j, \le m - 1$
5: **while** $i < m$ **and** $j < n$ **do**
6: **if** $U[i, j] \neq 0$ **then**
7: **for** $l = 0, \ldots, i - 1$ **do** $L_{l,k} := 0$
8: $L_{i,k} := U_{i,j}$
9: **for** $l = 0, \ldots, m - 1$ **do** $L_{l,k} := U_{l,j}$; $U_{l,j} := 0$
10: $i := i + 1$; $k := k + 1$
11: $j := j + 1$
12: **while** $k < m$ **do**
13: **for** $l = 0, \ldots, k - 1$ **do** $L_{l,k} := 0$
14: $L_{k,k} := 1$
15: **for** $l = k + 1, \ldots, m$ **do** $L_{l,k} := 0$
16: $k := k + 1$
17: **return** r, d, P_r, L, U

Example 1. *Consider matrix* $A \in \mathbb{B}^{4\times4}$ *where* $\mathbb{B} = \mathbb{Z}[x]$. $A =$

$$\begin{pmatrix} 11x^2 - 11x + 3 & -3(x-1)(2x-3) & 0 & 0 \\ 0 & 11x^2 - 11x + 3 & -3(x-1)(2x-3) & 0 \\ 0 & 0 & 11x^2 - 11x + 3 & -3(x-1)(2x-3) \\ -2x + 3 & 0 & 0 & -x \end{pmatrix}.$$

To compute the determinant of this matrix, Algorithm 1 starts with $d = 1$, $k = 0$, $c = 0$, $P_r = [0, 1, 2, 3]$, $A_{0,0} = 11x^2 - 11x + 3 \neq 0$, *and* $r = 1$. *After the first iteration, the nested for-loops update the (bottom-right) sub-matrix from the second row and column; we have* $A^{(1)} =$

$$\left(\begin{array}{c|ccc} A_{0,0} & -3(x-1)(2x-3) & 0 & 0 \\ \hline 0 & (A_{0,0})^2 & -3(x-1)(2x-3)A_{0,0} & 0 \\ 0 & 0 & (A_{0,0})^2 & -3(x-1)(2x-3)A_{0,0} \\ -2x+3 & -3(x-1)(2x-3)^2 & 0 & -xA_{0,0} \end{array}\right),$$

where $A^{(1)}_{1,2} = A^{(1)}_{2,3} = -3(x-1)(2x-3)(11x^2-11x+3)$. *In the second iteration of the while-loop, we have* $d = -11x^2+11x-3$, $k=1$, $c=1$, $A^{(1)}_{1,1} = (11x^2-11x+3)^2 \neq 0$, *and* $r=2$. *Then,* $A^{(2)} =$

$$\left(\begin{array}{cc|cc} A_{0,0} & A_{0,1} & 0 & 0 \\ 0 & (A_{0,0})^2 & -3(x-1)(2x-3)A_{0,0} & 0 \\ \hline 0 & 0 & (A_{0,0})^3 & -3(x-1)(2x-3)(A_{0,0})^2 \\ -2x+3 & (2x-3)A_{0,1} & -9(x-1)^2(2x-3)^3 & -x(A_{0,0})^2 \end{array}\right).$$

In the third iteration of the while-loop, we have $d = -(11x^2-11x+3)^2$, $k=2$, $c=2$, $A^{(2)}_{2,2} = -(11x^2-11x+3)^3 \neq 0$, *and* $r=3$. *And so,* $A^{(3)} =$

$$\left(\begin{array}{ccc|c} A_{0,0} & A_{0,1} & 0 & 0 \\ 0 & (A_{0,0})^2 & -3(x-1)(2x-3)A_{0,0} & 0 \\ 0 & 0 & (A_{0,0})^3 & -3(x-1)(2x-3)(A_{0,0})^2 \\ \hline -2x+3 & (2x-3)A_{0,1} & -9(x-1)^2(2x-3)^3 & A^{(3)}_{3,3} \end{array}\right),$$

where $A^{(3)}_{3,3} = -1763x^7 + 7881x^6 - 19986x^5 + 35045x^4 - 41157x^3 + 30186x^2 - 12420x + 2187$. *In fact, one can check that* $A^{(3)}_{3,3}$ *is the determinant of the full-rank* ($r=4$) *matrix* $A \in \mathbb{Z}[x]^{4\times 4}$.

In [6], Bareiss introduced an alternative version of this algorithm, known as *multi-step* Bareiss' algorithm to compute fraction-free LU decomposition. This method reduces the computation of row eliminations by adding three cheaper divisions to compute each row in the while-loop and removing one multiplication in each iteration of the nested for-loops; see the results in Table 1 and [12, Chapter 9] for more details.

In the next sections, we investigate optimizations of Algorithm 1 to compute the determinant of matrices over multivariate polynomials. These optimizations are achieved by reducing the cost of exact divisions by finding better pivots and utilizing the BPAS multithreaded interface to parallelize this algorithm.

2.1 Smart-Pivoting in FFLU Algorithm

Returning to Example 1, we performed exact divisions for the following divisors in the second and third iterations,

$$d^{(1)} = -11x^2 + 11x - 3,$$
$$d^{(2)} = -121x^4 + 242x^3 - 187x^2 + 66x - 9.$$

However, we could pick a polynomial with fewer terms as our pivot in every iteration to reduce the cost of these exact divisions. Such a method, which finds a polynomial with the minimum number of terms in each column as the pivot of each iteration, is referred to as column-wise smart-pivoting. For matrix A of

Example 1, one can pick $A_{3,0} = -2x+3$ as the first pivot. Applying this method yields, after the first iteration, $A^{(1)} =$

$$\left(\begin{array}{c|ccc} -2x+3 & 0 & 0 & -x \\ \hline 0 & -(2x-3)A_{0,0} & 3(x-1)(2x-3)^2 & 0 \\ 0 & 0 & -(2x-3)A_{0,0} & 3(x-1)(2x-3)^2 \\ A_{0,0} & 3(x-1)(2x-3)^2 & 0 & xA_{0,0} \end{array}\right),$$

where $d = 2x-3$. Continuing this method from Algorithm 1, we get the following matrix for $r = 4$, $A^{(4)} =$

$$\left(\begin{array}{ccc|c} -2x+3 & 0 & 0 & -x \\ 0 & -(2x-3)A_{0,0} & 3(x-1)(2x-3)^2 & 0 \\ 0 & 0 & -(2x-3)A_{0,0}^2 & 3(x-1)(2x-3)^2A_{0,0} \\ \hline A_{0,0} & 3(x-1)(2x-3)^2 & 9(x-1)^2(2x-3)^3 & A_{3,3}^{(4)} \end{array}\right),$$

where $A_{3,3}^{(4)} = 1763x^7 - 7881x^6 + 19986x^5 - 35045x^4 + 41157x^3 - 30186x^2 + 12420x - 2187$, $P_r = [3, 1, 2, 0]$, and we have $\text{DET}(A) = -A_{3,3}^{(4)}$ from Algorithm 2.

In *column-wise smart-pivoting*, we limited our search for the best pivot to the corresponding column of the current row. To extend this method, one can try searching for the best pivot in the sub-matrix starting from the next current row and column. To perform this method, referred to as *(fully) smart pivoting*, we need to use column-operations and a column-wise permutation matrix P_c. The column operations along with row operations are not cache-friendly. This is certainly an issue for matrices with (large) multivariate polynomial entries while this may not be an issue with (relatively small) matrices with numerical entries. Therefore, we avoid column swapping within the decomposition, and instead we keep track of column permutations in the list of column-wise permutation patterns P_c to calculate the parity check later in Algorithm 2.

Algorithm 4 presents the pseudo-code of the *smart pivoting* fraction-free LU decomposition utilizing both row-wise and column-wise permutation patterns P_r, P_c. This algorithm updates A in-place, to become the upper triangular matrix U, and returns the rank and denominator of the given matrix $A \in \mathbb{B}^{m\times n}$.

2.2 Parallel FFLU Algorithm

For further practical performance, we now investigate opportunities for parallelism alongside our schemes for cache-efficiency. In particular, notice that during the row reduction step (the `for` loops on lines 24–28 of Algorithm 4) the update of each element is independent. Implementing this step as a `parallel_for` loop is easily achieved with the multithreading support provided in the BPAS library; see further details in [4].

Algorithm 4. SPFFLU-HELPER(A)

Input: an $m \times n$ matrix $A = (a_{i,j})_{0 \le i < m,\ 0 \le j < n}$ over $\mathbb{B}$ ($a_{i,j} \in \mathbb{B}$).

Output: r, d, P_r, P_c where r is the rank, d is the denominator, so that, $d = s\ det(S)$ where S is an appropriate sub-matrix of A ($S = A$ if A is square and non-singular) $s \in (-1, 1)$ is decided by the parity of row and column permutations, P_r, P_c.

$k := 0$; $d := 1$; $\ell := 0$
$P_r := [0, 1, \ldots, m-1]$; $P_c := [0, 1, \ldots, n-1]$
while $k < m$ **and** $\ell < n$ **do**
    if $a_{k,\ell} = 0$ **then**
        $i := k + 1$
        while $i < m$ **do**
            if $a_{i,\ell} \neq 0$ **then**
                $(i, j) :=$ FINDBESTPIVOT(A, i, ℓ)
                SWAP i-th and k-th rows of A
                $P_r[i], P_r[k] := P_r[k], P_r[i]$
                $P_c[j], P_c[\ell] := P_c[\ell], P_c[j]$
                break
            $i := i + 1$
        if $m \le i$ **then**
            $\ell := \ell + 1$
            continue
    else
        $(i, j) :=$ FINDBESTPIVOT(A, k, ℓ)
        SWAP i-th and k-th rows of A
        $P_r[i], P_r[k] := P_r[k], P_r[i]$
        $P_c[j], P_c[\ell] := P_c[\ell], P_c[j]$
    $r := r + 1$
    for $i = k + 1, \ldots, m - 1$ **do**
        for $j = \ell + 1, \ldots, n - 1$ **do**
            $a_{i,P_c[j]} := a_{i,P_c[\ell]}\ a_{k,P_c[j]} - a_{i,P_c[j]}\ a_{k,P_c[\ell]}$
            if $k = 0$ **then** $a_{i,P_c[j]} := -a_{i,P_c[j]}$
            else $a_{i,P_c[j]} :=$ EXACTQUOTIENT$(a_{i,P_c[j]}, d)$
    $d := -a_{k,P_c[\ell]}$; $k := k + 1$; $\ell := \ell + 1$
return $r, -d, P_r, P_c$

Algorithm 5. PARALLEL-SPFFLU-HELPER(A)

```
// -snip-
parallel_for i = k+1, ..., m-1
    parallel_for j = ℓ+1, ..., n-1
        a_{i,P_c[j]} := a_{i,P_c[ℓ]} a_{k,P_c[j]} − a_{i,P_c[j]} a_{k,P_c[ℓ]}
        if k = 0 then a_{i,P_c[j]} := −a_{i,P_c[j]}
        else a_{i,P_c[j]} := ExactQuotient(a_{i,P_c[j]}, d)
    end for
end for
// -snip-
```

Algorithm 5 shows a naïve implementation of this parallel algorithm. Note that in a `parallel_for` loop, each iteration is (potentially) executed in parallel. In Algorithm 5, this means lines 3–5 are executed independently and in parallel for each possible value of (i, j). If that number of such possible values exceeds a pre-determined limit (e.g. the number of hardware threads supported), then the number of iterations will be divided as evenly as possible among the available threads.

A difficulty to this parallelization scheme is that the size of the sub-matrices decreases with each iteration. Therefore, the amount of work executed by each thread also decreases. In practice, to address this *load-balancing* and to maximize parallelism, we only parallelize the outer loop (line 1 of Algorithm 5).

2.3 Experimentation

In this section, we compare the fraction-free LU decomposition algorithms for Bézout matrix (Definition 2) of randomly generated, non-zero and sparse polynomials in $\mathbb{Z}[x_1, \ldots, x_v]$ for $v \geq 5$ in the BPAS library. We recall that methods based on the Bézout matrix have been observed (during development of the *RegularChains* library [16], and later in Sect. 3.3) to be well-suited for sparse polynomials with many variables. Throughout this paper, our benchmarks were collected on a machine running Ubuntu 18.04.4 and GMP 6.1.2, with an Intel Xeon X5650 processor running at 2.67 GHz, with 12×4 GB DDR3 memory at 1.33 GHz.

Table 1 shows the comparison between the standard implementation of the fraction-free LU decomposition (Algorithm 1; denoted `plain`), the *column-wise* smart pivoting (denoted `col-wise SP`), the *fully* smart-pivoting method (Algorithm 4; denoted `fully SP`), and Bareiss' multi-step technique added to Algorithm 4 (denoted `multi-step`). Here, $v = 5$ and the generated polynomials have a *sparsity ratio* (the fraction of zero terms to the total possible number of terms in a fully dense polynomial of the same partial degrees) of 0.98.

This table indicates that using smart-pivoting yields up to a factor of 3 speed-up. Comparing `col-wise SP` and `fully SP` shows that calculating P_c (column-wise permutation patterns) along with P_r (row-wise permutation patterns) does not cause any slow-down in the calculation of d.

Moreover, using both multi-step technique and smart-pivoting does not bring any additional speed-up. The smart-pivoting technique is already minimized the cost of exact divisions in each iteration. Table 2 shows $^{\text{plain}}/_{\text{fully SP}}$, $^{\text{plain}}/_{\text{multi-step}}$, and $^{\text{fully SP}}/_{\text{multi-step}}$ ratios from Table 1.

To analyze the performance of parallel FFLU algorithm, we compare Algorithm 5 and Algorithm 4 for $n \times n$ matrices of randomly generated non-zero univariate polynomials with integer coefficients and degree 1. Table 3 summarizes these results. For $n = 75$, 2.14× parallel speed-up is achived, and speed-up continues to increase with increasing n.

Table 1. Compare the execution time (in seconds) of fraction-free LU decomposition algorithms for Bézout matrix of randomly generated, non-zero and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_5]$ with $x_5 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, $\deg(a, x_2) = \deg(b, x_2)+1 = 5$, $\deg(a, x_3) = \deg(b, x_3) = 1$, $\deg(a, x_4) = \deg(b, x_4) = 1$, $\deg(a, x_5) = \deg(b, x_5) = 1$

d	plain	col-wise SP	fully SP	multi-step
6	0.048346	0.018623	0.021154	0.021257
7	2.379480	0.941655	0.954981	0.953532
8	3.997310	0.444759	0.426654	0.475043
9	73.860600	32.531600	31.764200	30.882500
10	2726.690000	1431.430000	1408.140000	1398.370000
11	9059.290000	5113.530000	4768.950000	5348.520000
12	5953.150000	3937.250000	3521.140000	3711.790000
13	81411.900000	42858.500000	42043.600000	41850.800000

Table 2. Ratios of FFLU algorithms for polynomials in Table 1

d	plain/fully SP	plain/multi-step	fully SP/multi-step
6	2.285431	2.274357	0.995155
7	2.491652	2.495438	1.001520
8	9.368973	8.414628	0.898138
9	2.325278	2.391665	1.028550
10	1.936377	1.949906	1.006987
11	1.899640	1.693794	0.891639
12	1.690688	1.603849	0.948637
13	1.936368	1.945289	1.004607

Table 3. Comparing the execution time (in seconds) of Algorithm 4 and Algorithm 5 for $n \times n$ matrices of random non-zero degree 1 univariate integer polynomials

n	*serial* FFLU	*parallel* FFLU	serial/parallel
10	00.11976	0.012765	0.938109
15	0.118972	0.076118	1.562994
20	0.628613	0.339738	1.850288
25	2.299270	1.126620	2.040857
30	6.241600	3.109840	2.007049
35	15.305100	7.552200	2.026575
40	33.831800	16.387200	2.064526
45	67.702600	32.307100	2.095595
50	127.438000	60.420000	2.109202
55	224.681000	106.043000	2.118773
60	392.795000	177.456000	2.213478
65	607.089000	284.659000	2.132689
70	947.805000	444.181000	2.133826
75	1432.180000	668.991000	2.140806

3 Bézout Subresultant Algorithms

In this section, we continue exploring the subresultant algorithms for multivariate polynomials based on calculating the determinant of (Hybrid) Bézout matrices.

3.1 Bézout Matrix and Subresultants

A traditional way to define subresultants is via computing determinants of submatrices of the Sylvester matrix (see, e.g. [5] or [11, Ch. 6]). Li [17] presented an elegant way to calculate subresultants directly from the following matrices. This method follows the same idea as subresultants based on Sylvester matrix.

Theorem 1. *The k-th subresultant* $S_k(a,b)$ *of* $a = \sum_{i=0}^{m} a_i y^i, b = \sum_{i=0}^{n} b_i y^i \in \mathbb{B}[y]$ *is calculated by the determinant of the following* $(m+n-k)\times(m+n-k)$ *matrix:*

$$E_k := \begin{array}{r} \left.\begin{array}{r} \\ \\ \\ \end{array}\right\} n-k \\ \left.\begin{array}{r} \\ \\ \\ \end{array}\right\} k \\ \left.\begin{array}{r} \\ \\ \\ \end{array}\right\} m-k \end{array} \begin{bmatrix} a_m & a_{m-1} & \cdots & a_2 & a_1 & a_0 & & & \\ & \ddots & & & & & \ddots & & \\ & & a_m & a_{m-1} & \cdots & a_2 & a_1 & a_0 & \\ & & & & & 1 & -y & & \\ & & & & & & \ddots & \ddots & \\ & & & & & & & 1 & -y \\ b_n & b_{n-1} & \cdots & b_2 & b_1 & b_0 & & & \\ & \ddots & & & & & \ddots & & \\ & & b_n & b_{n-1} & \cdots & b_2 & b_1 & b_0 & \end{bmatrix}, \tag{1}$$

so that,

$$S_k(a,b) = (-1)^{k(m-k+1)} det(E_k).$$

Proof. [17, Section 2]

Another practical division-free approach is through utilizing the Bézout matrix to compute the subresultant chain of multivariate polynomials by calculating the determinant of the Bézout matrix of the input polynomials [13]. From [7], we define the *symmetric* Bézout matrix as follows.

Definition 1. *The Bézout matrix associated with* $a, b \in \mathbb{B}[y]$*, where* $m := \deg(a) \geq n := \deg(b)$ *is the symmetric matrix:*

$$Bez(a,b) := \begin{pmatrix} c_{0,0} & \cdots & c_{0,m-1} \\ \vdots & \ddots & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,m-1} \end{pmatrix},$$

where the coefficients $c_{i,j}$, for $0 \leq i, j < m$, are defined by the so-called Cayley expression as follows,

$$\frac{a(x)b(y) - a(y)b(x)}{x - y} = \sum_{i,j=0}^{m-1} c_{i,j} y^i x^j.$$

The relations between the *Sylvester* and Bézout matrices have been studied for decades yielding an efficient algorithm to construct the Bézout matrix [2] using a so-called Hybrid Bézout matrix.

Definition 2. *The Hybrid Bézout matrix of $a = \sum_{i=0}^{m} a_i y^i$ and $b = \sum_{i=0}^{n} b_i y^i$ is defined as the $m \times m$ matrix*

$$HBez(a, b) := \begin{pmatrix} h_{0,0} & \cdots & h_{0,m-1} \\ \vdots & \ddots & \vdots \\ h_{m-1,0} & \cdots & h_{m-1,m-1} \end{pmatrix},$$

where the coefficients $h_{i,j}$, for $0 \leq i, j < m$, are defined as:

$$h_{i,j} = \text{coeff}(H_{m-i+1}, m - j) \text{ for } 1 \leq i \leq n,$$
$$h_{i,j} = \text{coeff}(x^{m-i} b, m - j) \text{ for } m + 1 \leq i \leq n,$$

with,

$$H_i = (a_m y^{i-1} + \cdots + a_{m-i+1})(b_{n-i} y^{m-i} + \cdots + b_0 y^{m-n}) - (a_{m-i} y^{m-i} + \cdots + a_0)(b_n y^{i-1} + \cdots + b_{n-i+1}).$$

Example 2. *Consider the polynomials $a = 5y^5 + y^3 + 2y + 1$ and $b = 3y^3 + y + 3$ in $\mathbb{Z}[y]$. The Sylvester matrix of a, b is:*

$$Sylv(a, b) = \begin{pmatrix} 5 & 0 & 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 5 & 0 & 1 & 0 & 2 & 1 & 0 \\ 0 & 0 & 5 & 0 & 1 & 0 & 2 & 1 \\ 3 & 0 & 1 & 3 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 1 & 3 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 & 1 & 3 \end{pmatrix},$$

and the Bézout matrix of a, b is:

$$Bez(a, b) = \begin{pmatrix} 0 & -15 & 0 & -5 & -15 \\ -15 & 0 & -5 & -15 & 0 \\ 0 & -5 & -15 & 5 & 0 \\ -5 & -15 & 5 & 0 & 0 \\ -15 & 0 & 0 & 0 & -5 \end{pmatrix},$$

while the Hybrid Bézout matrix of a, b is:

$$HBez(a,b) = \begin{pmatrix} 15 & -6 & 0 & -2 & -1 \\ 2 & 15 & -6 & -3 & 0 \\ 0 & 2 & 15 & -6 & -3 \\ 3 & 0 & 1 & 3 & 0 \\ 0 & 3 & 0 & 1 & 3 \end{pmatrix}.$$

Diaz-Toca and Gonzalez-Vega examined the relations between Bézout matrices and subresultants in [9]. Hou and Wang studied to apply the Hybrid Bézout matrix for the calculation of subresultants in [13].

Notation 1. *Let J_m denote the backward identity matrix of order m and let B and H be defined as follows:*

$$B := J_m \ Bez(a,b) \ J_m = \begin{pmatrix} c_{m-1,m-1} & \cdots & c_{m-1,0} \\ \vdots & \ddots & \vdots \\ c_{0,m-1} & \cdots & c_{0,0} \end{pmatrix},$$

$$H := J_m \ HBez(a,b) \quad = \begin{pmatrix} h_{m-1,0} & \cdots & h_{m-1,m-1} \\ \vdots & \ddots & \vdots \\ h_{0,0} & \cdots & h_{0,m-1} \end{pmatrix}.$$

Now, we can state how to compute the subresultants from Bézout matrices as follows.

Theorem 2. *For polynomials $a = \sum_{i=0}^{m} a_i y^i$ and $b = \sum_{i=0}^{n} b_i y^i$ in $\mathbb{B}[y]$, the k-th subresultant of a, b, i.e., $S_k(a,b)$, can be obtained from:*

$$(-1)^{(m-1)(m-k-1)/2} a_m^{m-n} S_k(a,b) = \sum_{i=0}^{k} B_{m-k,k-i} \ y^i,$$

where $B_{m-k,i}$ for $0 \leq i \leq k$ denotes the $(m-k) \times (m-k)$ minor extracted from the first $m-k$ rows, the first $m-k-1$ columns and the $(m-k+i)$-th column of B.

Proof. [2, Theorem 2.3]

Theorem 3. *For those polynomials $a, b \in \mathbb{B}[y]$, the k-th subresultant of a, b, i.e., $S_k(a,b)$, can be obtained from:*

$$(-1)^{(m-1)(m-k-1)/2} S_k(a,b) = \sum_{i=0}^{k} H_{m-k,k-i} \ y^i,$$

where $H_{m-k,i}$ for $0 \leq i \leq k$ denotes the $(m-k) \times (m-k)$ minor extracted from the first $m-k$ rows, the first $m-k-1$ columns and the $(m-k+i)$-th column of H.

Proof. [2, Theorem 2.3]
Abdeljaoued et al. in [2] study further this relation between subresultants and Bézout matrices. Theorem 4 is the main result of this paper.

Theorem 4. *For those polynomials $a, b \in \mathbb{B}[y]$, the k-th subresultant of a, b can be obtained from the following $m \times m$ matrices, where $\tau = (m-1)(m-k-1)/2$:*

$$(-1)^{\tau} a_m^{m-n} S_k(a,b) = (-1)^k \begin{vmatrix} c_{m-1,m-1} & c_{m-1,m-2} & \cdots & \cdots & \cdots & c_{m-1,0} \\ \vdots & \vdots & & \cdots & \cdots & \vdots \\ c_{k,m-1} & c_{k,m-2} & \cdots & \cdots & \cdots & c_{k,0} \\ & & 1 & -y & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -y \end{vmatrix},$$

$$(-1)^{\tau} S_k(a,b) = (-1)^k \begin{vmatrix} h_{m-1,0} & h_{m-1,1} & \cdots & \cdots & \cdots & h_{m-1,m-1} \\ \vdots & \vdots & & \cdots & \cdots & \vdots \\ h_{k,0} & h_{k,1} & \cdots & \cdots & \cdots & h_{k,m-1} \\ & & 1 & -y & & \\ & & & \ddots & \ddots & \\ & & & & 1 & -y \end{vmatrix}.$$

Proof. [2, Theorem 2.4]
The advantage of this aforementioned method is that one can compute the entire subresultant chain in a bottom-up fashion. This process starts from computing the determinant of matrix H (or B) in Definition 1 to calculate $S_0(a,b)$, the resultant of a, b, and update the last k rows of H (or B) to calculate $S_k(a,b)$ for $1 \leq k \leq n$.

Example 3. *Consider polynomials $a = -5y^4x + 3yx - y - 3x + 3$ and $b = -2y^3x + 3y^3 - x$ in $\mathbb{Z}[x,y]$ where $x < y$. From Definition 2, the Hybrid Bézout matrix of a, b is the matrix A from Example 1 on page 34. Recall from Example 1 that the determinant of this matrix can be calculated using the fraction-free LU decomposition schemes. Theorem 4, for $k = 0$, yields that,*

$$S_0(a,b) = -1763x^7 + 7881x^6 - 19986x^5 + 35045x^4 - 41157x^3 + 30186x^2 - 12420x + 2187.$$

For $k = 1$, one can calculate $S_1(a,b)$ from the determinant of:

$$H^{(1)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2 - 11x + 3 & -3(x-1)(2x-3) \\ 0 & 11x^2 - 11x + 3 & -3(x-1)(2x-3) & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

that is,

$$S_1(a,b) = -242x^5y + 132x^5 + 847x^4y - 660x^4 - 1100x^3y + 1257x^3 + 693x^2y - 1134x^2 - 216xy + 486x + 27y - 81.$$

We can continue calculating subresultants of higher indices with updating matrix $H^{(1)}$*. For instance, the* 2*nd and* 3*rd subresultants are, respectively, from the determinant of:*

$$H^{(2)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

and,

$$H^{(3)} = \begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 1 & -y & 0 & 0 \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix},$$

which are,

$$S_2(a,b) = 22yx^3 - 12x^3 - 55yx^2 + 48x^2 + 39yx - 63x - 9y + 27,$$
$$S_3(a,b) = -2y^3x + 3y^3 - x.$$

We further studied the performance of computing subresultants from Theorem 4 in comparison to the Hybrid Bézout matrix in Definition 2 for multivariate polynomials with integer coefficients. In our implementation, we took advantage of the *FFLU* schemes reviewed in Sect. 2 to compute the determinant of these matrices using *smart-pivoting* technique in *parallel*; see Sect. 3.3 for implementation details and results.

3.2 Speculative Bézout Subresultant Algorithms

In Example 3, the Hybrid Bézout matrix was used to compute subresultants of two polynomials in $\mathbb{Z}[x,y]$. We constructed the square matrix H from Definition 1 and updated the last $k \geq 0$ rows following Theorem 4. Thus, the kth subresultant could be directly computed from the determinant of this matrix.

Consider solving systems of polynomial equations by triangular decomposition, and particularly, regular chains. This method uses a *Regular GCD* subroutine (see [8]) which requires the computation of subresultants in a bottom-up fashion: for multivariate polynomials a, b (viewed as univariate in their main variable) compute $S_0(a,b)$, then possibly $S_1(a,b)$, then possibly $S_2(a,b)$, etc., to try and find a regular GCD. This bottom-up approach for computing subresultant chains is discussed in [5].

In the approach explained in the previous section, we would call the determinant algorithm twice for $H^{(0)} := H$ and $H^{(1)}$ to compute S_0, S_1 respectively. Here, we study a speculative approach to compute both S_0 and S_1 at the cost of computing only one of them. This approach can also be extended to compute any two successive subresultants S_k, S_{k+1} for $2 \leq k < \deg(b, x_n)$.

To compute S_0, S_1 of polynomials $a = -5y^4x + 3yx - y - 3x + 3$ and $b = -2y^3x + 3y^3 - x$ in $\mathbb{Z}[x, y]$ from Example 3, consider the $(m+1) \times m$ matrix, with $m = 4$, derived from the Hybrid Bézout matrix of a, b, $H^{(0,1)} =$

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & 0 & 11x^2-11x+3 & -3(x-1)(2x-3) \\ 0 & 11x^2-11x+3 & -3(x-1)(2x-3) & 0 \\ \mathbf{11x^2-11x+3} & \mathbf{-3(x-1)(2x-3)} & \mathbf{0} & \mathbf{0} \\ \textit{0} & \textit{0} & \textit{1} & -y \end{pmatrix}.$$

In this matrix, the first three rows are identical to the first three rows of $H^{(0)}$ and $H^{(1)}$, while the 4th (**bold**) row is the 4th row of $H^{(0)}$ and the 5th (*italicized*) row is the 4th row of $H^{(1)}$. A deeper look into the determinant algorithm reveals that the *Gaussian (row) elimination* for the first three rows in each iteration of the fraction-free LU decomposition is similar in both $H^{(0)}$ and $H^{(1)}$ and the only difference is within the 4th row.

Hence, managing these row eliminations in the fraction-free LU decomposition, we can compute determinants of $H^{(0)}$ and $H^{(1)}$ by using $H^{(0,1)}$ only calling the FFLU algorithm once. Indeed, when this algorithm tries to eliminate the last rows of $H^{(0)}$ and $H^{(1)}$, we should use the last two rows of $H^{(0,1)}$ separately and return two denominators corresponding to S_0, S_1.

We can further extend this speculative approach to compute S_2 and S_3 by updating the matrix $H^{(0,1)}$ to get the $(m+3) \times m$ matrix $H^{(2,3)} =$

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ \mathbf{0} & \mathbf{0} & \mathbf{11x^2-11x+3} & \mathbf{-3(x-1)(2x-3)} \\ \cancel{0} & \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} \\ \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} & \cancel{0} \\ \textit{1} & \textit{-y} & \textit{0} & \textit{0} \\ 0 & 1 & -y & 0 \\ 0 & 0 & 1 & -y \end{pmatrix}.$$

Therefore, to calculate subresultants of index 2 and 3, we should respectively consider the 2nd (**bold**) and 5th (*italicized*) rows of $H^{(2,3)}$ in the fraction-free LU decomposition while ignoring the 3rd and 4th (strikethrough) rows. An adaptation of the FFLU algorithm can then modify $H^{(2,3)}$ as follows to return $d_{(2)}$, ignoring the 5th and strikethrough rows.

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ 0 & -2x+3 & -y(-2x+3) & 0 \\ \cancel{0} & \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} \\ \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} & \cancel{0} \\ \textit{1} & \textit{-y} & \textit{0} & \textit{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{-22x^3+55x^2-39x+9} & \mathbf{3(x-1)(2x-3)^2} \\ 0 & 0 & -2x+3 & d_{(2)} \end{pmatrix},$$

where $d_{(2)} = -22x^3y+12x^3+55x^2y-48x^2-39xy+63x+9y-27$ and $S_2 = -d_{(2)}$. Note that the 2nd and 6th rows are swapped to find a proper pivot.

The adapted FFLU algorithm can also modify $H^{(2,3)}$ to rather return $d_{(3)}$, ignoring the 2nd (**bold**) and strikethrough rows,

$$\begin{pmatrix} -2x+3 & 0 & 0 & -x \\ \mathbf{0} & \mathbf{0} & \mathbf{11x^2-11x+3} & \mathbf{-3(x-1)(2x-3)} \\ \cancel{0} & \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} \\ \cancel{11x^2-11x+3} & \cancel{-3(x-1)(2x-3)} & \cancel{0} & \cancel{0} \\ 1 & -y(-2x+3) & 0 & x \\ 0 & -2x+3 & -y^2(-2x+3) & x \\ 0 & 0 & y(-2x+3) & d_{(3)} \end{pmatrix},$$

where $d_{(3)} = -2xy^3+3y^3-x$ and $S_3 = d_{(3)}$.

Generally, to compute subresultants of index k and $k+1$, one can construct the matrix $H^{(k,k+1)}$ from the previously constructed $H^{(k-2,k-1)}$ for $k > 1$. This *recycling* of previous information makes computing the next subresultants of index k and $k+1$ much more efficient, and is discussed below. We proceed with an adapted FFLU algorithm over:

- the first $m-k-1$ rows,
- the **bold** row for computing S_k, or the *italicized* row for computing S_{k+1}, and
- the last k rows

of matrix $H^{(k,k+1)} \in \mathbb{B}^{(m+k)\times k}$ with $\mathbb{B} = \mathbb{Z}[x_1, \ldots, x_v]$,

$$H^{(k,k+1)} = \begin{pmatrix} h_{m-1,0} & h_{m-1,1} & \cdots\cdots & h_{m-1,m-1} \\ \vdots & \vdots & \cdots\cdots & \vdots \\ \mathbf{h_{k,0}} & \mathbf{h_{k,1}} & \cdots\cdots & \mathbf{h_{k,m-1}} \\ \cancel{h_{k-1,0}} & \cancel{h_{k-1,1}} & \cancel{\cdots\cdots} & \cancel{h_{k-1,m-1}} \\ \cancel{\vdots} & \cancel{\vdots} & \cancel{\cdots\cdots} & \cancel{\vdots} \\ \cancel{h_{0,0}} & \cancel{h_{0,1}} & \cancel{\cdots\cdots} & \cancel{h_{0,m-1}} \\ & & 1 & -y \\ & & \ddots & \ddots \\ & & & 1 \quad -y \end{pmatrix}.$$

As seen in the last example, the FFLU algorithm, depending on the input polynomials, may create two completely different submatrices to calculate $d_{(2)}$ and $d_{(3)}$. Thus, the cost of computing S_k, S_{k+1} from $H^{(k,k+1)}$ speculatively may not necessarily be less than computing them successively from $H^{(k)}, H^{(k+1)}$ for some $k > 1$.

We improve the performance of computing S_k, S_{k+1} speculatively via *caching*, and then reusing, intermediate data calculated to compute S_{k-2}, S_{k-1} from $H^{(k-2,k-1)}$. In this approach, the adapted FFLU algorithm returns $d_{(k-2)}, d_{(k-1)}$ along with $H^{(k-2,k-1)}$, the reduced matrix $H^{(k-2,k-1)}$ to compute $d_{(k-1)}$, the list of permutation patterns and pivots.

Therefore, we can utilize $H^{(k-2,k-1)}$ to construct $H^{(k,k+1)}$. In addition, if the first $\delta := m - k - 1$ pivots are picked from the first δ rows of $H^{(k-2,k-1)}$, then one can use the first δ rows of the reduced matrix $H^{(k-2,k-1)}$ along with the list of permutation patterns and pivots to perform the first δ row eliminations of $H^{(k,k+1)}$ via recycling the first δ rows of the reduced matrix cached *a priori*.

3.3 Experimentation

In this section, we compare the subresultant algorithms based on (Hybrid) Bézout matrix against the Ducos' subresultant chain algorithm in BPAS and Maple 2020. In BPAS, our optimized Ducos' algorithm (denoted OptDucos), is detailed in [5].

Table 4 and Table 5 show the running time of plain and speculative algorithms for randomly generated, non-zero, sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_6]$ with $x_6 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 6$. Table 6 and Table 7 show the running time of plain, speculative and caching subresultant schemes for randomly generated, non-zero, and sparse polynomials $a, b \in \mathbb{Z}[x_1, x_2, \ldots, x_7]$ with $x_7 < \cdots < x_2 < x_1$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 7$.

Note that the Bézout algorithm in Maple computes the resultant of a, b ($S_0(a, b)$) meanwhile both Maple's and BPAS's Ducos' algorithm computes the entire subresultant chain. In BPAS, we have the following:

1. Bézout ($\rho = 0$) calculates the resultant ($S_0(a, b)$) via the determinant of Hybrid Bézout matrix of a, b;
2. Bézout ($\rho = 1$) calculates $S_1(a, b)$ following Theorem 4 from the Hybrid Bézout matrix of a, b;
3. SpecBézout ($\rho = 0$) calculates $S_0(a, b), S_1(a, b)$ speculatively using $H^{(0,1)}$;
4. SpecBézout ($\rho = 2$) calculates $S_2(a, b), S_3(a, b)$ speculatively using $H^{(2,3)}$;
5. SpecBézout$_{\texttt{cached}}$ ($\rho = 2$) calculates $S_2(a, b), S_3(a, b)$ speculatively via $H^{(2,3)}$ and the *cached* information calculated in *SpecBézout* ($\rho = 0$)
6. SpecBézout$_{\texttt{cached}}$ ($\rho = \texttt{all}$) calculates the entire subresultant chain using the speculative algorithm and caching.

To compute subresultants from Bézout matrices in Maple, we use the command `SubresultantChain( ... , 'representation'='BezoutMatrix')` from the `RegularChains` library. Our Bézout algorithm is up to 3× faster than the Maple implementation to calculate only S_0. Moreover, our results show that Bézout algorithms outperform the Ducos' algorithm in both BPAS and Maple for sparse polynomials with many variables.

Tables 4 and 6 show that the cost of computing subresultants S_0, S_1 speculatively is comparable to the running time of computing only one of them. Tables 5 and 7 indicate the importance of recycling cached data to compute higher subresultants speculatively. Our Bézout algorithms can calculate all subresultants speculatively in a comparable running time to the Ducos' algorithm.

Table 4. Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, sparse polynomials $a, b \in \mathbb{Z}[x_6 < \ldots < x_1]$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 6$

	Maple		BPAS			
d	Bézout $(\rho = 0)$	Ducos	Bézout $(\rho = 0)$	Bézout $(\rho = 1)$	SpecBézout $(\rho = 0)$	OptDucos
10	0.05128	0.03000	0.024299	0.026762	0.032166	0.045270
11	0.06001	0.04574	0.057312	0.068722	0.058843	0.049532
12	0.02515	0.05100	0.007223	0.019530	0.012792	0.061419
13	0.81209	16.81200	0.421278	0.739842	0.594225	9.527660
14	3.14360	112.280	2.414530	3.829530	3.250710	69.957100
15	518.380	7163.30	151.656	779.9240	512.260	3655.820

Table 5. Comparing the execution time (in seconds) of speculative subresultant algorithms for polynomials in Table 4

	BPAS			
d	SpecBézout$(\rho = 0)$	SpecBézout$(\rho = 2)$	SpecBézout$_{\texttt{cached}}(\rho = 2)$	SpecBézout$_{\texttt{cached}}(\rho = \texttt{all})$
10	0.032166	0.022125	0.016432	0.076283
11	0.058843	0.079425	0.043512	0.193512
12	0.012792	0.010566	0.004148	0.071435
13	0.594225	2.106280	1.535510	7.891180
14	3.250710	8.735510	4.133760	73.59940
15	512.260	953.1170	579.8580	4877.130

Table 6. Comparing the execution time (in seconds) of subresultant algorithms based on Bézout matrix for randomly generated, non-zero, sparse polynomials $a, b \in \mathbb{Z}[x_7 < \ldots < x_1]$, $\deg(a, x_1) = \deg(b, x_1) + 1 = d$, and $\deg(a, x_i) = \deg(b, x_i) = 1$ for $2 \leq i \leq 7$

	Maple		BPAS			
d	Bézout$(\rho = 0)$	Ducos	Bézout$(\rho = 0)$	Bézout$(\rho = 1)$	SpecBézout$(\rho = 0)$	OptDucos
6	0.00098	0.00372	0.001303	0.001427	0.001553	0.002444
7	0.01148	0.43145	0.080210	0.174460	0.095569	0.279023
8	15.1850	34.8540	7.057270	10.834100	8.380050	22.440500
9	74.1390	327.570	36.8450	66.8430	44.7160	194.4860
10	9941.20	inf	4130.980	6278.240	5686.060	14145.30

Table 7. Comparing the execution time (in seconds) of speculative and caching subresultant algorithms for polynomials in Table 6

	BPAS			
d	SpecBézout$_{(\rho=0)}$	SpecBézout$_{(\rho=2)}$	SpecBézout$_{\texttt{cached}(\rho=2)}$	SpecBézout$_{\texttt{cached}(\rho=\texttt{all})}$
6	0.001553	0.001812	0.001350	0.003519
7	0.095569	0.103801	0.053730	0.213630
8	8.380050	13.10210	5.7240	25.83050
9	44.7160	67.86560	31.12090	136.8930
10	5686.060	8853.10	3856.550	17569.20

As described in Sect. 3.1, polynomial system solving benefits from computing regular GCDs in a bottom-up approach. From a test suite of over 3000 polynomial systems, coming from the literature and collected from MAPLE user-data (see [4, Section 6]) we compare the benefits of (Speculative) Bézout methods for computing subresultants vs BPAS's optimized Ducos algorithm. Table 8 shows this data for some systems of the test suite with at least 5 variables. Table 9 shows systems which are very challenging to solve, requiring at least 50 s. For these hard systems, speculative methods achived a speed-up of up to 1.6× compared to Ducos' method. Note that, in some cases, the regular GCD has high degree and is thus equal to a subresultant of high index. Thus, Ducos' method to compute the entire subresultant chain may be more efficient than repeated calls to the speculative method.

Table 8. Comparing time (in seconds) to solve polynomial systems with `nvar` ≥ 5; system names come from a test suite detailed [4]

SysName	OptDucos	Bézout	SpecBézout	SpecBézout$_{\texttt{cached}}$	OptDucos/ SpecBézout	Bézout/ SpecBézout
Sys2922	7.91041	7.93589	7.95695	7.95698	0.994151	0.997353
Sys2880	5.55801	5.70138	5.46921	5.41538	1.016236	1.042450
Sys2433	8.75830	8.77473	8.75625	8.76812	1.000234	1.002110
Sys2161	1.08153	0.89279	0.56666	0.63128	1.908605	1.575530
Sys2642	8.06066	6.97177	4.89233	3.21021	1.647612	1.425041
Sys2695	3.18267	3.05706	2.98045	2.12872	1.067849	1.025704
Sys2238	8.75708	8.75923	8.75251	8.75813	1.000522	1.000768
Sys2943	6.70348	6.12246	4.54511	4.69512	1.474877	1.347043
Sys1935	4.14390	5.01831	3.01449	1.98466	1.374660	1.664729
Sys2882	2.42182	2.37203	2.38065	2.35716	1.017294	0.996379
Sys2588	4.49268	4.51135	4.49201	4.49792	1.000149	1.004305
Sys2449	1.23251	1.28321	1.24507	1.26588	0.989912	1.030633
Sys2874	6.99887	7.22326	6.99438	7.11027	1.000642	1.032723
Sys2932	6.27556	6.25798	6.31953	6.29113	0.993042	0.990260
Sys2269	1.03128	1.03253	1.03961	1.04012	0.991987	0.993190

Table 9. Comparing time (in seconds) to solve "hard" polynomial systems with `nvar` $\geq$ 5; system names come from a test suite detailed [4]

SysName	OptDucos	Bézout	SpecBézout	SpecBézout$_{\texttt{cached}}$	OptDucos/ SpecBézout	Bézout/ SpecBézout
Sys2797	466.4250	425.8670	386.3810	325.1170	1.207163	1.102194
Sys2539	55.8694	55.8531	55.5113	55.4933	1.006451	1.006157
Sys2681	458.6800	458.5810	458.5360	458.5780	1.000314	1.000098
Sys2745	599.8020	599.3290	599.0610	599.2150	1.001237	1.000447
Sys3335	6406.7400	5843.7300	4799.9700	4801.1200	1.334746	1.217451
Sys2703	322.2940	487.0120	485.8170	491.1520	0.663406	1.002460
Sys2000	55.7026	56.1724	56.3106	57.0079	0.989203	0.997546
Sys2877	2127.4900	1914.5200	1253.8200	1247.4400	1.696807	1.526950

References

1. Abdeljaoued, J., Diaz-Toca, G.M., Gonzalez-Vega, L.: Minors of Bézout matrices, subresultants and the parameterization of the degree of the polynomial greatest common divisor. Int. J. Comput. Math. **81**(10), 1223–1238 (2004)
2. Abdeljaoued, J., Diaz-Toca, G.M., González-Vega, L.: Bézout matrices, subresultant polynomials and parameters. Appl. Math. Comput. **214**(2), 588–594 (2009)
3. Asadi, M., et al.: Basic Polynomial Algebra Subprograms (BPAS) (2021). http://www.bpaslib.org
4. 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)
5. Asadi, M., Brandt, A., Moreno Maza, M.: Computational schemes for subresultant chains. In: Boulier, F., England, M., Sadykov, T.M., Vorozhtsov, E.V. (eds.) CASC 2021. LNCS, vol. 12865, pp. 21–41. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85165-1_3
6. Bareiss, E.H.: Sylvester's identity and multistep integer-preserving Gaussian elimination. Math. Comput. **22**(103), 565–578 (1968)
7. Bini, D., Pan, V.Y.: Polynomial and Matrix Computations: Fundamental Algorithms. Springer, New York (2012). https://doi.org/10.1007/978-1-4612-0265-3
8. Chen, C., Moreno Maza, M.: Algorithms for computing triangular decomposition of polynomial systems. J. Symb. Comput. **47**(6), 610–642 (2012)
9. Diaz-Toca, G.M., Gonzalez-Vega, L.: Various new expressions for subresultants and their applications. Appl. Algebra Eng. Commun. Comput. **15**(3–4), 233–266 (2004). https://doi.org/10.1007/s00200-004-0158-4
10. Ducos, L.: Optimizations of the subresultant algorithm. J. Pure Appl. Algebra **145**(2), 149–163 (2000)
11. von zur Gathen, J., Gerhard, J.: Modern Computer Algebra, 3rd edn. Cambridge University Press, Cambridge (2013)
12. Geddes, K.O., Czapor, S.R., Labahn, G.: Algorithms for Computer Algebra. Springer, New York (1992). https://doi.org/10.1007/b102438
13. Hou, X., Wang, D.: Subresultants with the Bézout matrix. In: Computer Mathematics, pp. 19–28. World Scientific (2000)

14. Jeffrey, D.J.: LU factoring of non-invertible matrices. ACM Commun. Comput. Algebra **44**(1/2), 1–8 (2010)
15. Kerber, M.: Division-free computation of subresultants using Bézout matrices. Int. J. Comput. Math. **86**(12), 2186–2200 (2009)
16. Lemaire, F., Moreno Maza, M., Xie, Y.: The RegularChains library in Maple. ACM SIGSAM Bull. **39**(3), 96–97 (2005)
17. Li, Y.B.: A new approach for constructing subresultants. Appl. Math. Comput. **183**(1), 471–476 (2006)
18. Olver, P.J., Shakiban, C.: Applied Linear Algebra. Prentice Hall, Upper Saddle River (2006)