Fast in-place accumulated bilinear formulae

Jean-Guillaume Dumas, Bruno Grenet

Fast in-place accumulated bilinear formulae

Jean-Guillaume Dumas* Bruno Grenet*

July 20, 2023

Contents

1 Introduction

Bilinear operations are ubiquitous in computer science and in particular in computer algebra and symbolic computation. One of the most fundamental arithmetic operation is the multiplication, and when applied to, e.g., polynomials or matrices, its result is a bilinear function of its inputs.

In terms of arithmetic operations, from the work of [14, 12, 17], many sub-quadratic (resp. sub-cubic) algorithms were developed for these tasks. But these fast algorithms come at the expense of (potentially large) extra temporary space to perform the computation. On the contrary, classical, quadratic (resp. cubic) algorithms, when computed sequentially, quite often require very few (constant) extra registers. Further work then proposed simultaneously "fast" and "in-place" algorithms, for both matrix and polynomial operations [3, 15, 11, 8, 9].

We here propose algorithms to extend the latter line of work for *accumulated* algorithms arising from a bilinear formula. Indeed one of the main ingredient of the latter line of work is to use the (free) space of the output as intermediate storage. When the result has to be accumulated, i.e., if the output is also part of the input, this free space does not even exist.

To be able to design accumulated in-place algorithms we thus relax the in-place model to allow algorithms to also modify their input, therefore to use them as intermediate storage for instance, *provided that they are restored to their initial state after completion of the procedure*. This is in fact a natural possibility in

*Université Grenoble Alpes. Laboratoire Jean Kuntzmann, CNRS, UMR 5224. 150 place du Torrent, IMAG - CS 40700, 38058 Grenoble, cedex 9 France. {firstname.lastname}@univ-grenoble-alpes.fr

many programming environments. Furthermore, this restoration allows for recursive combinations of such procedures, as the (non concurrent) recursive calls will not mess-up the state of their callers.

We propose here a generic technique transforming any bilinear algorithm into an in-place algorithm under this model. This then directly applies to polynomial and matrix multiplication algorithms, including fast ones.

Next we first detail our model for in-place computations in Section 1.1 and recall some quadratic in-place algorithms in Section 1.2. From this, we detail in Section 2 our novel technique for in-place accumulation. Then we apply this technique and further optimizations in order to derive new fast and in-place algorithms for the accumulated multiplication of matrices, Section 3, and polynomials, Section 4.

1.1 In-place model

There exist different models for in-place algorithms. We here choose to call *in-place* an algorithm using only **the space of its inputs, its outputs, and at most $\mathcal{O}(1)$ extra space**. But algorithms are only allowed to modify their inputs, **if their inputs are restored to their initial state** afterwards. This is a less powerful model than when the input is purely read-only, but it turns out to be crucial in our case, especially when we have accumulation operations.

The algorithms we describe are *in-place with accumulation*. The archetypical example is a multiply-accumulate operation $a \mathrel{+}= b \times c$. For such an algorithm, the condition is that b and c are restored to their initial states at the end of the computation, while a (which is also part of the input) is replaced by $a + bc$.

Also, as a variant, by *over-place*, we mean an algorithm where the output replaces (parts of) its input (e.g., like $\vec{a} = b \cdot \vec{a}$). Similarly, we allow all of the input to be modified, provided that the parts of the input that are not the output are restored afterwards. In the following we signal by a "**Read-only:**" tag the parts of the input that the algorithm is not allowed to modify (the other parts are modifiable as long as they are restored). Note that in-place algorithms with accumulation are a special case of over-place algorithms.

For recurive algorithms, some space may be required to store the recursive call stack. (This space is bounded by the recursion depth of the algorithms, and managed in practice by the compiler.) In our complexity summary (Tables 1 and 2), the space complexity includes the stack. Nonetheless, we call *in-place* a recursive algorithm whose only extra space is the call stack.

The main limitations of this model are for black-box inputs, or for different inputs whose representations share some data. For more details on these models, we refer to [15, 8].

1.2 In place and over-place classical algorithms

For the sake of completeness, we recall here, in Algorithm 1, that classical algorithms for matrix or polynomial operations can be performed strictly in-place.

Algorithm 1 Quadratic/cubic in-place accumulated polynomial/matrix multiplication

Input: $A(X)$, $B(X)$, $C(X)$ polynomials of degrees m, n, $m+n$.
Read-only: A, B.
Output: $C(X) \mathrel{+}= A(X)B(X)$

```
for i = 0 to m do
    for j = 0 to n do
        C[i + j] += A[i]B[j];
    end for
end for
```

Input: A, B, C matrices of dimensions $m \times \ell$, $\ell \times n$, $m \times n$.
Read-only: A, B.
Output: $C \mathrel{+}= AB$

```
for i = 0 to m do
    for j = 0 to n do
        for k = 0 to ℓ do
            C_ij += A_ik B_kj;
        end for
    end for
end for
```

2 In-place computation of accumulated bilinear formulae

Karatsuba polynomial multiplication and Strassen matrix multiplication are famous optimizations of bilinear formulae on their inputs: results are linear combinations of products of bilinear combinations of the inputs.

To compute recursively such a formula in-place, the idea is to perform each product one at a time. For each product, both factors are linearly combined in-place into one of the entry beforehand and restored afterwards. Then the product of both entries is accumulated in one part of the output and then distributed to the other parts. The difficulty is to perform this distribution in-place, without recomputing the product. For this we presubtract one output to the other, then accumulate the product to one product, and finally add the newly accumulated output to the other one: overall both outputs just have accumulated the product, in-place. Potential constant factors can also be dealt with pre divisions and post multiplications. Basically we need two kind of in-place operations, and their combinations:

1. In-place accumulation of a quantity multiplied by a (known in advance) invertible constant; this is shown in Eq. (1).
$$\mu\left(\left(\mu^{-1}c\right) \mathrel{+}= m\right) \quad \text{computes in-place} \quad c \leftarrow c + \mu \cdot m \tag{1}$$
2. In-place distribution of the same quantity, without recomputation, to several outputs; this is shown in Eq. (2).
$$\left.\begin{array}{l} d \mathrel{-}= c; \\ c \mathrel{+}= m; \\ d \mathrel{+}= c; \end{array}\right\} \quad \text{computes in-place} \quad \begin{cases} c & \leftarrow c + m \\ d & \leftarrow d + m \end{cases} \tag{2}$$

Example 1 shows how to combine several of these operations, also linearly combining parts of the input.

Example 1. *Suppose, that for some inputs/outputs a, b, c, d, r, s one wants to compute an intermediate product $p = (a + 3b) * (c + d)$ only once and then distribute and accumulate that product to two of its outputs (or results), such that we have both $r \leftarrow r + 5p$ and $s \leftarrow s + 2p$. One way to perform this in-place is first to accumulate $a \mathrel{+}= 3b$ and $c \mathrel{+}= d$, together with pre-dividing r by 5, as in Eq. (1). Then we will compute and accumulate p to r, knowing that now we directly have $p = ac$. But to distribute p to s without recomputing it we first need to prepare s: divide it by 2, and pre-subtract r. This is $s \mathrel{/}= 2$, $s \mathrel{-}= r$. Now we can compute and accumulate the product $r \mathrel{+}= ac$. After this, we can reciprocate (or unroll) all our precomputations (in order to distribute this product to the other result, s, and to restore the inputs, other than those that are also results, to their initial state). For s, another possibility is to directly pre-subtract $2r$ and to post-add $2r$. This is summarized in Eq. (3).*

$$\left.\begin{array}{lll} a \mathrel{+}= 3b; & c \mathrel{+}= d; & r \mathrel{/}= 5; \\ s \mathrel{-}= 2r; & r \mathrel{+}= ac; & s \mathrel{+}= 2r; \\ r \mathrel{\star}= 5; & c \mathrel{-}= d; & a \mathrel{-}= 3b; \end{array}\right\} \quad \textit{computes in-place} \quad \begin{cases} r & \leftarrow r + 5(a + 3b)(c + d) \\ s & \leftarrow s + 2(a + 3b)(c + d) \end{cases} \tag{3}$$

Algorithm 2 shows how to implement this in general, taking into account the constant (or read-only) multiplicative coefficients of all the linear combinations. We suppose that inputs are in three distinct sets: left-hand sides, $\vec{a}$, right-hand sides, $\vec{b}$, and those accumulated to the results, $\vec{c}$. We denote by $\otimes$ the pointwise multiplications of left-hand sides by right-hand sides. Then Algorithm 2 computes $\vec{c} \mathrel{+}= \mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \otimes (\beta\vec{b})$, with α, β and μ matrices of constants.

To simplify the counting of operations, we denote the addition or subtraction of elements, $+=$ or $-=$, by **ADD**, the (tensor) product of elements, $\cdot$, by **MUL**, and the scaling by constants, $\star=$ or $/=$, by **SCA**.

We also denote by $\#x$ (resp. $\sharp x$) the number of non-zero (resp. $\notin \{0, 1, -1\}$) elements in a matrix x.

Theorem 2. *Algorithm 2 is correct, in-place, and requires t* **MUL**, $2(\#\alpha + \#\beta + \#\mu) - 5t$ **ADD** and $2(\sharp\alpha + \sharp\beta + \sharp\mu)$ **SCA** operations.

Algorithm 2 In place bilinear formula

Input: $\vec{a} \in \mathbb{D}^m$, $\vec{b} \in \mathbb{D}^n$, $\vec{c} \in \mathbb{D}^s$; $\alpha \in \mathbb{D}^{t\times m}$, $\beta \in \mathbb{D}^{t\times n}$, $\mu \in \mathbb{D}^{s\times t}$, with no zero-rows in α, β, μ.
Read-only: α, β, μ.
Output: $\vec{c}$ += $\mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \otimes (\beta\vec{b})$.

1: **for** $\ell = 1$ **to** t **do**
2: Let i s.t. $\alpha_{\ell,i} \neq 0$;
3: $a_i \star= \alpha_{\ell,i}$; **for** $\lambda = 1$ **to** m, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** a_i += $\alpha_{\ell,\lambda} a_\lambda$ **end**
4: Let j s.t. $\beta_{\ell,j} \neq 0$;
5: $b_j \star= \beta_{\ell,j}$; **for** $\lambda = 1$ **to** n, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ **do** b_j += $\beta_{\ell,\lambda} b_\lambda$ **end**
6: Let k s.t. $\mu_{k,\ell} \neq 0$;
7: c_k /= $\mu_{k,\ell}$; **for** $\lambda = 1$ **to** s, $\lambda \neq k$, $\mu_{\lambda,\ell} \neq 0$ **do** c_λ −= $\mu_{\lambda,\ell} c_k$ **end**
8: c_k += $a_i \cdot b_j$ {This is the product m_ℓ}
9: **for** $\lambda = 1$ **to** s, $\lambda \neq k$, $\mu_{\lambda,\ell} \neq 0$ **do** c_λ += $\mu_{\lambda,\ell} c_k$ **end**; $c_k \star= \mu_{k,\ell}$; {undo 7}
10: **for** $\lambda = 1$ **to** n, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ **do** b_j −= $\beta_{\ell,\lambda} b_\lambda$ **end**; b_j /= $\beta_{\ell,j}$; {undo 5}
11: **for** $\lambda = 1$ **to** m, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** a_i −= $\alpha_{\ell,\lambda} a_\lambda$ **end**; a_i /= $\alpha_{\ell,i}$; {undo 3}
12: **end for**
13: **return** $\vec{c}$.

Proof. First, as the only used operations (+=, −=, $\star$=, /=) are in-place ones, the algorithm is in-place. Second, the algorithm is correct both for the input and the output: The input is well restored, as $(\alpha_{\ell,i}a_i + \sum \alpha_{\ell,\lambda}a_\lambda - \sum \alpha_{\ell,\lambda}a_\lambda)/\alpha_{\ell,i} = a_i$ and $(\beta_{\ell,j}b_j + \sum \beta_{\ell,\lambda}b_\lambda - \sum \beta_{\ell,\lambda}b_\lambda)/\beta_{\ell,j} = b_j$; The output is correct as $c_\lambda - \mu_{\lambda,\ell}c_k/\mu_{k,\ell} + \mu_{\lambda,\ell}(c_k/\mu_{k,\ell} + a_ib_j) = c_\lambda + \mu_{\lambda,\ell}a_ib_j$ and $(c_k/\mu_{k,\ell} + a_ib_j)\mu_{k,\ell} = c_k + \mu_{k,\ell}a_ib_j$. Third, for the number of operations, Line 3 requires one multiplication by a constant for each non-zero element a_λ in the row and one less addition. But multiplications and divisions by 1 are no-op, and by -1 can be dealt with subtraction. This gives the total of $\#\alpha - t$ additions and $\sharp\alpha$ constant multiplications. Line 5 is similar for each non-zero element b_λ. Line 7 is for each non-zero element in μ. Finally Line 8 performs t multiplications of elements and t additions. The remaining lines double the number of **ADD** and **SCA**. This is $t + 2(\#\alpha + \#\beta + \#\mu - 3t) = 2(\#\alpha + \#\beta + \#\mu) - 5t$ **ADD**. □

For instance, to use Algorithm 2 with matrices or polynomials, each product m_ℓ is in fact computed recursively (in an actual implementation of a fixed formula, one could of course combine more efficiently the pre- and post-computations over the main loop), see next sections.

Remark 3. *Similarly, slightly more generic operations of the form $\vec{c} \leftarrow \vec{\gamma} \otimes \vec{c} + \mu\vec{m}$, for a vector $\gamma \in \mathbb{D}^s$, can also be computed in-place: precompute first $\vec{c} \leftarrow \vec{\gamma} \otimes \vec{c}$, then call Algorithm 2.*

3 In-place Strassen matrix multiplication with accumulation

Considered as 2×2 matrices, the matrix product with accumulation C += $A \cdot B$ could be computed using Strassen-Winograd algorithm by performing the following computations:

$$\begin{aligned}
&\rho_1 \leftarrow a_{11}b_{11}, && \rho_2 \leftarrow a_{12}b_{21},\\
&\rho_3 \leftarrow (-a_{11} - a_{12} + a_{21} + a_{22})b_{22}, && \rho_4 \leftarrow a_{22}(-b_{11} + b_{12} + b_{21} - b_{22}),\\
&\rho_5 \leftarrow (a_{21} + a_{22})(-b_{11} + b_{12}), && \rho_6 \leftarrow (-a_{11} + a_{21})(b_{12} - b_{22}),\\
&\rho_7 \leftarrow (-a_{11} + a_{21} + a_{22})(-b_{11} + b_{12} - b_{22}), &&
\end{aligned} \tag{4}$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \mathrel{+}= \begin{bmatrix} \rho_1 + \rho_2 & \rho_1 - \rho_3 + \rho_5 - \rho_7 \\ \rho_1 + \rho_4 + \rho_6 - \rho_7 & \rho_1 + \rho_5 + \rho_6 - \rho_7 \end{bmatrix}.$$

This algorithm uses 7 multiplications of half-size matrices and $24 + 4$ additions (that can be factored into only $15 + 4$, see [20]: 4 involving A, 4 involving B and 7 involving the products, plus 4 for the accumulation). This can be used recursively on matrix blocks, halved at each iteration, to obtain a sub-cubic algorithm. To save on operations, it is of course interesting to compute the products only once, that is store them in

extra memory chunks. In order to reduce the overall memory footprint, it is then desirable to minimize the number (or the volume) of these extra variables. To date, the best versions that reduce the extra memory space, also overwriting the input matrices (but not putting them back in place) were proposed in [3]. There, an in-place algorithm for the product without accumulation was proposed. But for the accumulated product the best obtained memory footprint, for a sub-cubic algorithm, was 2 temporary blocks per recursive level, thus a total of extra memory required to be $\frac{2}{3}n^2$.

With Algorithm 2 we instead obtain a sub-cubic algorithm for accumulated matrix multiplication with $\mathcal{O}(1)$ extra space requirement.

From Eq. (4) indeed, we can extract the matrices μ, α and β to be used in Algorithm 2 as follows:

$$\mu = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & 1 & 1 & -1 \end{bmatrix} \quad \alpha = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 1 & 1 \end{bmatrix} \quad \beta = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 1 & 0 & -1 \end{bmatrix} \tag{5}$$

All coefficients being 1 or -1 the resulting in-place algorithm can of course compute the accumulation $C \mathrel{+}= AB$ without constant multiplications. It thus requires 7 recursive calls and, from Theorem 2, at most $2(\#\alpha + \#\beta + \#\mu - 3t) = 2(14 + 14 + 14 - 3 * 7) = 42$ block additions. Just like the 24 additions of Eq. (4) can be factored into 15, one can optimize also the in-place algorithm. For instance, looking at α we see that performing the products in the order ρ_6, ρ_7, ρ_3, ρ_5 and accumulating in a_{21} allows to perform all additions/subtractions in A with an optimal number of only 6 operations. This is similar for β if the order ρ_6, ρ_7, ρ_4, ρ_5 is used and accumulation is in b_{12}. Thus ordering for instance ρ_6, ρ_7, ρ_4, ρ_3, ρ_5 will reduce the number of block additions to 26. Now looking at μ, ρ_6 being used only with ρ_1 one can save two more additions in C if they are computed one right before the other. Then ρ_6 and ρ_7 together also saves two more additions in C.

So using, e.g. the ordering $\rho_1, \rho_6, \rho_7, \rho_4, \rho_3, \rho_5, \rho_2$ requires only 22 additions, as shown in Appendix A.

Thus, without thresholds and for powers of two, the dominant term of the overall arithmetic cost is $\frac{28}{3}n^{\log_2(7)}$, for the in-place version, roughly only half more operations than the $6n^{\log_2(7)}$ cost for the version using extra temporaries.

Any bilinear algorithm for matrix multiplication (see, e.g., `https://fmm.univ-lille.fr/`) can be dealt with similarly.

4 In-place polynomial multiplication with accumulation

Algorithm 2 can also be used for polynomial multiplication. One difficulty now is that this does not completely fits the setting, as multiplication of two size-n inputs will in general span a double size-$2n$ output. This is not an issue until one has to distribute separately the two halves of this $2n$ values (or more generally to different parts of different outputs). In the following we show that this can anyway always be done for polynomial multiplications.

4.1 In-place Karatsuba polynomial multiplication with accumulation

For instance, we can immediately obtain an in-place Karatsuba polynomial multiplication this way. Karatsuba polynomial multiplication, indeed, writes as:

$$(Ya_1 + a_0)(Yb_1 + b_0) = a_0b_0 + Y((a_1 + a_0)(b_1 + b_0) - a_0b_0 - a_1b_1) + Y^2(a_1b_1). \tag{6}$$

From Eq. (6) we can extract the associated μ, α, β matrices, as shown in Eq. (7).

$$\mu = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \qquad \alpha = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad \beta = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{7}$$

Then, with $Y = X^t$ and a_i, b_i, c_i polynomials in X (and a_0, b_0, c_0 of degree less than t), this can be detailed, with accumulation, in Eq. (8):

$$\begin{array}{l} A(Y) = Ya_1 + a_0; \quad B(Y) = Yb_1 + b_0; \quad C(Y) = Y^3c_{11} + Y^2c_{10} + Yc_{01} + c_{00}; \\ \quad m_0 = a_0 \cdot b_0 = m_{01}Y + m_{00}; \quad m_1 = (a_0 + a_1) \cdot (b_0 + b_1) = m_{11}Y + m_{10}; \quad m_2 = a_1 \cdot b_1 = m_{21}Y + m_{20}; \\ \quad t_{00} = c_{00} + m_{00}; \quad t_{01} = c_{01} + m_{01} + m_{10} - m_{00} - m_{20}; \\ \quad t_{10} = c_{10} + m_{11} + m_{20} - m_{01} - m_{21}; \quad t_{11} = c_{11} + m_{21}; \\ \textbf{then } C + AB \equiv Y^3t_{11} + Y^2t_{10} + Yt_{01} + t_{00} \end{array} \tag{8}$$

Thus, in order to deal with the distributions of half of the products of Eq. (8), each coefficient in μ in Eq. (7) can be expanded into 2×2 identity blocks, and the middle rows combined two by two, as each tensor product actually spans two sub-parts of the result; we obtain Eq. (9):

$$\mu^{(2)} = \left[\begin{array}{cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 \\ \hline 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{cc|cc|cc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \tag{9}$$

Finally, Eq. (8) then translates into an in-place algorithm thanks to Algorithm 2 and Eqs. (7) and (9).

The first point is that products double the degree: this corresponds to a constraint that the two blocks have to remain together when distributed.

In other words, this means that the $\mu^{(2)}$ matrix needs to be considered two consecutive columns by two consecutive columns. This is always possible if the two columns are of full rank 2. Indeed, consider a 2×2 invertible submatrix $M = \left[\begin{smallmatrix} v & w \\ x & y \end{smallmatrix}\right]$ of these two columns. Then computing $\left[\begin{smallmatrix} c_i \\ c_j \end{smallmatrix}\right] \mathrel{+}= M \left[\begin{smallmatrix} \rho_0 \\ \rho_1 \end{smallmatrix}\right]$ is equivalent to computing a 2×2 version of Eq. (1):

$$M \left(\left(M^{-1} \begin{bmatrix} c_i \\ c_j \end{bmatrix} \right) \mathrel{+}= \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix} \right) \tag{10}$$

The other rows of these two columns can be dealt with as before by pre- and post- multiplying/dividing by a constant and pre- and post- adding/subtracting the adequate c_i and c_j. Now to apply a matrix $M = \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ to a vector of results $\left[\begin{smallmatrix} \vec{u} \\ \vec{v} \end{smallmatrix}\right]$, it is sufficient that one of its coefficient is invertible. W.l.o.g suppose that its upper left element, a, is invertible. Then the in-place evaluation of Eq. (11) performs this application, using the two (known in advance) constants $x = ca^{-1}$ and $y = d - ca^{-1}b$:

$$\left.\begin{array}{l} \vec{u} \star= a \\ \vec{u} \mathrel{+}= b \cdot \vec{v} \\ \vec{v} \star= y \\ \vec{v} \mathrel{+}= x \cdot \vec{u} \end{array}\right\} \quad \text{computes in-place} \quad \begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix} \leftarrow \begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix} = \begin{bmatrix} a\vec{u} + b\vec{v} \\ c\vec{u} + d\vec{v} \end{bmatrix}, \text{ for } x = ca^{-1} \text{ and } y = d - xb \tag{11}$$

Remark 4. *In practice for 2×2 blocks, if a is not invertible, permuting the rows is sufficient since c has to be invertible for the matrix to be invertible: for $J = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$, if $\tilde{M} = \left[\begin{smallmatrix} c & d \\ 0 & b \end{smallmatrix}\right] = J \cdot M$, then $M = J \cdot \tilde{M}$ and $M^{-1} = \tilde{M}^{-1} \cdot J$ so that Eq. (10) just becomes $J \cdot \tilde{M} \left(\left(\tilde{M}^{-1} \left(J \cdot \left[\begin{smallmatrix} c_i \\ c_j \end{smallmatrix}\right] \right) \right) \mathrel{+}= \left[\begin{smallmatrix} \rho_0 \\ \rho_1 \end{smallmatrix}\right] \right)$*

We now have all the tools to produce in-place polynomial algorithms. We start, in Algorithm 3, with a version of Algorithm 2 that regroups the intermediate computations into consecutive blocks.

Algorithm 3 In place bilinear 2 by 2 formula

Input: $\vec{a} \in \mathbb{D}^m$, $\vec{b} \in \mathbb{D}^n$, $\vec{c} \in \mathbb{D}^s$; $\alpha \in \mathbb{D}^{t\times m}$, $\beta \in \mathbb{D}^{t\times n}$, $\mu \in \mathbb{D}^{s\times(2t)} = \begin{bmatrix} M_1 & \dots & M_t \end{bmatrix}$, with no zero-rows in α, β, μ, and s.t. $M_i \in \mathbb{D}^{s\times 2}$ is of full-rank 2 for $i = 1..t$.
Read-only: α, β, μ.
Output: $\vec{c}$ += $\mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \otimes (\beta\vec{b})$, such that $(a_i \cdot b_j)$ fits two result variables c_k, c_l.

1: **for** $\ell = 1$ **to** t **do**
2: Let i s.t. $\alpha_{\ell,i} \neq 0$;
3: $a_i \star= \alpha_{\ell,i}$; **for** $\lambda = 1$ **to** m, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** a_i += $\alpha_{\ell,\lambda} a_\lambda$ **end**
4: Let j s.t. $\beta_{\ell,j} \neq 0$;
5: $b_j \star= \beta_{\ell,j}$; **for** $\lambda = 1$ **to** n, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ **do** b_j += $\beta_{\ell,\lambda} b_\lambda$ **end**
6: Let k, f s.t. $M = \begin{bmatrix} \mu_{k,2\ell} & \mu_{k,2\ell+1} \\ \mu_{f,2\ell} & \mu_{f,2\ell+1} \end{bmatrix}$ is invertible;
7: $\begin{bmatrix} c_k \\ c_f \end{bmatrix} \leftarrow M^{-1} \begin{bmatrix} c_k \\ c_f \end{bmatrix}$ {Via Eq. (11) and Remark 4}
8: **for** $\lambda = 1$ **to** s, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell} \neq 0$ **do** c_λ −= $\mu_{\lambda,2\ell} c_k$ **end**
9: **for** $\lambda = 1$ **to** s, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell+1} \neq 0$ **do** c_λ −= $\mu_{\lambda,2\ell+1} c_f$ **end**
10: $\begin{bmatrix} c_k \\ c_f \end{bmatrix}$ += $a_i \cdot b_j$ {This is the accumulation of the product $\begin{bmatrix} m_k \\ m_f \end{bmatrix}$}
11: **for** $\lambda = 1$ **to** s, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell+1} \neq 0$ **do** c_λ += $\mu_{\lambda,2\ell+1} c_f$ **end** {undo 9}
12: **for** $\lambda = 1$ **to** s, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell} \neq 0$ **do** c_λ += $\mu_{\lambda,2\ell} c_k$ **end** {undo 8}
13: $\begin{bmatrix} c_k \\ c_f \end{bmatrix} \leftarrow M \begin{bmatrix} c_k \\ c_f \end{bmatrix}$ {Via Eq. (11) and Remark 4, undo 7}
14: **for** $\lambda = 1$ **to** n, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ **do** b_j −= $\beta_{\ell,\lambda} b_\lambda$ **end**; b_j /= $\beta_{\ell,j}$; {undo 5}
15: **for** $\lambda = 1$ **to** m, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** a_i −= $\alpha_{\ell,\lambda} a_\lambda$ **end**; a_i /= $\alpha_{\ell,i}$; {undo 3}
16: **end for**
17: **return** $\vec{c}$.

Theorem 5. *Algorithm 3 is correct, in-place, and requires* t **MUL**-2D, $2(\#\alpha + \#\beta + \#\mu - t)$ **ADD** and $2(\sharp\alpha + \sharp\beta + \sharp\mu + 2t)$ **SCA** operations.

Proof. Thanks to Eqs. (10) and (11) and Remark 4, correctness is similar to that of Algorithm 2 in Theorem 2. For the number of operations, Eq. (11) requires 4 **SCA** and 2 **ADD** operations and is called $2t$ times. The rest is similar to Algorithm 2 and amounts to $2t + 2(\#\alpha - t + \#\beta - t + \#\mu - 2t) + (2t)2$ **ADD** and $2(\sharp\alpha + \sharp\beta + \sharp\mu - 2t) + (2t)4$ **SCA** operations. □

There remains now to use a double expansion of the output matrix $\mu \in \mathbb{D}^{s\times t}$ to simulate the double size of the intermediate products (**MUL**-2D), producing $\mu \in \mathbb{D}^{s\times(2t)}$ matrix $\mu^{(2)}$, as in Eq. (9), that is used as an input in Algorithm 3. This is shown in Algorithm 4.

Algorithm 4 Double expansion of output matrix

Input: $\mu \in \mathbb{D}^{m\times n}$ representing the linear distribution of n values to m outputs.
Output: $\mu^{(2)} \in \mathbb{D}^{(m+1)\times(2n)}$, representing the linear distribution of n double-size values to $m+1$ outputs.

Let $\mu^{(2)} = 0^{(m+1)\times(2n)}$;
for $j = 0$ **to** n **do**
  for $i = 0$ **to** m **do**
    $\mu^{(2)}(i, 2j) = \mu(i, j)$;
    $\mu^{(2)}(i+1, 2j+1) = \mu(i, j)$;
  end for
end for
return $\mu^{(2)}$.

Lemma 6. *Algorithm 4 is correct.*

Proof. It is sufficient to note that by expanding each coefficient to a 2×2 identity, then adding two successive rows is just a 2×2 merging of the last row of an identity with the first row of another identity. Therefore the resulting matrix, $\mu^{(2)}$, has for non-zero entries, exactly twice those of μ. □

Now we prove, in Lemma 7, that in fact any double expansion of a representative matrix is suitable for the in-place computation of Algorithm 3.

Lemma 7. *If μ does not contain any zero column, then each pair of columns of $\mu^{(2)}$, resulting from the expansion of a single column in μ, contains an invertible lower triangular 2×2 submatrix.*

Proof. Consider the top most non-zero element of a column. It is expanded as a 2×2 identity matrix whose second row is merged with the first row of the next identity matrix: in picture, $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$ is expanded to $\left[\begin{smallmatrix} a & 0 \\ b & a \\ * & b \end{smallmatrix}\right]$. □

For instance with $m_{00} + Y m_{01} = a_0 b_0 = \rho_0 + Y \rho_1$, consider the upper left 2×2 block of $\mu^{(2)}$ in Eq. (9), that is $M = \left[\begin{smallmatrix} 1 & 0 \\ -1 & 1 \end{smallmatrix}\right]$, whose inverse is $M^{-1} = \left[\begin{smallmatrix} 1 & 0 \\ 1 & 1 \end{smallmatrix}\right]$. One has first to precompute $M^{-1} \left[\begin{smallmatrix} c_{00} \\ c_{01} \end{smallmatrix}\right]$, that is nothing on c_{00} and c_{01} += c_{00} for the second coefficient. Then, afterwards, the third row, for c_{10}, will just be $-m_{01}$: for this just pre-add c_{10} += c_{01}, and post-subtract c_{10} −= c_{01} after the product actual computation. This example is exactly lines 14 to 18 of Algorithm 5 thereafter. One could also consider instead the first and last rows, but in this particular case without any advantage in term of computations. To complete Eq. (8), the computation of m_2 is dealt with in the same manner, while that of m_1 is direct in the results (and requiring pre and post additions of its inputs). This gives then the whole of Algorithm 5.

The second point is to deal with unbalanced dimensions and degrees for $Y = X^t$ and recursive calls. For this, first separate the largest polynomial in two parts, so that two sub-products are performed: a large balanced one, and, recursively, a smaller unbalanced one. Then, for the balanced case, the idea is to ensure that three out of four parts of the result, t_{00}, t_{01} and t_{10}, have the same size and that the last one t_{11} is smaller. This ensures that all accumulations can be performed in-place. The obtained procedure is given in Algorithm 5.

Proposition 8. *Algorithm 5 is correct and requires $\mathcal{O}\big(mn^{\log_2(3)-1}\big)$ operations.*

Proof. With the above analysis, correctness comes from that of Algorithms 3 and 4 applied to Eq. (7). When $m = n$, with 3 recursive calls and $\mathcal{O}(n)$ extra operations, the algorithm thus requires overall $\mathcal{O}(n^{\log_2(3)})$ operations. Otherwise, it requires $\lfloor \frac{m}{n} \rfloor$ equal degree calls, then a recursive call with n and $m \bmod n$. Now, let $u_1 = m$ and $u_2 = n$ and if the Euclidean algorithm on them requires k steps, let u_i for $i = 1..k$ denote the successive residues within this Euclidean algorithm (and $u_k \in \{0, 1\}$). Let $\kappa = k - 1$ if $u_k = 0$ and $\kappa = k$ otherwise. With these notations, Algorithm 5 requires less than $\mathcal{O}(\sum_{i=1}^{\kappa-1} \lfloor \frac{u_i}{u_{i+1}} \rfloor u_{i+1}^{\log_2(3)}) \leq \mathcal{O}(\sum_{i=1}^{\kappa-1} u_i u_{i+1}^{\log_2(3)-1})$ operations. But, $u_{i+1} \leq u_2 = n$ and we let $s_i = u_i + u_{i+1}$ so that $u_i = s_i - u_{i+1} \leq s_i$. Now, from [10, Corollary 2.6], we have that $s_i \leq s_1 (2/3)^{i-1}$. Thus the number of operations is bounded by $\mathcal{O}(\sum_{i=1}^{\kappa-1} s_i n^{\log_2(3)-1}) \leq \mathcal{O}(n^{\log_2(3)-1} s_1 (\frac{1}{1-(2/3)} - 1)) = \mathcal{O}(n^{\log_2(3)-1}(m+n)) = \mathcal{O}(mn^{\log_2(3)-1})$. □

Algorithm 5 is compared with previous Karatsuba-like algorithms for polynomial multiplications designed to reduce their memory footprint in Table 1 (see also [7, Table 2.2]).

Table 1: Reduced-memory algorithms for Karatsuba polynomial multiplication

Alg.	Memory	inputs	accumulation
[18]	$n + 5 \log n$	read-only	✗
[15, 16]	$5 \log n$	read-only	✗
[8]	$\mathcal{O}(1)$	read-only	✗
Algorithm 5	$5 \log n$	mutable	✓

For the complexity bound, all coefficients being 1 or -1 the resulting in-place algorithm can thus compute in fact the accumulation C += AB without constant multiplications. Also, the de-duplication of each

Algorithm 5 In-place Karatsuba polynomial multiplication with accumulation

Input: $A(X)$, $B(X)$, $C(X)$ polynomials of degrees m, n, $m+n$ with $m \geq n$.
Output: C += AB

1: **if** $n \leq$ Threshold **then** {Constant-time if Threshold $\in \mathcal{O}(1)$}
2: **return** the quadratic in-place polynomial multiplication. {Algorithm 1}
3: **else if** $m > n$ **then**
4: Let $A(X) = A_0(X) + X^{n+1}A_1(X)$
5: $C_{0..2n}$ += A_0B {Recursive call}
6: **if** $m \geq 2n$ **then**
7: $C_{(n+1)..(n+m)}$ += A_1B {Recursive call}
8: **else**
9: $C_{(n+1)..(n+m)}$ += BA_1 {Recursive call}
10: **end if**
11: **else**
12: Let $t = \lceil (2n+1)/4 \rceil$; {$t - 1 \geq 2n - 3t$ and thus $t > n - t$}
13: Let $A = a_0 + X^t a_1$; $B = b_0 + X^t b_1$; $C = c_{00} + c_{01}X^t + c_{10}X^{2t} + c_{11}X^{3t}$; {$d^\circ c_{11} = 2n - 3t$}
{Iteration for m_0}
14: c_{01} += c_{00};
15: c_{10} += c_{01};
16: $\begin{bmatrix} c_{00} \\ c_{01} \end{bmatrix}$ += $a_0 \cdot b_0$ {Recursive call}
17: c_{10} −= c_{01}; {this is $c_{10} - m_{01}$}
18: c_{01} −= c_{00}; {this is $c_{01} + m_{01} - m_{00}$}
{Iteration for m_1}
19: a_0 += a_1; {$d^\circ a_0 = t \geq n - t = d^\circ a_1$}
20: b_0 += b_1;
21: $\begin{bmatrix} c_{01} \\ c_{10} \end{bmatrix}$ += $a_0 \cdot b_0$ {Recursive call}
22: b_0 −= b_1;
23: a_0 −= a_1;
{Iteration for m_2}
24: c_{10} += c_{11};
25: c_{01} += c_{10};
26: $\begin{bmatrix} c_{10} \\ c_{11} \end{bmatrix}$ += $a_1 \cdot b_1$ {Recursive call}
27: c_{01} −= c_{10}; {this is $c_{01} + m_{01} - m_{00} + m_{10} - m_{20}$}
28: c_{10} −= c_{11}; {this is $c_{10} - m_{01} + m_{11} + m_{20} - m_{21}$}
29: **end if**
30: **return** C.

recursive output enables some natural reuse, so in fact there is a cost of $2(\#\alpha - t + \#\beta - t) = 2(4-3+4-3)$ with $t = 3$, and $2(2(\#\mu - t) = 4(5-3) = 2(\#\mu^{(2)} - 2t)$, for a total of at most 12 block additions and 3 recursive accumulated calls. The simple application of Eq. (8) would require 8 additions. Thus, without thresholds and for powers of two, the dominant term of the overall cost only goes from $10n^{\log_2(3)}$, for the original version, to $14n^{\log_2(3)}$, for the fully in-place version.

4.2 Further bilinear polynomial multiplications

We have shown that any bilinear algorithm can be transformed into an in-place version. This approach thus also works for any Toom-k algorithm using $2k-1$ interpolations points instead of the three points of Karatsuba (Toom-2).

For instance Toom-3 uses interpolations at $0, 1, -1, 2, \infty$. Therefore, α and β are the Vandermonde matrices of these points for the 3 parts of the input polynomials and μ is the inverse of the Vandermonde matrix of these points for the 5 parts of the result, as shown in Eq. (12) thereafter.

$$\mu = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & -2 & 4 & -8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{3} & -1 & \frac{1}{6} & -2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ -\frac{1}{2} & \frac{1}{6} & \frac{1}{2} & -\frac{1}{6} & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
$$\alpha = \beta = \begin{bmatrix} 0^0 & 0^1 & 0^2 \\ 1^0 & 1^1 & 1^2 \\ (-1)^0 & (-1)^1 & (-1)^2 \\ (-2)^0 & (-2)^1 & (-2)^2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \tag{12}$$

With the same kind of duplication as in Eq. (9), apart from the recursive calls, the initially obtained operation count is $2(11+11-2*5)+2(2(16-5)) = 68$ additions and $2(2+2+2(11)) = 52$ scalar multiplications. Following the optimization of [2], we see in α and β that the evaluations at 1 and -1 (second and third rows) share one addition. As they are successive in our main loop, subtracting one at the end of the second iteration, then followed by re-adding it at the third iteration can be optimized out. This is 2 less operations. Together with shared coefficients in the rows of μ, some further optimizations of [2] can probably also be applied, where the same multiplicative constants appear at successive places.

4.3 Fast bilinear polynomial multiplication

When sufficiently large roots of unity exist, polynomial multiplications can be computed fast in our in-place model via a discrete Fourier transform and its inverse, as shown in Algorithm 6, for power of two dimensions, and in Algorithm 7, for general dimensions.

Let $F \in \mathbb{D}[X]$ of degree $< n$ and w be a principal n-th root of unity, where $n = 2^p$. The discrete Fourier transform of F at w is defined as $\mathsf{DFT}_n(F, w) = (F(w^0), F(w^1), \ldots, F(w^{n-1}))$. The map is invertible, of inverse $\mathsf{DFT}_n^{-1}(\cdot, w) = \frac{1}{n}\mathsf{DFT}_n(\cdot, w^{-1})$. It is known that the DFT can be computed over-place, replacing the input by the output [4]. Actually, for over-place algorithms and their extensions to the *truncated Fourier transform*, it is more natural to work with the *bit-reversed DFT*. To describe it, let $[i]_p$ be the length-p bit reversal of $i = \sum_{j=0}^{p-1} d_j 2^j$, $d_j \in \{0, 1\}$, defined by $[i]_p = \sum_{j=0}^{p-1} d_j 2^{p-j}$. The bit-reversed DFT is $\mathsf{brDFT}_n(F, w) = (F(w^{[0]_p}), F(w^{[1]_p}), \ldots, F(w^{[n-1]_p}))$. If $\pi : \{0, \ldots, 2^p - 1\} \to \{0, \ldots, 2^p - 1\}$ denotes the bit-reversal permutation (that is $\pi(i) = [i]_p$), we have $\mathsf{brDFT}_n(\cdot, w) = \pi \circ \mathsf{DFT}_n(\cdot, w)$. Its inverse is $\mathsf{brDFT}_n^{-1}(\cdot, w) = \frac{1}{n}\mathsf{DFT}_n(\cdot, w^{-1}) \circ \pi = \frac{1}{n}\mathsf{DFT}_n(\pi(\cdot), w^{-1})$ since π is an involution.

Remark 9. *The Fast Fourier Transform (FFT) algorithm has two main variants:* decimation in time *(DIT) and* decimation in frequency *(DIF). Both algorithms can be performed over-place, replacing the input by the*

output. Without applying any permutation to the entries of the input/output vector, the over-place DIF FFT algorithm naturally computes $\mathsf{brDFT}_n(\cdot, w)$*, while the over-place DIT FFT algorithm on* w^{-1} *computes* $\mathsf{brDFT}_n^{-1}(\cdot, w)$*.*

Algorithm 6 IP2pow: In-place power of two multiplication with accumulation

Input: $\vec{a}$, $\vec{b}$ and $\vec{c}$ of length 2^L, 2^L and 2^{L+1}, containing the coefficients of A, B, $C \in \mathbb{D}[X]$ respectively; $w \in \mathbb{D}$ primitive 2^{L+1}-th root of unity.

Output: $\vec{c}$ contains the coefficients of $C + A \cdot B$.

1: Let $n = 2^L$;
2: $\vec{c} \leftarrow \mathsf{brDFT}_{2n}(\vec{c}, w)$; {over-place}
3: $\vec{a} \leftarrow \mathsf{brDFT}_n(\vec{a}, w^2)$; $\vec{b} \leftarrow \mathsf{brDFT}_n(\vec{b}, w^2)$ {over-place}
4: **for** $i = 0$ **to** $n - 1$ **do** c_i += $a_i \times b_i$ **end**
5: $\vec{a} \leftarrow \mathsf{brDFT}_n^{-1}(\vec{a}, w^2)$; $\vec{b} \leftarrow \mathsf{brDFT}_n^{-1}(\vec{b}, w^2)$ {Undo 3 over-place}
6: **for** $i = 0$ **to** $n - 1$ **do** $a_i \star= w^i$; $b_i \star= w^i$ **end** {see Remark 10}
7: $\vec{a} \leftarrow \mathsf{brDFT}_n(\vec{a}, w^2)$; $\vec{b} \leftarrow \mathsf{brDFT}_n(\vec{b}, w^2)$ {over-place}
8: **for** $i = 0$ **to** $n - 1$ **do** c_{i+n} += $a_i \times b_i$ **end**
9: $\vec{a} \leftarrow \mathsf{brDFT}_n^{-1}(\vec{a}, w^2)$; $\vec{b} \leftarrow \mathsf{brDFT}_n^{-1}(\vec{b}, w^2)$ {Undo 7 over-place}
10: **for** $i = 0$ **to** $n - 1$ **do** a_i /= w^i; b_i /= w^i **end** {Undo 6}
11: **return** $\vec{c} \leftarrow \mathsf{brDFT}_{2n}^{-1}(\vec{c}, w)$

Remark 10. *To compute a geometric progression, in-place, as in Line 6 of Algorithm 6, one can of course use an extra variable (say initialized to* $x = 1$*) storing and updating* w^i *along the loop (using* $a_i \star= x$*;* $x \star= w$*; at each step). This requires* $\mathcal{O}(1)$ *extra space. One can further reduce this extra space to nothing, if needed, by using a non-zero input slot instead. For instance, suppose indeed that* $\exists j, a_j \neq 0$*. Then pre-divide every other coefficients by* a_j *and use this particular variable to update and store* $a_j w^i$ *(just computing* $a_i \star= a_j$*;* $a_j \star= w$*; at each step). Finally re-divide* a_j *by* w*,* $n - j + 1$ *times.*

Theorem 11. *Using an over-place* brDFT *algorithm with complexity bounded by* $\mathcal{O}(n \log n)$*, Algorithm 6 is correct, in-place and has complexity bounded by* $\mathcal{O}(n \log n)$*.*

Proof. The algorithm follows the pattern of the standard FFT-based multiplication algorithm. Our goal is to compute $\mathsf{brDFT}_{2n}(A, w)$, $\mathsf{brDFT}_{2n}(B, w)$ and $\mathsf{brDFT}_{2n}(C, w)$, then obtain $\mathsf{brDFT}_{2n}(C + AB, w)$ and finally $C + AB$ using an inverse brDFT. Computations on C and then $C + AB$ are performed over-place using any standard over-place brDFT algorithm. The difficulty happens for A and B that are stored in length-n arrays. We use the following property of the bit reversed order: for $k < n/2$, $[k]_p = 2[k]_{p-1}$, and for $k \geq n/2$, $[k]_p = 2[k - n/2]_{p-1} + 1$. Therefore, the first n coefficients of $\mathsf{brDFT}_{2n}(A, w)$ are $(A(w^{2[0]_{p-1}}), \dots A(w^{2[\frac{n}{2}-1]_{p-1}})) = \mathsf{brDFT}_n(A, w^2)$. Similarly, the next n coefficients are $\mathsf{brDFT}_n(A(wX), w^2)$. Therefore, one can compute $\mathsf{brDFT}(A, w^2)$ and $\mathsf{brDFT}(B, w^2)$ in $\vec{a}$ and $\vec{b}$ respectively, and update the first n entries of $\vec{c}$. Next we restore $\vec{a}$ and $\vec{b}$ using $\mathsf{brDFT}_n^{-1}(\cdot, w^2)$. We compute $A(wX)$ and $B(wX)$ and again $\mathsf{brDFT}(A(wX), w^2)$ and $\mathsf{brDFT}(B(wX), w^2)$ to update the last n entries of $\vec{c}$. Finally, we restore $\vec{a}$ and $\vec{b}$ and perform the inverse brDFT on $\vec{c}$. The cost is dominated by the ten $\mathsf{brDFT}^{\pm 1}$ computations. □

The case where n is not a power of two is loosely similar, using as a routine a truncated Fourier transform (TFT) rather than a DFT [19]. Let w be an N-th root of unity for some $N = 2^p$. The length-n (bit-reversed) TFT of a polynomial $F \in \mathbb{D}[X]$, $n < N$, is $\mathsf{brTFT}_n(F, w) = (F(w^{[0]_p}), \dots, F(w_p^{[n-1]}))$, that is the n first coefficients of $\mathsf{brDFT}_N(F, w)$. As for the (bit-reversed) DFT, the (bit-reversed) TFT and its inverse can be computed over-place [11, 16, 1, 5].

Given inputs A and $B \in \mathbb{D}[X]$ of respective lengths m and n and an output $C \in \mathbb{D}[X]$ of length $m+n-1$, we aim to replace C by $C + AB$. The idea is first to replace C by $\mathsf{brTFT}_{m+n-1}(C, w)$ where w is a 2^p-th principal root of unity, $2^p \geq m+n-1$. That is, the vector $\vec{c}$ now contains as its i-th entry the value $C(w^{[i]_p})$.

The goal is then to replace $C(w^{[i]_p})$ by $C(w^{[i]_p}) + A(w^{[i]_p})B(w^{[i]_p})$, for $i = 0$ to $m+n-2$. We cannot compute the length $m+n-1$ brTFT's of A and B since this takes too much space. Instead, we will progressively compute some parts of these brTFT's by means of (standard) brDFT's, and update $\vec{c}$ accordingly. The starting point of this strategy is the following lemma.

Lemma 12 ([11, 16])**.** *Let $F \in \mathbb{D}[X]$, ℓ and s be two integers such that 2^ℓ divides s and w be a 2^p-th principal root of unity. Define $F_{s,\ell}(X) = F(w^{[s]_p}X) \bmod X^{2^\ell}-1$. Then*

$$\mathsf{brDFT}_{2^\ell}(F_{s,\ell}, w^{2^{p-\ell}}) = (F(w^{[s]_p}), \ldots, F(w^{[s+2^\ell-1]_p})).$$

Proof. Let $w_\ell = w^{2^{p-\ell}}$. This is a principal 2^ℓ-th root of unity since w is a principal 2^p-th root of unity. In particular, for any $i < 2^\ell$, $F_{s,\ell}(w_\ell^{[i]_\ell}) = F(w^{[s]_p} w_\ell^{[i]_\ell})$. Now, $w_\ell^{[i]_\ell} = w^{[i]_p}$ since $2^{p-\ell}[i]_\ell = [i]_p$. Furthermore, $[s]_p + [i]_p = [s+i]_p$ since $i < 2^\ell$ and 2^ℓ divides s. Finally, $F_{s,\ell}(w_\ell^{[i]_\ell}) = F(w^{[s+i]_p})$. □

Corollary 13. *Let $F \in \mathbb{D}[X]$ stored in an array $\vec{f}$ of length n, ℓ and k be two integers and w be a 2^p-th principal root of unity, with $2^\ell \le n$ and $(k+1)2^\ell \le 2^p$. There exist an algorithm $\mathsf{partTFT}_{k,\ell}(\vec{f}, w)$ that replaces the first 2^ℓ entries of $\vec{f}$ by $F(w^{[k\cdot 2^\ell]_p})$, …, $F(w^{[(k+1)\cdot 2^\ell-1]_p})$, and an inverse algorithm $\mathsf{partTFT}^{-1}_{k,\ell}$ that restores $\vec{f}$ to its initial state. Both algorithms use $O(1)$ extra space and have complexity $O(n + \ell \cdot 2^\ell)$.*

Proof. Algorithm $\mathsf{partTFT}_{k,\ell}(\vec{f}, w)$ is the following:

1: **for** $i = 0$ **to** $n-1$ **do** $f_i \star= w^{i[k\cdot 2^\ell]_p}$ **end**
2: **for** $i = 2^\ell$ **to** $n-1$ **do** f_{i-2^ℓ} += f_i **end**
3: $\vec{f}_{0..2^\ell-1} \leftarrow \mathsf{brDFT}_{2^\ell}(\vec{f}_{0..2^\ell-1}, w^{2^{p-\ell}})$

Its correctness is ensured by Lemma 12. Its inverse algorithm $\mathsf{partTFT}^{-1}_{k,\ell}(\vec{f}, w)$ does the converse:

1: $\vec{f}_{0..2^\ell-1} \leftarrow \mathsf{brDFT}^{-1}_{2^\ell}(\vec{f}_{0..2^\ell-1}, w^{2^{p-\ell}})$
2: **for** $i = 2^\ell$ **to** $n-1$ **do** f_{i-2^ℓ} −= f_i **end**
3: **for** $i = 0$ **to** $n-1$ **do** f_i /= $w^{i[k\cdot 2^\ell]_p}$ **end**

In both algorithms, the call to $\mathsf{brDFT}^{\pm 1}$ has cost $O(\ell \cdot 2^\ell)$, and the two other steps have cost $O(n)$. □

To implement the previously sketched strategy, we assume that $m \le n$ for simplicity. We let ℓ, t be such that $2^\ell \le m < 2^{\ell+1}$ and $2^{\ell+t} \le n < 2^{\ell+t+1}$. Using $\mathsf{partTFT}^{\pm 1}$, we are able to compute $(A(w^{[k\cdot 2^\ell]_p}), \ldots, A(w^{[(k+1)\cdot 2^\ell-1]_p}))$ for any k, and restore A in its initial state afterwards. Similarly, we can compute $(B(w^{[k\cdot 2^{\ell+t}]_p}), \ldots, B(w^{[(k+1)\cdot 2^{\ell+t}-1]_p}))$ and restore B.

Theorem 14. *Algorithm 7 is correct and in-place. If the algorithm brDFT used inside $\mathsf{partTFT}$ has complexity $O(n \log n)$, its running time is $O((m+n)\log(m+n))$.*

Proof. The fact that the algorithm is in-place comes from Corollary 13. The only slight difficulty is to produce, fast and in-place, the relevant roots of unity. This is actually dealt with in the original over-place TFT algorithm [11] and can be done the same way here.

To assess its correctness, first note that the values of line 4 are computed so that $2^\ell \le r, m$ and $2^{\ell+t} \le r, n$. One iteration of the while loop update the entries c_k to $c_{k+2^{\ell+t}-1}$ where $k = m+n-1-r$. To this end, we first compute $B(w^{[k\cdot 2^{\ell+t}]_p})$ to $B(w^{[(k+1)\cdot 2^{\ell+t}-1]_p})$ in $\vec{b}$ using $\mathsf{partTFT}$. Then, since $\vec{a}$ may be too small to store $2^{\ell+t}$ values, we compute the corresponding evaluations of A by groups of 2^ℓ, using a smaller $\mathsf{partTFT}$. After each computation in $\vec{a}$, we update the corresponding entries in $\vec{c}$ and restore $\vec{a}$. Finally, at the end of the iteration, entries k to $k+2^{\ell+t}-1$ of $\vec{c}$ have been updated and $\vec{b}$ can be restored. This proves the correctness of the algorithm.

We now bound the complexity of the algorithm. Since $m \le n$, our aim is to bound it by $O(n \log n)$. Let us first bound the number of iterations of the while loop. We identify two phases, first iterations where $r \ge n$ and then iterations with $r < n$. During the first phase, $2^{\ell+t} > \frac{n}{2}$ entries of $\vec{c}$ are updated at each

Algorithm 7 In-place fast polynomial multiplication with accumulation

Input: $\vec{a}$, $\vec{b}$ and $\vec{c}$ of length m, n and $m+n-1$, $m \leq n$, containing the coefficients of A, B, $C \in \mathbb{D}[X]$ respectively; $w \in \mathbb{D}$ principal 2^p-th root of unity with $2^{p-1} < m+n-1 < 2^p$
Output: $\vec{c}$ contains the coefficients of $C + A \cdot B$.

$\vec{c} \leftarrow \mathsf{brTFT}_{m+n-1}(\vec{c}, w)$; {over-place}
$r \leftarrow m+n-1$
while $r \geq 0$ **do**
  $\ell \leftarrow \lfloor \log_2(\min(r,m)) \rfloor$; $t \leftarrow \lfloor \log_2 \min(r,n) \rfloor - \ell$; $k \leftarrow m+n-1-r$
  $\vec{b} \leftarrow \mathsf{partTFT}_{k,\ell+t}(\vec{b}, w)$ {over-place: $B(w^{[k\cdot 2^{\ell+t}]_p}), \ldots, B(w^{[(k+1)\cdot 2^{\ell+t}-1]_p})$}
  for $s = 0$ **to** $2^t - 1$ **do**
    $\vec{a} \leftarrow \mathsf{partTFT}_{s+k\cdot 2^t,\ell}(\vec{a}, w)$ {over-place: $A(w^{[(k\cdot 2^t+s)2^\ell]_p}), \ldots, A(w^{[(k\cdot 2^t+s+1)2^\ell-1]_p})$}
    for $i = 0$ **to** $2^\ell - 1$ **do** $c_{i+(k\cdot 2^t+s)2^\ell}$ += $a_i b_{i+s\cdot 2^\ell}$ **end**
    $\vec{a} \leftarrow \mathsf{partTFT}^{-1}_{s+k\cdot 2^t,\ell}(\vec{a}, w)$ {Undo 7 over-place}
  end for
  $\vec{b} \leftarrow \mathsf{partTFT}^{-1}_{k,\ell+t}(\vec{b}, w)$ {Undo 5 over-place}
  r -= $2^{\ell+t}$
end while
return $\vec{c} \leftarrow \mathsf{brTFT}^{-1}_{m+n-1}(\vec{c}, w)$

iteration, hence the first phase has at most 3 iterations. In the second phase, $2^{\ell+t} > \frac{r}{2}$ entries are updated per iteration. The second phase starts with $r < n$ and each iteration decreases r by half, hence the second phase has at most $\log n$ iterations.

In one iteration, the costs come from calls to $\mathsf{partTFT}^{\pm 1}$. One call to $\mathsf{partTFT}^{\pm 1}$ with an input of size m and a transform of length 2^ℓ has cost $O(m + \ell \cdot 2^\ell)$. To compute the cost of one iteration, we separately count the contributions due to the linear term m and to the non-linear term $\ell \cdot 2^\ell$ in this complexity. In one iteration, there are two calls to $\mathsf{partTFT}^{\pm 1}$ on $\vec{b}$ and 2^{t+1} calls to $\mathsf{partTFT}^{\pm 1}$ on $\vec{a}$. The contribution of the linear terms for these calls is thus $O(n + m \cdot 2^t) = O(n)$ since $m \cdot 2^t < 2^{\ell+1+t} \leq 2n$. Since there are $\log n$ iterations, the global cost due to these linear terms is $O(n \log n)$.

The cost due to the non-linear terms in one iteration is $O((\ell + t) \cdot 2^{\ell+t})$. In the first phase, $2^{\ell+t} \leq n$ and these costs sum to $O(n \log n)$. In the second phase, $2^{\ell+t} \leq r$. Let r_i be the value of r during the i-th iteration of the second phase: $r_1 < n$ and $r_{i+1} \leq r_i/2$ hence $r_i < n/2^{i-1}$. The global cost of the DFT's of the second phase is then $O(\sum_i \frac{n}{2^i} \log \frac{n}{2^i}) = O(n \log n)$. □

Algorithm 7 is compared with previous FFT-based algorithms for polynomial multiplications designed to reduce their memory footprint in Table 2 (see also [7, Table 2.2]).

Table 2: Reduced-memory algorithms for FFT polynomial multiplication

Alg.	Memory	inputs	accumulation
[4]	$2n$	read-only	✗
[15]	$\mathcal{O}(2^{\lceil \log_2 n \rceil} - n)$	read-only	✗
[11]	$\mathcal{O}(1)$	read-only	✗
Algorithm 7	$\mathcal{O}(1)$	mutable	✓

5 Conclusion

We here provide a generic technique mapping any bilinear formula into an in-place algorithm. This allows us for instance to provide the first accumulated in-place Strassen-like matrix multiplication algorithm. This

also allows use to provide fast in-place accumulated polynomial multiplications algorithms.

Many further accumulated algorithm can then be reduced to these fundamental building blocks, see for instance Toeplitz, circulant, convolutions or remaindering operations in [6].

References

[1] Andrew Arnold. A new truncated Fourier transform algorithm. In Manuel Kauers, editor, *ISSAC'2013, Proceedings of the 2013 International Symposium on Symbolic and Algebraic Computation, Boston, USA*, pages 15–22, New York, June 2013. ACM Press. doi:10.1145/2465506.2465957.

[2] Marco Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, volume 4547 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2007. doi:10.1007/978-3-540-73074-3_10.

[3] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In May [13], pages 135–143. doi:10.1145/1576702.1576713.

[4] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965. doi:10.1090/S0025-5718-1965-0178586-1.

[5] Nicholas Coxon. An in-place truncated Fourier transform. *Journal of Symbolic Computation*, 110:66–80, 2022. doi:https://doi.org/10.1016/j.jsc.2021.10.002.

[6] Jean-Guillaume Dumas and Bruno Grenet. In-place sub-quadratic polynomial modular remainder. Technical report, IMAG-hal-03979016, July 2023. URL: https://hal.science/hal-03979016.

[7] Pascal Giorgi. Efficient algorithms and implementation in exact linear algebra. (algorithmes et implantations efficaces en algèbre linéaire exacte), 2019. Habilitation, University of Montpellier, France. URL: https://tel.archives-ouvertes.fr/tel-02360023.

[8] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. Generic reductions for in-place polynomial multiplication. In James H. Davenport, Dongming Wang, Manuel Kauers, and Russell J. Bradford, editors, *ISSAC'2019, Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation, Beijing, China*, pages 187–194, New York, July 2019. ACM Press. doi:10.1145/3326229.3326249.

[9] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. Fast in-place algorithms for polynomial operations: division, evaluation, interpolation. In Ioannis Z. Emiris, Lihong Zhi, and Anton Leykin, editors, *ISSAC'2020, Proceedings of the 2020 International Symposium on Symbolic and Algebraic Computation, Kalamata, Greece*, pages 210–217, New York, July 2020. ACM Press. doi:10.1145/3373207.3404061.

[10] Bruno Grenet and Ilya Volkovich. One (more) line on the most ancient algorithm in history. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 15–17, 2020. doi:10.1137/1.9781611976014.3.

[11] David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In Wolfram Koepf, editor, *ISSAC'2010, Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation, Munich, Germany*, page 325–329, New York, July 2010. ACM Press. doi:10.1145/1837934.1837996.

[12] Hsiang-Tsung Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22(5):341–348, October 1974. doi:10.1007/BF01436917.

[13] John P. May, editor. *ISSAC'2009, Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, Seoul, Korea*, New York, July 2009. ACM Press.

[14] Robert Moenck and Allan Borodin. Fast modular transforms via division. In *13th Annual Symposium on Switching and Automata Theory (Swat 1972)*, pages 90–96, October 1972. doi:10.1109/SWAT.1972.5.

[15] Daniel S. Roche. Space-and time-efficient polynomial multiplication. In May [13], pages 295–302. doi:10.1145/1576702.1576743.

[16] Daniel S. Roche. *Efficient Computation with Sparse and Dense Polynomials*. PhD thesis, University of Waterloo, Ontario, Canada, 2011. URL: http://hdl.handle.net/10012/5869.

[17] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. doi:10.1007/BF02165411.

[18] Emmanuel Thomé. Karatsuba multiplication with temporary space, September 2002. URL: https://hal.inria.fr/hal-02396734.

[19] Joris van der Hoeven. The Truncated Fourier Transform and Applications. In Jaime Gutierrez, editor, *ISSAC'2004, Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*, pages 290–296, New York, July 2004. ACM Press. doi:10.1145/1005285.1005327.

[20] S. Winograd. La complexité des calculs numériques. *La Recherche*, 8:956–963, 1977.

A In-place accumulated matrix-multiplication with 7 recursive calls and 22 additions

```
C11 := C11 - C21;
C12 := C12 - C21;
C22 := C22 - C21;
C21 := C21 + A11 * B11;
C12 := C12 + C21;
C11 := C11 + C21;
A21 := A21 - A11;
B12 := B12 - B22;
C21 := C21 + A21 * B12;
A21 := A21 + A22;
B12 := B12 - B11;
C12 := C12 - C21;
C21 := C21 - A21 * B12;
C12 := C12 + C21;
C22 := C22 + C21;
B12 := B12 + B21;
C21 := C21 + A22 * B12;
B12 := B12 + B22;
B12 := B12 - B21;
A21 := A21 - A12;
C12 := C12 - A21 * B22;
A21 := A21 + A12;
A21 := A21 + A11;
C22 := C22 - C12;
C12 := C12 + A21 * B12;
C22 := C22 + C12;
A21 := A21 - A22;
B12 := B12 + B11;
C11 := C11 + A12 * B21;
```