# In-place accumulation of fast multiplication formulae

Jean-Guillaume Dumas, Bruno Grenet

# In-place accumulation of fast multiplication formulae

Jean-Guillaume Dumas* Bruno Grenet*

June 27, 2024

**Abstract**

This paper deals with simultaneously fast and in-place algorithms for formulae where the result has to be linearly accumulated: some output variables are also input variables, linked by a linear dependency. Fundamental examples include the in-place accumulated multiplication of polynomials or matrices, $C \mathrel{+}= AB$. The difficulty is to combine in-place computations with fast algorithms: those usually come at the expense of (potentially large) extra temporary space, but with accumulation the output variables are not even available to store intermediate values. We first propose a novel automatic design of fast and in-place accumulating algorithms for any bilinear formulae (and thus for polynomial and matrix multiplication) and then extend it to any linear accumulation of a collection of functions. For this, we relax the in-place model to any algorithm allowed to modify its inputs, provided that those are restored to their initial state afterwards. This allows us, in fine, to derive unprecedented in-place accumulating algorithms for fast polynomial multiplications and for Strassen-like matrix multiplications.

## 1 Introduction

Multiplication is one of the most fundamental arithmetic operations in computer science and in particular in computer algebra and symbolic computation. In terms of arithmetic operations, for instance, from the work of [22, 26, 27], many sub-quadratic (resp. sub-cubic) algorithms were developed for polynomial (resp. matrix) multiplication. But these fast algorithms usually come at the expense of (potentially large) extra temporary space to perform the computation that could hinder their practical efficiency, due for instance to cache misses. 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 matrix or polynomial operations [4, 24, 16, 12, 13].

We here extend the latter line of work for *accumulating* algorithms. Actually, one of the main ingredients on non-accumulating algorithms is to use the (free) space of the output as intermediate storage. But 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 accumulating 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, *provided that they are restored to their initial state after completion of the procedure*. This is in fact a natural possibility in 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 thus propose a generic technique transforming any bilinear algorithm into an in-place algorithm under this model. This directly applies to accumulating polynomial and matrix multiplication algorithms, including fast ones. Further, the technique actually generalizes to any linear accumulation, *i.e.*, not only bilinear formulae, provided that the input of the accumulation can be itself reversibly computed in-place (therefore also potentially in-place of some of its own input if needed).

Next, we give our model for in-place computations and recall classical in-place algorithms in Section 2. We then detail in Section 3 our novel technique for in-place accumulation. Finally, we apply this technique and

*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

further optimizations in order to derive new fast and in-place algorithms for the accumulating multiplication of matrices, Section 4, and of polynomials, Section 5.

## 2 Computational model

Our computational model is an *algebraic RAM*. Inputs and outputs are arrays of ring elements. (For simplicity, our algorithms are described over a finite field $\mathbb{F}$, unless otherwise stated.) The machine is made of *algebraic registers* that each contain one ring element, and *pointer registers* that each contain one pointer, that is one integer. Atomic operations are ring operations on the algebraic registers and basic pointer arithmetic. We assume that the pointer registers are large enough to store the length of the input/output arrays.

Both inputs and outputs have read/write permissions. But algorithms are only allowed to modify their inputs **if their inputs are restored to their initial state** afterwards. In this model, we call *in-place* an algorithm using only **the space of its inputs, its outputs, and at most $\mathcal{O}(1)$ extra space**. For recursive algorithms, some space may be required to store the recursive call stack. (This stack is only made of pointers and its size is bounded by the recursion depth of the algorithms. In practice, it is managed by the compiler.) Nonetheless, we call *in-place* a recursive algorithm whose only extra space is the call stack. In our complexity summaries (Tables 2 and 3), we include the size of the stack.

The main limitations of this model are for black-box inputs, or for inputs whose representations share some data. A model with read-only inputs would be more powerful, but mutable inputs turn out to be necessary in our case. In particular, 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 $a + bc$. As a variant, we describe *over-place* algorithms, that replace (parts of) the input by the output (e.g., $\vec{a} \leftarrow b \cdot \vec{a}$). Similarly, all of the input can 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. Our model is somewhat similar to catalytic machines and transparent space [6], but using only the input and output as catalytic space. Also, we do preserve the (not in-place) time complexity, up to a (quasi)-linear overhead. We refer to [6, 24, 12] for more details.

Classical algorithms for matrix or polynomial operations can be performed in-place, without any call stack, as recalled in Algorithm 1.

**Algorithm 1** Quadratic/cubic in-place accumulating multiplications.

| | |
|---|---|
| **Input:** $A$, $B$, $C$, deg. $m$, $n$, $m+n$. | **Input:** $A$, $B$, $C$, $m \times \ell$, $\ell \times n$, $m \times n$. |
| **Read-only:** Polynomials $A$, $B$. | **Read-only:** Matrices $A$, $B$. |
| **Output:** $C(X) \mathrel{+}= A(X)B(X)$ | **Output:** $C \mathrel{+}= AB$ |
| 1: **for** $i$, $j$ **do** | 1: **for** $i$, $j$, $k$ **do** |
| 2: $\quad C[i+j] \mathrel{+}= A[i]B[j]$; | 2: $\quad C_{ij} \mathrel{+}= A_{ik}B_{kj}$; |
| 3: **end for** | 3: **end for** |

## 3 In-place linear accumulation

Karatsuba polynomial multiplication [22] and Strassen matrix multiplication [27] 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, we perform each product one at a time. For each product, both factors are then linearly combined in-place into one of the entry beforehand and restored afterwards. The product of both entries is at that point 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*. Our idea is to pre-subtract one output from the other, then accumulate the product to one output, and finally re-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 kinds of in-place operations, and their combinations. First, as shown in Eq. (1), an in-place accumulation of a quantity multiplied by a (known in advance) invertible constant:

$$\{c \mathrel{/}= \mu;\ c \mathrel{+}= m;\ c \mathrel{*}= \mu;\} \text{ computes in-place } c \leftarrow c + \mu \cdot m. \tag{1}$$

Second, as shown in Eq. (2), an in-place distribution of the same quantity, without recomputation, to several outputs:

$$\{d \mathrel{-}= c;\ c \mathrel{+}= m;\ d \mathrel{+}= c;\} \text{ computes in-place } \begin{cases} c \leftarrow c + m; \\ d \leftarrow d + m. \end{cases} \tag{2}$$

Example 1 shows how to combine several of these operations, while 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 $s \leftarrow s + 2p$. To perform this in-place, first accumulate $a \mathrel{+}= 3b$ and $c \mathrel{+}= d$, then pre-divide $r$ by 5, as in Eq. (1). Now we directly have $p = ac$ and it can be computed once, and then accumulated to $r$, and to $s$, if the latter is prepared: divide it by 2, and pre-subtract $r$ or, equivalently, pre-subtract $2r$. This is $s \mathrel{-}= 2r$ followed by $r \mathrel{+}= ac$. After this, we can reciprocate (or unroll) the precomputations: this distributes the product to the other result and restores the read-only inputs to their initial state. This is summarized as:*

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

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 $\odot$ the point-wise multiplications of left-hand sides by right-hand sides. Then Algorithm 2 computes $\vec{c} \mathrel{+}= \mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \odot (\beta\vec{b})$, with $\alpha$, $\beta$ and $\mu$ matrices of constants.

**Remark 2.** *Lines 2 to 7 and 9 to 14 of Algorithm 2 are acting on independent parts of the input, $\vec{a}$ and $\vec{b}$, and of the output $\vec{c}$. If needed they could therefore be computed in parallel or in different orders, and even potentially grouped or factorized across the main loop (on $\ell$).*

To simplify the counting of operations, we denote by **ADD** both the addition or subtraction of elements, $\mathrel{+}=$ or $\mathrel{-}=$; by **MUL** the (tensor) product of elements, $\odot$; and by **SCA** the scaling by constants, $\mathrel{*}=$ or $\mathrel{/}=$. We also denote by $\#x$ (resp. $\sharp x$) the number of non-zero (resp. $\notin \{0, 1, -1\}$) elements in a matrix $x$.

**Theorem 3.** *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.

*Proof.* First, as the only used operations ($\mathrel{+}=$, $\mathrel{-}=$, $\mathrel{*}=$, $\mathrel{/}=$) 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, Lines 2 and 3 require one multiplication by a constant for each non-zero element $a_\lambda$ 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 is $\#\alpha - t$ additions and $\sharp\alpha$ constant multiplications. Lines 4 and 5

**Algorithm 2** In-place bilinear formula.

**Input:** $\vec{a} \in \mathbb{F}^m$, $\vec{b} \in \mathbb{F}^n$, $\vec{c} \in \mathbb{F}^s$; $\alpha \in \mathbb{F}^{t \times m}$, $\beta \in \mathbb{F}^{t \times n}$, $\mu \in \mathbb{F}^{s \times t}$.
**Read-only:** $\alpha$, $\beta$, $\mu$ (all 3 without zero-rows).
**Output:** $\vec{c}$ += $\mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \odot (\beta\vec{b})$.

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

(resp. Lines 6 and 7) are similar for each non-zero element in $b_\lambda$ (resp. in $\mu$). 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**. □

**Remark 4.** *Similarly, slightly more generic accumulation operations of the form $\vec{c} \leftarrow \vec{\gamma} \odot \vec{c} + \mu\vec{m}$, for a vector $\gamma \in \mathbb{F}^s$, can also be computed in-place: precompute first $\vec{c} \leftarrow \vec{\gamma} \odot \vec{c}$, then call Algorithm 2.*

For instance, to use Algorithm 2 with matrices or polynomials, each product $m_\ell$ is in fact computed recursively. Further, in an actual implementation of a fixed formula, one can combine more efficiently the pre- and post-computations over the main loop on $\ell$, as in Remark 2. See Sections 4 and 5 for examples of recursive calls, together with sequential optimizations and combinations.

In fact the method for accumulation, computing each bilinear multiplication once is generalizable. With the notations of Algorithm 2, any algorithm of the form $\vec{c}$ += $\mu\vec{m}$ can benefit from this technique, provided that each $m_j$ can be obtained from a function that can be computed in-place. Let $F_j : \Omega \to \mathbb{F}$ be such a function on some inputs from a space $\Omega$, for which an in-place algorithm exists. Then we can accumulate it in-place, *if it satisfies the following constraint*, that it is not using its output space as an available intermediary memory location. Further, this function can be in-place in different models: it can follow our model of Section 2, if there is a way to put its input back into their initial states, or some other model, again provided that it follows the above constraint. Then, the idea is just to keep from Algorithm 2 the Lines 6 to 10, replacing Line 8 by the in-place call to $F_j$, potentially surrounding that call by manipulations on the inputs of $F_j$ (just like the one performed on $\vec{a}$ and $\vec{b}$ in Algorithm 2). We give examples of the application of the generalized method of Theorem 5 to non-bilinear formulae in Section 4.2, and we can thus show that:

**Theorem 5.** *Let $\vec{c} \in \mathbb{F}^s$ and $\mu \in \mathbb{F}^{s \times t}$, without zero-rows. Let $\vec{F} = (F_j : \Omega \to \mathbb{F})_{j=1..t}$ be a collection of functions and $\omega \in \Omega$. If all these functions are computable in-place, without using their output space as an intermediary memory location, then there exists an in-place algorithm computing $\vec{c}$ += $\mu\vec{F}(\omega)$ in-place, requiring a single call to each $F_j$, together with $(2\#\mu - t)$* **ADD** *and $2\sharp\mu$* **SCA** *ops.*

# 4 In-place Strassen matrix multiplication with accumulation

## 4.1 7 recursive calls and 18 additions

Considered as $2 \times 2$ matrices, the matrix product with accumulation $C \mathrel{+}= A \cdot B$ could be computed using Strassen-Winograd (S.-W.) algorithm by performing the following computations:

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

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \mathrel{+}= \begin{bmatrix} \rho_1 + \rho_2 & \rho_1 - \rho_3 + \rho_5 - \rho_7 \\ \rho_1 + \rho_4 + \rho_6 - \rho_7 & \rho_1 + \rho_5 + \rho_6 - \rho_7 \end{bmatrix}. \tag{3}$$

This algorithm uses 7 multiplications of half-size matrices and $24 + 4$ additions (that can be factored into only $15 + 4$ [30]: 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. To date, up to our knowledge, the best versions that reduced this extra memory space (also overwriting the input matrices but not putting them back in place) were proposed in [4]: their best sub-cubic accumulating product used 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 an in-place sub-cubic algorithm for accumulating matrix multiplication, without extra temporary field element. From Eq. (3) indeed (see also the representation in [18, 5], denoted HM), we can extract the matrices

$$\mu = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & 1 & 1 & -1 \end{bmatrix}, \quad \alpha = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & -1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 1 & 0 \\ -1 & 0 & 1 & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 1 & 0 & -1 \end{bmatrix}. \tag{4}$$

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 3, at most $2(\#\alpha + \#\beta + \#\mu - 3t) = 2(14 + 14 + 14 - 3 * 7) = 42$ block additions. Just like the 24 additions of Eq. (3) can be factored into 15, one can optimize also the in-place algorithm. For instance, looking at $\alpha$ we see that performing the products in the order $\rho_6$, $\rho_7$, $\rho_3$, $\rho_5$ and accumulating in $a_{21}$ enables to perform all additions/subtractions in $A$ with only 6 operations (this is in fact optimal, see Prop. 8). This is similar for $\beta$ if the order $\rho_6$, $\rho_7$, $\rho_4$, $\rho_5$ is used and accumulation is in $b_{12}$. Thus ordering for instance $\rho_6$, $\rho_7$, $\rho_4$, $\rho_3$, $\rho_5$ will reduce the number of block additions to 26. Now looking at $\mu$ (more precisely at its transpose, see [21]), a similar reduction can be obtained, e.g., if one of the orders ($\rho_6$, $\rho_7$, $\rho_1$, $\rho_5$) or ($\rho_5$, $\rho_7$, $\rho_1$, $\rho_6$) is used and accumulation is in $c_{22}$.

Therefore, using the ordering $\rho_6, \rho_7, \rho_1, \rho_4, \rho_3, \rho_5, \rho_2$ requires only 18 additions (plus 7 accumulations in $C$), as shown with Algorithm 3. Thus, without thresholds and for powers of two, the dominant term of the overall arithmetic cost is $8n^{\log_2(7)}$, for the in-place version, roughly a third more operations than the $6n^{\log_2(7)}$ dominant term of the cost for the version using extra temporaries.

We here give an in-place version of Strassen-Winograd algorithm for matrix multiplication. We first directly apply our Algorithm 2 to the classical, not in-place Strassen-Winograd algorithm, following the specific scheduling strategy of Section 4. This strategy enables to reduce the number of additions obtained when calling Algorithm 2, from $42 + 7$ to $18 + 7$: mostly remove successive additions/subtractions that are reciprocal on either sub-matrices. This optimized version is given in Algorithm 3 and reaches the minimal possible number of extra additions/subtractions, as shown in Theorem 9.

**Algorithm 3** In-place accumulating S.-W. matrix-multiplication.

**Input:** $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$, $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$.
**Output:** $C \mathrel{+}= AB$.

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

The number of temporary blocks of dimensions $\frac{n}{2} \times \frac{n}{2}$ required for the computation in Algorithm 3 is compared to that of previously known algorithms in Table 1.

Table 1: Reduced-memory accumulating S.-W. multiplication.

| Alg. | Temp. blocks | inputs | accumulation |
|---|---|---|---|
| [20] | 3 | read-only | " |
| [4] | 2 | read-only | " |
| Algorithm 3 | 0 | mutable | " |

We now prove that 18 additions is the minimal number of additions required by an in-place algorithm resulting from any bilinear algorithm for matrix multiplication using only 7 multiplications. For this we consider elementary operations on variables (similar to elementary linear algebra operators): *variable-switching* (swapping variable $i$ and variable $j$); *variable-multiplying* (multiplying a variable by a constant); *variable-addition* (adding one variable, potentially multiplied by a constant, to another variable). An *elementary program* is a program using only these three kind of operations. Now, the in-place implementation of a linear function on its input, for $\alpha \in \mathbb{F}^{t \times m}$ and $\vec{a} \in \mathbb{F}^m$, is the computation of each of the $t$ coefficients of $\alpha\vec{a}$, using only elementary operations and only the variables of $\vec{a}$ as temporary variables. We start by proving in Lemma 6 that in any bilinear algorithm for matrix multiplication using only 7 multiplications, the columns of the associated matrices $\alpha, \beta, \mu$ (as in Eq. (4)) cannot contain too many zeroes.

**Lemma 6.** *If $(\alpha, \beta, \mu) \in \mathbb{F}^{7\times4} \times \mathbb{F}^{7\times4} \times \mathbb{F}^{4\times7}$ is the* HM *representation of a bilinear algorithm for matrix multiplication, then none of $\alpha, \beta, \mu^\intercal$ contains a zero column vector, nor a multiple of a standard basis vector.*

*Proof.* The dimensions of the matrices indicate that the multiplicative complexity of the algorithm is 7. From [15] we know that all such bilinear algorithms can be obtained from one another. Following [5, Lemma 6], then any associated $\alpha, \beta, \mu^\intercal$ matrix is some row or column permutation, or the multiplication by some $G \otimes H$ (the Kronecker product of two invertible 2×2 matrices), of the matrices of Eq. (4). By duality [19], see also [5, Eq. (3)], it is also sufficient to consider any one of the 3 matrices. We thus let

$K = G \otimes H$. Then any column of $K$ is of the form $[ux, uy, vx, vy]^\intercal$, where $\begin{bmatrix} u \\ v \end{bmatrix}$ is a column of $G$ and $\begin{bmatrix} x \\ y \end{bmatrix}$ is a column of $H$. Further as $G$ is invertible, $u$ and $v$ cannot be both zero simultaneously and, similarly, $x$ and $y$ cannot be both zero simultaneously. Now consider for instance $\alpha \cdot K$, with $\alpha$ of Eq. (4). Then any column $\vec{\theta}$ of $\alpha \cdot K$ is of the form:
$[ux, uy, -ux - uy + vx + vy, vy, vx + vy, -ux + vx, -ux + vx + vy]^\intercal$.
For such a column to be a multiple of a standard basis vector or the zero vector, at least 6 of its 7 coefficients must be zero. For instance, this means that at least two out of rows 1, 2 and 4 must be zero: or that at least two of $ux$, $uy$ or $vy$ must be zero. This limits us to three cases: (1) $u = 0$, (2) $y = 0$ or (3) $x = v = 0$. If $u = 0$, then $\vec{\theta} = v[0, 0, x + y, y, x + y, x, x + y]^\intercal$; at least one of rows 4 or 6 has to be zero, thus, w.l.o.g. suppose $x = 0$, we obtain that $\vec{\theta} = vy[0, 0, 1, 1, 1, 0, 1]^\intercal$ with none of $v$ nor $y$ being zero (otherwise $G$ or $H$ is not invertible); such a column cannot be a multiple of a standard basis vector nor the zero vector. Similarly, if $y = 0$, then $\vec{\theta} = x[u, 0, -u + v, 0, v, -u + v, -u + v]^\intercal$; at least one of rows 1 or 5 has to be zero, thus, w.l.o.g. suppose $v = 0$, we obtain that $\vec{\theta} = ux[1, 0, -1, 0, 0, -1, -1]^\intercal$; such a column cannot be a multiple of a standard basis vector nor the zero vector. Finally, if $x = v = 0$, then $\vec{\theta} = uy[0, 1, -1, 0, 0, 0, 0]^\intercal$; again that column cannot be a multiple of a standard basis vector nor the zero vector. □

Now we show that any in-place elementary algorithm requires at least 1 extra operation to put back the input in its initial state.

**Lemma 7.** *Let $\vec{a} \in \mathbb{F}^m$ and $\alpha \in \mathbb{F}^{t \times m}$ with at least one row which is neither the zero row, nor a standard basis vector. Now suppose that, without any constraints in terms of temporary registers, $k$ is the minimal number of elementary operations required to compute $\alpha\vec{a}$. Then any algorithm computing all the $t$ values of $\alpha\vec{a}$, in-place of $\vec{a}$, requires at least $k + 1$ elementary operations.*

*Proof.* Consider an in-place algorithm realizing $\alpha\vec{a}$ in $f$ operations. Any zero or standard basis vector row can be realized without any operations on $\vec{a}$. Now take this algorithm at the moment where the last of the other rows of $\alpha$ are realized (at that point all the $t$ values are realized). Then this last realization (a non-trivial linear combination of the initial values of $\vec{a}$) has to have been stored in one variable of $\vec{a}$, say $a_i$. Therefore, at this point, the in-place algorithm has to perform at least one more operation to put back $a_i$ to its initial state. Therefore, by replacing all the in-place computations by operations on extra registers and omitting the operation(s) that restore this $a_i$, we obtain an algorithm with less than $f - 1$ elementary operations that realizes $\alpha\vec{a}$ and thus: $(f - 1) \geq k$. □

**Proposition 8.** *For the in-place realization of each of the two linear operators $\alpha$ and $\beta$, of any bilinear matrix multiplication algorithm using only 7 multiplications, and the restoration of the initial states of their input, at least 6 operations are needed.*

*Proof.* A bilinear matrix multiplication algorithm has to compute $\alpha\vec{a}$, with $\vec{a}$ the entries of the left input of the matrix multiplication, while $\beta$ deals with the right input. These $\alpha$ and $\beta$ matrices cannot contain a (4-dimensional) zero row: otherwise there would exist an algorithm using less than 6 multiplications, but 7 is minimal [29]. If $\alpha$ or $\beta$ contain at least 5 rows that are not standard basis vectors, then they require at least 5 non-trivial operations to be computed, and therefore at least 6 elementary operations with an in-place algorithm, by Lemma 7. The matrices also cannot contain more than 3 multiples of standard basis vectors, by [5, Lemma 8]. There thus remains now only to consider matrices with exactly 3 rows that are multiple of standard basis vectors. Let $M$ be the 4×4 sub-matrix obtained from $\alpha$ (or $\beta$) by removing those 3 standard basis vectors. By Lemma 6, no column of $M$ can be the zero column: otherwise a 7-dimensional column of $\alpha$ (or $\beta$) would be either a multiple of a standard basis vector, or the zero vector. This means that every variable of $\vec{a}$ has to be used at least once to realize the 4 operations of $M\vec{a}$. Now suppose that there exists an in-place algorithm realizing $M\vec{a}$ in 5 elementary operations. Any operations among these 5 that, as its results, puts back a variable into its initial state, does not realize any row of $M\vec{a}$ (because putting back a variable to its initial state is the trivial identity on this initial variable, and this would be represented by a 4-dimensional standard basis vector, which $M$ do not contain, by construction). Therefore, at most one

among these 5 operations puts back a variable of $\vec{a}$ into its initial state (otherwise $M\vec{a}$, and therefore $\alpha\vec{a}$ or $\beta\vec{a}$, would be realizable in strictly less than 4 operations). Thus, at most one variable of $\vec{a}$ can be modified during the algorithm (otherwise the algorithm would not be able to put back all its input variables into their initial state).

W.l.o.g suppose this only modified variable is $a_1$. Finally, as all the other 3 variables must be used in at least one of the 5 elementary operations, at least 3 operations are of the form $a_1 \mathrel{+}= \lambda_i a_i$ for $i = 2, 3, 4$ and some constants $\lambda_i$. After those, to put back $a_1$ into its initial state, each one of these 3 independent variables, $a_2$, $a_3$ and $a_4$, must be "removed" from $a_1$ at some point of the elementary program. But, with a total of 5 operations, there remains only 2 other possible elementary operations, each one of those modifying only $a_1$. Therefore not all 3 variables can be removed and thus no in-place algorithm can use only 5 operations. □

Finally, there remains to consider the linear combinations of the 7 multiplications to conclude that Algorithm 3 realizes the minimal number of operations for any in-place algorithm with 7 multiplications.

**Theorem 9.** *At least* 25 *additions are required to compute in-place any bilinear matrix multiplication algorithm using only* 7 *multiplications and to restore its input matrices to their initial states afterwards.*

*Proof.* Proposition 8 shows that at least 6 operations are required to realize $\alpha$ (or $\beta$). For $\mu$, we in fact compute $\vec{c} \mathrel{+}= \mu\vec{\rho}$, so we need to consider the matrix $P = \begin{bmatrix} I_4 & \mu \end{bmatrix} \in \mathbb{F}^{4\times 11}$ and the vector $\vec{\xi} = \begin{bmatrix} \vec{c} \\ \vec{\rho} \end{bmatrix}$. Consider now an elementary program that realizes $P\vec{\xi}$, in-place of $\vec{c}$ only. This implies for instance that if $\vec{\rho}$ is zero, $\vec{c}$ should ultimately be put back to its initial state. Finally, consider the transposed program $P^\intercal \underline{\vec{c}}$: it must be in-place of $\underline{\vec{c}}$, while putting back $\underline{\vec{c}}$ to its initial state afterwards. By Prop. 8, $\mu^\intercal$, thus $P^\intercal \in \mathbb{F}^{11\times 4}$, requires at least 6 elementary operations to be performed. By Tellegen's transposition principle, see also [21, Theorem 7], computing the transposed program requires at least $6 + (11 - 4) = 13$ operations. This gives a total of at least $6 + 6 + 13 = 25$ additions. □

Theorem 9 thus shows that our Algorithm 3 with 18 elementary additions and 7 recursive calls (thus 7 more, and a total of $18 + 7 = 25$ additions) is an optimal in-place bilinear matrix multiplication algorithm using only 7 multiplications.

To go beyond our minimality result for operations, one could try an alternate basis of [23]. But an argument similar to that of Prop. 8 shows that alternate basis does not help for the in-place case.

Any bilinear algorithm for matrix multiplication (see, e.g., `https://fmm.univ-lille.fr/`) can be dealt with similarly. Further, even the accumulating version of the non-bilinear algorithm of [10] can benefit from our techniques of in-place accumulation.

## 4.2 In-place Square & Rank-k Update

Thanks to Algorithm 3 and with some care on transposes, the same technique can be adapted to, e.g., [10, Alg. 12], which performs the multiplication of a matrix by its transpose. With an accumulation, this is a classical *Symmetric Rank-k Update* (or SYRK): $C \leftarrow \alpha AA^\intercal + \beta C$.

Following the notations of the latter algorithm, which is not a bilinear algorithm on its single input matrix, the in-place accumulating version is shown in Algorithm 4, for $\alpha = \beta = 1$, using any (fast to apply) skew-unitary $Y \in \mathbb{F}^{n\times n}$. It has been obtained automatically by the method of Theorem 5, and it thus preserves the need of only 5 multiplications $P_1$ to $P_5$. It has then been scheduled to reduce the number of extra operations.

Algorithm 4 requires 3 recursive calls, 2 multiplications of two independent half matrices, 4 multiplications by a skew-unitary half matrix, 8 additions (of half inputs), 12 semi-additions (of half triangular outputs). Provided that the multiplication by the skew-unitary matrix can be performed in-place in negligible time, this gives a dominant term of the complexity bound for Algorithm 4 of a fraction $\frac{2}{2^\omega - 3}$ of the cost of the full in-place algorithm. This is a factor $\frac{1}{2}$, when Algorithm 3 is used for the two block multiplications of independent matrices ($P4$ and $P5$).

**Algorithm 4** In-place accumulating multiplication by its transpose.

**Input:** $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in \mathbb{F}^{m \times 2n}$; symmetric $C = \begin{bmatrix} c_{11} & c_{21}{}^{\intercal} \\ c_{21} & c_{22} \end{bmatrix} \in \mathbb{F}^{m \times m}$.
**Output:** $\mathrm{Low}(C)$ += $\mathrm{Low}(A \cdot A^{\intercal})$. {update bottom left triangle}

```
Low(C22):=Low(C22)−Low(C11);   Low(C21):=Low(C21)−Low(C11);
 Up(C21):= Up(C21)−Low(C11)ᵀ;
Low(C11):=Low(C11)+Low(A11*A11ᵀ);          # P1 Rec.
 Up(C21):= Up(C21)+Low(C11)ᵀ;
Low(C21):=Low(C21)+Low(C11);   Low(C22):=Low(C22)+Low(C11);
Low(C11):=Low(C11)+Low(A12*A12ᵀ);          # P2 Rec.
A11 := A11*Y;   A21 := A21*Y; A11 := A11−A21; A21 := A21−A22;
Low(C22):=Low(C22)−Low(C21);   Low(C22):=Low(C22)−Low(C21ᵀ);
C21 := C21+A11*A21ᵀ;                       # P4 (e.g., Algorithm 3)
Low(C22):=Low(C22)+Low(C21ᵀ);
A21 := A21−A11;
 Up(C21):= Up(C21)−Low(C21)ᵀ;
Low(C21):=Low(C21)+Low(A21*A21ᵀ);          # P5 Rec.
 Up(C21):= Up(C21)+Low(C21)ᵀ;   Low(C22):=Low(C22)+Low(C21);
A21 := A21+A12;
C21 := C21+A22*A21ᵀ;                       # P3 (e.g., Algorithm 3)
A21 := A21−A12; A21 := A21+A11; A21 := A21+A22; A11 := A11+A21;
A21 := A21*Y⁻¹; A11 := A11*Y⁻¹;
```

Now, the skew-unitary matrices used in [10], are either a multiple of the identify matrix, or the Kronecker product of $\begin{bmatrix} a & b \\ -b & a \end{bmatrix}$ by the identity matrix, for $a^2 + b^2 = -1$ and $a \neq 0$. The former is easily performed in-place in time $\mathcal{O}(n^2)$. For the latter, it is sufficient to use Eq. (10): the multiplication $\begin{bmatrix} a & b \\ -b & a \end{bmatrix} \vec{u}$ can be realized in place by the algorithm: $u_1 *= a$; $u_1$ += $b \cdot u_2$; $u_2 *= (a + b^2 a^{-1})$; $u_2$ += $\left(-ba^{-1}\right) \cdot u_1$. The same technique can be used on the symmetric algorithm for the square of matrices given in [3]. The resulting in-place algorithm is given in Algorithm 5.

# 5 In-place polynomial multiplication with accumulation

Algorithm 2 can also be used for polynomial multiplication. An additional difficulty is that this does not completely fit 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.

**Algorithm 5** In-place accumulating S.-W. matrix-square.

**Input:** $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$.
**Output:** $C \mathrel{+}= A^2$.

```
A22 := A22 - A21; C12 := C12 + C22;
C22 := C22 + A22 * A22;
A22 := A22 + A12; A22 := A22 - A11;
C12 := C12 - A22 * A12;
C21 := C21 - A21 * A22;
C21 := C21 - C22; A22 := A22 + A11;
C22 := C22 - A22 * A22;
C11 := C11 + C22;
C22 := C22 - A12 * A21;
A22 := A22 + A21; C12 := C12 - C22; C11 := C11 - C22;
C22 := C22 + A22 * A22;
A22 := A22 - A12; C21 := C21 + C22;
C11 := C11 + A11 * A11;
```

## 5.1 In-place accumulating Karatsuba

For instance, we immediately obtain an in-place Karatsuba polynomial multiplication since it writes as in Eq. (5), from which we can extract the associated $\mu$, $\alpha$, $\beta$ matrices shown in Eq. (6).

$$\begin{aligned}(Ya_1 + a_0)(Yb_1 + b_0) &= a_0b_0 + Y^2(a_1b_1) \\ &\quad + Y(a_0b_0 + a_1b_1 - (a_0 - a_1)(b_0 - b_1))\end{aligned} \tag{5}$$

$$\mu = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & -1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \alpha = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \qquad \beta = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \tag{6}$$

Then, with $Y = X^\delta$ and $a_i$, $b_i$, $c_i$ polynomials in $X$ (and $a_0$, $b_0$, $c_0$ of degree less than $t$), this is detailed, with accumulation, in Eq. (7):

$$\begin{aligned}&A(Y) = Ya_1 + a_0; \quad B(Y) = Yb_1 + b_0; \\ &C(Y) = Y^3c_{11} + Y^2c_{10} + Yc_{01} + c_{00}; \\ &\quad m_0 = a_0 \cdot b_0 = m_{01}Y + m_{00}; \quad m_1 = a_1 \cdot b_1 = m_{11}Y + m_{10}; \\ &\quad m_2 = (a_0 - a_1) \cdot (b_0 - b_1) = m_{21}Y + m_{20}; \\ &\quad t_{00} = c_{00} + m_{00}; \quad t_{01} = c_{01} + m_{01} + m_{00} + m_{10} - m_{20}; \\ &\quad t_{10} = c_{10} + m_{10} + m_{01} + m_{11} - m_{21}; \quad t_{11} = c_{11} + m_{11}; \\ &\textbf{then} \quad C + AB = Y^3t_{11} + Y^2t_{10} + Yt_{01} + t_{00}\end{aligned} \tag{7}$$

To deal with the distributions of each half of the products of Eq. (7), each coefficient in $\mu$ in Eq. (6) 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. (8):

$$\mu^{(2)} = \begin{bmatrix} I_2 & 0_2 & 0_2 \\ 0_2 & 0_2 & 0_2 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ I_2 & I_2 & -I_2 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0_2 & 0_2 & 0_2 \\ 0_2 & I_2 & 0_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}. \tag{8}$$

Finally, Eq. (7) then translates into an in-place algorithm thanks to Algorithm 2 and Eqs. (6) and (8). 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 matrix $\mu^{(2)}$ 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 $2 \times 2$ invertible sub-matrix $M = \begin{bmatrix} v & w \\ x & y \end{bmatrix}$ of these two columns. Then computing $\begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{+}= M \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}$ is equivalent to computing a $2 \times 2$ version of Eq. (1):

$$\left\{ \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= M^{-1}; \quad \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{+}= \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}; \quad \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= M \right\}. \tag{9}$$

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 $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to a vector of results $\begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix}$, it is sufficient that one of its coefficients is invertible. W.l.o.g suppose that its upper left element, $a$, is invertible. Thus, $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ ca^{-1} & 1 \end{bmatrix} \begin{bmatrix} a & b \\ 0 & d - ca^{-1} \end{bmatrix}$. Then the in-place evaluation of Eq. (10) performs this application, using the two (known in advance) constants $x = ca^{-1}$ and $y = d - ca^{-1}b$:

$$\left.\begin{array}{l} \vec{u} \mathrel{*}= a \\ \vec{u} \mathrel{+}= b \cdot \vec{v} \\ \vec{v} \mathrel{*}= y \\ \vec{v} \mathrel{+}= x \cdot \vec{u} \end{array}\right\} \quad \begin{array}{l} \text{computes in-place:} \\ \begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix} \leftarrow \begin{bmatrix} a & b \\ c & d \end{bmatrix} \odot \begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix} = \begin{bmatrix} a\vec{u} + b\vec{v} \\ c\vec{u} + d\vec{v} \end{bmatrix} \\ \text{for } x = ca^{-1} \text{ and } y = d - xb \end{array} \tag{10}$$

**Remark 10.** *In practice for $2 \times 2$ blocks, if $a$ is not invertible, permuting the rows is sufficient since $c$ has to be invertible for the matrix to be invertible: for $J = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, if $\tilde{M} = \begin{bmatrix} c & d \\ 0 & b \end{bmatrix} = J \cdot M$, then $M = J \cdot \tilde{M}$ and $M^{-1} = \tilde{M}^{-1} \cdot J$ so that Eq. (9) just becomes:*
$\begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= J; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= \tilde{M}^{-1}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{+}= \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= \tilde{M}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \mathrel{*}= J.$

We now have the tools for in-place polynomial algorithms. We start, in Algorithm 6, with a version of Algorithm 2 for which the multiplications are accumulated into two consecutive blocks (denoted **MUL**-2D). A C++ implementation of Algorithms 2 and 6 (`trilplacer`) is available in the PLinOpt library: `https://github.com/jgdumas/plinopt`.

**Theorem 11.** *Algorithm 6 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. (9) and (10) and Remark 10, correctness is similar to that of Algorithm 2 in Theorem 3. Then, Eq. (10) 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 to use a double expansion of the output $\mu \in \mathbb{F}^{s \times t}$ to simulate the double size of the intermediate products (**MUL**-2D), producing $\mu^{(2)} \in \mathbb{F}^{s \times (2t)}$, as in Eq. (8), that is used as an input in Algorithm 6. This double expansion matrix is obtained by the following duplication: $\mu^{(2)}(i, 2j) = \mu(i, j)$ and $\mu^{(2)}(i+1, 2j+1) = \mu(i, j)$ for $i = 1..s$ and $j = 1..t$. We prove, in Lemma 12, that in fact any such double expansion of a representative matrix is suitable for the in-place computation of Algorithm 6.

**Lemma 12.** *If $\mu$ does not contain any zero column, then each pair of columns of $\mu^{(2)}$, resulting from the expansion of a single column in $\mu$, contains an invertible lower triangular $2 \times 2$ sub-matrix.*

**Algorithm 6** In-place bilinear 2 by 2 formula.

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

1: **for** $\ell = 1$ **to** $t$ **do**
2: Let $i$ s.t. $\alpha_{\ell,i} \neq 0$; $a_i$ *= $\alpha_{\ell,i}$;
3: **for** $\lambda = 1$ **to** $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** $a_i$ += $\alpha_{\ell,\lambda} a_\lambda$ **end for**
4: Let $j$ s.t. $\beta_{\ell,j} \neq 0$; $b_j$ *= $\beta_{\ell,j}$;
5: **for** $\lambda = 1$ **to** $n$, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ **do** $b_j$ += $\beta_{\ell,\lambda} b_\lambda$ **end for**
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. (10) and Remark 10}
8: **for** $\lambda = 1$ **to** $s$, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell} \neq 0$ **do** $c_\lambda$ −= $\mu_{\lambda,2\ell} c_k$ **end for**
9: **for** $\lambda = 1$ **to** $s$, $\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell+1} \neq 0$ **do** $c_\lambda$ −= $\mu_{\lambda,2\ell+1} c_f$ **end for**
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** $c_\lambda$ += $\mu_{\lambda,2\ell+1} c_f$ **end for**
12: **for** $\lambda = 1$ **to** $s$,$\lambda \notin \{f, k\}$, $\mu_{\lambda,2\ell} \neq 0$ **do** $c_\lambda$ += $\mu_{\lambda,2\ell} c_k$ **end for**
13: $\begin{bmatrix} c_k \\ c_f \end{bmatrix} \leftarrow M \begin{bmatrix} c_k \\ c_f \end{bmatrix}$ {via Eq. (10) and Remark 10, 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 for** ; $b_j$ /= $\beta_{\ell,j}$;
15: **for** $\lambda = 1$ **to** $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ **do** $a_i$ −= $\alpha_{\ell,\lambda} a_\lambda$ **end for** ; $a_i$ /= $\alpha_{\ell,i}$;
16: **end for**
17: **return** $\vec{c}$.

*Proof.* The top most non-zero element of a column is expanded as a 2×2 identity matrix whose second row is merged with the first row of the next identity matrix: $\begin{bmatrix} a \\ b \end{bmatrix}$ is expanded to $\begin{bmatrix} a & 0 \\ b & a \\ * & b \end{bmatrix}$. □

For instance with $m_{00} + Ym_{01} = a_0 b_0 = \rho_0 + Y\rho_1$, consider the upper left $2 \times 2$ block of $\mu^{(2)}$ in Eq. (8), that is $M = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, whose inverse is $M^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$. One has first to precompute $M^{-1} \begin{bmatrix} c_{00} \\ c_{01} \end{bmatrix}$, that is nothing to $c_{00}$ 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-subtract $c_{10}$ −= $c_{01}$, and post-add $c_{10}$ += $c_{01}$ after the product actual computation. This example is in lines 15 and 20 of Algorithm 7 thereafter. To complete Eq. (7), the computation of $m_1$ is dealt with in the same manner, while that of $m_2$ is direct in the results. Note that as both $t_{01}$ and $t_{10}$ receive together $m_{01} + m_{10}$, some pre- and post-additions are simplified out in Algorithm 7. The second point is to deal with unbalanced dimensions and degrees for $Y = X^\delta$ and recursive calls. First split the largest polynomial into two parts, so that two sub-products are performed: a large balanced one, and, recursively, a smaller unbalanced one. Second, 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. Details can be found in Algorithm 7.

**Proposition 13.** *Algorithm 7 is correct and requires $\mathcal{O}(mn^{\log_2(3)-1})$ operations.*

*Proof.* With the above analysis, correctness comes from that of Algorithm 6 applied to Eq. (6). 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 $(m \bmod n)$. Let $u_1 = m$, $u_2 = n$, $u_3$, …, $u_k$ denote the successive residues in the Euclidean algorithm on inputs $m$ and $n$ (where $u_k$ is the last nonzero residue). Then, Algorithm 7 requires less than $\mathcal{O}(\sum_{i=1}^{k-1} \lfloor \frac{u_i}{u_{i+1}} \rfloor u_{i+1}^{\log_2(3)}) \leq \mathcal{O}(\sum_{i=1}^{k-1} u_i u_{i+1}^{\log_2(3)-1})$

**Algorithm 7** In-place Karatsuba polynomial multiplication with accumulation

**Input:** $A$, $B$, $C$ polynomials of degrees $m$, $n$, $m+n$ with $m \geq n$.
**Output:** $C \mathrel{+}= AB$

**if** $n \leq$ Threshold **then** {Constant-time if Threshold $\in \mathcal{O}(1)$}
  **return** the quadratic in-place multiplication. {Algorithm 1}
**else if** $m > n$ **then**
  Let $A(X) = A_0(X) + X^{n+1}A_1(X)$
  $C_{0..2n} \mathrel{+}= A_0B$ {Recursive call}
  **if** $m \geq 2n$ **then**
    $C_{(n+1)..(n+m)} \mathrel{+}= A_1B$ {Recursive call}
  **else**
    $C_{(n+1)..(n+m)} \mathrel{+}= BA_1$ {Recursive call}
  **end if**
**else** {Now $m = n$}
  Let $\delta = \lceil (2n+1)/4 \rceil$; {$\delta - 1 \geq 2n - 3\delta$ and thus $\delta > n - \delta$}
  Let $A = a_0 + X^\delta a_1$; $B = b_0 + X^\delta b_1$;
  Let $C = c_{00} + c_{01}X^\delta + c_{10}X^{2\delta} + c_{11}X^{3\delta}$; {$d^\circ c_{11} = 2n - 3\delta$}
  $c_{01} \mathrel{-}= c_{00}$; $c_{10} \mathrel{-}= c_{01}$;
  $\begin{bmatrix} c_{00} \\ c_{01} \end{bmatrix} \mathrel{+}= a_0 \cdot b_0$ {Recursive call for $m_0$}
  $c_{11} \mathrel{-}= c_{10[0..2n-3\delta]}$; {first $2n - 3\delta + 1$ coefficients of $c_{10}$}
  $\begin{bmatrix} c_{01} \\ c_{10} \end{bmatrix} \mathrel{+}= a_1 \cdot b_1$ {Recursive call for $m_1$}
  $c_{11} \mathrel{+}= c_{10[0..2n-3\delta]}$; {as $d^\circ m_{11} \leq 2n - 3\delta$}
  $c_{10} \mathrel{+}= c_{01}$; $c_{01} \mathrel{+}= c_{00}$;
  $a_0 \mathrel{-}= a_1$; $b_0 \mathrel{-}= b_1$; {$d^\circ a_0 = \delta - 1 \geq n - \delta = d^\circ a_1$}
  $\begin{bmatrix} c_{01} \\ c_{10} \end{bmatrix} \mathrel{-}= a_0 \cdot b_0$ {Recursive call[a] for $m_2$}
  $b_0 \mathrel{+}= b_1$; $a_0 \mathrel{+}= a_1$;
**end if**
**return** $C$.

[a] Variant that computes $C \mathrel{-}= AB$, obtained by changing signs at each recursive call.

operations. But, $u_{i+1} \leq u_2 = n$ and if we let $s_i = u_i + u_{i+1}$, $u_i \leq s_i$. Thus, the number of operations is bounded by $\mathcal{O}(\sum_{i=1}^{k-1} s_i n^{\log_2(3)-1})$. From [14, Corollary 2.6], we have that $s_i \leq s_1(2/3)^{i-1}$. Therefore, $\sum_{i=1}^{k-1} s_i \leq s_1 \sum_{i \geq 0}(2/3)^i = \mathcal{O}(m+n)$, and the number of operations is $\mathcal{O}(mn^{\log_2(3)-1})$. □

Note that all coefficients of $\alpha$, $\beta$ and $\mu^{(2)}$ being 1 or $-1$, Algorithm 7 does compute the accumulation $C \mathrel{+}= AB$ without constant multiplications. Also, the de-duplication enables some natural reuse. There is thus a cost of $2(\#\alpha - t + \#\beta - t) = 2(4-3+4-3) = 4$ additions with $a_0, a_1, b_0, b_1$. Then $2(2(\#\mu - t) - 1) = 2(\#\mu^{(2)} - 2t - 1) = 2(10-6-1)$ additions with $c_{ij}$ (*i.e.*, $10-6$ minus the one saved by factoring $m_{01} + m_{10}$). This is a total of 3 recursive accumulating calls and at most 10 half-block additions. For degree $n-1$ (size $n$) polynomials, this is between $5n - \frac{1}{2}$ and $5n - 5$ additional additions, and with 2 operations for an accumulating base case, this gives a dominant term between $11.75n^{\log_2(3)}$ and $9.5n^{\log_2(3)}$. A careful Karatsuba implementation for both polynomials of *size* $n$ a power of two requires instead $6.5n^{\log_2(3)}$ operations [24]. Now in practice, the supplementary operations are in fact, at least mostly, compensated by the gain in memory allocations or movements: depending on the compilator and flags we obtain some slight speed-up or slow-down, about $\pm 10\%$, when compared to the state of the art Karatsuba modular (60 bits prime) polynomial multiplication of the NTL (`https://libntl.org`) library.

We compare in Table 2 the procedure given in Algorithm 7 (obtained via the automatic application

of Algorithm 6) with previous Karatsuba-like algorithms for polynomial multiplications, designed to reduce their memory footprint (see also [11, Table 2.2]).

Table 2: Reduced-memory algorithms for Karatsuba polynomial multiplication

| Algorithm | Memory Algebraic | Reg. Pointer | Inputs | Accumulation |
|---|---|---|---|---|
| [28] | $n$ | $5\log n$ | read-only | % |
| [24, 25] | 0 | $5\log n$ | read-only | % |
| [12] | $\mathcal{O}(1)$ | | read-only | % |
| Algorithm 7 | 0 | $5\log n$ | mutable | " |

## 5.2 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, $\alpha$ and $\beta$ are the Vandermonde matrices of these points for the 3 parts of the input polynomials and $\mu$ is the inverse of the Vandermonde matrix of these points for the 5 parts of the result, as shown in Eq. (11) thereafter.

$$\mu = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{3} & -\frac{1}{6} & 2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} & \frac{1}{6} & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \ \alpha = \beta = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

With the same kind of duplication as in Eq. (8), 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 $\alpha$ and $\beta$ 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 two fewer operations. Together with shared coefficients in the rows of $\mu$, some further optimizations of [2] can probably also be applied, where the same multiplicative constants appear at successive places. Currently, for instance, PLinOpt produces a program with only 56 additions and 51 scalar multiplications.

## 5.3 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. For simplicity, we consider a ring $\mathbb{D}$ that contains a principal $N$-th root of unity $\omega \in \mathbb{D}$ for some $N = 2^p$. (In particular, 2 is a unit in $\mathbb{D}$.)

Let $F \in \mathbb{D}[X]$ of degree $< N$. The discrete Fourier transform of $F$ at $\omega$ is defined as $\mathsf{DFT}_N(F, \omega) = (F(\omega^0), F(\omega^1), \ldots, F(\omega^{N-1}))$. The map is invertible, of inverse $\mathsf{DFT}_N^{-1}(\cdot, \omega) = \frac{1}{N}\mathsf{DFT}_N(\cdot, \omega^{-1})$. Further, the DFT can be computed over-place, replacing the input by the output [7]. Actually, for over-place algorithms and their extensions to the *truncated Fourier transform*, it is more natural to work with the *bit-reversed DFT* defined by $\mathsf{brDFT}_N(F, \omega) = (F(\omega^{[0]_p}), F(\omega^{[1]_p}), \ldots, F(\omega^{[N-1]_p}))$ where $[i]_p = \sum_{j=0}^{p-1} d_j 2^{p-j}$ is the length-$p$ bit reversal of $i = \sum_{j=0}^{p-1} d_j 2^j$, $d_j \in \{0, 1\}$. Its inverse is $\mathsf{brDFT}_N^{-1}(\Lambda, \omega) = \frac{1}{N}\mathsf{DFT}_N((\Lambda_{[0]_p}, \ldots, \Lambda_{[N-1]_p}), \omega^{-1})$.

**Remark 14.** *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* $\mathsf{brDFT}_N(\cdot,\omega)$*, while the over-place DIT-FFT algorithm on* $\omega^{-1}$ *computes* $N \cdot \mathsf{brDFT}_N^{-1}(\cdot,\omega)$*.*

**Algorithm 8** In-place power of two accumulating multiplication.

**Input:** $\vec{a}$, $\vec{b}$ and $\vec{c}$ of respective lengths $n$, $n$ and $N = 2n$, containing the coefficients of $A$, $B$, $C \in \mathbb{D}[X]$ respectively; $\omega \in \mathbb{D}$ principal $N$-th root of unity, with $N = 2^p$.
**Output:** $\vec{c}$ contains the coefficients of $C + A \cdot B$.

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

**Theorem 15.** *Using an over-place* brDFT *algorithm with complexity bounded by* $\mathcal{O}(n \log n)$*, Algorithm 8 is correct, in-place and has complexity bounded by* $\mathcal{O}(n \log n)$*.*

*Proof.* Algorithm 8 follows the pattern of the standard FFT-based multiplication algorithm. Our goal is to compute $\mathsf{brDFT}_{2n}(A,\omega)$, $\mathsf{brDFT}_{2n}(B,\omega)$ and $\mathsf{brDFT}_{2n}(C,\omega)$, then obtain $\mathsf{brDFT}_{2n}(C+AB,\omega)$ 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 $k < n$, $[k]_p = 2[k]_{p-1}$, and for $k \geq n$, $[k]_p = 2[k-n]_{p-1}+1$. Therefore, the first $n$ coefficients of $\mathsf{brDFT}_{2n}(A,\omega)$ are $(A(\omega^{2[0]_{p-1}}), \dots A(\omega^{2[n-1]_{p-1}})) = \mathsf{brDFT}_n(A,\omega^2)$. Similarly, the next $n$ coefficients are $\mathsf{brDFT}_n(A(\omega X),\omega^2)$. Therefore, one can compute $\mathsf{brDFT}_n(A,\omega^2)$ and $\mathsf{brDFT}_n(B,\omega^2)$ 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 $\mathsf{brDFT}_n^{-1}(\cdot,\omega^2)$. We compute $A(\omega X)$ and $B(\omega X)$ and again $\mathsf{brDFT}_n(A(\omega X),\omega^2)$ and $\mathsf{brDFT}_n(B(\omega X),\omega^2)$ to update the last $n$ entries of $\vec{c}$. Finally, we restore $\vec{a}$ and $\vec{b}$ and perform $\mathsf{brDFT}^{-1}$ on $\vec{c}$. The cost is dominated by the ten $\mathsf{brDFT}^{\pm 1}$ computations. □

The standard (not in-place) algorithm uses two brDFT and one $\mathsf{brDFT}^{-1}$ in size $2n$. Since our in-place variant uses 2 size-$2n$ and 8 size-$n$ $\mathsf{brDFT}^{\pm 1}$, the dominant term is twice as large.

The case where the sizes are not powers of two is loosely similar, using as a routine a truncated Fourier transform (TFT) rather than a DFT [17]. Let $\omega$ still be an $N$-th root of unity for some $N = 2^p$, and $n < N$. The length-$n$ (bit-reversed) TFT of a polynomial $F \in \mathbb{D}[X]$ at $\omega$ is $\mathsf{brTFT}_n(F,\omega) = (F(\omega^{[0]_p}), \dots, F(\omega^{[n-1]_p}))$, that is the $n$ first coefficients of $\mathsf{brDFT}_N(F,\omega)$. As for the DFT, the (bit-reversed) TFT and its inverse can be computed over-place [16, 25, 1, 8].

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 $m + n - 1 \leq N$, we aim to replace $C$ by $C + AB$. As in the power-of-two case, we first replace $C$ by $\mathsf{brTFT}_{m+n-1}(C,\omega)$ in $\vec{c}$. Then we progressively update $\vec{c}$ using small brTFT's on the inputs, using the following lemma.

**Lemma 16** ([16, 25])**.** *Let* $F \in \mathbb{D}[X]$*,* $\ell$*,* $s \in \mathbb{Z}_{>0}$ *where* $2^\ell$ *divides* $s$*, and* $\omega$ *be a* $2^p$*-th principal root of unity. If* $F_{s,\ell}(X) = F(\omega^{[s]_p} X) \bmod X^{2^\ell - 1}$*,* $\mathsf{brDFT}_{2^\ell}(F_{s,\ell}, \omega^{2^{p-\ell}}) = (F(\omega^{[s]_p}), \dots, F(\omega^{[s+2^\ell-1]_p}))$*.*

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

**Corollary 17.** *Let $F \in \mathbb{D}[X]$ stored in an array $\vec{f}$ of length $n$, $\ell$, $k \in \mathbb{Z}_{>0}$ and $\omega$ be a $2^p$-th principal root of unity, with $2^\ell \leq n$ and $(k+1)2^\ell \leq 2^p$. There exists an algorithm, $\mathsf{partTFT}_{k,\ell}(\vec{f},\omega)$, that replaces the first $2^\ell$ entries of $\vec{f}$ by $F(\omega^{[k\cdot 2^\ell]_p}), \ldots, F(\omega^{[(k+1)\cdot 2^\ell-1]_p})$, and an inverse algorithm $\mathsf{partTFT}^{-1}_{k,\ell}$ that restores $\vec{f}$ to its initial state. Both algorithms are in-place have complexity bounded by $\mathcal{O}(n+\ell\cdot 2^\ell)$.*

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

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

Its correctness is ensured by Lemma 16. Its inverse algorithm $\mathsf{partTFT}^{-1}_{k,\ell}(\vec{f},\omega)$ does the converse:

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

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

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

$$(B(\omega^{[k\cdot 2^{\ell+t}]_p}), \ldots, B(\omega^{[(k+1)\cdot 2^{\ell+t}-1]_p})).$$

**Algorithm 9** In-place fast accumulating polynomial multiplication.

**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; $\omega \in \mathbb{D}$ principal $2^p$-th root of unity with $2^{p-1} < m+n-1 < 2^p$
**Output:** $\vec{c}$ contains the coefficients of $C + A\cdot B$.

1: $\vec{c} \leftarrow \mathsf{brTFT}_{m+n-1}(\vec{c},\omega)$; {over-place}
2: $r \leftarrow m+n-1$
3: **while** $r \geq 0$ **do**
4: $\quad \ell \leftarrow \lfloor \log_2 \min\{r,m\} \rfloor$; $t \leftarrow \lfloor \log_2 \min\{r,n\} \rfloor - \ell$;
5: $\quad k \leftarrow m+n-1-r$ {over-place: $B(\omega^{[k\cdot 2^{\ell+t}]_p}), \ldots, B(\omega^{[(k+1)\cdot 2^{\ell+t}-1]_p})$}
6: $\quad \vec{b} \leftarrow \mathsf{partTFT}_{k,\ell+t}(\vec{b},\omega)$
7: $\quad$ **for** $s=0$ **to** $2^t-1$ **do** {over-place: $A(\omega^{[(k\cdot 2^t+s)2^\ell]_p}), \ldots, A(\omega^{[(k\cdot 2^t+s+1)2^\ell-1]_p})$}
8: $\quad\quad \vec{a} \leftarrow \mathsf{partTFT}_{s+k\cdot 2^t,\ell}(\vec{a},\omega)$
9: $\quad\quad$ **for** $i=0$ **to** $2^\ell-1$ **do** $c_{i+(k\cdot 2^t+s)2^\ell} \mathrel{+}= a_i b_{i+s\cdot 2^\ell}$ **end for**
10: $\quad\quad \vec{a} \leftarrow \mathsf{partTFT}^{-1}_{s+k\cdot 2^t,\ell}(\vec{a},\omega)$ {undo 8 over-place}
11: $\quad$ **end for**
12: $\quad \vec{b} \leftarrow \mathsf{partTFT}^{-1}_{k,\ell+t}(\vec{b},\omega)$ {undo 6 over-place}
13: $\quad r \mathrel{-}= 2^{\ell+t}$
14: **end while**
15: **return** $\vec{c} \leftarrow \mathsf{brTFT}^{-1}_{m+n-1}(\vec{c},\omega)$

**Theorem 18.** *Algorithm 9 is correct and in-place. If the algorithm* brDFT *used inside* partTFT *has complexity $\mathcal{O}(n\log n)$, then the running time of Algorithm 9 is $\mathcal{O}(n\log n)$.*

*Proof.* The fact that the algorithm is in-place comes from Corollary 17. 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 [16] and can be done the same way here.

To assess its correctness, first note that the values of Lines 4 and 5 are computed so that $2^\ell \leq r, m$ and $2^{\ell+t} \leq r, n$. One iteration of the while loop updates the entries $c_k$ to $c_{k+2^{\ell+t}-1}$ where $k = m+n-1-r$. To this end, we first compute $B(\omega^{[k\cdot 2^{\ell+t}]_p})$ to $B(\omega^{[(k+1)\cdot 2^{\ell+t}-1]_p})$ 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 $2^\ell$, 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.

To bound its complexity, we first bound the number of iterations of the while loop. We identify two phases, first iterations where $r \geq n$ and then iterations with $r < n$. The first phase has at most 3 iterations since $2^{\ell+t} > \frac{n}{2}$ entries of $\vec{c}$ are updated per iteration. The second phase starts with $r < n$ and each iteration updates $2^{\ell+t} > \frac{r}{2}$ entries. That is, $r$ is halved and this second phase has at most $\log_2 n$ iterations. The cost of an iteration is dominated by the calls to $\mathsf{partTFT}^{\pm 1}$. The cost of a call to $\mathsf{partTFT}^{\pm 1}_{k,\ell}$ with a size-$m$ input is the sum of a linear term $\mathcal{O}(m)$ and a non-linear term $\mathcal{O}(\ell \cdot 2^\ell)$. At each 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 linear terms sum to $\mathcal{O}(n + m \cdot 2^t) = \mathcal{O}(n)$ since $m \cdot 2^t < 2^{\ell+1+t} \leq 2n$. Over the $\log_2 n$ iterations, the global cost due to these linear terms is $\mathcal{O}(n \log n)$. The cost due to the non-linear terms in one iteration is $\mathcal{O}((\ell+t)\cdot 2^{\ell+t})$. In the first iterations, $2^{\ell+t} \leq n$ and these costs sum to $\mathcal{O}(n \log n)$. In the next iterations, $2^{\ell+t} \leq r < n$. Since $r$ is halved at each iteration, the non-linear costs in these iterations sum to $\mathcal{O}\left(\sum_i \frac{n}{2^i} \log \frac{n}{2^i}\right) = \mathcal{O}(n \log n)$. □

Then, Algorithm 9 is compared with previous FFT-based algorithms for polynomial multiplications designed to reduce their memory footprint in Table 3 (see also [11, Table 2.2]). Note that no call stack is needed for computing the FFT, therefore these algorithms only require $\mathcal{O}(1)$ pointer registers.

Table 3: Reduced-memory algorithms for FFT polynomial multiplication

| Algorithm | Algebraic Reg. | Inputs | Accumulation |
|---|---|---|---|
| [7] | $2n$ | read-only | % |
| [24] | $\mathcal{O}(2^{\lceil \log_2 n \rceil} - n)$ | read-only | % |
| [16] | $\mathcal{O}(1)$ | read-only | % |
| Algorithm 9 | $\mathcal{O}(1)$ | mutable | " |

# 6 Conclusion

We here provide a generic technique mapping any bilinear formula (and more generally any linear accumulation) into an in-place algorithm. This allows us for instance to provide the first accumulating in-place Strassen-like matrix multiplication algorithm. This also allows use to provide fast in-place accumulating polynomial multiplications algorithms.

Many further accumulating algorithm can then be reduced to these fundamental building blocks, see for instance Toeplitz, circulant, convolutions or remaindering operations in [9].

# References

[1] Andrew Arnold. A new truncated Fourier transform algorithm. In *ISSAC'13, Boston, USA*, pages 15–22, June 2013. doi:10.1145/2465506.2465957.

*This material is based on work supported in part by the Agence Nationale pour la Recherche under Grants ANR-21-CE39-0006, ANR-15-IDEX-0002 and ANR-22-PECY-0010.*

[2] Marco Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In *WAIFI 2007*, volume 4547 of *LNCS*, pages 116–133, 2007. URL: https://doi.org/10.1007/978-3-540-73074-3_10.

[3] Marco Bodrato. A Strassen-like matrix multiplication suited for squaring and higher power computation. In *ISSAC'10, Munich, Germany*, pages 273–280, July 2010. doi:10.1145/1837934.1837987.

[4] Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *ISSAC'09, Seoul, Korea*, pages 135–143, July 2009. doi:10.1145/1576702.1576713.

[5] Nader H. Bshouty. On the additive complexity of 2×2 matrix multiplication. *Information Processing Letters*, 56(6):329–335, December 1995. doi:10.1016/0020-0190(95)00176-X.

[6] Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: catalytic space. In *STOC'14*, pages 857–866. ACM, 2014. doi:10.1145/2591796.2591874.

[7] 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.

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

[9] Jean-Guillaume Dumas and Bruno Grenet. In-place fast polynomial modular remainder. In *ISSAC'24, Raleigh, NC, USA*, July 2024. URL: https://hal.science/hal-03979016.

[10] Jean-Guillaume Dumas, Clément Pernet, and Alexandre Sedoglavic. Some fast algorithms multiplying a matrix by its adjoint. *Journal of Symbolic Computation*, 115:285–315, March 2023. doi:10.1016/j.jsc.2022.08.009.

[11] Pascal Giorgi. Efficient algorithms and implementation in exact linear algebra, 2019. Habil. U. of Montpellier, France. URL: https://theses.hal.science/tel-02360023.

[12] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. Generic reductions for in-place polynomial multiplication. In *ISSAC'19, Beijing, China*, pages 187–194, July 2019. doi:10.1145/3326229.3326249.

[13] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. Fast in-place algorithms for polynomial operations: division, evaluation, interpolation. In *ISSAC'20, Kalamata, Greece*, pages 210–217, July 2020. doi:10.1145/3373207.3404061.

[14] 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.

[15] Hans-Friedich de Groote. On varieties of optimal algorithms for the computation of bilinear mappings II. *Theoretical Computer Science*, 7(2):127–148, 1978. doi:10.1016/0304-3975(78)90045-2.

[16] David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC'10, Munich, Germany*, page 325–329, July 2010. doi:10.1145/1837934.1837996.

[17] Joris van der Hoeven. The Truncated Fourier Transform and Applications. In *ISSAC'04, Santander, Spain*, pages 290–296, July 2004. doi:10.1145/1005285.1005327.

[18] John E. Hopcroft and Jean Musinski. Duality applied to the complexity of matrix multiplications and other bilinear forms. In *STOC*, pages 73–87, April 30 - May 2 1973. doi:10.1137/0202013.

[19] John E. Hopcroft and Jean E. Musinski. Duality applied to the complexity of matrix multiplication and other bilinear forms. *SIAM J. Comput.*, 2(3):159–173, 1973. doi:10.1137/0202013.

[20] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In *Supercomputing '96*, 1996. doi:10.1145/369028.369096.

[21] Michael Kaminski, David G. Kirkpatrick, and Nader H. Bshouty. Addition requirements for matrix and transposed matrix products. *Journal of Algorithms*, 9(3):354–364, 1988. doi:10.1016/0196-6774(88)90026-0.

[22] Anatolii Karatsuba and Yuri P. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[23] Elaye Karstadt and Oded Schwartz. Matrix multiplication, a little faster. *J. ACM*, 67(1):1:1–1:31, 2020. doi:10.1145/3364504.

[24] Daniel S. Roche. Space-and time-efficient polynomial multiplication. In *ISSAC'09, Seoul, Korea*, pages 295–302, July 2009. doi:10.1145/1576702.1576743.

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

[26] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7(3):281–292, 1971. doi:10.1007/BF02242355.

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

[28] Emmanuel Thomé. Karatsuba multiplication with temporary space of size $\leq n$, September 2002. URL: https://hal.inria.fr/hal-02396734.

[29] Shmuel Winograd. On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4(4):381–388, 1971. doi:10.1016/0024-3795(71)90009-7.

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