



HAL
open science

In-place accumulation of fast multiplication formulae

Jean-Guillaume Dumas, Bruno Grenet

► **To cite this version:**

Jean-Guillaume Dumas, Bruno Grenet. In-place accumulation of fast multiplication formulae. Univ. Grenoble Alpes. 2023. hal-04167499v2

HAL Id: hal-04167499

<https://hal.science/hal-04167499v2>

Submitted on 24 Oct 2023 (v2), last revised 28 Jun 2024 (v4)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

In-place accumulation of fast multiplication formulae

Jean-Guillaume Dumas*

Bruno Grenet*

October 24, 2023

Abstract

We here propose simultaneously fast and in-place algorithms for problems where the result of some formula has to be linearly accumulated. In other words, some of the output variables are also input variables, with a linear dependency between these both read and write accesses. Fundamental examples include the accumulated multiplication of polynomials or matrices, $C += AB$, that is where there is a side-effect on the output C which receives in fine $C + AB$. The difficulty is to combine in-place computations, here the output variables are not even directly available to store intermediate values, with fast algorithms, that usually come at the expense of (potentially large) extra temporary space.

We first propose the 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.

This generation requires a relaxation of the in-place model to any algorithm that is allowed to modify its inputs provided that those 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.

From this generic method we are for instance able to derive unprecedented in-place accumulating algorithms for fast polynomial multiplications as well as for Strassen-like matrix multiplications; the latter coming with an associated optimality result.

Contents

1	Introduction	2
1.1	In-place model	2
1.2	In place and over-place classical algorithms	3
2	In-place computation of linear accumulation	3
2.1	In-place computation of accumulating bilinear formulae	3
2.2	Generalization to the in-place accumulation of in-place formulae	5
3	In-place Strassen matrix multiplication with accumulation	5
4	In-place polynomial multiplication with accumulation	6
4.1	In-place Karatsuba polynomial multiplication with accumulation	7
4.2	Further bilinear polynomial multiplications	11
4.3	Fast bilinear polynomial multiplication	11
5	Conclusion	14
A	In-place accumulating matrix-multiplication with 7 recursive calls and 18 additions	17

*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

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 [19, 17, 22], 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. 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, 20, 14, 10, 11].

We here propose algorithms to extend the latter line of work for *accumulating* algorithms. 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 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 first propose here a generic technique transforming any bilinear algorithm into an in-place algorithm under this model. This then 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 first detail our model for in-place computations in Section 1.1 and recall some classical 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 accumulating 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 $a += b \times c$. 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 recursive 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 [20, 10].

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 accumulating polynomial/matrix multiplication

Input: $A(X), B(X), C(X)$ polynomials of degrees $m, n, m + n$.

Read-only: A, B .

Output: $C(X) += A(X)B(X)$

```

1: for  $i = 0$  to  $m$  do
2:   for  $j = 0$  to  $n$  do
3:      $C[i + j] += A[i]B[j]$ ;
4:   end for
5: end for

```

Input: A, B, C matrices of dimensions $m \times \ell, \ell \times n, m \times n$.

Read-only: A, B .

Output: $C += AB$

```

1: for  $i = 0$  to  $m$  do
2:   for  $j = 0$  to  $n$  do
3:     for  $k = 0$  to  $\ell$  do
4:        $C_{ij} += A_{ik}B_{kj}$ ;
5:     end for
6:   end for
7: end for

```

2 In-place computation of linear accumulation

2.1 In-place computation of accumulating 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).

$$\left. \begin{array}{l} c /= \mu; \\ c += m; \\ c \star = \mu; \end{array} \right\} \text{ computes in-place } c \leftarrow c + \mu \cdot m \quad (1)$$

2. In-place distribution of the same quantity, without recomputation, to several outputs; this is shown in Eq. (2).

$$\left. \begin{array}{l} d -= c; \\ c += m; \\ d += c; \end{array} \right\} \text{ computes in-place } \begin{cases} c \leftarrow c + m \\ d \leftarrow d + m \end{cases} \quad (2)$$

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 $a += 3b$ and $c += d$, 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 $s /= 2$, $s -= r$. Now we can compute and accumulate the product $r += ac$. 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).

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

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

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 $\vec{m} = (\alpha \vec{a}) \odot (\beta \vec{b})$.

```

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

Remark 2. Lines 2 to 4, as well as Lines 6 to 8 of Algorithm 2 are in fact 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 ℓ).

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. $\#x$) 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 $2(\#\alpha + \#\beta + \#\mu)$ **SCA** operations.

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 $c_\lambda - \mu_{\lambda,\ell} c_k / \mu_{k,\ell} + \mu_{\lambda,\ell} (c_k / \mu_{k,\ell} + a_i b_j) = c_\lambda + \mu_{\lambda,\ell} a_i b_j$ and $(c_k / \mu_{k,\ell} + a_i b_j) \mu_{k,\ell} = c_k + \mu_{k,\ell} a_i b_j$. Third, for the number of operations, Line 2 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 $\#\alpha$ constant multiplications. Line 3 is similar for each non-zero element b_λ . Line 4 is for each non-zero element in μ . Finally Line 5 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**. \square

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 $\gamma \in \mathbb{D}^s$, can also be computed in-place: precompute in-place 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_ℓ is in fact computed recursively. Further, in an actual implementation of a fixed formula, one could of course combine more efficiently the pre- and post-computations over the main loop on ℓ , as in Remark 2. See next Sections 3 and 4 for examples of recursive calls, together with and sequential optimizations reusing several times the same subparts of the input pre- and post-computations.

2.2 Generalization to the in-place accumulation of in-place formulae

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 $F_j : \Omega \times \mathbb{D}$ be such a function on some inputs from a space Ω , 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 1.1, 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 4 to 6, replacing Line 5 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 have shown the following Theorem 5.

Theorem 5. *Let $\vec{c} \in \mathbb{D}^s$ and $\mu \in \mathbb{D}^{s \times t}$, with no zero-rows. Let $\vec{F} = (F_j : \Omega_j \times \mathbb{D})_{j=1..t}$ be a collection of functions. If these t functions are all computable in-place, without using their output space as an available intermediary memory location, then there exists an in-place algorithm computing $\vec{c} += \mu \vec{F}$ in-place, that requires a single call to each one of the F_j , together with $(2\#\mu - t)$ **ADD** and $2\#\mu$ **SCA** operations.*

We give an example application of the generalized method of Theorem 5 to a non bilinear formula in Appendix B.

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:

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

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

This algorithm uses 7 multiplications of half-size matrices and $24 + 4$ additions (that can be factored into only $15 + 4$, see [25]: 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 accumulating 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 accumulating matrix multiplication with $\mathcal{O}(1)$ extra space requirement.

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

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

All coefficients being 1 or -1 the resulting in-place algorithm can of course compute the accumulation $C += AB$ 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. (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 $\rho_6, \rho_7, \rho_3, \rho_5$ 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 $\rho_6, \rho_7, \rho_4, \rho_5$ 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 μ (more precisely at its transpose, see [16]), 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} .

So using, e.g. the ordering $\rho_6, \rho_7, \rho_1, \rho_4, \rho_3, \rho_5, \rho_2$ requires only 18 additions, as shown in Appendix A.

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.

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 [8] (but developed from a bilinear one) can benefit from our techniques of in-place accumulation. See Appendix B for this example.

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.

*The third row of α , corresponding to ρ_3 , contains 4 elements and thus requires at least 3 additions; adding 3 more additions to put back the input coefficients to their initial state means that 6 operations is minimal, see Proposition 20 in Appendix A.

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:

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

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

$$\mu = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \quad \alpha = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \beta = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \quad (7)$$

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):

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

$$\mu^{(2)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

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 $M = \begin{bmatrix} v & w \\ x & y \end{bmatrix}$ of these two columns. Then computing $\begin{bmatrix} c_i \\ c_j \end{bmatrix} += M \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}$ is equivalent to computing a 2×2 version of Eq. (1):

$$\begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = M^{-1}; \quad \begin{bmatrix} c_i \\ c_j \end{bmatrix} += \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}; \quad \begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = M. \quad (10)$$

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 $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to a vector of results $\begin{bmatrix} \vec{u} \\ \vec{v} \end{bmatrix}$, it is sufficient that one of its coefficient 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. (11) performs this application, using the two (known in advance) constants $x = ca^{-1}$ and $y = d - ca^{-1}b$:

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

Remark 6. 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 $J = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, 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. (10) just becomes $\begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = J; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = \tilde{M}^{-1}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} += \begin{bmatrix} \rho_0 \\ \rho_1 \end{bmatrix}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = \tilde{M}; \begin{bmatrix} c_i \\ c_j \end{bmatrix} \star = J$.

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: $\vec{a} \in \mathbb{D}^m, \vec{b} \in \mathbb{D}^n, \vec{c} \in \mathbb{D}^s; \alpha \in \mathbb{D}^{t \times m}, \beta \in \mathbb{D}^{t \times n}, \mu \in \mathbb{D}^{s \times (2t)} = [M_1 \dots M_t]$, 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 $\vec{m} = (\alpha \vec{a}) \odot (\beta \vec{b})$, 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 6}
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
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
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 {undo 9}
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 {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 6, 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 7. Algorithm 3 is correct, in-place, and requires t MUL-2D, $2(\#\alpha + \#\beta + \#\mu - t)$ ADD and $2(\#\alpha + \#\beta + \#\mu + 2t)$ SCA operations.

Proof. Thanks to Eqs. (10) and (11) and Remark 6, correctness is similar to that of Algorithm 2 in Theorem 3. 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 $2(\#\alpha + \#\beta + \#\mu - 2t) + (2t)4$ SCA operations. \square

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.

Lemma 8. 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 . \square

Now we prove, in Lemma 9, that in fact any double expansion of a representative matrix is suitable for the in-place computation of Algorithm 3.

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 9. *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, $\begin{bmatrix} a \\ b \end{bmatrix}$ is expanded to $\begin{bmatrix} a & 0 \\ b & a \\ * & b \end{bmatrix}$. \square

For instance with $m_{00} + Ym_{01} = a_0b_0 = \rho_0 + Y\rho_1$, consider the upper left 2×2 block of $\mu^{(2)}$ in Eq. (9), 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 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 10. *Algorithm 5 is correct and requires $\mathcal{O}(mn^{\log_2(3)-1})$ 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 [12, Corollary 2.6], we have that $s_i \leq s_1(2/3)^{i-1}$. Thus the number of operations is bounded by $\mathcal{O}(\sum_{i=1}^{\kappa-1} s_i n^{\log_2(3)-1}) \leq \mathcal{O}(n^{\log_2(3)-1} s_1 (\frac{1}{1-(2/3)} - 1)) = \mathcal{O}(n^{\log_2(3)-1} (m+n)) = \mathcal{O}(mn^{\log_2(3)-1})$. \square

Algorithm 5 is compared with previous Karatsuba-like algorithms for polynomial multiplications designed to reduce their memory footprint in Table 1 (see also [9, Table 2.2]).

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 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

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  $n \leq \text{Threshold}$  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  $C$ .

```

Table 1: Reduced-memory algorithms for Karatsuba polynomial multiplication

Alg.	Memory	inputs	accumulation
[23]	$n + 5 \log n$	read-only	✗
[20, 21]	$5 \log n$	read-only	✗
[10]	$\mathcal{O}(1)$	read-only	✗
Algorithm 5	$5 \log n$	mutable	✓

recursive accumulating 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.

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

$$\alpha = \beta = \begin{bmatrix} 0^0 & 0^1 & 0^2 \\ 1^0 & 1^1 & 1^2 \\ (-1)^0 & (-1)^1 & (-1)^2 \\ (-2)^0 & (-2)^1 & (-2)^2 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & -2 & 4 \\ 0 & 0 & 1 \end{bmatrix}$$

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 $\text{DFT}_n(F, w) = (F(w^0), F(w^1), \dots, F(w^{n-1}))$. The map is invertible, of inverse $\text{DFT}_n^{-1}(\cdot, w) = \frac{1}{n} \text{DFT}_n(\cdot, w^{-1})$. It is known that the DFT can be computed over-place, replacing the input by the output [5]. 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 $\text{brDFT}_n(F, w) = (F(w^{[0]_p}), F(w^{[1]_p}), \dots, F(w^{[n-1]_p}))$. If $\pi : \{0, \dots, 2^p - 1\} \rightarrow \{0, \dots, 2^p - 1\}$ denotes the bit-reversal permutation (that is $\pi(i) = [i]_p$), we have $\text{brDFT}_n(\cdot, w) = \pi \circ \text{DFT}_n(\cdot, w)$. Its inverse is $\text{brDFT}_n^{-1}(\cdot, w) = \frac{1}{n} \text{DFT}_n(\cdot, w^{-1}) \circ \pi = \frac{1}{n} \text{DFT}_n(\pi(\cdot), w^{-1})$ since π is an involution.

Remark 11. *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 $\text{brDFT}_n(\cdot, w)$, while the over-place DIT FFT algorithm on w^{-1} computes $\text{brDFT}_n^{-1}(\cdot, w)$.*

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:  $\vec{c} \leftarrow \text{brDFT}_{2n}(\vec{c}, w)$ ; {over-place}  
3:  $\vec{a} \leftarrow \text{brDFT}_n(\vec{a}, w^2)$ ;  $\vec{b} \leftarrow \text{brDFT}_n(\vec{b}, w^2)$  {over-place}  
4: for  $i = 0$  to  $n - 1$  do  $c_i += a_i \times b_i$  end  
5:  $\vec{a} \leftarrow \text{brDFT}_n^{-1}(\vec{a}, w^2)$ ;  $\vec{b} \leftarrow \text{brDFT}_n^{-1}(\vec{b}, w^2)$  {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 12}  
7:  $\vec{a} \leftarrow \text{brDFT}_n(\vec{a}, w^2)$ ;  $\vec{b} \leftarrow \text{brDFT}_n(\vec{b}, w^2)$  {over-place}  
8: for  $i = 0$  to  $n - 1$  do  $c_{i+n} += a_i \times b_i$  end  
9:  $\vec{a} \leftarrow \text{brDFT}_n^{-1}(\vec{a}, w^2)$ ;  $\vec{b} \leftarrow \text{brDFT}_n^{-1}(\vec{b}, w^2)$  {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  $\vec{c} \leftarrow \text{brDFT}_{2n}^{-1}(\vec{c}, w)$ 
```

Remark 12. 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 13. 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 $\text{brDFT}_{2n}(A, w)$, $\text{brDFT}_{2n}(B, w)$ and $\text{brDFT}_{2n}(C, w)$, then obtain $\text{brDFT}_{2n}(C + AB, w)$ 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 $\text{brDFT}_{2n}(A, w)$ are $(A(w^{2^{[0]_{p-1}}}), \dots, A(w^{2^{[\frac{n}{2}-1]_{p-1}}})) = \text{brDFT}_n(A, w^2)$. Similarly, the next n coefficients are $\text{brDFT}_n(A(wX), w^2)$. Therefore, one can compute $\text{brDFT}(A, w^2)$ and $\text{brDFT}(B, w^2)$ 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 $\text{brDFT}_n^{-1}(\cdot, w^2)$. We compute $A(wX)$ and $B(wX)$ and again $\text{brDFT}(A(wX), w^2)$ and $\text{brDFT}(B(wX), w^2)$ 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 $\text{brDFT}^{\pm 1}$ computations. \square

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 [24]. 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 $\text{brTFT}_n(F, w) = (F(w^{[0]_p}), \dots, F(w^{[n-1]_p}))$, that is the n first coefficients of $\text{brDFT}_N(F, w)$. As for the (bit-reversed) DFT, the (bit-reversed) TFT and its inverse can be computed over-place [14, 21, 1, 6].

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 $\text{brTFT}_{m+n-1}(C, w)$ 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 14 ([14, 21]). 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

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

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, $F_{s,\ell}(w_\ell^{[i]_\ell}) = F(w^{[s+i]_p})$. \square

Corollary 15. 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 $2^\ell \leq n$ and $(k+1)2^\ell \leq 2^p$. There exist an algorithm $\text{partTFT}_{k,\ell}(\vec{f}, w)$ that replaces the first 2^ℓ entries of \vec{f} by $F(w^{[k \cdot 2^\ell]_p}), \dots, F(w^{[(k+1) \cdot 2^\ell - 1]_p})$, and an inverse algorithm $\text{partTFT}_{k,\ell}^{-1}$ 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 $\text{partTFT}_{k,\ell}(\vec{f}, w)$ 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** $f_{i-2^\ell} += f_i$ **end**
- 3: $\vec{f}_{0..2^\ell-1} \leftarrow \text{brDFT}_{2^\ell}(\vec{f}_{0..2^\ell-1}, w^{2^{p-\ell}})$

Its correctness is ensured by Lemma 14. Its inverse algorithm $\text{partTFT}_{k,\ell}^{-1}(\vec{f}, w)$ does the converse:

- 1: $\vec{f}_{0..2^\ell-1} \leftarrow \text{brDFT}_{2^\ell}^{-1}(\vec{f}_{0..2^\ell-1}, w^{2^{p-\ell}})$
- 2: **for** $i = 2^\ell$ **to** $n - 1$ **do** $f_{i-2^\ell} -= f_i$ **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 $\text{brDFT}^{\pm 1}$ has cost $O(\ell \cdot 2^\ell)$, and the two other steps have cost $O(n)$. \square

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

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 $C + A \cdot B$.

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

Theorem 16. *Algorithm 7 is correct and in-place. If the algorithm brDFT used inside 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 15. 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 [14] and can be done the same way here.

To assess its correctness, first note that the values of line 4 are computed so that $2^\ell \leq r, m$ and $2^{\ell+t} \leq r, n$. 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 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 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 $m \leq n$, 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 $r \geq n$ and then iterations with $r < n$. During the first phase, $2^{\ell+t} > \frac{n}{2}$ entries of \vec{c} are updated at each 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 partTFT $^{\pm 1}$. One call to 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 partTFT $^{\pm 1}$ on \vec{b} and 2^{t+1} calls to 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 DFTs of the second phase is then $O(\sum_i \frac{n}{2^i} \log \frac{n}{2^i}) = O(n \log n)$. \square

Algorithm 7 is compared with previous FFT-based algorithms for polynomial multiplications designed to reduce their memory footprint in Table 2 (see also [9, Table 2.2]).

Table 2: Reduced-memory algorithms for FFT polynomial multiplication

Alg.	Memory	inputs	accumulation
[5]	$2n$	read-only	\times
[20]	$O(2^{\lceil \log_2 n \rceil} - n)$	read-only	\times
[14]	$O(1)$	read-only	\times
Algorithm 7	$O(1)$	mutable	\checkmark

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 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 [7].

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 [18], pages 135–143. doi:10.1145/1576702.1576713.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Jean-Guillaume Dumas and Bruno Grenet. In-place fast polynomial modular remainder. Technical report, IMAG-hal-03979016, September 2023. URL: <https://hal.science/hal-03979016>.
- [8] 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.
- [9] Pascal Giorgi. Efficient algorithms and implementation in exact linear algebra. (algorithmes et implémentations efficaces en algèbre linéaire exacte), 2019. Habilitation, University of Montpellier, France. URL: <https://tel.archives-ouvertes.fr/tel-02360023>.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Hans Friedich Groote, de. On varieties of optimal algorithms for the computation of bilinear mappings II. Optimal algorithms for 2×2 -matrix multiplication. *Theoretical Computer Science*, 7(2):127–148, 1978. doi:10.1016/0304-3975(78)90045-2.
- [14] 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.

- [15] 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.
- [16] 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.
- [17] Hsiang-Tsung Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22(5):341–348, October 1974. doi:10.1007/BF01436917.
- [18] 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.
- [19] 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.
- [20] Daniel S. Roche. Space-and time-efficient polynomial multiplication. In May [18], pages 295–302. doi:10.1145/1576702.1576743.
- [21] 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>.
- [22] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. doi:10.1007/BF02165411.
- [23] Emmanuel Thomé. Karatsuba multiplication with temporary space, September 2002. URL: <https://hal.inria.fr/hal-02396734>.
- [24] 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.
- [25] S. Winograd. La complexité des calculs numériques. *La Recherche*, 8:956–963, 1977.
- [26] 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.

A In-place accumulating matrix-multiplication with 7 recursive calls and 18 additions

We here give an in-place version of Strassen-Winograd algorithm for matrix multiplication. This algorithm results first from the direct application of our Algorithm 2 to the classical, not in-place Strassen-Winograd algorithm, following the specific scheduling strategy of Section 3. This strategy allows 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 8, and we prove afterwards in Theorem 21 that this version actually reaches the minimal possible number of extra additions/subtractions when considering any bilinear matrix multiplication algorithm using only 7 multiplications.

Algorithm 8 In-place accumulating matrix-multiplication with 7 multiplications

Input: $A \in \mathbb{D}^{m \times k} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B \in \mathbb{D}^{k \times n} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$, $C \in \mathbb{D}^{m \times n} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$.

Output: $C+ = AB$.

```

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

```

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 need to consider the following elementary operations on variables (similar to elementary linear algebra operators):

Definitions 17. An elementary operation on variables is one of the following three types of operations:

- 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, in-place: this is for instance $a += \lambda b$ for a constant λ and variables a and b .*

An elementary program is a program making use of only these three kind of operations.

Then we consider the in-place implementation of a linear function on its input: for $\alpha \in \mathbb{D}^{t \times m}$ and $\vec{a} \in \mathbb{D}^m$, we want to compute at least once each of the t coefficients of $\alpha\vec{a}$, but using only elementary operations and using only the variables of \vec{a} as temporary variables.

We start by proving in Lemma 18 that in any bilinear algorithm for matrix multiplication using only 7 multiplications, the associated matrices α, β, μ (as in Eq. (5)) cannot contain too many zeroes.

Lemma 18. *If $(\alpha, \beta, \mu) \in \mathbb{D}^{7 \times 4} \times \mathbb{D}^{7 \times 4} \times \mathbb{D}^{4 \times 7}$ is a bilinear algorithm for matrix multiplication, then none of α, β, μ^\top contain a zero column vector, nor a multiple of a vector of the canonical basis.*

Proof. The dimensions of the matrices indicate that the multiplicative complexity of the algorithm is 7. From [13] we know that all such bilinear algorithms can be obtained from one another. Following [4, Lemma 6], then any associated α, β, μ^\top 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. (5). By duality [15], see also [4, Eq. (3)], it is also sufficient to consider any one of the 3 matrices.

We thus let $G = \begin{bmatrix} g_{1,1} & g_{1,2} \\ g_{2,1} & g_{2,2} \end{bmatrix}$, $H = \begin{bmatrix} h_{1,1} & h_{1,2} \\ h_{2,1} & h_{2,2} \end{bmatrix}$ and $K = G \otimes H$ their Kronecker product.

Then any column of K is of the form $[ux \quad uy \quad vx \quad vy]^\top$, for $u \in \{g_{1,1}, g_{1,2}\}$, $v \in \{g_{2,1}, g_{2,2}\}$, with u and v both in the same column of G and $x \in \{h_{1,1}, h_{1,2}\}$, $y \in \{h_{2,1}, h_{2,2}\}$, with x and y both in the same 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 α of Eq. (5). Then any column $\vec{\theta}$ of $\alpha \cdot K$ is of the form:

$$\vec{\theta} = [ux \quad uy \quad -ux - uy + vx + vy \quad vx \quad vy \quad -ux + vx \quad -ux + vx + vy]^\top \quad (13)$$

For such a column to be a multiple of a vector of the canonical basis or the zero vector, at least 6 of the coefficients of this dimension-7 column 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) $x = v = 0$:

1. If $u = 0$, then $\vec{\theta} = v[0 \quad 0 \quad x+y \quad y \quad x+y \quad x \quad x+y]^\top$; at least one of rows 4 or 6 has to be zero, thus, w.l.o.g. suppose $x = 0$, we obtain that $\vec{\theta} = vy[0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1]^\top$ with none of v nor y being zero (otherwise G or H is not invertible); such a column cannot be a multiple of a vector of the canonical basis nor the zero vector.
2. Similarly, if $y = 0$, then $\vec{\theta} = x[u \quad 0 \quad -u+v \quad 0 \quad v \quad -u+v \quad -u+v]^\top$; at least one of rows 1 or 5 has to be zero, thus, w.l.o.g. suppose $v = 0$, we obtain that $\vec{\theta} = ux[1 \quad 0 \quad -1 \quad 0 \quad 0 \quad -1 \quad -1]^\top$; such a column cannot be a multiple of a vector of the canonical basis nor the zero vector.
3. Finally, if $x = v = 0$, then $\vec{\theta} = uy[0 \quad 1 \quad -1 \quad 0 \quad 0 \quad 0 \quad 0]^\top$; such a column cannot be a multiple of a vector of the canonical basis nor the zero vector.

□

Now we show that any in-place linear algorithm requires at least one more operation than an optimal not in-place equivalent algorithm.

Lemma 19. *Let $\vec{a} \in \mathbb{D}^m$ and $\alpha \in \mathbb{D}^{t \times m}$ with at least one row which is neither the zero row, nor a vector of the canonical basis. 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 row, or any row that is a vector of the canonical basis can be realized without any operations on \vec{a} . Now take the in-place algorithm at the moment where the last of the other rows are realized. At that point all the t values are realized, but this last other row realization has to have been stored in one variable of \vec{a} , say a_i . Also, this last other row realization is a non trivial linear combination of the initial values of \vec{a} . 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. This means that by replacing all the in-place computations by operations on extra registers and omitting the operation(s) that put back a_i into its initial state, we obtain an algorithm with less than $f - 1$ elementary operations that realizes $\alpha\vec{a}$, without any constraints in terms of temporary registers. In other words $(f - 1) \geq k$ or $f \geq k + 1$. \square

Proposition 20. *At least 6 operations are needed to compute in-place of the input, and to restore their initial states, for the realization of each of the two linear operators α and β , of any bilinear matrix multiplication algorithm using only 7 multiplications.*

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 β deals with the right input matrix. These α and β matrices cannot contain a (4-dimensional) zero row: otherwise there would exist an algorithm using less than 6 multiplications, while 7 is minimal [26]. If these matrices contain at least 5 rows that are not (4-dimensional) vectors of the canonical basis, 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 19. The matrices also cannot contain more than 3 multiple of vectors of the canonical basis, by [4, Lemma 8] (and thus require at least 4 operations to be computed). There thus remains now only to consider matrices with exactly 3 rows that are multiple of vectors of the canonical basis.

Let M be the 4×4 sub-matrix obtained from α or β by removing those 3 vectors of the canonical basis. By Lemma 18, no column of M can be the zero column, otherwise a column of α or β would be a dimension 7 multiple of a column vector of the canonical basis, or the 7-dimensional 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 vector of the canonical basis: but, by construction, M does not contain any row that is a 4-dimensional vector of the canonical basis). 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). This means that 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 .

Also, all the other 3 variables must be used in at least one of the 5 elementary operations. This means that at least 3 operations are of the form $a_1 += \lambda_i a_i$ for $i = 2, 3, 4$ and some constants λ_i . Now, generically, to put back a_1 into its initial state, each one of these 3 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 . This shows that no in-place algorithm can use only 5 operations. \square

Finally, there remains to consider the linear combinations of the 7 multiplications to conclude that Algorithm 8 realizes the minimal number of operations for any in-place algorithm with 7 multiplications in our model.

Theorem 21. *At least 25 additions are required to compute in-place any bilinear matrix multiplication algorithm using only 7 multiplications and to restore its two input matrices to their initial states afterwards.*

Proof. We need 6 operations at least to realize α and β , from Proposition 20. For μ , we in fact compute $\vec{c} += \mu\vec{\rho}$, so we need to consider the matrix $P = \begin{bmatrix} I_4 & \mu \end{bmatrix} \in \mathbb{D}^{4 \times 11}$ 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 be put back to its initial state at the end of the program. Finally, thus consider the transposed program $P^\top\vec{c}$: it must be in-place of \vec{c} , while putting back \vec{c} to its initial state afterwards. Now, by Proposition 20, μ^\top , and thus $P^\top \in \mathbb{D}^{11 \times 4}$, requires at least 6 elementary operations to be performed. By Tellegen's principle, see also [16, Theorem 7], computing the transposed program thus requires at least $6 + (11 - 4) = 13$ operations. This gives a total of at least $6 + 6 + 13 = 25$ additions. \square

Theorem 21 thus shows that our Algorithm 8 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.

B In-place accumulating multiplication by its transpose

Thanks to Algorithm 8 and with some care on transposes, the same technique can be adapted to, e.g., [8, Alg. 12], which performs the multiplication of a matrix by its transpose. 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 9. It has been obtained automatically by the method of Section 2.2 and then scheduled so as to reduce the number of extra operations.

Algorithm 9 In-place accumulating multiplication by its transpose

Input: $A \in \mathbb{D}^{m \times 2n}$, symmetric $C = \begin{bmatrix} c_{11} & c_{21}^\top \\ c_{21} & c_{22} \end{bmatrix} \in \mathbb{D}^{m \times m}$, skew-unitary $Y \in \mathbb{D}^{n \times n}$.

Output: $\text{Low}(C) += \text{Low}(A \cdot A^\top)$. {as C remains symmetric, only update its lower triangular part}

```

Low(C22) := Low(C22) - Low(C11);
Low(C21) := Low(C21) - Low(C11);
Up(C21) := Up(C21) - Low(C11)^\top;
Low(C11) := Low(C11) + Low(A11 * A11^\top); # P1 Rec. Low(C11) += Low(P1)
Up(C21) := Up(C21) + Low(C11)^\top; # C21 + Low(P1)^\top
Low(C21) := Low(C21) + Low(C11); # C21 + Low(P1) + Low(P1)^\top
Low(C22) := Low(C22) + Low(C11); # Low(C22) += Low(P1)

Low(C11) := Low(C11) + Low(A12 * A12^\top); # P2 Rec. Low(C11) += Low(P2)

A11 := A11 * Y;
A21 := A21 * Y;
A11 := A11 - A21; # -S1
A21 := A21 - A22; # -S2
Low(C22) := Low(C22) - Low(C21);
Low(C22) := Low(C22) - Low(C21)^\top;
C21 := C21 + A11 * A21^\top; # P4 In-Place Strassen-Winograd
Low(C22) := Low(C22) + Low(C21)^\top; # Low(C22) += Low(P4)^\top

A21 := A21 - A11; # S3
Up(C21) := Up(C21) - Low(C21)^\top;
Low(C21) := Low(C21) + Low(A21 * A21^\top); # P5 Rec. Low(C21) += Low(P5)
Up(C21) := Up(C21) + Low(C21)^\top; # C21 += Low(P5) + Low(P5)^\top
Low(C22) := Low(C22) + Low(C21); # Low(C22) += Low(P4) + Low(P5)

A21 := A21 + A12; # S4
C21 := C21 + A22 * A21^\top; # P3 In-Place Strassen-Winograd
A21 := A21 - A12;

A21 := A21 + A11;
A21 := A21 + A22;
A11 := A11 + A21;
A21 := A21 * Y^{-1}; # initial A21
A11 := A11 * Y^{-1}; # initial A11

```

Algorithm 9 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 9 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 8 is used for the two block multiplications of independent matrices ($P4$ and $P5$).

Now, the skew-unitary matrices used in [8], are either a multiple of the identity 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. (11): the multiplication $\begin{bmatrix} a & b \\ -b & a \end{bmatrix} \vec{u}$ can be realized in place by the algorithm: $u_1 \star= a$; $u_1 += b \cdot u_2$; $u_2 \star= (a + b^2 a^{-1})$; $u_2 += (-b a^{-1}) \cdot u_1$.