Fast in-place accumulation

Jean-Guillaume Dumas[*] Bruno Grenet[*]

October 22, 2025

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, $C \mathrel{+}= AB$ (that is with only $\mathcal{O}(1)$ extra space). 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 to ultimately derive unprecedented in-place accumulating algorithms for fast polynomial multiplications and for Strassen-like matrix multiplications.

We then consider the simultaneously fast and in-place computation of the Euclidean polynomial modular remainder $R(X) \equiv A(X) \mod B(X)$. Fast algorithms for this usually also come at the expense of a linear amount of extra temporary space. In particular, they require one to first compute and store the whole quotient $Q(X)$ such that $A = BQ + R$. We here propose an *in-place* algorithm to compute the remainder only. If A and B have respective degree $m + n$ and n, and $\mathfrak{M}(k)$ denotes the complexity of a (not-in-place) algorithm to multiply two degree-k polynomials, our algorithm uses at most $\mathcal{O}\big(\frac{m}{n}\mathfrak{M}(n)\log(n)\big)$ arithmetic operations. In this particular case this is a factor $\log(n)$ more than the not-in-place algorithm. But if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$, then our algorithms do match the not-in-place complexity bound of $\mathcal{O}\big(\frac{m}{n}\mathfrak{M}(n)\big)$. We also propose variants that compute – still in-place and with the same kind of complexity bounds – the over-place remainder $A(X) \equiv A(X) \mod B(X)$, the accumulated remainder $R(X) \mathrel{+}= A(X) \mod B(X)$ and the accumulated modular multiplication $R(X) \mathrel{+}= A(X)C(X) \mod B(X)$, that is multiplication in a polynomial extension of a finite field.

To achieve this, we develop techniques for Toeplitz matrix operations, for generalized convolutions, short product and power series division and remainder whose output is also part of the input.

[*] 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

Contents

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 early work of [42, 52, 53] to the most recent work of [13, 36, 2], 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. It is an intriguing question whether there is a necessary trade-off between space and time, that is whether fast algorithms always require extra space to perform their computations. The use of extra space could hinder the practical efficiency of the algorithms, due for instance to cache misses. For parallel computations, it is easier to get good parallel speedups if the sub-computations run in place, to avoid memory management. Some initial lower bounds suggested that fast algorithms do indeed require extra space [1]. But further work then proposed simultaneously "fast" and "in-place" algorithms, for matrix or polynomial operations [9, 51, 37, 29, 28].

We here extend the latter line of work to *accumulating* algorithms. Actually, one of the main ingredients on non-accumulating algorithms is to use the (free) space of the output as intermediate storage. This makes it possible to avoid existing lower bounds. But when the result has to be accumulated, *i.e.*, if the output is also part of the input, this free space does not 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 such as Karatsuba's [42] or Strassen's [53]. 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).

Then we use this technique to develop in-place modular methods for dense univariate polynomials over a finite ring. For instance, we compute in-place the remainder only of the Euclidean division. This means that, e.g., with respect to [28, Algorithm 3], we obtain the remainder without needing any space for the quotient.

As polynomials and Toeplitz matrices are indeed different representations of the same objects, see, e.g., [4, 5, 29], we develop as building blocks fast methods for Toeplitz matrix operations, in-place *with accumulation* or *over-place*, as well as for generalized convolutions, short product and power series division and remainder, where the output is also part of the input.

As a direct application of these techniques we finally obtain in-place algorithms for the multiplication in a polynomial extension of a finite field.

This paper extends the results of [20, 21] as follows:

- We describe an implementation of the first accumulating and in-place Strassen-like matrix multiplication algorithm and show that it compares favorably in practice with the standard not-in-place variants.
- We describe an implementation of the first in-place variant of Karatsuba polynomial multiplication and show that it has very close performance to that of the state-of-the-art library NTL.
- We propose fast in-place algorithms for most of the basic linear algebra subroutines (triangular multiplication and solving, Gaussian elimination, inverse, nullspace, rank updates) through reductions to matrix multiplication.
- We propose fast in-place algorithms for more structured matrices, *i.e.*, Toeplitz-like matrix-vector multiplication.
- We give here a full interpretation of Toeplitz matrix-vector operations in terms of polynomial or power series operations, in order to propose polynomial only versions of all our in-place modular polynomial routines.
- We improve on the algorithms for fast in-place polynomial short products. In particular, we improve their complexity bounds and remove the need of a call-stack for this operation.

Our paper is organized as follows. We give our model for in-place computations and recall classical in-place algorithms in Section 1.1. We then introduce our main notations in Section 1.2 and recall the classical (quadratic or cubic) algorithms from the perspective of in-place accumulating computations in Section 1.3. We then detail in Section 2 our novel technique for in-place accumulation. With this technique and further optimizations, we can also derive new fast and in-place algorithms for the accumulating multiplication of matrices, Section 3, and of polynomials, Section 4. Then, in Sections 5 and 6, we derive novel in-place algorithms for circulant and Toeplitz matrices. Finally, in Section 7 we present fast in-place algorithms computing just the polynomial remainder, and for an accumulated modular multiplication.

Acknowledgements

We are grateful to the anonymous reviewers for their comments and suggestions, which have helped improve the quality of our article.

1.1 Computational model

Our computational model is an *algebraic RAM*. Inputs and outputs are arrays of ring elements. Ring elements are assumed to require bounded space. (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 3 and 4), 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. It is also not suitable for rings whose elements have unbounded length such as the ring of integers. 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, the whole 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 [12], 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 [12, 51, 29] for more details. Another closely related model is Goldreich's global storage model [30, 31] that uses one global space for inputs and outputs, and some local space as extra space for the computations.

Remark 1. *The vocabulary for algorithms that use no extra space is diverse and sometimes inconsistent between papers. We summarize our three main notions:*

- *An algorithm is* in-place *if it uses $O(1)$ extra space, in addition to the input and output space. In this paper, we still call in-place recursive algorithms that require a call stack. It will be easily checked that these call stacks are made of $O(\log n)$ pointers only where n is the input size. Another term for in-place is* constant-space.
- *An algorithm is* over-place *if it replaces (part of) its input by the output. This does not assume anything on the extra space required for the algorithm. Some authors use* in-place *for this notion.*
- *An* accumulating *algorithm is a special case of an over-place algorithm, where the result of some computation is accumulated into part of the input. A typical example is a computation $a \mathrel{+}= b \times c$.*

We draw the attention of the reader to the fact that "in-place" can therefore refer to either the first or the second concept in the literature.

1.2 Notations

Polynomials over a finite field $\mathbb{F}$ are denoted by capital letters A, B, ... A degree-n polynomial $A \in \mathbb{F}[X]$ has $(n+1)$ coefficients denoted $a_0, \ldots, a_n$. Given A, $B \in \mathbb{F}[X]$, $A \operatorname{div} B$ and $A \bmod B$ denote the quotient and remainder in the Euclidean division of A by B, respectively. In particular, $A = (A \operatorname{div} B) \times B + (A \bmod B)$, with $\deg(A \bmod B) < \deg(B)$. Matrices over $\mathbb{F}$ are also denoted by capital letters A, B, ... Entries of a matrix $A \in \mathbb{F}^{m\times n}$ are denoted A_{ij} (or $A_{i,j}$ for better readability), $1 \le i \le m$, $1 \le j \le n$.

In algorithms, a degree-n polynomial A is stored in a size-$(n+1)$ array of coefficients. By a slight abuse of notation, a polynomial is identified with its array representation and similarly the notation a_i denotes both the coefficient of degree i of A and the ith cell of its array representation. This is similar for matrices. There is no real distinction between inputs and outputs: Every array manipulated by the algorithm is part of the input. Instead of an output, to each algorithm is associated a "**Result:**" which indicates the array that has been modified. (Other arrays are implicitly assumed to be finally restored.)

We denote by $\mathfrak{M}(n)$ a *multiplication time* [26] for $\mathbb{F}[X]$ such that two degree-n polynomials over $\mathbb{F}$ can be multiplied using at most $\mathfrak{M}(n)$ operations in $\mathbb{F}$ by a *bilinear algorithm*. The restriction to bilinear algorithms is benign since all known *algebraic* algorithms are bilinear. As customary, we assume that $\mathfrak{M}(n)/n$ is non-decreasing and $\mathfrak{M}(mn) \le m^2\mathfrak{M}(n)$. One can take $\mathfrak{M}(n) = O(n \log n \log\log n)$ [13], *cf.* also [36] for more precise bounds.

Similarly, we denote by $\mathfrak{MM}(m;k;n)$ the cost of a bilinear algorithm to multiply an $m\times k$ matrix by a $k\times n$ one. We write $\mathfrak{MM}(m)$ if $m = k = n$. We also denote by $\omega > 2$, the exponent of the dominant term of this cost when the matrices are square.[1] Strassen's algorithm allows taking $\omega = \log(7) \simeq 2.807$ [53] while the most recent results show that one can take $\omega < 2.371339$ [2].

Finally, for a vector $\vec{c} \in \mathbb{F}^n$, we denote by $\overleftarrow{c} \in \mathbb{F}^n$ the *reversed* vector defined as $(\overleftarrow{c})_i = (\vec{c})_{n-i+1}$. In algorithms, instructions such as $\overleftarrow{c} \mathrel{+}= \vec{a}$ means that the vector $\vec{a}$ is accumulated into $\vec{c}$ but in reversed order.

1.3 Classical algorithms

Classical algorithms for matrix and polynomial operations can be performed in-place, without any call stack, as recalled in Algorithms 2 and 3.

Algorithm 2 Quadratic i-p. accumulating polynomial multiplication.

```
Inputs: $A(X)$, $B(X)$, $C(X) \in \mathbb{F}[X]$ of respective degrees $m$, $n$, $m+n$.
Read-only: $A$, $B$.
Result: $C(X) \mathrel{+}= A(X)B(X)$
for $0 \le i \le m$, $0 \le j \le n$ do
    $c_{i+j} \mathrel{+}= a_i b_j$;
end for
```

Also classical, quadratic, algorithms for polynomial remaindering and triangular matrix operations can be performed in-place, as recalled in Algorithms 4 and 5.

[1] If we let $\{n_1, n_2, n_3\} = \{m, k, n\}$ with $n_1 \le n_2 \le n_3$, then by cutting the matrices to the smallest dimension, it is straightforward to see that we therefore always have $\mathfrak{MM}(m;k;n) \le \lceil \frac{m}{n_1} \rceil \lceil \frac{k}{n_1} \rceil \lceil \frac{n}{n_1} \rceil \mathfrak{MM}(n_1) = \mathcal{O}(n_1^{\omega-2} n_2 n_3)$.

Algorithm 3 Cubic i-p. accumulating matrix multiplication.

```
Inputs: $A \in \mathbb{F}^{m\times\ell}$, $B \in \mathbb{F}^{\ell\times n}$, $C \in \mathbb{F}^{m\times n}$.
Read-only: $A$, $B$.
Result: $C \mathrel{+}= AB$
for $1 \le i \le m$, $1 \le j \le n$, $1 \le k \le \ell$ do
    $C_{ij} \mathrel{+}= A_{ik}B_{kj}$;
end for
```

For any field $\mathbb{F}$ we have for instance the following over-place algorithms for triangular matrix operations, given in Algorithm 4.

Algorithm 4 Over-place quadratic triangular matrix operations
(left: matrix-vector multiplication; right: triangular system solve)

```
Inputs: $U \in \mathbb{F}^{m\times m}$ upper triangular and $\vec{v} \in \mathbb{F}^m$
Read-only: $U$
Result: $\vec{v} \leftarrow U \cdot \vec{v}$
for $i = 1$ to $m$ do
    for $j = 1$ to $i - 1$ do
        $v_j \mathrel{+}= U_{ji}v_i$
    end for
    $v_i \leftarrow U_{ii}v_i$;
end for
```

```
Result: $\vec{v} \leftarrow U^{-1} \cdot \vec{v}$
for $i = m$ down-to $1$ do
    for $j = m$ down-to $i + 1$ do
        $v_i \mathrel{-}= U_{ij}v_j$
    end for
    $v_i \leftarrow U_{ii}^{-1}v_i$;                {if $U_{ii} \in \mathbb{F}^*$}
end for
```

The classical long-division algorithm provides a quadratic in-place algorithm for computing the remainder of two polynomials without computing the quotient, see Algorithm 5.

Algorithm 5 In-place quadratic polynomial remainder.

```
Inputs: $A(X)$, $B(X)$, $R(X)$ in $\mathbb{F}[X]$, of respective degrees $N$, $M$ and $M - 1$.
Read-only: $A(X)$, $B(X)$.
Result: $R(X) = A(X) \mod B(X)$.
Let $\bar{B} = B \bmod X^M$ and $n = \max(N - M, -1)$
$R \leftarrow A \operatorname{div} X^{n+1}$;
for $i = n$ down-to $0$ do
    $q \leftarrow r_{M-1} \cdot b_M^{-1}$;                {leading coefficients of $R$ and $B$}
    $R \leftarrow a_i + X \cdot R \bmod X^M$;
    $R \mathrel{-}= q \cdot \bar{B}$;
end for
```

Algorithm 5 can be made over-place of its input A by considering that R is just a “pointer” to some position in A (with q in r_M): This is then close to the in-place quadratic version given in [48].

2 In-place linear accumulation

Karatsuba polynomial multiplication [42] and Strassen matrix multiplication [53] 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 inputs 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 Equation (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 Equation (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 6 shows how to combine several of these operations, while also linearly combining parts of the input.

Example 6. *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 Equation (1). Now we directly have $p = ac$ that 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 follows:*

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

Algorithm 7 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 (their Hadamard product). Then Algorithm 7 computes $\vec{c} \mathrel{+}= \mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \odot (\beta\vec{b})$, with α, β and μ matrices of constants. These matrices define the HM *representation* of a bilinear algorithm, as in [20, 24] and references therein. They can also be denoted by $L = \alpha$, $R = \beta$, $P = \mu$ as they act respectively on the left-hand side, right-hand side and post (or product) side.

$$\vec{c} \mathrel{+}= P \cdot (L \cdot \vec{a}) \odot (R \cdot \vec{b}) \tag{3}$$

Algorithm 7 In-place bilinear formula.

```
Inputs: $\vec{a} \in \mathbb{F}^m$, $\vec{b} \in \mathbb{F}^n$, $\vec{c} \in \mathbb{F}^s$; $\alpha \in \mathbb{F}^{t\times m}$, $\beta \in \mathbb{F}^{t\times n}$, $\mu \in \mathbb{F}^{s\times t}$.
Read-only: $\alpha$, $\beta$, $\mu$ (all 3 without zero-rows).
Result: $\vec{c} \mathrel{+}= \mu\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \odot (\beta\vec{b})$.
 1: for $\ell = 1$ to $t$ do
 2:     Find one $i$ s.t. $\alpha_{\ell,i} \neq 0$; $a_i \mathrel{*}= \alpha_{\ell,i}$;
 3:     for $\lambda = 1$ to $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ do $a_i \mathrel{+}= \alpha_{\ell,\lambda} a_\lambda$ end for
 4:     Find one $j$ s.t. $\beta_{\ell,j} \neq 0$; $b_j \mathrel{*}= \beta_{\ell,j}$;
 5:     for $\lambda = 1$ to $n$, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ do $b_j \mathrel{+}= \beta_{\ell,\lambda} b_\lambda$ end for
 6:     Find one $k$ s.t. $\mu_{k,\ell} \neq 0$; $c_k \mathrel{/}= \mu_{k,\ell}$;
 7:     for $\lambda = 1$ to $s$, $\lambda \neq k$, $\mu_{\lambda,\ell} \neq 0$ do $c_\lambda \mathrel{-}= \mu_{\lambda,\ell} c_k$ end for
 8:     $c_k \mathrel{+}= a_i \cdot b_j$                {this is the product $m_\ell$, computed only once}
 9:     for $\lambda = 1$ to $s$, $\lambda \neq k$, $\mu_{\lambda,\ell} \neq 0$ do $c_\lambda \mathrel{+}= \mu_{\lambda,\ell} c_k$ end for    {undo 7}
10:     $c_k \mathrel{*}= \mu_{k,\ell}$;                                         {undo 6}
11:     for $\lambda = 1$ to $n$, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ do $b_j \mathrel{-}= \beta_{\ell,\lambda} b_\lambda$ end for    {undo 5}
12:     $b_j \mathrel{/}= \beta_{\ell,j}$;                                         {undo 4}
13:     for $\lambda = 1$ to $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ do $a_i \mathrel{-}= \alpha_{\ell,\lambda} a_\lambda$ end for    {undo 3}
14:     $a_i \mathrel{/}= \alpha_{\ell,i}$;                                         {undo 2}
15: end for
```

Remark 8. *Lines 2 to 7 and 9 to 14 of Algorithm 7 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 ℓ). A C++ implementation of Algorithm 7 (`trilplacer`) is available in the* PLinOpt *library [18].*

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 9. *Algorithm 7 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_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 is $\#\alpha - t$ additions and $\sharp\alpha$ constant multiplications. Lines 4 and 5 (resp. Lines 6 and 7) are similar for each non-zero element in b_λ

(resp. in μ). Finally, Line 8 performs t multiplications of elements and t additions. The remaining lines double the number of **ADD** and **SCA**. This is $t+2(\#\alpha+\#\beta+\#\mu-3t) = 2(\#\alpha+\#\beta+\#\mu)-5t$ **ADD**. □

Remark 10. *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 7.*

For instance, to use Algorithm 7 with matrices or polynomials, each product m_ℓ 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 ℓ, as in Remark 8. See Sections 3 and 4 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 7, any algorithm of the form $\vec{c} \mathrel{+}= \mu\vec{m}$ 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 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 7 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 7). We give examples of the application of the generalized method of Theorem 11 to non-bilinear formulae in Section 3.4, and we can thus show that:

Theorem 11. *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 $\vec{c} \mathrel{+}= \mu\vec{F}(\omega)$ in-place, requiring a single call to each F_j, together with $(2\#\mu - t)$* **ADD** and $2\sharp\mu$ **SCA** ops.

3 In-place Strassen matrix multiplication with accumulation

Considered as 2×2 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:

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

This algorithm uses 7 multiplications of half-size matrices and 24+4 additions (that can be factored into only 15 + 4 [55]: 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 [9]: their best sub-cubic accumulating product used 2 temporary blocks per recursive level, thus a total of extra memory required to be $\frac{2}{3}n^2$.

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

With Algorithm 7 we instead obtain an in-place sub-cubic algorithm for accumulating matrix multiplication, with $\mathcal{O}(1)$ extra temporary field element. From Equation (4) indeed (see also the representation in [39, 11]), we can extract the HM matrices

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

All coefficients being 1 or -1 the resulting in-place algorithm can compute the accumulation $C \mathrel{+}= AB$ without constant multiplications. It thus requires 7 recursive calls and, from Theorem 9, at most $2(\#\alpha + \#\beta + \#\mu - 3t) = 2(14 + 14 + 14 - 3 * 7) = 42$ block additions. Just like the 24 additions of Equation (4) can be factored into 15, one can also optimize the in-place algorithm. For instance, looking at α we see that performing the products in the order ρ_6, ρ_7, ρ_3, ρ_5 and accumulating in a_{21} enables to perform all additions/subtractions in A with only 6 operations (this is in fact optimal, see Proposition 15). This is similar for β if the order ρ_6, ρ_7, ρ_4, ρ_5 is used and accumulation is in b_{12}. Thus ordering for instance ρ_6, ρ_7, ρ_4, ρ_3, ρ_5 will reduce the number of block additions to 26. Now looking at μ (more precisely at its transpose, see [41]), a similar reduction can be obtained, e.g., if one of the orders (ρ_6, ρ_7, ρ_1, ρ_5) or (ρ_5, ρ_7, ρ_1, ρ_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). This is shown in Algorithm 12, which can be obtained (after potentially trying several random optimizations to get the optimal cost) via the PLinOpt library with:

```
./bin/trilplacer data/2x2x2_7_Winograd_{L,R,P}.sms
```

This strategy enables us to reduce the number of additions obtained when calling Algorithm 7, from $42 + 7$ to $18 + 7$: Mostly remove successive additions or subtractions that are reciprocal on either sub-matrices. This optimized version is given in Algorithm 12 and reaches the minimal possible number of extra additions/subtractions, as shown in Theorem 16.

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

The obtained number of temporary blocks of dimensions $\frac{n}{2} \times \frac{n}{2}$ required for the computation in Algorithm 12 is compared to that of previously known accumulating algorithms in Table 1.

Now for the number of operations and practical efficiency. Using 18 extra additions, without thresholds and for powers of two, the dominant term of the overall arithmetic cost is $8n^{\log_2(7)}$, for

Algorithm 12 In-place accumulating S.-W. matrix multiplication.

```
Inputs: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$, $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$.
Result: $C \mathrel{+}= AB$.
$A_{21} \mathrel{-}= A_{11}$; $B_{12} \mathrel{-}= B_{22}$; $C_{21} \mathrel{-}= C_{22}$;
$C_{22} \mathrel{+}= A_{21} * B_{12}$;
$A_{21} \mathrel{+}= A_{22}$; $B_{12} \mathrel{-}= B_{11}$; $C_{12} \mathrel{-}= C_{22}$;
$C_{22} \mathrel{-}= A_{21} * B_{12}$;
$C_{11} \mathrel{-}= C_{22}$;
$C_{22} \mathrel{+}= A_{11} * B_{11}$;
$C_{11} \mathrel{+}= C_{22}$; $B_{12} \mathrel{+}= B_{21}$; $C_{21} \mathrel{+}= C_{22}$;
$C_{21} \mathrel{+}= A_{22} * B_{12}$;
$B_{12} \mathrel{+}= B_{22}$; $B_{12} \mathrel{-}= B_{21}$; $A_{21} \mathrel{-}= A_{12}$;
$C_{12} \mathrel{-}= A_{21} * B_{22}$;
$A_{21} \mathrel{+}= A_{12}$; $A_{21} \mathrel{+}= A_{11}$;
$C_{22} \mathrel{+}= A_{21} * B_{12}$;
$C_{12} \mathrel{+}= C_{22}$; $B_{12} \mathrel{+}= B_{11}$; $A_{21} \mathrel{-}= A_{22}$;
$C_{11} \mathrel{+}= A_{12} * B_{21}$;
```

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

Alg.	Temp. blocks	inputs	accumulation
[40]	3	read-only	✓
[9]	2	read-only	✓
Algorithm 12	0	mutable	✓

the in-place version. This is roughly a third more operations than the $6n^{\log_2(7)}$ dominant term of the cost for the version using extra temporaries.

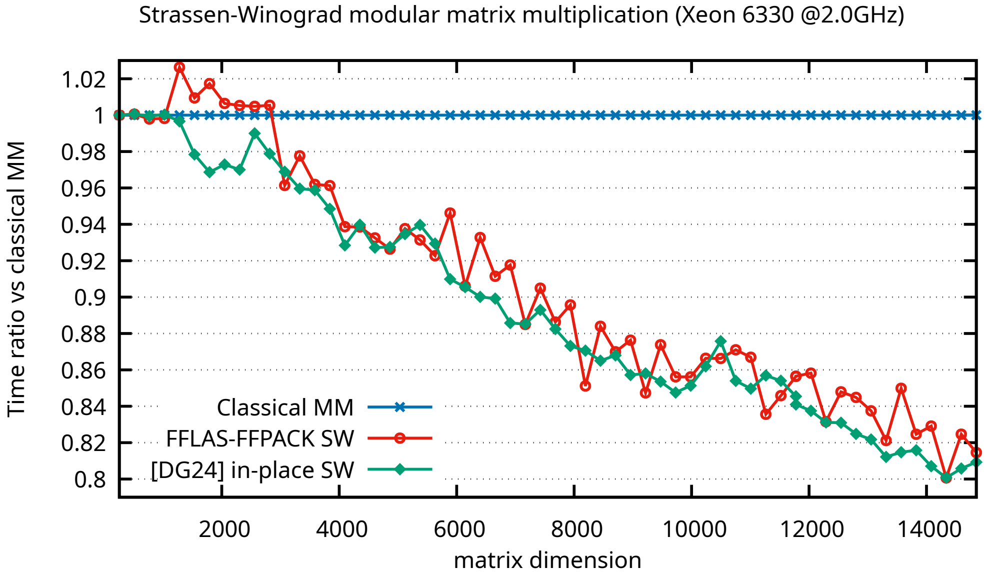
But it turns out that in practice, there is no penalty of using these extra operations to reduce memory, in terms of performance: Figure 1 compares the time of the reference FFLAS[2] Strassen-Winograd matrix multiplication to that of Algorithm 12. The multiplications are performed modulo 131071 on a single core of a Xeon 6330 CPU @2.00GHz. The reference implementation first computes the product $T = AB$ in a temporary block T using 2 extra half-dimension blocks per recursive call via [17], and then accumulates $C \mathrel{+}= T$. This is a total of $n^2 + \sum_{i=1}^{\log_2(n)} 2\left(\frac{n}{2^i}\right)^2 = \frac{5}{3}n^2$ extra memory usage. Figure 1 shows that using extra memory does not provide a faster routine: Algorithm 12 indeed shows the same kind of acceleration with respect to the conventional algorithm and seems even less sensitive to particularities of the dimensions of the matrices.

3.2 In-place additive complexity

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

[2] http://linbox-team.github.io/fflas-ffpack.

Figure 1: Optimal In-place accumulating Strassen-Winograd MM.



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 13 that in any bilinear algorithm for matrix multiplication using only 7 multiplications, the columns of the associated matrices α, β, μ (as in Equation (5)) cannot contain too many zeroes.

Lemma 13. *If $(\alpha, \beta, \mu) \in \mathbb{F}^{7\times 4} \times \mathbb{F}^{7\times 4} \times \mathbb{F}^{4\times 7}$ 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 [34] we know that all such bilinear algorithms can be obtained from one another. Following [11, 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 Equation (5). By duality [39], see also [11, 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 $\begin{bmatrix}ux, uy, vx, vy\end{bmatrix}^\intercal$, where $\left[\begin{smallmatrix}u\\v\end{smallmatrix}\right]$ is a column of G and $\left[\begin{smallmatrix}x\\y\end{smallmatrix}\right]$ 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 α of Equation (5). Then any column $\vec{\theta}$ of $\alpha \cdot K$ is of the form:

$$\begin{bmatrix}ux, uy, -ux - uy + vx + vy, vy, vx + vy, -ux + vx, -ux + vx + vy\end{bmatrix}^\intercal.$$

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 $\vec{\theta} = v\big[0, 0, x + y, y, x + y, x, x + y\big]^\intercal$; at least one of rows 4 or 6 has to be zero, thus, w.l.o.g. suppose $x = 0$, we obtain that $\vec{\theta} = vy\big[0, 0, 1, 1, 1, 0, 1\big]^\intercal$ 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 $\vec{\theta} = x\big[u, 0, -u + v, 0, v, -u + v, -u + v\big]^\intercal$; at least one of rows 1 or 5 has to be zero, thus, w.l.o.g. we can suppose that $v = 0$. We then obtain that $\vec{\theta} = ux\big[1, 0, -1, 0, 0, -1, -1\big]^\intercal$; such a column cannot be a multiple of a standard basis vector nor the zero vector. Finally, if $x = v = 0$, then $\vec{\theta} = uy\big[0, 1, -1, 0, 0, 0, 0\big]^\intercal$; 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 14. *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 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 α 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 entry 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 15. *For the in-place realization of each of the two linear operators α and β, 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 β deals with the right input. These α and β matrices cannot contain a (4-dimensional) zero row: otherwise there would exist an algorithm using less than 6 multiplications, but 7 is minimal [56]. If α or β 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 14. The matrices also cannot contain more than 3 multiples of standard basis vectors, by [11, Lemma 8]. There thus remains now only to consider matrices with exactly 3 rows that are multiples of standard basis vectors. Let M be the 4×4 sub-matrix obtained from α (or β) by removing those 3 standard basis vectors. By Lemma 13, no column of M can be the zero column: otherwise a 7-dimensional column of α (or β) 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 λ_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, it remains to consider the linear combinations of the 7 multiplications to conclude that Algorithm 12 realizes the minimal number of operations for any in-place algorithm with 7 multiplications.

Theorem 16. *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 15 shows that at least 6 operations are required to realize α (or β). For μ, we in fact compute $\vec{c} \mathrel{+}= \mu\vec{\rho}$, so we need to consider the matrix $P = [I_4\ \mu] \in \mathbb{F}^{4\times 11}$ 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 Proposition 15, $\mu^\intercal$, 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 [41, 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 16 thus shows that our Algorithm 12 with 18 elementary additions and 7 from the 7 recursive calls (for 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 [43]. But an argument similar to that of Proposition 15 shows that alternate basis does not help for the in-place case.

Proposition 17. *For the in-place realization of each of the linear operators arising from the sparsification of any bilinear matrix multiplication algorithm using only* 7 *multiplications, and for the restoration of the initial states of their input, at least* 6 *operations are needed.*

Proof. The alternate basis method of [43] consists in sparsifying the matrices of Equation (5), via right multiplication by 4×4 invertible matrices. The sparsest obtained matrices are given in:

$$\begin{bmatrix}1&1&0&0&0&0&0\\0&0&0&-1&1&0&0\\0&0&1&0&0&1&0\\1&0&0&0&1&1&-1\end{bmatrix},\quad \begin{bmatrix}0&0&1&0\\1&0&0&0\\-1&0&0&1\\0&1&0&1\\0&0&1&1\\0&-1&0&0\\0&0&0&1\end{bmatrix},\quad \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&-1\\0&1&0&1\\0&0&1&0\\1&0&0&1\\0&0&0&1\end{bmatrix}. \tag{6}$$

These sparse matrices are obtained from the ones in Equation (5) via the following respective change of bases:

$$\begin{bmatrix}1&0&0&0\\0&0&-1&1\\0&-1&0&1\\0&0&0&1\end{bmatrix},\quad \begin{bmatrix}0&1&0&0\\1&0&-1&0\\1&0&0&0\\-1&0&1&1\end{bmatrix},\quad \begin{bmatrix}1&0&0&0\\0&0&1&0\\-1&1&0&0\\-1&1&0&-1\end{bmatrix}. \tag{7}$$

We then follow the same line of reasoning as in Proposition 15, where we mostly need to adapt Lemma 13. W.l.o.g, we consider a 4×4 transformation M of the middle matrix in Equation (6).

If the resulting product matrix has only 3 rows that are multiple of a standard basis vector, then 6 multiplications are minimal by Proposition 15. The only other possibility is thus, as in Equation (6), that it contains exactly 4 rows that are multiples of a standard basis vector.

Now, each one of the four columns of the resulting product has the form:

$$\begin{bmatrix}c & a & d-a & b+d & c+d & -b & d\end{bmatrix}^\intercal.$$

But such a column can never be a zero-column, nor a multiple of a standard basis vector.

Therefore, as in Proposition 15 all 4 variables must appear in the 3 rows that are not standard basis vectors and at most one variable can be modified. So again at least 3 operations are of the form $a_1 \mathrel{+}= \lambda_i a_i$ for $i = 2, 3, 4$ and some constants λ_i. And then again at least 3 operations are required to put back a_1 in its initial state and no in-place algorithm can use strictly less than 6 operations. □

3.3 Fast over-place linear algebra

In [19, 25], many fast dense linear algebra block recursive routines (system solving, factorizations, etc.) were designed to use as extra memory almost only that of the accumulating matrix multiplication that they used. With our new in-place technique we now have fast in-place accumulating matrix multiplications, from any fast bilinear multiplication. We can therefore now combine these results to get fully in-place fast linear algebra.

In the following, we express the complexities in terms of the (bilinear) matrix multiplication time $\mathfrak{MM}(m; k; n)$ as defined in Section 1.2. Indeed, Theorem 9 shows that any bilinear matrix multiplication algorithm gives rise to an in-place accumulating variant with same asymptotic complexity. And Sections 3.1 and 3.2 provide a finer analysis of the algorithm obtained from the original Strassen-Winograd multiplication algorithm.

3.3.1 Over-place TRMM and TRSM

This is the case for instance for triangular matrix multiplication (TRMM) and triangular system solving (TRSM), as shown in Algorithms 18 and 19, both adapted from [19, § 4].

Algorithm 18 Over-place TRMM.

```
Inputs: $T \in \mathbb{F}^{m \times m}$ upper-triangular, $B \in \mathbb{F}^{m \times n}$;
Result: $B \leftarrow T \cdot B$.
if $m \leq$ Threshold then
    Apply the quadratic in-place TRMM.
else
    {split the matrices as: $T = \begin{bmatrix} T_1 & T_2 \\ & T_3 \end{bmatrix}$, $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$}
    $B_1 \leftarrow TRMM(T_1, B_1)$                {recursive call}
    $B_1 \mathrel{+}= T_2 \cdot B_2$                {Algorithm 12}
    $B_2 \leftarrow TRMM(T_3, B_2)$                {recursive call}
end if
```

Algorithm 19 Over-place TRSM.

```
Inputs: $T \in \mathbb{F}^{m \times m}$ upper-triangular, $B \in \mathbb{F}^{m \times n}$;
Result: $B \leftarrow T^{-1}B$.
if $m \leq$ Threshold then
    Apply the quadratic in-place TRSM.
else
    {split the matrices as: $T = \begin{bmatrix} T_1 & T_2 \\ & T_3 \end{bmatrix}$, $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$}
    $B_2 \leftarrow TRSM(T_3, B_2)$                {recursive call}
    $B_1 \mathrel{-}= T_2 \cdot B_2$                {Algorithm 12}
    $B_1 \leftarrow TRSM(T_1, B_1)$                {recursive call}
end if
```

3.3.2 Over-place PLUQ

Now, combining matrix multiplication with triangular solving, we can extend our method to the PLUQ Gaussian factorization of [25, Algorithm 1]: The latter indeed uses only recursive calls, matrix multiplication, TRSM and permutations. If the matrix has generic rank profile, then $P = Q = I_m$ and by using our fast in-place matrix multiplication and fast over-place TRSM, this gives a fast over place PLUQ factorization. Otherwise, it still does not need any arithmetic registers, but has still to store the permutation matrices, as $O(n)$ extra pointer registers.

Remark 20. *If the field has more than mn elements, then each arithmetic register could in principle store two indices (the two coordinates addressing any matrix entry). In this case, it is possible to store both P and Q permutation matrices within the elements of the initial matrix. Consider for instance a (block recursive) factorization revealing the row rank profile. At any point of the factorization, if some permutation is required, it means that there is a zero at the location of a* *pivot and that a non-zero pivot in the same row has to be chosen instead, if it exists, or that one has to use another row. By the rules of Gaussian elimination, this row with at least one zero, will not be modified anymore during the factorization. In other words the zero at the pivot location will remain a zero coefficient within the final LU factorization. The idea is thus as follows: Use only two extra pointer registers to store the coordinates of the first (row-major) pivot not in the default position. This induces the location of a zero element in the factorization. Then store the coordinates of the second pivot not in the default position instead of this arithmetic zero, and so on. By this technique, only two extra pointer registers are needed to recover all the information of both the P and Q matrices. Note that in an actual implementation of this trick, the coordinates could change along the course of the elimination, at least virtually, following the subsequent permutations. All the coordinates would then have to be updated accordingly, thus potentially all along this path of coordinates.*

3.3.3 Over-place KERN, INVT and INV

Further, we now also consider the null-space (KERN), triangular inverse (INVT) and inverse (INV) algorithms of [19, § 6]. Those make use of only recursive calls, TRSM, PLUQ and accumulating matrix multiplication. Therefore, again combining with our technique, one obtains fast over-place algorithms for these three other routines.

3.4 Fast in-place square and rank-k update

Now, some non-bilinear algorithm can also be transformed into accumulating and in-place algorithms: This is true for instance for both the square of a matrix or the multiplication of a matrix by its transpose.

3.4.1 In-place SQUARE

Both are shown next, and, first on the symmetric algorithm for the square of matrices given in [6]. The resulting in-place algorithm is given in Algorithm 21.

3.4.2 In-place SYRK

Second, thanks to Algorithm 12 and with some care on transposes, the same technique can be adapted to, e.g., [23, 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): $C \leftarrow \alpha A A^{\intercal} + \beta C$.

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 22, using any (fast to apply) skew-unitary $Y \in \mathbb{F}^{n\times n}$. It has been obtained automatically by the method of Theorem 11, 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 22 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 22 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 12 is used for the two block multiplications of independent matrices (P_4 and P_3).

Algorithm 21 In-place accumulating S.-W. matrix-square.

```
Inputs: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$, $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$.
Result: $C \mathrel{+}= A^2$.
$A_{22} \mathrel{-}= A_{21}$; $C_{12} \mathrel{+}= C_{22}$;
$C_{22} \mathrel{+}= A_{22} * A_{22}$;
$A_{22} \mathrel{+}= A_{12}$; $A_{22} \mathrel{-}= A_{11}$;
$C_{12} \mathrel{-}= A_{22} * A_{12}$;
$C_{21} \mathrel{-}= A_{21} * A_{22}$;
$C_{21} \mathrel{-}= C_{22}$; $A_{22} \mathrel{+}= A_{11}$;
$C_{22} \mathrel{-}= A_{22} * A_{22}$;
$C_{11} \mathrel{+}= C_{22}$;
$C_{22} \mathrel{-}= A_{12} * A_{21}$;
$A_{22} \mathrel{+}= A_{21}$; $C_{12} \mathrel{-}= C_{22}$; $C_{11} \mathrel{-}= C_{22}$;
$C_{22} \mathrel{+}= A_{22} * A_{22}$;
$A_{22} \mathrel{-}= A_{12}$; $C_{21} \mathrel{+}= C_{22}$;
$C_{11} \mathrel{+}= A_{11} * A_{11}$;
```

Algorithm 22 In-place accumulating mult. by its transpose (SYRK).

```
Inputs: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbb{F}^{m \times 2n}$; symmetric $C = \begin{bmatrix} C_{11} & {C_{21}}^\intercal \\ C_{21} & C_{22} \end{bmatrix} \in \mathbb{F}^{m \times m}$.
Result: $\mathrm{Low}(C) \mathrel{+}= \mathrm{Low}(A \cdot A^\intercal)$.        {update bottom left triangle}
$\mathrm{Low}(C_{22}) \mathrel{-}= \mathrm{Low}(C_{11})$; $\mathrm{Low}(C_{21}) \mathrel{-}= \mathrm{Low}(C_{11})$;
$\mathrm{Up}(C_{21}) \mathrel{-}= \mathrm{Low}(C_{11})^\intercal$;
$\mathrm{Low}(C_{11}) \mathrel{+}= \mathrm{Low}(A_{11} * A_{11}^\intercal)$;        {$P_1$ Rec.}
$\mathrm{Up}(C_{21}) \mathrel{+}= \mathrm{Low}(C_{11})^\intercal$;
$\mathrm{Low}(C_{21}) \mathrel{+}= \mathrm{Low}(C_{11})$; $\mathrm{Low}(C_{22}) \mathrel{+}= \mathrm{Low}(C_{11})$;
$\mathrm{Low}(C_{11}) \mathrel{+}= \mathrm{Low}(A_{12} * A_{12}^\intercal)$;        {$P_2$ Rec.}
$A_{11} \mathrel{*}= Y$; $A_{21} \mathrel{*}= Y$; $A_{11} \mathrel{-}= A_{21}$; $A_{21} \mathrel{-}= A_{22}$;
$\mathrm{Low}(C_{22}) \mathrel{-}= \mathrm{Low}(C_{21})$; $\mathrm{Low}(C_{22}) \mathrel{-}= \mathrm{Low}(C_{21}^\intercal)$;
$C_{21} \mathrel{+}= A_{11} * A_{21}^\intercal$;        {$P_4$ (e.g., Algorithm 12)}
$\mathrm{Low}(C_{22}) \mathrel{+}= \mathrm{Low}(C_{21}^\intercal)$;
$A_{21} \mathrel{-}= A_{11}$;
$\mathrm{Up}(C_{21}) \mathrel{-}= \mathrm{Low}(C_{21})^\intercal$;
$\mathrm{Low}(C_{21}) \mathrel{+}= \mathrm{Low}(A_{21} * A_{21}^\intercal)$;        {$P_5$ Rec.}
$\mathrm{Up}(C_{21}) \mathrel{+}= \mathrm{Low}(C_{21})^\intercal$; $\mathrm{Low}(C_{22}) \mathrel{+}= \mathrm{Low}(C_{21})$;
$A_{21} \mathrel{+}= A_{12}$;
$C_{21} \mathrel{+}= A_{22} * A_{21}^\intercal$;        {$P_3$ (e.g., Algorithm 12)}
$A_{21} \mathrel{-}= A_{12}$; $A_{21} \mathrel{+}= A_{11}$; $A_{21} \mathrel{+}= A_{22}$; $A_{11} \mathrel{+}= A_{21}$;
$A_{21} \mathrel{*}= Y^{-1}$; $A_{11} \mathrel{*}= Y^{-1}$;
```

Now, the skew-unitary matrices used in [23], are either a multiple of the identify matrix, or the Kronecker product of $\left[\begin{smallmatrix} a & b \\ -b & a \end{smallmatrix}\right]$ 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, the multiplication $\left[\begin{smallmatrix} a & b \\ -b & a \end{smallmatrix}\right] \vec{u}$ can be realized in place by the algorithm: $u_1 \mathrel{*}= a$; $u_1 \mathrel{+}= b{\cdot}u_2$; $u_2 \mathrel{*}= (a+b^2a^{-1})$; $u_2 \mathrel{+}= \left(-ba^{-1}\right) \cdot u_1$. (This corresponds to Equation (13) in Section 4.2.)

3.4.3 In-place SYR2K and symmetric factorization

The next step is to be able to perform an over place symmetric factorization. For this, the algorithm of [22] uses recursive calls, accumulating matrix multiplication, TRMM, TRSM, SYRK, diagonal scaling, permutations and also the symmetric rank 2k update. This latter SYR2K computes the upper or lower triangular part of $\text{Low}\,(C) \mathrel{+}= \text{Low}\,(A \cdot B^{\intercal} + B \cdot A^{\intercal})$ for any $A \in \mathbb{F}^{m \times k}$ and $B \in \mathbb{F}^{m \times k}$. If the symmetric matrix C can actually use a square space, a fast way to compute the latter is to start by zeroing the upper triangular part of C. Then compute one accumulated multiplication and end by accumulating the upper part of the result on the lower part ($\text{Up}\,(C) = 0$; $C \mathrel{+}= A \cdot B^{\intercal}$; $\text{Low}\,(C) \mathrel{+}= \text{Up}\,(C)$). That way the cost of the computation is dominated by exactly one matrix multiplication $\mathfrak{MM}(m;k;m)$. But this requires some space not needed in a symmetric matrix, either its upper or its lower part. Rather, Algorithm 23 is fully in-place and deals with only one of the two triangles of C. The drawback is that the computational cost is modified to be dominated by $\frac{2}{2^{\omega-1}-2}\mathfrak{MM}(m;k;m)$. The latter is just again $\mathfrak{MM}(m;k;m)$ if $\omega = 3$, but the ratio increases to more than 1 for smaller $\omega > 2$.

Algorithm 23 Over-place SYR2K.

```
Inputs: $A \in \mathbb{F}^{m\times k}$; $B \in \mathbb{F}^{m\times k}$; symmetric $C \in \mathbb{F}^{m\times m}$.
Result: $\text{Low}\,(C) \mathrel{+}= \text{Low}\,(A \cdot B^{\intercal} + B \cdot A^{\intercal})$.
if $m \leq$ Threshold then
    Apply the quadratic in-place SYR2K.
else
    {split the matrices as: $A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$, $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$, $C = \begin{bmatrix} C_1 & \\ C_2 & C_3 \end{bmatrix}$}
    $\text{Low}\,(C_1) \mathrel{+}= \text{Low}\,(A_1 \cdot {B_1}^{\intercal} + B_1 \cdot {A_1}^{\intercal})$        {recursive call}
    $\text{Low}\,(C_3) \mathrel{+}= \text{Low}\,(A_2 \cdot {B_2}^{\intercal} + B_2 \cdot {A_2}^{\intercal})$        {recursive call}
    $C_2 \mathrel{+}= A_2 \cdot {B_1}^{\intercal}$        {Algorithm 12}
    $C_2 \mathrel{+}= B_2 \cdot {A_1}^{\intercal}$        {Algorithm 12}
end if
```

Now, with Algorithms 12, 18, 19 and 23 one can directly implement the in-place version of TRSSYR2K from [22, Algorithm 1]. This with the in-place versions of GEMM, TRMM, TRSM, SYRK and SYR2K of Algorithms 12, 18, 19, 22 and 23, now provide all the sub-routines for symmetric indefinite triangular factorization [22, Algorithm 2 and 3].

To conclude this section, we recall in Table 2 the different over-place linear algebra routines obtained, together with the dominant term of their associated complexity bound.

For the first algorithms, the dominant term of the bound is that of [19]. For SYRK, we use the improvement of [23]. Then Algorithm 23 for SYR2K requires two recursive calls and two rectangular multiplications. This a cost bounded as $T_n(m) \leq 2T_n(\frac{m}{2}) + 2\mathfrak{MM}(\frac{m}{2};n;\frac{m}{2})$ and thus dominated by $\dfrac{2}{2^{\omega}-4}\mathfrak{MM}(m;n;m)$.

Finally, for the symmetric factorization, for the sake of simplicity, we consider here only the generic rank profile case. Then the factorization reduces to two recursive calls, one TRSM call and one SYRK call on matrices of half-dimension [19, § 6.4]. This is a cost bounded by $T(m) \leq 2T(\frac{m}{2}) + TRSM(\frac{m}{2}) + SYRK(\frac{m}{2}) = 2T(\frac{m}{2}) + \left(\frac{2}{2^\omega-4} + \frac{2}{2^\omega-3}\right)\mathfrak{MM}(\frac{m}{2}) + o(n^\omega)$. The latter is $T(m) \leq \frac{2(2^{\omega+1}-7)}{(2^w-3)(2^w-4)(2^w-2)} = \frac{2^{\omega+2}-14}{8^\omega-9\cdot4^\omega+26\cdot2^\omega-24}$.

Table 2: Over-place fast linear algebra algorithms.

Algorithm	Memory registers Algebraic	 Pointer	Complexity bound dominant term
TRMM, TRSM	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\frac{2}{2^\omega - 4}\left\lceil\frac{n}{m}\right\rceil\mathfrak{MM}(m)$
INVT	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\frac{4}{(2^\omega-2)(2^\omega-4)}\mathfrak{MM}(n)$
PLUQ, KERN	$\mathcal{O}(1)$	$\mathcal{O}(n)^\star$	$\mathcal{O}(mnr^{\omega-2})$
INV	$\mathcal{O}(1)$	$\mathcal{O}(n)^\star$	$\frac{3\times2^\omega}{(2^\omega-2)(2^\omega-4)}\mathfrak{MM}(n)$
SYRK	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\frac{2}{2^\omega-3}\mathfrak{MM}(m;n;m)$
SYR2K	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\frac{2}{2^\omega-4}\mathfrak{MM}(m;n;m)$
$PLDL^\intercal P^\intercal$	$\mathcal{O}(1)$	$\mathcal{O}(n)^\star$	$\frac{2^{\omega+2}-14}{8^\omega-9\cdot4^\omega+26\cdot2^\omega-24}\mathfrak{MM}(n)$

* If $|\mathbb{F}| \geq mn$, Remark 20 can be used to reduce the number of pointer registers to $\mathcal{O}(\log n)$.

4 Fast in-place polynomial multiplication with accumulation

Algorithm 7 can also be used for polynomial multiplication. An additional difficulty is that this does not completely fit the setting, as a multiplication of two size-n inputs will in general span a (double) size-$2n$ output. This is not an issue until one has to distribute separately the two halves of this $2n$ values (or more generally to different parts of different outputs). In the following we show that this can anyway always be done for polynomial multiplications.

4.1 Double-size output in polynomial multiplication

To illustrate the handling of double-sized output, we consider, for instance, an in-place Karatsuba polynomial multiplication. We start with

$$\begin{aligned}(Ya_1 + a_0)(Yb_1 + b_0)&\\ &= a_0b_0 + Y^2(a_1b_1) + Y(a_0b_0 + a_1b_1 - (a_0 - a_1)(b_0 - b_1)). \quad (8)\end{aligned}$$

From Equation (8), we can express the coefficient vector of the result as $\mu\left(\alpha\left[\begin{smallmatrix}a_0\\a_1\end{smallmatrix}\right]\odot\beta\left[\begin{smallmatrix}b_0\\b_1\end{smallmatrix}\right]\right)$, where

$$\mu=\begin{bmatrix}1&0&0\\1&1&-1\\0&1&0\end{bmatrix},\qquad\alpha=\begin{bmatrix}1&0\\0&1\\1&-1\end{bmatrix},\qquad\beta=\begin{bmatrix}1&0\\0&1\\1&-1\end{bmatrix}.\tag{9}$$

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 Equation (10):

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

To deal with the distributions of each half of the products of Equation (10), each coefficient of μ in Equation (9) 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 Equation (11):

$$\mu^{(2)}=\begin{bmatrix}I_2&0_2&0_2\\0&0&0\\0&0&0\end{bmatrix}+\begin{bmatrix}0&0&0\\I_2&I_2&-I_2\\0&0&0\end{bmatrix}+\begin{bmatrix}0&0&0\\0&0&0\\0_2&I_2&0_2\end{bmatrix}=\begin{bmatrix}1&0&0&0&0&0\\1&1&1&0&-1&0\\0&1&1&1&0&-1\\0&0&0&1&0&0\end{bmatrix}\tag{11}$$

Finally, Equation (10) then translates into an in-place algorithm thanks to Algorithm 7 and Equations (9) and (11). We formalize this process in the following Section 4.2.

4.2 In-place bilinear accumulation in consecutive blocks

The first point of polynomial multiplications is that products double the degree: This corresponds to a constraint that the two blocks obtained after a multiplication 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 2×2 invertible sub-matrix $M=\left[\begin{smallmatrix}v&w\\x&y\end{smallmatrix}\right]$ of these two columns. Then computing $\left[\begin{smallmatrix}c_i\\c_j\end{smallmatrix}\right]$ += $M\left[\begin{smallmatrix}\rho_0\\\rho_1\end{smallmatrix}\right]$ is equivalent to computing a 2×2 version of Equation (1):

$$\left\{\begin{bmatrix}c_i\\c_j\end{bmatrix}\mathrel{*}= M^{-1};\quad\begin{bmatrix}c_i\\c_j\end{bmatrix}\mathrel{+}=\begin{bmatrix}\rho_0\\\rho_1\end{bmatrix};\quad\begin{bmatrix}c_i\\c_j\end{bmatrix}\mathrel{*}= M\right\}.\tag{12}$$

The other rows of these two columns can be dealt with as before by pre- and post-multiplying or dividing by a constant and pre- and post-adding or subtracting the adequate c_i and c_j. Now to apply a matrix $M=\left[\begin{smallmatrix}a&b\\c&d\end{smallmatrix}\right]$ to a vector of results $\left[\begin{smallmatrix}\vec{u}\\\vec{v}\end{smallmatrix}\right]$, it is sufficient that one of its coefficients is invertible. W.l.o.g suppose that its upper left element, a, is invertible. Thus, $\left[\begin{smallmatrix}a&b\\c&d\end{smallmatrix}\right]=\left[\begin{smallmatrix}1&0\\ca^{-1}&1\end{smallmatrix}\right]\left[\begin{smallmatrix}a&b\\0&d-ca^{-1}b\end{smallmatrix}\right]$.

Then the in-place evaluation of Equation (13) performs this application, using the two (known in advance) constants $\beta = ca^{-1}$ and $\alpha = d - ca^{-1}b$:

$$\boxed{\left.\begin{array}{l}\vec{u} \mathrel{*}= a\\ \vec{u} \mathrel{+}= b\cdot\vec{v}\\ \vec{v} \mathrel{*}= \alpha\\ \vec{v} \mathrel{+}= \beta\cdot\vec{u}\end{array}\right\} \quad \begin{array}{l}\text{computes in-place:}\\ \begin{bmatrix}\vec{u}\\ \vec{v}\end{bmatrix} \leftarrow \begin{bmatrix}a & b\\ c & d\end{bmatrix} \odot \begin{bmatrix}\vec{u}\\ \vec{v}\end{bmatrix} = \begin{bmatrix}a\vec{u}+b\vec{v}\\ c\vec{u}+d\vec{v}\end{bmatrix}\\ \text{for } \beta = ca^{-1} \text{ and } \alpha = d - \beta b\end{array}} \tag{13}$$

Remark 24. *In practice for 2×2 blocks, if a is not invertible, permuting the rows is sufficient (since c has then to be invertible for the matrix to be invertible): For $J = \left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right]$, if $\tilde{M} = \left[\begin{smallmatrix}c & d\\ 0 & b\end{smallmatrix}\right] = J{\cdot}M$, then $M = J{\cdot}\tilde{M}$ and $M^{-1} = \tilde{M}^{-1}{\cdot}J$.*

This can be simplified further by setting $\gamma = cd^{-1}$, $\delta = -\gamma b$, if d is also invertible, so that now the algorithm becomes: $\vec{v} \mathrel{*}= d$; $\vec{v} \mathrel{+}= b\cdot\vec{u}$; $\vec{u} \mathrel{*}= \delta$; $\vec{u} \mathrel{+}= \gamma\cdot\vec{v}$.

Finally, if d is not invertible, an additional step seems to be required, with for instance: $\vec{v} \mathrel{*}= b$; $\vec{u} \mathrel{*}= (-c)$; $\vec{u} \mathrel{+}= \vec{v}$; $\vec{v} \mathrel{-}= \vec{u}$; $\vec{u} \mathrel{+}= \vec{v}$.

We now have the tools for in-place polynomial algorithms. We start, in Algorithm 25, with a version of Algorithm 7 for which the multiplications are accumulated into two consecutive blocks (denoted **MUL**-2D).

Algorithm 25 In-place bilinear 2 by 2 formula.

```
Inputs: $\vec{a} \in \mathbb{F}^m$, $\vec{b} \in \mathbb{F}^n$, $\vec{c} \in \mathbb{F}^s$; $\alpha \in \mathbb{F}^{t\times m}$, $\beta \in \mathbb{F}^{t\times n}$, $\mu^{(2)} \in \mathbb{F}^{s\times(2t)} = [M_1 \cdots M_t]$, with no zero-rows
    in $\alpha$, $\beta$, $\mu^{(2)}$, 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: $\alpha, \beta, \mu^{(2)}$.
Result: $\vec{c} \mathrel{+}= \mu^{(2)}\vec{m}$, for $\vec{m} = (\alpha\vec{a}) \odot (\beta\vec{b})$
 1: for $\ell = 1$ to $t$ do
 2:     Let $i$ s.t. $\alpha_{\ell,i} \neq 0$; $a_i \mathrel{*}= \alpha_{\ell,i}$;
 3:     for $\lambda = 1$ to $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ do $a_i \mathrel{+}= \alpha_{\ell,\lambda} a_\lambda$ end for
 4:     Let $j$ s.t. $\beta_{\ell,j} \neq 0$; $b_j \mathrel{*}= \beta_{\ell,j}$;
 5:     for $\lambda = 1$ to $n$, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ do $b_j \mathrel{+}= \beta_{\ell,\lambda} b_\lambda$ end for
 6:     Let $k$, $f$ s.t. $M = \left[\begin{smallmatrix}\mu^{(2)}_{k,2\ell} & \mu^{(2)}_{k,2\ell+1}\\ \mu^{(2)}_{f,2\ell} & \mu^{(2)}_{f,2\ell+1}\end{smallmatrix}\right]$ is invertible;
 7:     $\left[\begin{smallmatrix}c_k\\ c_f\end{smallmatrix}\right] \leftarrow M^{-1} \left[\begin{smallmatrix}c_k\\ c_f\end{smallmatrix}\right]$                    {via Equation (13) and Remark 24}
 8:     for $\lambda = 1$ to $s$, $\lambda \notin \{f,k\}$, $\mu^{(2)}_{\lambda,2\ell} \neq 0$ do $c_\lambda \mathrel{-}= \mu^{(2)}_{\lambda,2\ell} c_k$ end for
 9:     for $\lambda = 1$ to $s$, $\lambda \notin \{f,k\}$, $\mu^{(2)}_{\lambda,2\ell+1} \neq 0$ do $c_\lambda \mathrel{-}= \mu^{(2)}_{\lambda,2\ell+1} c_f$ end for
10:     $\left[\begin{smallmatrix}c_k\\ c_f\end{smallmatrix}\right] \mathrel{+}= a_i \cdot b_j$          {this is the accumulation of the product $\left[\begin{smallmatrix}m_k\\ m_f\end{smallmatrix}\right]$}
11:     for $\lambda = 1$ to $s$, $\lambda \notin \{f,k\}$, $\mu^{(2)}_{\lambda,2\ell+1} \neq 0$ do $c_\lambda \mathrel{+}= \mu^{(2)}_{\lambda,2\ell+1} c_f$ end for
12:     for $\lambda = 1$ to $s$, $\lambda \notin \{f,k\}$, $\mu^{(2)}_{\lambda,2\ell} \neq 0$ do $c_\lambda \mathrel{+}= \mu^{(2)}_{\lambda,2\ell} c_k$ end for
13:     $\left[\begin{smallmatrix}c_k\\ c_f\end{smallmatrix}\right] \leftarrow M \left[\begin{smallmatrix}c_k\\ c_f\end{smallmatrix}\right]$            {via Equation (13) and Remark 24, undo 7}
14:     for $\lambda = 1$ to $n$, $\lambda \neq j$, $\beta_{\ell,\lambda} \neq 0$ do $b_j \mathrel{-}= \beta_{\ell,\lambda} b_\lambda$ end for ; $b_j \mathrel{/}= \beta_{\ell,j}$;
15:     for $\lambda = 1$ to $m$, $\lambda \neq i$, $\alpha_{\ell,\lambda} \neq 0$ do $a_i \mathrel{-}= \alpha_{\ell,\lambda} a_\lambda$ end for ; $a_i \mathrel{/}= \alpha_{\ell,i}$;
16: end for
```

A C++ implementation of Algorithm 25 (`trilplacer -e`) is also available in the PLinOpt library.

Theorem 26. *Algorithm 25 is correct, in-place, and requires t* **MUL**-2D, $2(\#\alpha+\#\beta+\#\mu^{(2)}-t)$ **ADD** and $2(\sharp\alpha+\sharp\beta+\sharp\mu^{(2)}+2t)$ **SCA** operations.

Proof. Thanks to Equations (12) and (13) and Remark 24, correctness is similar to that of Algorithm 7 in Theorem 9. Then, Equation (13) requires 4 **SCA** and 2 **ADD** operations and is called $2t$ times. The rest is similar to Algorithm 7 and amounts to $2t+2(\#\alpha-t+\#\beta-t+\#\mu^{(2)}-2t)+(2t)2$ **ADD** and $2(\sharp\alpha+\sharp\beta+\sharp\mu^{(2)}-2t)+(2t)4$ **SCA** operations. □

There remains to use a double expansion of the post matrix μ to simulate the double size of the intermediate products (**MUL**-2D), producing $\mu^{(2)}$, as in Equation (11) (that can then be used as an input in Algorithm 25). This double expansion matrix $\mu^{(2)}\in\mathbb{F}^{s\times(2t)}$ is simply obtained by duplicating and interleaving all the entries of $\mu\in\mathbb{F}^{(s-1)\times t}$ as follows:

$$\forall i\in[1..(s-1)],\forall j\in[1..t],\begin{cases}\mu^{(2)}(1,2j)=0\\ \mu^{(2)}(i,2j-1)=\mu(i,j)\\ \mu^{(2)}(i+1,2j)=\mu(i,j)\\ \mu^{(2)}(s,2j-1)=0\end{cases}\tag{14}$$

We prove, in Lemma 27, that in fact any such double expansion of a representative matrix is suitable for the in-place computation of Algorithm 25.

Lemma 27. *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, as in Equation (14), contains an invertible lower triangular 2×2 sub-matrix.*

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 $\begin{bmatrix}a&0\\b&a\\*&b\end{bmatrix}$. □

4.3 In-place accumulating Karatsuba multiplication

We now can come back to the Karatsuba case.

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 Equation (11), 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 28 thereafter. To complete Equation (10), 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 28. 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. The in-place algorithm can be obtained automatically (up to some sign swaps) via the combination of Equation (14) and Algorithm 25 with the PLinOpt library, running:

```
./bin/trilplacer -e data/1o1o2_3_Karatsuba_{L,R,P}.sms
```

The resulting details can be found in Algorithm 28.

Algorithm 28 In-place accumulating Karatsuba polynomial multiplication.

```
Inputs: $A$, $B$, $C$ polynomials of degrees $m$, $n$, $m+n$ with $m \geq n$.
Result: $C \mathrel{+}= AB$
if $n \leq$ Threshold then                              {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place multiplication.                                    {Algorithm 2}
else if $m > n$ then
    Let $A(X) = A_0(X) + X^{n+1}A_1(X)$
    $C_{0..2n} \mathrel{+}= A_0 B$                                                 {recursive call}
    if $m \geq 2n$ then
        $C_{(n+1)..(n+m)} \mathrel{+}= A_1 B$                                      {recursive call}
    else
        $C_{(n+1)..(n+m)} \mathrel{+}= B A_1$                                      {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
```

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

Proposition 29. *Algorithm 28 is correct and requires $\mathcal{O}(mn^{\log_2(3)-1})$ operations.*

Proof. With the above analysis, correctness comes from that of Equation (14) and Algorithm 25 applied to Equation (9). 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 non-zero residue). Then, Algorithm 28 requires less than $\mathcal{O}(\sum_{i=1}^{k-1} \lfloor \frac{u_i}{u_{i+1}} \rfloor u_{i+1}^{\log_2(3)}) \leq \mathcal{O}(\sum_{i=1}^{k-1} u_i u_{i+1}^{\log_2(3)-1})$ operations. But, $u_{i+1} \leq u_2 = n$ and if we let $s_i = u_i + u_{i+1}$, $u_i \leq s_i$. This means that the number of operations is bounded by $\mathcal{O}(\sum_{i=1}^{k-1} s_i n^{\log_2(3)-1})$. From [33, 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 α, β and $\mu^{(2)}$ being 1 or -1, Algorithm 28 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 [51]. Now in practice, the supplementary operations are in fact, at least mostly, compensated by the gain in memory allocations or movements as shown in Figure 2. We indeed have implemented Algorithm 28 with the NTL library,[3] and we compare it to two other Karatsuba modular (60 bit prime) implementations:

- a multiplication that recursively allocates temporary polynomials when needed;
- the state-of-the-art library NTL multiplication implementation that pre-allocates once a stack larger than the extra memory requirements at any point in the recursive algorithm.

The observed results are dependent on the architecture and the compiler. We provide two graphs with two different architectures, both using the same compiler clang-17 which appears to produce faster code. In both cases, the implementation with on-the-fly allocations is significantly slower than the others. On one architecture, the in-place version is very close to the stacked implementation, in fact never more than 10% slower. And on the second architecture, the stacked implementation is at least 20% slower.

We also compare in Table 3 the procedure given in Algorithm 28 (obtained via the automatic application of Equation (14) and Algorithm 25) with previous Karatsuba-like algorithms for polynomial multiplications, designed to reduce their memory footprint (see also [27, Table 2.2]).

Table 3: Reduced-memory algorithms for Karatsuba polynomial multiplication

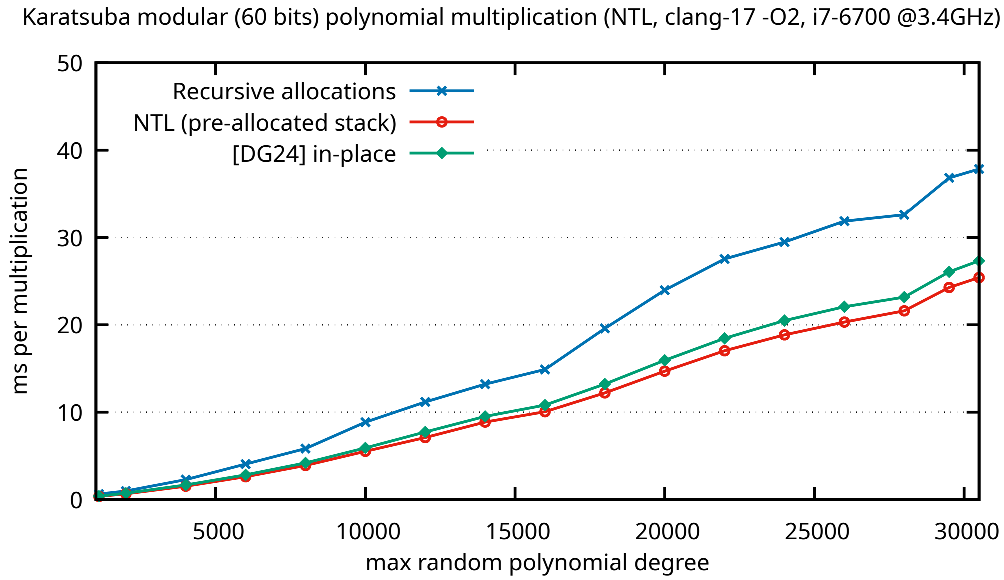
Algorithm	Memory Reg.		Inputs	Accumulation
	Algebraic	Pointer		
[46]	$2n$	$5\log n$	read-only	✗
[54]	n	$5\log n$	read-only	✗
[51, 50]	0	$5\log n$	read-only	✗
[29]	$\mathcal{O}(1)$		read-only	✗
Algorithm 28	0	$5\log n$	mutable	✓

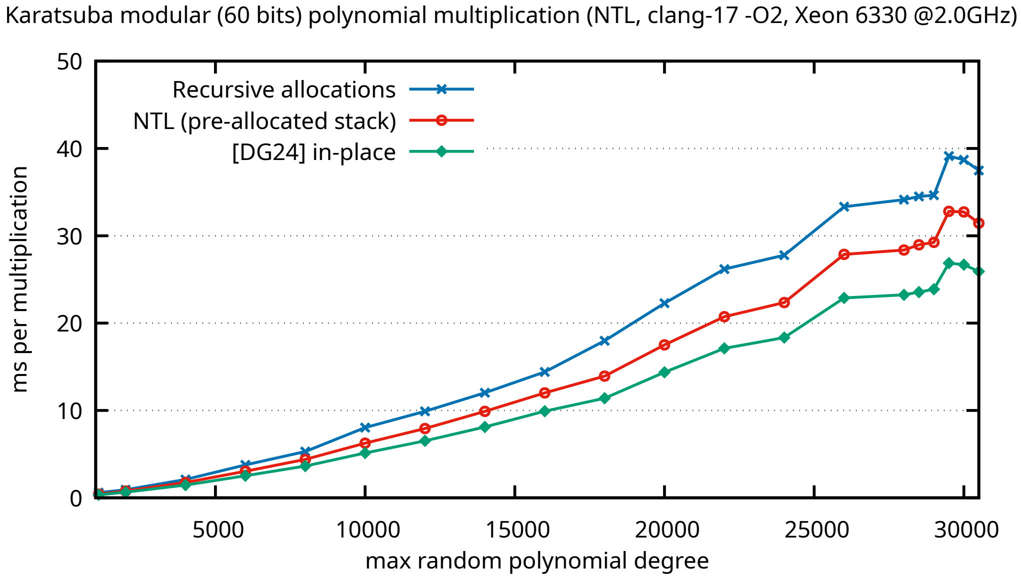
4.4 In-place accumulating Toom-Cook multiplications

We have shown that any bilinear algorithm for polynomial multiplication can be transformed into an in-place version. This approach thus also works for any Toom-k algorithm using $2k-1$ interpolations

[3] https://libntl.org.

Figure 2: In-place Karatsuba polynomial multiplication.





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 Equation (15) thereafter.

$$\mu = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & -\frac{1}{3} & -\frac{1}{6} & 2 \\ -1 & \frac{1}{2} & \frac{1}{2} & 0 & -1 \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{6} & \frac{1}{6} & -2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix};\ \alpha = \beta = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 2 & 4 \\ 0 & 0 & 1 \end{bmatrix} \tag{15}$$

It is then possible to apply Equation (14) and obtain $\mu^{(2)}$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & 1 & 1 & 0 & -\frac{1}{3} & 0 & -\frac{1}{6} & 0 & 2 & 0 \\ -1 & -\frac{1}{2} & \frac{1}{2} & 1 & \frac{1}{2} & -\frac{1}{3} & 0 & -\frac{1}{6} & -1 & 2 \\ \frac{1}{2} & -1 & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{6} & \frac{1}{2} & \frac{1}{6} & 0 & -2 & -1 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} & 0 & -\frac{1}{6} & 0 & \frac{1}{6} & 1 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

Applying Theorem 26 to α and β in Equation (15), and $\mu^{(2)}$ in Equation (16) gives an operation count of $2(11{+}11{-}2{*}5){+}2(2(16{-}5)) = 68$ additions and $2(2{+}2{+}2(11)) = 52$ scalar multiplications.

The PLinOpt implementation of Algorithm 25 on these matrices is:

```
./bin/trilplacer -e data/2o2o4_5_Toom3_{L,R,P}.sms
```

This automatically generates an in-place straight-line program (SLP) using only 58 additions, as it simplifies some intermediate post and pre-additions. Further direct simplifications can be found (running PLinOpt's `./bin/optimizer` for the operations in-between recursive calls), to produce an in-place SLP using then only 43 extra additions and 34 extra scalings.

Further optimizations should be tried to further reduce the complexity, for instance, trying all possible orderings of the recursive calls and incorporating the tricks presented in [7, 8, 10, 27].

4.5 In-place accumulating FFT-based 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 Nth 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 less than N. The discrete Fourier transform of F at ω 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 [15]. 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 30. *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* ω^{-1} *computes* $N \cdot \mathsf{brDFT}_N^{-1}(\cdot, \omega)$ *[5, Exercise 49].*

We start with the in-place Algorithm 31 for the power of two case.

Algorithm 31 In-place power of two accumulating multiplication.

```
Inputs: $\vec{a}$, $\vec{b}$ 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 $N = 2^p$.
Result: $\vec{c}$ contains the coefficients of $C + A \cdot B$.
 1: $\mathsf{brDFT}_{2n}(\vec{c}, \omega)$;                                              {over-place}
 2: $\mathsf{brDFT}_n(\vec{a}, \omega^2)$; $\mathsf{brDFT}_n(\vec{b}, \omega^2)$                        {over-place}
 3: for $i = 0$ to $n-1$ do $c_i$ += $a_i \times b_i$ end for
 4: $\mathsf{brDFT}_n^{-1}(\vec{a}, \omega^2)$; $\mathsf{brDFT}_n^{-1}(\vec{b}, \omega^2)$                      {undo 2}
 5: for $i = 0$ to $n-1$ do $a_i$ *= $\omega^i$; $b_i$ *= $\omega^i$ end for
 6: $\mathsf{brDFT}_n(\vec{a}, \omega^2)$; $\mathsf{brDFT}_n(\vec{b}, \omega^2)$                        {over-place}
 7: for $i = 0$ to $n-1$ do $c_{i+n}$ += $a_i \times b_i$ end for
 8: $\mathsf{brDFT}_n^{-1}(\vec{a}, \omega^2)$; $\mathsf{brDFT}_n^{-1}(\vec{b}, \omega^2)$                      {undo 6}
 9: for $i = 0$ to $n-1$ do $a_i$ /= $\omega^i$; $b_i$ /= $\omega^i$ end for                     {undo 5}
10: $\mathsf{brDFT}_{2n}^{-1}(\vec{c}, \omega)$
```

Theorem 32. *Using an over-place* brDFT *algorithm with complexity bounded by $\mathcal{O}(n \log n)$, Algorithm 31 is correct, in-place and has complexity bounded by $\mathcal{O}(n \log n)$.*

Proof. Algorithm 31 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 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 brDFT^{-1} in size $2n$. A $\mathsf{brDFT}^{\pm 1}$ in size n requires $\frac{3}{2} n \log n + \mathcal{O}(n)$ ring operations if the required powers of ω are precomputed [26]. Therefore, the standard algorithm uses $9n \log n + \mathcal{O}(n)$ ring operations. By contrast, our in-place variant uses 2 size-$2n$ and 8 size-n $\mathsf{brDFT}^{\pm 1}$ and cannot precompute the powers of ω. Each call to $\mathsf{brDFT}^{\pm 1}$ in size n then requires $2n \log n + \mathcal{O}(n)$ ring operations, and the full algorithm requires $24n \log n + \mathcal{O}(n)$ ring operations. The dominant term is therefore $\frac{8}{3}$ times as large in the in-place variant.

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 [38]. Let ω still be an Nth 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 ω 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 [37, 50, 3, 16].

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 33 ([37, 50])**.** *Let $F \in \mathbb{D}[X]$, ℓ, $s \in \mathbb{Z}> 0$ where 2^ℓ divides s, and ω be a 2^pth principal root of unity. If $F_{s,\ell}(X) = F(\omega^{[s]_p} X) \bmod X^{2^\ell - 1}$, $\mathsf{brDFT}_{2^\ell}(F_{s,\ell}, \omega^{2^{p-\ell}}) = (F(\omega^{[s]_p}), \ldots, F(\omega^{[s+2^\ell-1]_p}))$.*

Proof. Let $\omega_\ell = \omega^{2^{p-\ell}}$. This is a principal 2^ℓth root of unity since ω is a principal 2^pth 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^ℓ divides s. Finally, $F_{s,\ell}(\omega_\ell^{[i]_\ell}) = F(\omega^{[s+i]_p})$. □

Corollary 34. *Let $F \in \mathbb{D}[X]$ stored in an array $\vec{f}$ of length n, ℓ, $k \in \mathbb{Z}_{>0}$ and ω be a 2^pth 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^ℓ entries of $\vec{f}$ by $F(\omega^{[k \cdot 2^\ell]_p})$, …, $F(\omega^{[(k+1)\cdot 2^\ell - 1]_p})$, and an inverse algorithm $\mathsf{partTFT}_{k,\ell}^{-1}$ 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:

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

Its correctness is ensured by Lemma 33, while, $\mathsf{partTFT}_{k,\ell}^{-1}(\vec{f}, \omega)$, its inverse algorithm, does the converse:

```
$\mathsf{brDFT}_{2^\ell}^{-1}(\vec{f}_{0..2^\ell-1}, \omega^{2^{p-\ell}})$
**for** $i = 2^\ell$ **to** $n-1$ **do** $f_{i-2^\ell} \mathrel{-}= f_i$ **end for**
**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 ℓ, 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. Similarly, it is possible to compute $(B(\omega^{[k\cdot 2^{\ell+t}]_p}), \ldots, B(\omega^{[(k+1)\cdot 2^{\ell+t} - 1]_p}))$ and restore B.

Theorem 36. *Algorithm 35 is correct and in-place. If the algorithm brDFT used inside $\mathsf{partTFT}$ has complexity $\mathcal{O}(n \log n)$, then the running time of Algorithm 35 is $\mathcal{O}(n \log n)$.*

Proof. The fact that the algorithm is in-place comes from Corollary 34. 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 [37] 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 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^ℓ, using a smaller $\mathsf{partTFT}$. After each computation in $\vec{a}$, we update the corresponding

Algorithm 35 In-place fast accumulating polynomial multiplication.

Inputs: $\vec{a}$, $\vec{b}$ 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^pth root of unity with $2^{p-1} < m+n-1 < 2^p$

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

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

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 35 is compared with previous FFT-based algorithms for polynomial multiplications designed to reduce their memory footprint in Table 4 (see also [27, 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 4: Reduced-memory algorithms for FFT polynomial multiplication

Algorithm	Algebraic Reg.	Inputs	Accumulation
[15]	$2n$	read-only	✗
[51]	$\mathcal{O}(2^{\lceil \log_2 n \rceil} - n)$	read-only	✗
[37]	$\mathcal{O}(1)$	read-only	✗
Algorithm 35	$\mathcal{O}(1)$	mutable	✓

5 Fast in-place convolution with accumulation

In this section, we reduce in-place accumulating convolutions to in-place multiplication with accumulation. This allows us in the next section to describe in-place algorithms for structured matrix operations, and to ultimately obtain an in-place fast polynomial remainder in the subsequent section.

From algorithms for polynomial multiplications at a lower level, one can devise an algorithm for the generalized accumulated convolution $C \mathrel{+}= AB \mod (X^n - f)$. To compute such convolutions, one can reduce the computations to full in-place products. The previous section proves that these computations have cost $O(\mathfrak{M}(n))$. Then, the initial idea is to unroll a first recursive iteration and then to call any in-place accumulated polynomial multiplication on halves.

We describe several algorithms. In Section 5.1, we deal with the case where n is even. Algorithm 37 uses a Karatsuba-like iteration in the case $f \notin \{0, 1\}$ while Algorithm 38 deals with the case $f = 1$. Section 5.2 treats the case of an odd n when f is invertible, and Section 5.3 the remaining cases of a short product, that is when $f = 0$. Section 5.4 summarizes these algorithms and provides the required proofs.

5.1 Even dimension

Suppose indeed first that n is even. Let $t = n/2$, and $A(X) = a_0(X) + X^t a_1(X)$, $B(X) = b_0(X) + X^t b_1(X)$, $C(X) = c_0(X) + X^t c_1(X)$, all of degree $n-1 = 2t-1$ (so that all of a_0, a_1, b_0, b_1, c_0 and c_1 are of degree at most $t-1$). Then let $\tau_0 + X^t \tau_1 = a_0 b_1 + a_1 b_0$. Since $X^{2t} = X^n \equiv f \mod X^n - f$, we have that: $C + AB \mod (X^n - f) = C + a_0 b_0 + X^t \tau_0 + f \cdot \tau_1 + f \cdot a_1 b_1$. This can be computed with 4 full accumulated sequential products, each of degree no more than $2t - 2 = n - 2$, and by exchanging the lower and upper parts when accumulating $a_0 b_1 X^t$ and $a_0 b_1 X^t$: this comes from the fact that $(\tau_0 + X^t \tau_1) X^t \equiv X^t \tau_0 + f \cdot \tau_1 \mod X^n - f$.

À la Karatsuba, a more efficient version could use only 3 full polynomial products: Let instead $m_0 = m_{00} + m_{01} X^t = a_0 b_0$, $m_1 = m_{10} + m_{11} X^t = (a_0 + a_1)(b_0 + b_1)$ and $m_2 = m_{20} + m_{21} X^t = a_1 b_1$ be these 3 full products, to be computed and accumulated in-place. As $X^{2t} = X^n \equiv f \mod X^n$, then $C + AB \mod (X^n - f)$ is also:

$$\begin{multline} c_0 + m_{00} + f m_{20} + f(m_{11} - m_{01} - m_{21}) \\ + \Big(c_1 + m_{01} + f m_{21} + (m_{10} - m_{00} - m_{20})\Big) X^t. \quad (17) \end{multline}$$

Equation (17) is an in-place accumulating linear computation: it computes $\vec{c} \mathrel{+}= \mu \vec{m}$ where $\vec{m} = (\alpha \vec{a}) \odot (\beta \vec{b})$, for $\alpha = \beta = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \in \mathbb{F}^{3 \times 2}$ and $\mu = \left[\begin{array}{cc|cc|cc} 1 & -f & 0 & f & f & -f \\ -1 & 1 & 1 & 0 & -1 & f \end{array}\right] \in \mathbb{F}^{2 \times 6}$. If $f \notin \{0, 1\}$ and

$\mu = [M_0|M_1|M_2]$, $M_0^{-1} = (1-f)^{-1}\left[\begin{smallmatrix}1 & f\\ 1 & 1\end{smallmatrix}\right]$, $M_1^{-1} = \left[\begin{smallmatrix}0 & 1\\ f^{-1} & 0\end{smallmatrix}\right]$, and $M_2^{-1} = (f^2-f)^{-1}\left[\begin{smallmatrix}f & f\\ 1 & f\end{smallmatrix}\right]$. From this, Equation (14) and Algorithm 25 derives an accumulating in-place algorithm, using 2×2 invertible blocks, that computes $\left[\begin{smallmatrix}c_i\\ c_j\end{smallmatrix}\right] \mathrel{+}= M\left[\begin{smallmatrix}\rho_0\\ \rho_1\end{smallmatrix}\right]$, via the equivalent algorithm: $\left[\begin{smallmatrix}c_i\\ c_j\end{smallmatrix}\right] \mathrel{*}= M^{-1}$; $\left[\begin{smallmatrix}c_i\\ c_j\end{smallmatrix}\right] \mathrel{+}= \left[\begin{smallmatrix}\rho_0\\ \rho_1\end{smallmatrix}\right]$; $\left[\begin{smallmatrix}c_i\\ c_j\end{smallmatrix}\right] \mathrel{*}= M$. After some simplifications described below, we obtain Algorithm 37.

Algorithm 37 In-place even degree accumulating f-convolution.

```
Inputs: $A(X)$, $B(X)$, $C(X)$ polynomials of odd degree $n-1$; $f \in \mathbb{F} \setminus \{0,1\}$.
Result: $C \mathrel{+}= AB \mod (X^n - f)$
if $n \le$ Threshold then                                {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place polynomial multiplication.
else
    Let $t = n/2$;                                                             {$n$ is even},
    Let $A = a_0 + X^t a_1$; $B = b_0 + X^t b_1$; $C = c_0 + X^t c_1$;                 {degrees $< n/2$}
    $c_1 \mathrel{+}= c_0$;
    $c_1 \mathrel{/}= (1-f)$;                                                  {simplification (i)}
    $c_0 \mathrel{+}= f \cdot c_1$;
    $\begin{bmatrix}c_0\\ c_1\end{bmatrix} \mathrel{+}= a_0 \cdot b_0$;          {acc. full prod. $m_0$ with $a_0 b_0 = \begin{bmatrix}m_{00}\\ m_{01}\end{bmatrix}$}
    $c_0 \mathrel{/}= f$;                                             {simplifications (ii) and (iii)}
    $\begin{bmatrix}c_1\\ c_0\end{bmatrix} \mathrel{-}= a_1 \cdot b_1$;          {acc. full prod. $m_2$ with $a_1 b_1 = \begin{bmatrix}m_{20}\\ m_{21}\end{bmatrix}$}
    $c_0 \mathrel{-}= c_1$;                          {this is $c_0/f + m_{00}/f - m_{01} + m_{20} - m_{21}$}
    $c_1 \mathrel{*}= (1-f)$;                                                 {simplification (iv)}
    $c_1 \mathrel{-}= f \cdot c_0$;                       {this is $c_1 - m_{00} + m_{01} - m_{20} + f m_{21}$}
    $a_0 \mathrel{+}= a_1$;
    $b_0 \mathrel{+}= b_1$;
    $\begin{bmatrix}c_1\\ c_0\end{bmatrix} \mathrel{+}= a_0 \cdot b_0$;   {acc. full prod. $m_1$ with $(a_0+a_1)(b_0+b_1) = \begin{bmatrix}m_{10}\\ m_{11}\end{bmatrix}$}
    $b_0 \mathrel{-}= b_1$;
    $a_0 \mathrel{-}= a_1$;
    $c_0 \mathrel{*}= f$;                                                      {simplification (v)}
end if
```

The algorithm is obtained from the output of Equation (14) and Algorithm 25 after applying the following simplifications:

(i) As $1 + f(1-f)^{-1} = (1-f)^{-1}$, then $M_0^{-1}\left[\begin{smallmatrix}c_0\\ c_1\end{smallmatrix}\right] = (1-f)^{-1}\left[\begin{smallmatrix}1 & f\\ 1 & 1\end{smallmatrix}\right]\left[\begin{smallmatrix}c_0\\ c_1\end{smallmatrix}\right]$ can be sequentially computed via $c_1 \mathrel{+}= c_0$; $c_1 \mathrel{/}= (1-f)$; $c_0 \mathrel{+}= f c_1$.

(ii) As $(M_2^{-1} \cdot M_0) = \begin{bmatrix}0 & -1\\ -f^{-1} & 0\end{bmatrix} = -\begin{bmatrix}1 & 0\\ 0 & f^{-1}\end{bmatrix}\left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right]$, then computing M_2 right after M_0 allows simplifying the intermediate in-place operations into a variable swap, negations and multiplication by f^{-1}.

(iii) That negation can be delayed, since $(-M \cdot c) \mathrel{+}= \rho$ is equivalent to $-((M \cdot c) \mathrel{-}= \rho)$.

(iv) Similarly, $-(M_1^{-1} \cdot M_2) = \left[\begin{smallmatrix}1 & -f\\ -1 & 1\end{smallmatrix}\right]$. Up to some swaps, this is simplified as $\left[\begin{smallmatrix}1 & 0\\ -f & 1\end{smallmatrix}\right] \cdot \left[\begin{smallmatrix}1 & 0\\ 0 & 1-f\end{smallmatrix}\right] \cdot \left[\begin{smallmatrix}1 & -1\\ 0 & 1\end{smallmatrix}\right]$, equivalent to the sequential computation: $c_0 \mathrel{-}= c_1$; $c_1 \mathrel{*}= (1-f)$; $c_1 \mathrel{-}= f c_0$.

(v) Finally, $M_1 = \left[\begin{smallmatrix} f & 0 \\ 0 & 1 \end{smallmatrix}\right] \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ and thus this combination is again just a swap of variables and a multiplication by f.

Overall, we obtain Algorithm 37 that works for even n with $f \notin \{0, 1\}$. We then need to design variant algorithms for the other cases. Algorithm 38 deals with the case where $f = 1$ (as M_0 and M_2 are not invertible when $f = 1$). We present here a version with 4 full products, but a more efficient version with 3 products only, like Algorithm 37, could also be derived.

Algorithm 38 In-place even degree accumulating 1-convolution.

```
Inputs: $A(X)$, $B(X)$, $C(X)$ polynomials of odd degree $n-1$;
Result: $C \mathrel{+}= AB \mod (X^n - 1)$
if $n \le$ Threshold then                                  {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place polynomial multiplication.
else
    Let $t = n/2$;                                         {$n$ is even, let $Y = X^t$, so that $Y^2 \equiv 1$},
    Let $A = a_0 + X^t a_1$; $B = b_0 + X^t b_1$; $C = c_0 + c_1 X^t$;
    $\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \mathrel{+}= a_0 \cdot b_0$      {acc. full prod. of degree $2t - 2 \le n - 1$}
    $\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \mathrel{+}= a_1 \cdot b_1$      {acc. full prod. since $Y^2 \equiv 1$}
    $\begin{bmatrix} c_1 \\ c_0 \end{bmatrix} \mathrel{+}= a_0 \cdot b_1$      {acc. full prod. since $(u + vY)Y \equiv v + uY$}
    $\begin{bmatrix} c_1 \\ c_0 \end{bmatrix} \mathrel{+}= a_1 \cdot b_0$      {acc. full prod. since $(u + vY)Y \equiv v + uY$}
end if
```

5.2 Odd dimension

Algorithm 39 deals with the odd-n case where f is invertible. For the sake of simplicity, we also only present the version with 4 full products. The additional difficulty here is that the degrees of the lower and upper parts are different. Therefore, Algorithm 39 ensures that there is always the correct space to accumulate.

5.3 Short product

Finally, we also have to deal with the case $f = 0$, that is with the in-place short product $C \mathrel{+}= AB \mod X^n$. A first idea would be to again split the input polynomials in two parts, $A(Y) = X^{n/2}a_1 + a_0$ and $B(Y) = X^{n/2}b_1 + b_0$, then compute $C(Y) \mathrel{+}= (a_0 b_0) + (a_1 b_0 + a_0 b_1 \mod X^{n/2})$. This would be one full in-place polynomial multiplication with accumulation, for $a_0 b_0$, and two recursive calls of half degree, for $a_1 b_0$ and $a_0 b_1$. Unfortunately, these two recursive calls when splitting in two would induce some extra logarithmic factors in the complexity when the full multiplication is not $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$.

We have obtained different alternative solutions, either with less recursive calls, or splitting the input polynomials in more parts, and some of them with restrictions on the field characteristic. We first present one variant in detail in Section 5.3.1. We then give only the HM representations of more variants, and compare their respective complexity bounds in Sections 5.3.2 and 5.3.3.

Algorithm 39 In-place odd degree accumulating f-convolution.

```
Inputs: $A(X)$, $B(X)$, $C(X)$ polynomials of even degree $n-1$; $f \in \mathbb{F}^*$.
Result: $C \mathrel{+}= AB \mod (X^n - f)$
if $n \le$ Threshold then                                    {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place polynomial multiplication.
else
    Let $t = (n+1)/2$;                                        {$n$ is odd, so that $X^{2t} \equiv fX$},
    Let $A = a_0 + X^t a_1$; $B = b_0 + X^t b_1$; $C = c_0 + c_1 X^t$;
                                                              {$a_1$, $b_1$, $c_1$ of degree $n-1-t = t-2$}
    $\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \mathrel{+}= a_0 \cdot b_0$                {acc. full prod. of degree $2t-2 \le n-1$}
    $\begin{bmatrix} c_{1..(t-1)} \\ c_1 \end{bmatrix} \mathrel{+}= a_1 \cdot b_1$       {acc. full prod. of degree $(2t-4)+1 \le n-1$}
    $c_{0..(t-2)} \mathrel{/}= f$;
    $\begin{bmatrix} c_{t..(2t-2)} \\ c_{0..(t-2)} \end{bmatrix} \mathrel{+}= a_0 \cdot b_1$    {acc. full prod. since $\left\{ \begin{smallmatrix} 2t-3<n-1 \\ (u+vX^{t-1})X^t \equiv fv+uX^t \end{smallmatrix} \right\}$
    $\begin{bmatrix} c_{t..(2t-2)} \\ c_{0..(t-2)} \end{bmatrix} \mathrel{+}= a_1 \cdot b_0$    {acc. full prod. since $\left\{ \begin{smallmatrix} 2t-3<n-1 \\ (u+vX^{t-1})X^t \equiv fv+uX^t \end{smallmatrix} \right\}$
    $c_{0..(t-2)} \mathrel{*}= f$;
end if
```

Remark 40. *The algorithms described below have a better complexity than [21, Alg. 4]. Another approach, suitable over fields with at least three elements, is to compute the short product as a sum of two convolutions modulo $X^n - f$ for two non-zero distinct values f,* cf. *[21, Remark 1]. This approach is also slightly less efficient than the ones presented here.*

5.3.1 Splitting in three, with a single recursive call

We here split the input and output polynomials in 3 blocks of size close to $n/3$. A first possibility is given in Equation (18), with overlapping intermediate polynomials t_i:

$$\boxed{\begin{array}{l} A(Y) = Y^2 a_2 + Y a_1 + a_0; B(Y) = Y^2 b_2 + Y b_1 + b_0; C(Y) = Y^2 c_2 + Y c_1 + c_0; \\ \quad m_0 = a_0(b_0 - b_2); m_1 = a_0 b_2; m_2 = (a_0 + a_1) b_1; m_3 = a_1(b_0 - b_1); m_4 = (a_1 + a_2) b_0; \\ \quad t_0 = c_0 + m_0 + m_1; \quad t_1 = c_1 + m_2 + m_3; \quad t_2 = c_2 + m_1 - m_3 + m_4; \\ \textbf{then } C + AB \mod Y^3 \equiv Y^2 t_2 + Y t_1 + t_0 \mod Y^3 \end{array}} \tag{18}$$

The HM representation corresponding to the complete operations of Equation (18) (*i.e.*, $Y^2 t_2 + Y t_1 + t_0$ without discarding the degrees larger than Y^3) is given in Equation (19):

$$\mu = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & -1 & 1 \end{bmatrix}; \quad \alpha = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}; \quad \beta = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & -1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{19}$$

Splitting the five products $m_i = m_{i0} + m_{i1} Y$, as in Equation (11), we obtain the 2×2 expansion of μ in Equation (19) as a 4×10 matrix $\mu^{(2)}$. The technique to get a short product algorithm from this, is to not compute the high degree coefficients: this is just discarding the last row of $\mu^{(2)}$, to obtain the 3×10 matrix given in Equation (20):

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \mathrel{+}= \left[\begin{array}{cc|cc|cc|cc|cc} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -1 & 1 & 1 & 0 \end{array}\right] \cdot \begin{bmatrix} m_{00} \\ m_{01} \\ \vdots \\ m_{41} \end{bmatrix}. \tag{20}$$

Now the four first pairs of columns of Equation (20) can be dealt with the technique of Algorithm 25 and the last pair is $m_{40} \mod X^{n/3}$, thus performed via a single recursive call of degree $n/3$. Note that one can directly read off the matrix μ in Equation (19) that there is a single recursive call: the number of recursive calls is indeed the number of columns having a single non-zero in the last row (after expansion and discarding, this column becomes a pair of non-full rank columns in Equation (20)).

Remark 41. *Note that since m_0 is used only once, one could modify Equation (18) to get a seemingly simplified version as follows: replace $m_0 = a_0(b_0 - b_2)$; $m_1 = a_0 b_2$; $\dots$ $t_0 = c_0 + m_0 + m_1$ by instead $m_0' = a_0 b_0$; $m_1 = a_0 b_2$; $\dots$ $t_0 = c_0 + m_0'$. But this would now require a second recursive call, as the high degree part of m_1 can not be computed nor stored anymore.*

For the first four calls to full-products, it is sufficient to extract the first four columns of μ and the first four rows of α and β in Equation (19):

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} \mathrel{+}= \mu^{(2)}_{1..3,1..8} \cdot \left(\alpha_{1..4,*} \begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} \odot \beta_{1..4,*} \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix}\right)$$

Algorithm 25 can then be called on these sub-matrices via running the PLinOpt library with:

```
./bin/trilplacer -e data/2o2o2_4_partSP_{L,R,P}.sms
```

It is then sufficient to extract only the computations c_0, c_1, c_2 and remove that of c_3 (the highest degrees of the result are not used when computing modulo X^n – or $Y^3 = (X^{n/3})^3$ in Equation (18)). This part thus needs only 10 additions and 4 calls to in-place accumulating products of degrees $n/3$ to compute the first four columns of Equation (20). This is shown in lines 6 to 14 of Algorithm 42.

Then, the recursive call is the low degree part of the last column of μ and the last row of α and β: $c_2 \mathrel{+}= \{(a_1 + a_2) \cdot b_0\}_{\text{low}}$. Finally one deals with degrees above $3\lceil n/3 \rceil$ directly and, overall, we now obtain Algorithm 42.

Remark 43. *Even though Algorithm 42 only makes one recursive call, this is not a tail recursive call. Therefore, a call stack is* a priori *needed. Yet, it is easily seen that this call stack can be removed. Indeed, Algorithm 42 could be split in two parts: the first one is a tail recursive algorithm made of all instructions up to the recursive call; the second one is not recursive, and it only restores A (Line 17) and computes the coefficients c_{3t} and c_{3t+1} if needed. This way, no call stack is required since the only recursive algorithm is tail recursive.*

Theorem 44. *Using an in-place polynomial multiplication with complexity bounded by $\mathfrak{M}(n)$, Algorithm 42 is correct, in-place and has complexity bounded by $\frac{4}{3^{1+\epsilon}-1}\mathfrak{M}(n) + \mathcal{O}(n)$.*

Proof. Algorithm 42 uses 4 polynomial multiplications with polynomials of degree $n/3$, then 1 recursive call of a short product of degree $n/3$, and $\mathcal{O}(n)$ extra computations. Solving the recursion for this cost, gives an overall complexity bound of $\frac{4}{3^{1+\epsilon}-1}\mathfrak{M}(n) + \mathcal{O}(n)$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for $0 < \epsilon \leq 1$. If instead $\mathfrak{M}(n) = \tilde{\mathcal{O}}(n)$, one then gets $2\mathfrak{M}(n) + \mathcal{O}(n)$ which is also $\frac{4}{3^{1+\epsilon}-1}\mathfrak{M}(n)$ with $\epsilon = 0$. □

Algorithm 42 In-place accumulating short product with 1 recursive call.

Inputs: $A(X)$, $B(X)$, $C(X)$ polynomials of degree $< n$ in $\mathbb{F}[X]$;
Result: $C \mathrel{+}= AB \mod X^n$.

```
if $n \leq$ Threshold then                                   {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place polynomial multiplication.
else
    Let $t = \lfloor n/3 \rfloor$;
    Let $A \mod X^{3t} = a_0 + X^t a_1 + X^{2t} a_2$; $B \mod X^{3t} = b_0 + X^t b_1 + X^{2t} b_2$; $C \mod X^{3t} = c_0 + c_1 X^t + X^{2t} c_2$;
    {First four columns of Equation (20) via Algorithm 25:}
    $b_0 \mathrel{-}= b_2$;
    $\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \mathrel{+}= a_0 \cdot b_0$;                                   {acc. full prod.}
    $b_0 \mathrel{+}= b_2$; $c_2 \mathrel{-}= c_0$;
    $\begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \mathrel{+}= a_0 \cdot b_2$;                                   {acc. full prod.}
    $a_1 \mathrel{+}= a_0$;
    $\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \mathrel{+}= a_1 \cdot b_1$;                                   {acc. full prod.}
    $a_1 \mathrel{-}= a_0$; $b_0 \mathrel{-}= b_1$; $c_2 \mathrel{+}= c_1$;
    $\begin{bmatrix} c_1 \\ c_2 \end{bmatrix} \mathrel{+}= a_1 \cdot b_0$;                                   {acc. full prod.}
    $b_0 \mathrel{+}= b_1$; $c_2 \mathrel{-}= c_1$; $c_2 \mathrel{+}= c_0$;
    {Fifth pair of columns of Equation (20):}
    $a_1 \mathrel{+}= a_2$;
    $c_2 \mathrel{+}= a_1 \cdot b_0 \mod X^t$                    {recursive call: $c_2 \mathrel{+}= m_4 \mod X^t$}
    $a_1 \mathrel{-}= a_2$;
    {Directly computing highest degrees:}
    if $n \geq 3t+1$ then                                        {scalar accumulations}
        for $i = 0$ to $3t$ do $c_{3t} \mathrel{+}= A_i \cdot B_{3t-i}$ end for
    end if
    if $n = 3t+2$ then                                           {scalar accumulations}
        for $i = 0$ to $3t+1$ do $c_{3t+1} \mathrel{+}= A_i \cdot B_{3t+1-i}$ end for
    end if
end if
```

5.3.2 Short product varying the number of recursive calls

By setting up a polynomial system with the coefficients of the three matrices in an HM representation as indeterminates, one can solve for matrices that produce a short product. With this we performed a search on the obtained solutions for various splittings (in 2 or 3) and associated tensor rank (respectively 3 and 5) and various recursive calls.

From this we obtained that:

- Splitting in 2, there is no algorithm for the short product with no recursive calls (and 3 full multiplications) working in even characteristic;
- Splitting in 3, there is no algorithm for the short product with strictly more than 2 recursive calls (and tensor rank 5);
- Splitting in 3, there is no algorithm for the short product with no recursive calls (and 5 full multiplications) working in even characteristic.

Then for the other situations we have found several possibilities and present some of them next, showing only their initial HM representation.

Splitting in 3 with 2 recursive calls and 3 full multiplications We found Equation (21):

$$\mu = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}, \quad \alpha = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}. \tag{21}$$

On the one hand, Equation (21) uses 3 polynomial multiplications with polynomials of degree $n/3$, then 2 recursive calls of short products of degree $n/3$, and $\mathcal{O}(n)$ extra computations. This gives an overall complexity bound of $\frac{3}{3^{1+\epsilon}-2}\mathfrak{M}(n) + \mathcal{O}(n)$. For instance with a quadratic full polynomial multiplication ($\epsilon = 1$) the in-place short product is computed at a cost lower than half that of the full one (more precisely, the ratio is 3/7).

In terms of ϵ, the cut-off point between Equation (21) and Algorithm 42 is then at $1 + \epsilon = \log_3(5) \approx 1.465$ (the exponent of Toom-3 algorithms).

Split in 3 with no recursive calls, in odd characteristic

$$\mu = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}, \quad \alpha = \begin{bmatrix} 2 & 1 & -1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \\ -1 & -1 & 1 \\ 0 & -1 & 0 \end{bmatrix}, \quad \beta = \begin{bmatrix} 1 & 0 & 0 \\ 1 & -2 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & -1 & 1 \end{bmatrix}. \tag{22}$$

Split in 2 with 1 recursive call

$$\mu = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 1 & 0 \end{bmatrix}, \quad \alpha = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 1 \end{bmatrix}, \quad \beta = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}. \tag{23}$$

Split in 2 with no recursive calls, in odd characteristic (and 1 recursive call in even characteristic)

$$\mu = \begin{bmatrix} -2 & 1 & 1 \\ 1 & -1 & 0 \end{bmatrix}, \qquad \alpha = \begin{bmatrix} 1 & 2 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \qquad \beta = \begin{bmatrix} 0 & 1 \\ -1 & 2 \\ 1 & 2 \end{bmatrix}. \tag{24}$$

5.3.3 Short product efficiency

We now give in Table 5 the dominant term of the complexity bounds for the different short product algorithms of Section 5.3.2. For this we just present the ratios when used with an underlying full polynomial multiplication of complexity $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$, for $0 < \epsilon \leq 1$. Now, when $\mathfrak{M}(n) = \tilde{\mathcal{O}}(n)$, letting $\epsilon = 0$, also provides the correct ratio.

Table 5: Short product complexity ratio with $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$.

Alg.	$\frac{n}{2}$, 0 rec.	$\frac{n}{2}$, 1 rec.	$\frac{n}{3}$, 0 rec.	$\frac{n}{3}$, 1 rec.	$\frac{n}{3}$, 2 rec.
Equation	(24)	(23)	(22)	(19)	(21)
Char.	odd	any	odd	any	any
Extra Mem. reg.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Extra Ptr. reg.	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Ratio	$\frac{3}{2^{1+\epsilon}}$	$\frac{2}{2^{1+\epsilon}-1}$	$\frac{5}{3^{1+\epsilon}}$	$\frac{4}{3^{1+\epsilon}-1}$	$\frac{3}{3^{1+\epsilon}-2}$
$1+\epsilon=2$	$\frac{3}{4}$	$\frac{2}{3}$	$\frac{5}{9}$	$\frac{1}{2}$	$\frac{3}{7}$
Karatsuba	1	1	0.88	0.85	0.81
Toom-3	1.09	1.14	1	1	1
		$\vdots$			
$\mathfrak{M}(n) = \tilde{\mathcal{O}}(n)$	$\frac{3}{2}$	2	$\frac{5}{3}$	2	3

First, as noted in Remark 40, Algorithm 42 (from Equation (19)) is always better than the in-place algorithms we gave in [21, Alg. 4] and [21, Remark 1], with ratio $\frac{4}{3^{1+\epsilon}-1}$ compared to, respectively, $\frac{5}{3^{1+\epsilon}-2}$ and 2.

Then, as long as the number of recursive calls is not larger than 1, the short product algorithms are as in-place as the underlying full polynomial multiplication, thanks to Remark 43. For instance when used with Algorithm 35, the algorithms obtained from Equations (19) and (22) to (24) are fully in-place ($\mathcal{O}(1)$ memory *and* pointer registers); while Equation (21) would require $\mathcal{O}(1)$ memory registers but $\mathcal{O}(\log(n))$ pointer registers.

As expected, on the one hand, we also see that when the underlying polynomial multiplication is closer to quasi-linear, then performing the least possible number of recursive call is more interesting. On the other hand, when the underlying polynomial multiplication is closer to quadratic, then performing as many recursive calls as possible is better.

Finally, there is most probably room for improvements if one considers splitting the initial polynomials in more than three blocks.

5.4 Fast in-place convolution with accumulation

Finally, Algorithms 37 to 39 and 42 all together provide a complete solution for the fast in-place convolution with accumulation, as given in Algorithm 45.

Algorithm 45 In-place convolution with accumulation.

```
Inputs: $A(X)$, $B(X)$, $C(X)$ polynomials of degree $< n$; $f \in \mathbb{F}$.
Result: $C \mathrel{+}= AB \mod (X^n - f)$
if $f = 0$ then Apply Algorithm 42
else if $n$ is odd then Apply Algorithm 39                    $\{f \neq 0\}$
else if $f = 1$ then Apply Algorithm 38                       $\{n$ is even$\}$
else Apply Algorithm 37. end if                               $\{f \notin \{0, 1\}\}$
```

Theorem 46. *Using an in-place polynomial multiplication with complexity bounded by $\mathfrak{M}(n)$, Algorithm 45 is correct, in-place and has complexity bounded by $\mathcal{O}(\mathfrak{M}(n))$.*

Proof. The correctness of Algorithm 37 comes from that of Equation (17). The correctness of Algorithms 38 and 39 is direct from the degrees of the sub-polynomials (as given in the comments, line by line). The correctness of Algorithm 42 comes from that of Equation (18). Also, all four algorithms are in-place as they use only in-place atomic operations or in-place polynomial multiplication.

The complexities of Algorithms 37 to 39 satisfy $T(n) \leq 4\mathfrak{M}(n/2) + \mathcal{O}(n)$ since they use only a linear number of operations plus up to four calls to polynomial multiplications of half-degrees. Since $\mathfrak{M}(n)/n$ is non-decreasing, $T(n) = \mathcal{O}(\mathfrak{M}(n))$. Finally, Algorithm 42 use a linear number of atomic operations, three or four degree-$n/3$ polynomial multiplications and only two or one recursive calls with degree-$n/3$ polynomials. Thus, their complexity bounds satisfy $T(n) \leq 2T(n/3) + 3\mathfrak{M}(n/3) + \mathcal{O}(n)$ or $T(n) \leq 1T(n/3) + 4\mathfrak{M}(n/3) + \mathcal{O}(n)$, whence in both cases $T(n) = \mathcal{O}(\mathfrak{M}(n))$ since $\mathfrak{M}(n)/n$ is non-decreasing. □

6 Circulant and Toeplitz matrix operations with accumulation

Toeplitz matrix-vector multiplication can be reduced to circulant matrix-vector multiplication, via an embedding into a double-size circulant matrix. But this is not immediately in-place, since doubling the size requires a double space. We see in the following how we can instead double the operations while keeping the same dimension. We start by the usual definitions, also extending circulant matrices to f-circulant matrices, following, e.g., [49, Theorem 2.6.4].

Definition 47. *For $\vec{a} \in \mathbb{F}^m$, $\mathscr{C}(\vec{a})$ is the* circulant matrix *represented by $\vec{a}$, that is, the $m \times m$ matrix (C_{ij}), such that $C_{1j} = a_j$ and the $(i+1)$st row is the cyclic right shift by 1 of the ith row.*

Definition 48. *For $f \in \mathbb{F}$ and $\vec{a} \in \mathbb{F}^m$, the (lower) f-*circulant matrix *represented by $\vec{a}$, $\mathscr{C}_{\mathrm{f}}(\vec{a})$, is the $m \times m$ matrix (Γ_{ij}), such that:*

$$\text{for } C = \mathscr{C}(\vec{a}), \begin{cases} \Gamma_{ij} = C_{ij} & \text{if } i \leq j, \\ \Gamma_{ij} = f \cdot C_{ij} & \text{otherwise.} \end{cases}$$

6.1 Fast in-place accumulating (f-)circulant matrix vector product

It is well known that circulant matrices are diagonalized by a discrete Fourier transform, and hence can be manipulated via fast Fourier transforms as $\mathscr{C}_1(\vec{a}) = F_m^{-1}\mathrm{diag}(F_m\vec{a})F_m$, for F_m a DFT-matrix, see, e.g., [32, § 4.7.7]. This gives an alternative way to compute circulant matrix-vector multiplication in-place (using and restoring afterwards both the matrix and the vector). Indeed, the (even truncated) Fourier transform and its inverse can be computed in-place [51, 37, 16]. This gives us an in-place algorithm to compute the *accumulation* $\vec{c} \mathrel{+}= \mathscr{C}_1(\vec{a}) \cdot \vec{b}$ as:

```
$\vec{a} \leftarrow F_m\vec{a}$, $\vec{b} \leftarrow F_m\vec{b}$ and $\vec{c} \leftarrow F_m\vec{c}$;
$\vec{c} \mathrel{+}= \mathrm{diag}(\vec{a}) \cdot \vec{b}$;
$\vec{c} \leftarrow F_m^{-1}\vec{c}$; $\vec{b} \leftarrow F_m^{-1}\vec{b}$ and $\vec{a} \leftarrow F_m^{-1}\vec{a}$.
```

Now, this diagonalization enables us to compute fast in-place accumulated circulant matrix-vector only when primitive roots of sufficiently large order exist. In the following we present more generic fast in-place algorithms that work over any field.

The algebra of f-circulant matrices is in fact isomorphic to the algebra of polynomials modulo $X^n - f$ [49, Theorem 2.6.1]. This means that the product of an f-circulant matrix by a vector is obtained by the convolution of this vector by the vector representing the f-circulant matrix, and thus via Algorithm 45. Note that, as recalled above, the case $f = 1$ can also be computed using a discrete Fourier transform, but only when primitive roots of sufficiently large order exist. By contrast, Algorithm 45 has no restriction on f and is a reduction to any accumulated in-place polynomial multiplication (including DFT ones, as, e.g., Algorithm 35).

6.2 Fast in-place accumulating Toeplitz matrix-vector product

Definition 49. *For $\vec{a} \in \mathbb{F}^{2m-1}$, $\mathcal{T}(\vec{a})$ is the (square)* Toeplitz matrix *represented by $\vec{a}$, that is, the $m \times m$ matrix (T_{ij}), such that $T_{ij} = a_{m+j-i}$. Similarly, for $\vec{a} \in \mathbb{F}^{m+n-1}$, we denote by $\mathcal{T}_{\mathrm{m,n}}(\vec{a})$ the $m \times n$ rectangular Toeplitz matrix defined by its first column, $\vec{a}_{1..m}$ bottom to top, and its first row, $\vec{a}_{m..(m+n-1)}$, left to right.*

The matrix-vector product for rectangular Toeplitz matrices and the middle product of polynomials are the same task, see, e.g., [29, § 3.1]. We also immediately see that with these notations, we have for instance $\mathscr{C}_1(\vec{a}) = \mathscr{C}(\vec{a})$, $\mathscr{C}_0(\vec{a}) = \frac{1}{2}(\mathscr{C}_1(\vec{a}) + \mathscr{C}_{\text{-}1}(\vec{a}))$, or also $\mathscr{C}_{\mathrm{f}}(\vec{a}) = \mathcal{T}([f{\cdot}\vec{a}_{2..m}, \vec{a}])$, where $[\vec{u}, \vec{v}]$ denotes the vector obtained by concatenation of $\vec{u}$ and $\vec{v}$.

Fast algorithms for f-circulant matrices then provide algorithms, by reduction, for accumulation with triangular and square Toeplitz matrices first, as sums of f-circulant in Algorithms 50 and 52, and then for any Toeplitz matrix, again as sums of triangular Toeplitz matrices in Algorithm 54.

Algorithm 50 In-place accumulating Upp. Triang. Toeplitz m-v. mult.

Inputs: $\vec{a}, \vec{b}, \vec{c} \in \mathbb{F}^m$.
Result: $\vec{c} \mathrel{+}= \mathcal{T}([\vec{0}, \vec{a}]) \cdot \vec{b}$.

```
$\vec{c} \mathrel{+}= \mathscr{C}_0(\vec{a}) \cdot \vec{b}$.                                   {Algorithm 45}
```

Remark 51. *Similarly to Algorithm 45, we can design an algorithm for accumulating lower triangular Toeplitz matrix-vector product. This algorithm is automatically obtained from Algorithm 45 by reversing the indices in the matrix and vectors. More generally, given an algorithm that computes* *a matrix-vector product $\vec{c} = A \cdot \vec{b}$, the* reversed *algorithm obtained by reversing all indices computes the reversed vector $\overleftarrow{c} = A^\intercal \cdot \overleftarrow{b}$.*

In particular, we here present f-circulant matrices where the coefficient acts on the lower left part of the matrix (excluding the diagonal). Algorithms can be reversed to deal with the other type of f-circulant matrices, where the coefficient would on the upper right part of the matrix (excluding the diagonal).

Algorithm 52 In-place accumulating square Toeplitz m-v. mult.

```
Inputs: $\vec{a_1} \in \mathbb{F}^m$, $\vec{a_2}, \vec{b}, \vec{c} \in \mathbb{F}^{m+1}$,
Result: $\vec{c} \mathrel{+}= \mathcal{T}([\vec{a_1}, \vec{a_2}]) \cdot \vec{b}$.
Let $\vec{b_1} = \vec{b}_{1..m}$ and $\vec{c_2} = \vec{c}_{2..m+1}$;
$\overleftarrow{c_2} \mathrel{+}= \mathscr{C}_0(\overleftarrow{a_1}) \cdot \overleftarrow{b_1}$;    {Algorithm 45 (reversed)}
$\vec{c} \mathrel{+}= \mathscr{C}_0(\vec{a_2}) \cdot \vec{b}$;    {Algorithm 45}
```

Lemma 53. *Algorithms 50 and 52 are correct and have complexity bounded by $\mathcal{O}(\mathfrak{M}(n))$.*

Proof. The complexity bound comes from that of Algorithm 45. Correctness is obtained directly looking at the values of the matrices. First, for Algorithm 50, we have that $\mathcal{T}([\vec{0}, \vec{a}]) = \mathscr{C}_0(\vec{a})$. Second, for Algorithm 52, $\mathcal{T}([\vec{a_1}, \vec{a_2}]) = \mathscr{C}_0(\vec{a_2}) + \begin{bmatrix} \vec{0}^\intercal & 0 \\ \mathscr{C}_0(\overleftarrow{a_1})^\intercal & \vec{0} \end{bmatrix}$. The computation $\mathscr{C}_0(\overleftarrow{a_1})^\intercal \cdot \vec{b_1}$ is the reversed algorithm of $\mathscr{C}_0\, \overleftarrow{a_1} \cdot \overleftarrow{b_1}$ as explained in Remark 51. □

From this, we give in Algorithm 54 an in-place rectangular Toeplitz matrix-vector multiplication.

Algorithm 54 In-place accumulating rectangular Toeplitz matrix-vector multiplication.

```
Inputs: $\vec{a} \in \mathbb{F}^{m+n-1}$, $\vec{b} \in \mathbb{F}^n$, $\vec{c} \in \mathbb{F}^m$,
Result: $\vec{c} \mathrel{+}= \mathcal{T}_{\mathrm{m,n}}(\vec{a}) \cdot \vec{b}$.
if $m = n$ then
    $\vec{c} \mathrel{+}= \mathcal{T}(\vec{a}) \cdot \vec{b}$;    {Algorithm 52}
else if $m > n$ then
    Let $c_1 = \vec{c}_{1..n}$ and $c_2 = \vec{c}_{(n+1)..m}$;
    $c_1 \mathrel{+}= \mathcal{T}(\vec{a}_{(m-n+1)..(m+n-1)}) \cdot \vec{b}$;    {Algorithm 52}
    $c_2 \mathrel{+}= \mathcal{T}_{\mathrm{m-n,n}}(\vec{a}_{1..(m-1)}) \cdot \vec{b}$;    {recursive call}
else
    Let $b_1 = \vec{b}_{1..m}$ and $b_2 = \vec{b}_{(m+1)..n}$;
    $c \mathrel{+}= \mathcal{T}(\vec{a}_{1..(2m-1)}) \cdot b_1$;    {Algorithm 52}
    $c \mathrel{+}= \mathcal{T}_{\mathrm{m,n-m}}(\vec{a}_{(m+1)..(m+n-1)}) \cdot b_2$;    {recursive call}
end if
```

Proposition 55. *Algorithm 54 is correct. Its complexity is bounded by $\mathcal{O}\left(\frac{\max\{m,n\}}{\min\{m,n\}}\mathfrak{M}(\min\{m,n\})\right)$ operations.*

Proof. If $m > n$, there are $\lfloor m/n \rfloor$ calls to Algorithm 52 in size n, requiring $\mathcal{O}((m/n)\mathfrak{M}(n)) \leq \mathcal{O}(\mathfrak{M}(m))$ operations. The remaining recursive call is then negligible. This is similar when $m < n$. □

Remark 56. *Algorithms 50, 52 and 54 and their reversed versions can be combined to compute in-place the accumulation $\vec{c} \mathrel{+}= T \cdot \vec{b}$ for any $\vec{c} \in \mathbb{F}^m$, $\vec{b} \in \mathbb{F}^n$, and band Toeplitz matrix $T = \mathcal{T}_{\text{m,n}}([\vec{0}_k, \vec{a}, \vec{0}_t])$ where $\vec{a} \in \mathbb{F}^\ell$ and $t = m + n - k - \ell - 1$. The cost is $\mathcal{O}\left(\frac{\max(m,n)}{\min(m,n)}\mathfrak{M}(\min(m,n))\right)$.*

The idea is to tile the matrix T with upper triangular, lower triangular, and rectangular Toeplitz matrices. Note that Algorithm 54 cannot be used directly on T since it requires the vectors $\vec{a_1}$ and $\vec{a_2}$ to be writable, while the zeroes of T are not stored.

6.3 Fast over-place Toeplitz matrix operations

These in-place accumulated Toeplitz matrix-vector multiplications allow us to obtain both over-place triangular Toeplitz multiplication and system solve, given in Algorithms 57 and 59. If $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$, then the obtained algorithm have the same asymptotic cost as they not-in-place counterparts. Otherwise, an extra logarithmic factor now appears in the complexity bounds.

Algorithm 57 Over-place triang. Toeplitz m-v. mult.

```
Inputs: $\vec{a}, \vec{b} \in \mathbb{F}^m$, s.t. $a_1 \in \mathbb{F}^*$.
Result: $\vec{b} \leftarrow \mathcal{T}([\vec{a}, \vec{0}]) \cdot \vec{b}$.
if $m \leq$ Threshold then                    {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place triang. m-v. mult.                    {Algorithm 4}
else
    Let $k = \lceil m/2 \rceil$, $b_1 = \vec{b}_{1..k}$ and $b_2 = \vec{b}_{(k+1)..m}$;
    $b_2 \leftarrow \mathcal{T}([\vec{a}_{(k+1)..m}, \vec{0}]) \cdot b_2$;                    {recursive call}
    $b_2 \mathrel{+}= \mathcal{T}_{\text{m-k,k}}([a_1, \ldots, a_{m-1}]) \cdot b_1$;                    {Algorithm 54}
    $b_1 \leftarrow \mathcal{T}([a_{(m-k+1)..m}, \vec{0}]) \cdot b_1$;                    {recursive call}
end if
```

Proposition 58. *Algorithm 57 is correct and its complexity his bounded by $\mathcal{O}(\mathfrak{M}(m)\log(m))$ operations, or $\mathcal{O}(\mathfrak{M}(m))$ if $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$.*

Proof. For the correctness, let $T = \mathcal{T}([\vec{a}, \vec{0}])$ and consider it as blocks $T_1 = \mathcal{T}([a_{(m-k+1)..m}, \vec{0}])$, $T_2 = \mathcal{T}([\vec{a}_{(k+1)..m}, \vec{0}])$ and $G = \mathcal{T}_{\text{m-k,k}}([a_1, \ldots, a_{m-1}])$. Then $T = \left[\begin{smallmatrix} T_1 & 0 \\ G & T_2 \end{smallmatrix}\right]$. Thus $T\vec{b} = \left[\begin{smallmatrix} T_1 b_1 \\ G b_1 + T_2 b_2 \end{smallmatrix}\right]$. Let $\bar{b}_1 = T_1 b_1$, $\hat{b}_2 = T_2 b_2$ and $\bar{b}_2 = G b_1 + T_2 b_2$. Then $\bar{b}_2 = \hat{b}_2 + G b_1$ and the algorithm is correct. Now for the complexity bound, the cost function is $T(m) \leq 2T(m/2) + \mathcal{O}(\mathfrak{M}(m))$, that is $\mathcal{O}(\mathfrak{M}(m)\log(m))$, or $\mathcal{O}(\mathfrak{M}(m))$ if $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$. □

Proposition 60. *Algorithm 59 is correct and has its complexity bounded by $\mathcal{O}(\mathfrak{M}(m)\log(m))$ operations, or $\mathcal{O}(\mathfrak{M}(m))$ if $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$.*

Proof. First, let $T = \mathcal{T}([\vec{0}, \vec{a}])$ and consider it as blocks $T_1 = \mathcal{T}([\vec{0}, \vec{a}_{1..k}])$, $T_2 = \mathcal{T}([\vec{0}, \vec{a}_{1..(m-k)}])$ and $G = \mathcal{T}_{\text{k,m-k}}(\vec{a}_{2..m})$. Then $T = \left[\begin{smallmatrix} T_1 & G \\ 0 & T_2 \end{smallmatrix}\right]$. Now define H, s.t. $T^{-1} = \left[\begin{smallmatrix} T_1^{-1} & H \\ 0 & T_2^{-1} \end{smallmatrix}\right]$. Then H satisfies $T_1^{-1}G + HT_2 = 0$. Also, we have $T^{-1}\vec{b} = \left[\begin{smallmatrix} T_1^{-1}b_1 + Hb_2 & T_2^{-1}b_2 \end{smallmatrix}\right]^\intercal$. Let $\bar{b}_2 = T_2^{-1}b_2$ and $\bar{b}_1 = T_1^{-1}b_1 + Hb_2$. Then $\bar{b}_1 = T_1^{-1}b_1 + HT_2\bar{b}_2 = T_1^{-1}b_1 - T_1^{-1}G\bar{b}_2 = T_1^{-1}\left(b_1 - G\bar{b}_2\right)$ and this shows that the algorithm is correct. Now for the complexity bound, the cost function is $T(m) \leq 2T(m/2) + \mathcal{O}(\mathfrak{M}(m))$, that is $T(m) = \mathcal{O}(\mathfrak{M}(m)\log(m))$, or $\mathcal{O}(\mathfrak{M}(m))$ if $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$. □

Algorithm 59 Over-place triang. Toeplitz system solve.

```
Inputs: $\vec{a}, \vec{b} \in \mathbb{F}^m$, s.t. $a_1 \in \mathbb{F}^*$.
Result: $\vec{b} \leftarrow \mathcal{T}([\vec{0}, \vec{a}])^{-1} \cdot \vec{b}$.
if $m \leq$ Threshold then                      {constant-time if Threshold $\in \mathcal{O}(1)$}
    Apply the quadratic in-place triang. syst. solve.                       {Algorithm 4}
else
    Let $k = \lceil m/2 \rceil$, $b_1 = \vec{b}_{1..k}$ and $b_2 = \vec{b}_{(k+1)..m}$;
    $b_2 \leftarrow \mathcal{T}([\vec{0}, \vec{a}_{1..(m-k)}])^{-1} \cdot b_2$;              {recursive call}
    $b_1 \mathrel{-}= \mathcal{T}_{\text{k,m-k}}(\vec{a}_{2..m}) \cdot b_2$;                   {Algorithm 54}
    $b_1 \leftarrow \mathcal{T}([\vec{0}, \vec{a}_{1..k}])^{-1} \cdot b_1$;                  {recursive call}
end if
```

Note that by means of Remark 51, we also have over-place algorithms for *upper* triangular Toeplitz matrix-vector multiplication and for *lower* triangular Toeplitz system solving.

6.4 Fast in-place accumulating Toeplitz-like matrix-vector product

Structured matrices such as Toeplitz and circulant matrices can be generalized and unified with the notion of displacement rank [49]. In particular, a matrix $A \in \mathbb{F}^{m \times n}$ is said Toeplitz-like of displacement rank α if the matrix $\nabla_Z(A) = A - Z_m A Z_n{}^{\intercal}$ has rank α, where $Z_m \in \mathbb{F}^{m \times m}$ is defined by $(Z_m)_{i,i-1} = 1$ and $(Z_m)_{i,j} = 0$ if $j \neq i-1$. In such a case, A admits *generators* $G \in \mathbb{F}^{m \times \alpha}$ and $H \in \mathbb{F}^{n \times \alpha}$ such that $\nabla_Z(A) = GH^{\intercal}$.

We describe here an algorithm that takes as inputs the generators G, H of a Toeplitz-like matrix A and two vectors $\vec{b}$, $\vec{c}$, and computes the accumulation $\vec{c} \mathrel{+}= A\vec{b}$ in place. The tool for this is the so-called *ΣLU-formula*: $\nabla_Z(A) = GH^{\intercal}$ if and only if

$$A = \sum_{i=1}^{\alpha} L(\vec{g}_i) \cdot U(\vec{h}_i)$$

where $\vec{g}_i$ (resp. $\vec{h}_i$) is the ith column of G (resp. H), and $L(\vec{g}) = \mathcal{T}_{\text{m},\ell}([\bar{g}, \vec{0}])$ and $U(\vec{h}) = \mathcal{T}_{\ell,\text{n}}([\vec{0}, \vec{h}])$ for $\ell = \min(m, n)$.

Proposition 62. *Algorithm 61 is in-place, correct and requires*

$$\mathcal{O}\left(\left(\frac{\max(m,n)}{\min(m,n)} + \log \min(m,n)\right) \mathfrak{M}(\min(m,n))\right)$$

operations, or $\mathcal{O}(\frac{\max(m,n)}{\min(m,n)} \mathfrak{M}(\min(m,n)))$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.

Proof. To prove the correctness of the algorithm, we first consider the ΣLU-formula, rewritten as $A = \sum_{i=1}^{\alpha} L_i U_i$ where $L_i = \mathcal{T}_{\text{m},\ell}([\bar{g}_i, \vec{0}])$ and $U_i = \mathcal{T}_{\ell,\text{n}}([\vec{0}, \vec{h}_i])$. The goal is to compute $A \cdot \vec{b}$ as $\sum_i L_i U_i \vec{b}$ by first computing $U_i \vec{b}$ into $\vec{b}$, then accumulating $L_i \vec{b}$ into $\vec{c}$ and finally restoring $\vec{b}$. If the first non-zero entry of $\vec{h}_i$ has index k, the $k-1$ left columns of U_i are made of zeroes. Therefore, $L_i U_i \vec{b} = LU\vec{b}_{k..n}$ where $L = \mathcal{T}_{\text{m},\ell}([\bar{g}_i, \vec{0}])$ and $U = \mathcal{T}_{\ell,\text{n-k+1}}([\vec{0}, (\vec{h}_i)_{k..n}])$. Now if $\ell = m$, the matrix U is rectangular. We split it into $[U_1 | U_2]$ where U_1 is an upper triangular square Toeplitz matrix. Then $U \cdot \vec{b}_{k..n} = U_1 \vec{b}_1 + U_2 \vec{b}_2$. This can be computed in $\vec{b}_1$, and $\vec{b}_1$ can be then restored.

Algorithm 61 In-place accumulating Toeplitz-like m-v. mult.

```
Inputs: Generators $G \in \mathbb{F}^{m\times\alpha}$ and $H \in \mathbb{F}^{n\times\alpha}$ of a Toeplitz-like matrix $A \in \mathbb{F}^{m\times n}$, $\vec{b} \in \mathbb{F}^{n}$, $\vec{c} \in \mathbb{F}^{m}$,
Result: $\vec{c} \mathrel{+}= A \cdot \vec{b}$.
 1: for $i = 1$ to $\alpha$ do
 2:   Let $\vec{g}_i$, $\vec{h}_i$ be the $i$th columns of $G$ and $H$
 3:   Let $k$ be the index of the first non-zero entry of $\vec{h}_i$
 4:   Let $\ell = \min(m, n - k + 1)$
 5:   Let $L = \mathcal{T}_{\text{m},\ell}([\vec{g},\vec{0}])$, $U_1 = \mathcal{T}_{\ell,\ell}([\vec{0},(\vec{h}_i)_{k,k+\ell-1}])$ and $\vec{b}_1 = \vec{b}_{k..k+\ell-1}$
 6:   Let $U_2 = \mathcal{T}_{\ell,\text{n-k+1-}\ell}([\vec{0},(\vec{h}_i)_{k+\ell,n}])$ and $\vec{b}_2 = \vec{b}_{k+\ell..n}$      {only if $\ell = m$}
 7:   $\vec{b}_1 \leftarrow U_1 \cdot \vec{b}_1$                                         {Algorithm 57 (reversed)}
 8:   $\vec{b}_1 \mathrel{+}= U_2 \cdot \vec{b}_2$                                       {Algorithm 54, only if $\ell = m$}
 9:   $\vec{c} \mathrel{+}= L \cdot \vec{b}$                                            {Remark 56}
10:   $\vec{b}_1 \mathrel{-}= U_2 \cdot \vec{b}_2$                                       {undo Line 8, Algorithm 54}
11:   $\vec{b}_1 \leftarrow U_1^{-1} \cdot \vec{b}_1$                                    {undo Line 7, Algorithm 59}
12: end for
```

For the complexity, we consider that no $\vec{h}_i$ starts with a zero, and let $\ell = \min(m, n)$. The over-place triangular Toeplitz matrix computations (Algorithms 57 and 59) have cost $\mathcal{O}(\mathfrak{M}(\ell)\log(\ell))$ or $\mathcal{O}(\mathfrak{M}(\ell))$ if $\mathfrak{M}(\ell) = \Theta(\ell^{1+\epsilon})$ for some $\epsilon > 0$. If $\ell = m$, operations with U_2 and L have total cost $\mathcal{O}(\frac{n}{m}\mathfrak{M}(m))$. If $\ell = n$, U_2 is empty and the cost of $\vec{c} \mathrel{+}= L \cdot \vec{b}_1$ is $\mathcal{O}(\frac{m}{n}\mathfrak{M}(n))$. Altogether, we obtain the announced cost. □

7 Fast in-place modular remainder

We consider now the fast in-place (resp. over-place) computation of the Euclidean polynomial modular remainder $R = A \bmod B$ (resp. $A = A \bmod B$) with A and B of respective degrees $m + n$ and n. Standard algorithms for the remainder require $\mathcal{O}\big(\frac{m}{n}\mathfrak{M}(n)\big)$ arithmetic operations and, apart from that of A and B, at least $\mathcal{O}(m)$ extra memory [28] to store the whole quotient Q such that $A = BQ + R$ with $\deg R < \deg B$. We first show how to avoid the storage of the whole quotient (as was hinted in [28]), and propose an algorithm still using $\mathcal{O}\big(\frac{m}{n}\mathfrak{M}(n)\big)$ arithmetic operations but only n extra space.

Second, we combine this with the techniques of Sections 5 and 6 and use the input space of A or B for intermediate computations in order to derive in-place and over-place algorithms for the modular remainder using at most $\mathcal{O}(\mathfrak{M}(m)\log(m))$ arithmetic operations, or $\mathcal{O}(\mathfrak{M}(m))$ if $\mathfrak{M}(m) = \Theta(m^{1+\epsilon})$ for some $\epsilon > 0$. In practice, this means that the asymptotic arithmetic cost of not-in-place algorithms is preserved by our in-place versions, except when the not-in-place base case is in the FFT regime, for which we have an extra $\log n$ factor.

Our first step is to interpret the results of Sections 6.2 and 6.3 in terms of polynomial and power series operations.

Remark 63. *A linear-algebraic derivation of the results of this section is presented in the conference paper [21].*

7.1 Toeplitz computations as polynomial operations

In Sections 6.2 and 6.3, several in-place and over-place algorithms for Toeplitz matrix-vector operations have been described. We give here their interpretation in terms of polynomial or power series operations, in order to use them in our in place remainder algorithms.

We identify a polynomial $A \in \mathbb{F}[X]$ with its vector of coefficients $\vec{a} \in \mathbb{F}^n$ where $\deg(A) = n-1$. Then $A = \sum_{i=0}^{n-1} a_i X^i$. (Note that here vectors are indexed from 0 to $n-1$.)

Let $\vec{a}$, $\vec{b}$, $\vec{c} \in \mathbb{F}^n$ and A, B, $C \in \mathbb{F}[X]$ be their corresponding polynomials of degree $< n$. For the lower triangular Toeplitz matrix $L = \mathcal{T}([\overleftarrow{a}, \vec{0}])$, the matrix-vector product $\vec{c} \leftarrow L \cdot \vec{b}$ corresponds to the *short product* $C \leftarrow A \cdot B \bmod X^n$. This is the natural product for truncated power series known at precision n.

We shall need the *reversed* operation, sometimes called a *high short product*, as well as its inverse. For a polynomial A, let $\overleftarrow{A} = X^{\deg(A)} A(1/X)$ be its reversed polynomial.

Definition 64. *Let $\vec{a}$, $\vec{b}$, $\vec{c} \in \mathbb{F}^n$ and A, B, $C \in \mathbb{F}[X]$ their corresponding polynomials of degree $< n$, and assume that a_{n-1} is non-zero. Let $U_{\overleftarrow{a}} = \mathcal{T}([\vec{0}, \overleftarrow{a}])$ be an upper triangular Toeplitz matrix.*

- *The* reversed short product *of A and B, denoted $A \overleftarrow{\mathrm{mul}} B$, is the polynomial C of degree $< n$ defined by one of the following equivalent equations: $C = A \cdot B \operatorname{div} X^{n-1}$, $\overleftarrow{C} = \overleftarrow{A} \cdot \overleftarrow{B} \bmod X^n$, or $\vec{c} = U_{\overleftarrow{a}} \cdot \vec{b}$.*

- *The* reversed power series division *of A and B, denoted $A \overleftarrow{\mathrm{div}} B$, is the polynomial C of degree $< n$ defined by one of the following equivalent equations: $\overleftarrow{C} = \overleftarrow{A}/\overleftarrow{B} \bmod X^n$, or $\vec{c} = U_{\overleftarrow{a}}^{-1} \cdot \vec{b}$.*

Let now $\vec{a} \in \mathbb{F}^{m+n-1}$, $\vec{b} \in \mathbb{F}^n$ and $\vec{c} \in \mathbb{F}^m$ and let $T = \mathcal{T}([\overleftarrow{a}])$ be a rectangular Toeplitz matrix. The matrix-vector product $\vec{c} \leftarrow T \cdot \vec{b}$ corresponds to the *middle product* $C \leftarrow (A \cdot B) \bmod X^{m+n-1} \operatorname{div} X^n$. In particular, when $m = n$ this is the standard middle product of polynomials [35].

Proposition 65. *Let A, B, $C \in \mathbb{F}[X]$ of degrees $< n$. Then*

(i) Algorithm 50 computes $C \mathrel{+}= B \overleftarrow{\mathrm{mul}} A$ in-place in $\mathcal{O}(\mathfrak{M}(n))$ operations;

(ii) Algorithm 57 computes $B \leftarrow B \cdot A \bmod X^n$ over-place in $\mathcal{O}(\mathfrak{M}(n) \log n)$ operations, or $\mathfrak{M}(n)$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$;

(iii) Algorithm 59 computes $B \leftarrow B \overleftarrow{\mathrm{div}} A$ over-place in $\mathcal{O}(\mathfrak{M}(n) \log n)$ operations, or $\mathfrak{M}(n)$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.

(iv) If $\deg(A) = m+n-2$, $\deg(B) = n-1$ and $\deg(C) = m-1$, Algorithms 52 and 54 compute $C \mathrel{+}= (A \cdot B) \bmod X^{m+n-1} \operatorname{div} X^n$ in $\mathcal{O}(\frac{\mu}{\nu}\mathfrak{M}(\nu))$ operations where $\mu = \max(m,n)$ and $\nu = \min(m,n)$.

Remark 66. *By Remark 51, the reversed version of Algorithm 57 computes $B \leftarrow B \overleftarrow{\mathrm{mul}} A$ in place, in the same complexity. By Remark 56, Algorithm 57 also works in place in the same complexity if $\deg(A) < n-1$.*

7.2 Small-space remainder by overwriting the quotient

The starting point of our derivation is the standard long division algorithm applied block by block, when A is large compared to B. Let $n = \deg(B)$ and write $A = \sum_{i=0}^{k-1} A_i \cdot X^{ni}$ where each A_i has degree at most $n-1$. The long division algorithm recalled in Algorithm 67 computes the Euclidean division of A by B as follows.

Algorithm 67 Long division of polynomials.

```
Inputs: $A$, $B$, $Q$, $R$ in $\mathbb{F}[X]$ of respective degrees $m+n$, $n$, $m$ and $n-1$.
Read-only: $A$, $B$.
Result: $Q = A \operatorname{div} B$ and $R = A \bmod B$.
1: Let $k = \lfloor \frac{m}{n} \rfloor$ and write $A = \sum_{i=0}^{k-1} A_i X^{ni}$        {polynomials of deg. $n-1$}
2: $R \leftarrow A_{k-1}$
3: $Q \leftarrow 0$
4: for $i = k-2$ down-to $0$ do
5:     $R \leftarrow RX^n + A_i$
6:     $T \leftarrow R \operatorname{div} B$
7:     $R \mathrel{-}= BT$
8:     $Q \leftarrow X^n Q + T$
9: end for
```

The first remark is that to compute the remainder, one does not need to retain Q but only T. The second aspect is to dive into the quotient computation $R \operatorname{div} B$. The fast algorithm consists in a power series division. More precisely, at Line 6, $\overleftarrow{T} = \overleftarrow{R}/\overleftarrow{B} \bmod X^n$ where the division is a (truncated) power series division (*cf.* for instance [26, Chapter 9]). Since the computation is at precision n, only the first n coefficients of $\overleftarrow{R}$ are needed. That is, the update at Line 5 can be postponed and merged with Line 7, and Line 6 becomes a (reversed) power series division $T = R \mathbin{\overleftarrow{\operatorname{div}}} B$. Line 7 becomes $R = RX^n + A_i - BT$ and since the result of this computation has degree $< n$, the computation can be performed modulo X^n. It becomes $R = A_i - (BT \bmod X^n)$. Altogether, we obtain Algorithm 68.

Theorem 69. *Algorithm 68 is correct and requires $\mathcal{O}\big(\frac{m}{n}\mathfrak{M}(n)\big)$ arithmetic operations and n extra memory space.*

Proof. Correctness follows from the previous discussion. For the complexity bounds, (reversed) power series division can be computed in place in $\mathcal{O}(\mathfrak{M}(n))$ ring operations if the dividend can be erased [28, Corollary 2.6]. Here R can be safely erased. Then the short product can also be computed in-place in $O(\mathfrak{M}(n))$ ring operations [29, Theorem 5.1]. Altogether, the time complexity is $O(k\mathfrak{M}(n)) = O(\frac{m}{n}\mathfrak{M}(n))$. The only required extra space is the degree-$(n-1)$ polynomial T. □

Remark 70. *For a polynomial B, denote by $\overleftarrow{\operatorname{div}}_B$ the operator defined by $\overleftarrow{\operatorname{div}}_B(A) = A \mathbin{\overleftarrow{\operatorname{div}}} B$, and mul_B the operator defined by $\operatorname{mul}_B(A) = A \cdot B \bmod X^{\deg(B)}$. Then Algorithm 68 gives rise to a formula for the remainder, namely*

$$A \bmod B = \sum_{i=0}^{k-1} \Big(\operatorname{mul}_{B_*} \circ \overleftarrow{\operatorname{div}}_{B^*} \Big)^i (A_i)$$

Algorithm 68 Overwritten-quotient Euclidean remainder.

```
Inputs: $A$, $B$, $R$ in $\mathbb{F}[X]$ of respective degrees $m+n$, $n$ and $n-1$.
Read-only: $A$, $B$.
Result: $R = A \bmod B$.
 1: if $m < 0$ then
 2:     $R \leftarrow A$;
 3: else
 4:     Let $k = \lfloor \frac{m}{n} \rfloor$ and write $A = \sum_{i=0}^{k-1} A_i X^{ni}$        {polynomials of deg. $n-1$}
 5:     Let $B^* = B \operatorname{div} X$ and $B_* = B \bmod X^n$
 6:     $R \leftarrow A_{k-1}$
 7:     for $i = k-2$ down-to $0$ do
 8:         $T \leftarrow R \overleftarrow{\operatorname{div}} B^*$;        {rev. power series division}
 9:         $R \leftarrow -B_* T \bmod X^n$;        {short product}
10:         $R \mathrel{+}= A_i$;
11:     end for
12: end if
```

where the exponent notation denotes the ith iterate of a function. This formula is the exact counterpart of the linear-algebraic formula derived in [21, Eq. (11)].

7.3 In-place remainders via Toeplitz techniques

We now derive algorithms that use only $\mathcal{O}(1)$ extra memory space in the in-place model of Section 1.1: Modifying the inputs is possible if and only if all inputs are restored to their initial state after the completion of the algorithm. This allows us to store some intermediate results, overwriting the inputs, provided that we can afterwards recompute the initial inputs in their entirety. Further, this enables recursive calls, as intermediate values are used but restored along the recursive descent. The general idea is then to use variants of Algorithm 68 based on results from Sections 5 and 6.

We present three variants of in-place polynomial remaindering:

- IPER: given A and B, it computes in-place $R = A \bmod B$ using only the output space for R and that of the modulus B (*i.e.*, A is read-only and B is restored to its initial state after completion);
- OPER: given A and B, it computes both $Q = A \operatorname{div} B$ and $R = A \bmod B$ (such that $A = BQ + R$), replacing A by $\langle Q, R \rangle$ (with B restored after completion);
- APER: given A, B and R, it computes $R \mathrel{+}= A \bmod B$, accumulating the remainder into A (with both A and B restored after completion).

We present IPER in Algorithm 71: this variant replaces only Lines 8 to 10 of Algorithm 68 by their over-place variants, Algorithms 57 and 59, that modify and restore parts of B.

Theorem 72. *Algorithm 71 is correct, in-place and has complexity bounded by $\mathcal{O}(\frac{m}{n}\mathfrak{M}(n)\log(n))$ operations, or $\mathcal{O}(\frac{m}{n}\mathfrak{M}(n))$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.*

Algorithm 71 IPER (R, A, B): In-place Polynomial Euclidean Remainder.

```
Inputs: $A$, $B$, $R$ in $\mathbb{F}[X]$ of respective degrees $m+n$, $n$ and $n-1$.
Read-only: $A$.
Result: $R = A \bmod B$.
if $m < 0$ then
    $R \leftarrow A$;
else
    Let $k = \lfloor \frac{m}{n} \rfloor$ and write $A = \sum_{i=0}^{k-1} A_i X^{ni}$        {polynomials of deg. $n-1$}
    Let $B^* = B \operatorname{div} X$ and $B_* = B \bmod X^n$
    $R \leftarrow A_{k-1}$;
    for $i = k-2$ down-to $0$ do
        $R \leftarrow R \overleftarrow{\operatorname{div}} B^*$;        {Proposition 65 (iii)}
        $R \leftarrow -B_* R \bmod X^n$;        {Proposition 65 (ii)}
        $R$ += $A_i$;
    end for
end if
```

Proof. Algorithm 71 calls $\mathcal{O}(k) = \mathcal{O}(\frac{m}{n})$ times Algorithms 57 and 59. And each call requires $\mathcal{O}(\mathfrak{M}(n)\log(n))$ operations, or $\mathcal{O}(\mathfrak{M}(n))$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$, by Propositions 58 and 60. □

7.4 Over-place and accumulating remainders

We give two variants of Algorithm 71. In OPER (Algorithm 73), A is replaced by $[A \operatorname{div} B, A \bmod B]$ while B is ultimately restored. Algorithm APER (Algorithm 74) computes R += $A \bmod B$.

Algorithm 73 OPER(A, B): Over-place Polynomial Euclidean Quotient and Remainder.

```
Inputs: $A$, $B$ in $\mathbb{F}[X]$ of respective degrees $m+n$ and $n$.
Result: $A = [A \operatorname{div} B, A \bmod B]$ of degrees $m$ and at most $n-1$.
if $m \geq 0$ then
    Let $k = \lfloor \frac{m}{n} \rfloor$ and write $A = \sum_{i=0}^{k-1} A_i X^{ni}$        {polynomials of deg. $n-1$}
    Let $B^* = B \operatorname{div} X$ and $B_* = B \bmod X^n$
    Let $s = m \bmod n$ be the degree of $A_{k-1}$ and $B_1 = B \operatorname{div} X^{n-s-1}$
    $A_{k-1} \leftarrow A_{k-1} \overleftarrow{\operatorname{div}} B_1$        {Proposition 65 (iii)}
    $A_{k-2}$ −= $B_* A_{k-1} \bmod X^n$        {Remark 66}
    for $i = k-2$ down-to $1$ do
        $A_i \leftarrow A_i \overleftarrow{\operatorname{div}} B^*$        {Proposition 65 (iii)}
        $A_{i-1}$ −= $B_* A_i \bmod X^n$        {Proposition 65 (ii)}
    end for
end if
```

The idea of OPER is to compute the remainder progressively in the blocks of A, making use of the *over-place* algorithm for reversed power series division and the in-place accumulated short product. A nice property of OPER is that it is reversible since each operation is reversible. The operation A_{i-1} −= $B_* A_i \bmod X^n$ is undone by A_{i-1} += $B_* A_i \bmod X^n$ and $A_i \leftarrow A_i \overleftarrow{\operatorname{div}} B^*$ by

$A_i \leftarrow A_i \overleftarrow{\mathrm{mul}}\, B^*$. Performing the reverses in reversed order recovers A. We denote by OPER^{-1} this recovery.

With this, APER is now just the application of OPER and OPER^{-1} interleaved with an update of the remainder. This is shown in Algorithm 74.

Algorithm 74 $\mathrm{APER}(R, A, B)$: Accumulated in-place Polynomial Euclidean Remainder.

```
Inputs: $R$, $A$, $B$ in $\mathbb{F}[X]$ of resp. degrees $n-1$, $m+n$ and $n$.
Result: $R \mathrel{+}= A \bmod B$ of degree at most $n-1$.
1: if $m \geq 0$ then $\mathrm{OPER}(A, B)$; end if          {Algorithm 73}
2: $R \mathrel{+}= A \bmod X^n$;                             {contains the remainder}
3: if $m \geq 0$ then $\mathrm{OPER}^{-1}(A, B)$; end if     {undo Line 1}
```

Theorem 75. *Algorithms 73 and 74 are correct, in-place, and their complexity is bounded by $\mathcal{O}(\frac{m}{n}\mathfrak{M}(n)\log(n))$ operations, or $\mathcal{O}(\mathfrak{M}(n))$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.*

7.5 Fast in-place modular multiplication

The goal is to compute $R \mathrel{+}= AC \bmod B$ where $\deg(R), \deg(A), \deg(C) < n$ and $\deg(B) = n$. This corresponds to multiplication in polynomial extensions of finite fields for instance. Let $D = AC$ of degree $2n-2$. Write $D = D_1X^n + D_0$ where $\deg(D_1) = n-2$ and $\deg(D_0) < n$. Then $D \bmod B = D_0 + (X^nD_1 \bmod B)$. To compute $X^nD_1 \bmod B$, we first compute the degree-$(n-2)$ quotient as $Q = D_1 \overleftarrow{\mathrm{div}}\, B^*$ where $B^* = B \operatorname{div} X^2$. Then we get $X^nD_1 \bmod B$ as $(X^nD_1 - Q \cdot B) \bmod X^n = -(Q \cdot B) \bmod X^n$. Since D is known only as $A \cdot C$, we first compute D_1 into C as $D_1 = C^* \overleftarrow{\mathrm{mul}}\, A^*$ where $A^* = A \operatorname{div} X$ and $C^* = C \operatorname{div} X$ and undo the computation afterwards.

Algorithm 76 $\mathrm{AXPYIN}(R, A, C, B)$: Accumulated in-place modular multiplication.

```
Inputs: $R$, $A$, $C$, $B$ in $\mathbb{F}[X]$ of resp. degrees $n-1$, $n-1$, $n-1$ and $n$
Result: $R \mathrel{+}= AC \bmod B$ of degree at most $n-1$.
1: Let $B_* = B \bmod X^n$ and $B^* = B \operatorname{div} X^2$
2: Let $A^* = A \bmod X$ and $C^* = C \bmod X$
3: $C^* \leftarrow C^* \overleftarrow{\mathrm{mul}}\, A^*$      {compute $D_1$ into $C^*$ by Proposition 65 (i)}
4: $C^* \leftarrow C^* \overleftarrow{\mathrm{div}}\, B^*$      {compute $Q$ into $C^*$ by Proposition 65 (iii)}
5: $R \mathrel{-}= B_* \cdot C^* \bmod X^n$                     {subtract $BQ \bmod X^n$ using Algorithm 42}
6: $C^* \leftarrow C^* \overleftarrow{\mathrm{mul}}\, B^*$      {undo Line 4}
7: $C^* \leftarrow C^* \overleftarrow{\mathrm{div}}\, A^*$      {undo Line 3}
8: $R \mathrel{+}= (A \cdot C) \bmod X^n$                       {add $D_0$ using Algorithm 42}
```

Proposition 77. *Algorithm 76 is correct and performs $\mathcal{O}(\mathfrak{M}((n)\log(n))$ operations, or $\mathcal{O}(\mathfrak{M}(n))$ only if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.*

Proof. The correctness is justified by the paragraph preceding the algorithm. The complexity is given by Proposition 65. □

Combining this algorithm with OPER let us perform any accumulating modular multiplication.

Algorithm 78 Accumulated in-place modular multiplication.

```
Inputs: $R$, $A$, $C$, $B$ in $\mathbb{F}[X]$ of resp. degrees $n-1$, $\ell$, $m$ and $n$
Result: $R \mathrel{+}= AC \bmod B$ of degree at most $n-1$.
OPER$(A, B)$                         {$A \bmod B$ in $A_0$ via Algorithm 73, only if $\ell \geq n$}
OPER$(C, B)$                         {$C \bmod B$ in $C_0$ via Algorithm 73, only if $m \geq n$}
AXPYIN$(R, A_0, C_0, B)$             {Algorithm 76}
OPER$^{-1}(C, B)$                    {recover $C$, only if $\ell \geq n$}
OPER$^{-1}(A, B)$                    {recover $A$, only if $m \geq n$}
```

Proposition 79. *Algorithm 78 is correct and performs $\mathcal{O}(\frac{\ell+m}{n}\mathfrak{M}(n)\log n))$ operations, or only $\mathcal{O}(\frac{\ell+m}{n}\mathfrak{M}(n))$ if $\mathfrak{M}(n) = \Theta(n^{1+\epsilon})$ for some $\epsilon > 0$.*

Proof. This is the combination of Theorem 75 and Proposition 77. □

8 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 algorithm compares favorably in practice with the standard not-in-place variants (Figure 1). We also extend the result to many in-place and over-place linear algebra routines, some of which are actually not bilinear.

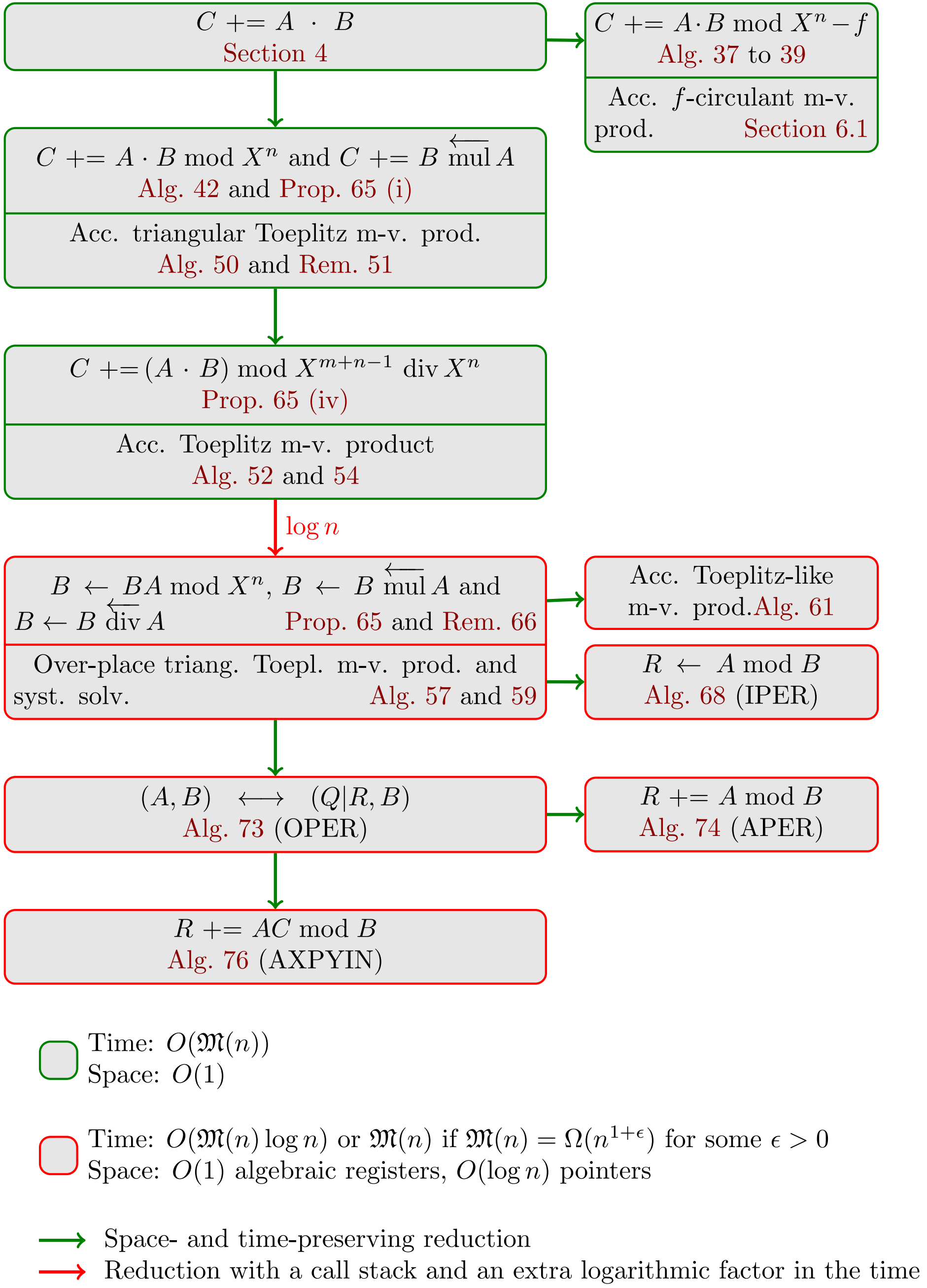
We apply the same technique to provide fast in-place accumulating polynomial multiplication algorithms. Our implementation of an in-place variant of Karatsuba polynomial multiplication has very close performance to that of the state-of-the-art library NTL (Figure 2). From these polynomial multiplication algorithms, we provide a series of reductions to get fast in-place accumulating algorithms for generalized convolutions, short product and power series division and remainder. We also get over-place variants, and in particular describe a fast algorithm that replaces the dividend by the quotient and remainder in the Euclidean division. These results are obtained through their equivalent representations as f-circulant or Toeplitz matrix-vector products, or system solving. The web of reductions is depicted on Figure 3.

We have here the first fast in-place, over-place and accumulating algorithms computing only the remainder of the polynomial Euclidean division. There, one open problem remains: to remove the extra logarithmic factor in the complexity that appears in this case, when $\mathfrak{M}(n)$ is not $\Omega(n^{1+\epsilon})$ for some $\epsilon > 0$.

Finally, Section 7.5 also shows a direct application of our techniques for the multiplication in a polynomial extension of a finite field.

Another possible improvement would be to study the behavior of these algorithms with floating point numbers. In this case our "undoing" approach might not restore the exact original states. It would then be necessary to study not only the accuracy of the obtained programs but also the effect on their inputs.

Figure 3: Main polynomial reductions in the paper.


$C \mathrel{+}= A \cdot B$
Section 4
$C \mathrel{+}= A \cdot B \bmod X^n - f$
Alg. 37 to 39
Acc. f-circulant m-v. prod. Section 6.1
$C \mathrel{+}= A \cdot B \bmod X^n$ and $C \mathrel{+}= B \overleftarrow{\mathrm{mul}}\, A$
Alg. 42 and Prop. 65 (i)
Acc. triangular Toeplitz m-v. prod.
Alg. 50 and Rem. 51
$C \mathrel{+}= (A \cdot B) \bmod X^{m+n-1} \operatorname{div} X^n$
Prop. 65 (iv)
Acc. Toeplitz m-v. product
Alg. 52 and 54
$\log n$
$B \leftarrow BA \bmod X^n$, $B \leftarrow B \overleftarrow{\mathrm{mul}}\, A$ and $B \leftarrow B \overleftarrow{\mathrm{div}}\, A$ Prop. 65 and Rem. 66
Over-place triang. Toepl. m-v. prod. and syst. solv. Alg. 57 and 59
Acc. Toeplitz-like m-v. prod.Alg. 61
$R \leftarrow A \bmod B$
Alg. 68 (IPER)
$(A, B) \longleftrightarrow (Q|R, B)$
Alg. 73 (OPER)
$R \mathrel{+}= A \bmod B$
Alg. 74 (APER)
$R \mathrel{+}= AC \bmod B$
Alg. 76 (AXPYIN)
Time: $O(\mathfrak{M}(n))$
Space: $O(1)$
Time: $O(\mathfrak{M}(n) \log n)$ or $\mathfrak{M}(n)$ if $\mathfrak{M}(n) = \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$
Space: $O(1)$ algebraic registers, $O(\log n)$ pointers
Space- and time-preserving reduction
Reduction with a call stack and an extra logarithmic factor in the time

References

[1] Karl Abrahamson. “Time-Space Tradeoffs for Branching Programs Contrasted with Those for Straight-Line Programs”. *27th Annual Symposium on Foundations of Computer Science (Sfcs 1986)*. Oct. 1986, pp. 402–409. DOI: 10.1109/SFCS.1986.58.

[2] Josh Alman et al. “More Asymmetry Yields Faster Matrix Multiplication”. *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2025, pp. 2005–2039. DOI: 10.1137/1.9781611978322.63.

[3] Andrew Arnold. “A New Truncated Fourier Transform Algorithm”. *ISSAC'2013, Proceedings of the 2013 International Symposium on Symbolic and Algebraic Computation, Boston, USA*. Ed. by Manuel Kauers. New York: ACM Press, June 2013, pp. 15–22. DOI: 10.1145/2465506.2465957.

[4] Dario Bini and Victor Y. Pan. “Fast Parallel Polynomial Division via Reduction to Triangular Toeplitz Matrix Inversion and to Polynomial Inversion Modulo a Power”. *Inf. Process. Lett.* 21.2 (1985), pp. 79–81. DOI: 10.1016/0020-0190(85)90037-7. URL: https://doi.org/10.1016/0020-0190(85)90037-7.

[5] Dario Bini and Victor Y. Pan. *Polynomial and matrix computations, 1st Edition*. Vol. 12. Progress in theoretical computer science. Birkhäuser, 1994. ISBN: 3764337869. DOI: 10.1007/978-1-4612-0265-3.

[6] Marco Bodrato. “A Strassen-like matrix multiplication suited for squaring and higher power computation”. *ISSAC'2010, Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation, Munich, Germany*. Ed. by Wolfram Koepf. New York: ACM Press, July 2010, pp. 273–280. ISBN: 978-1-4503-0150-3. DOI: 10.1145/1837934.1837987.

[7] Marco Bodrato. “Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0”. *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*. Ed. by Claude Carlet and Berk Sunar. Vol. 4547. Lecture Notes in Computer Science. Springer, 2007, pp. 116–133. DOI: 10.1007/978-3-540-73074-3_10. URL: https://doi.org/10.1007/978-3-540-73074-3%5C_10.

[8] Marco Bodrato and Alberto Zanoni. “Integer and polynomial multiplication: towards optimal toom-cook matrices”. *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*. ISSAC '07. Waterloo, Ontario, Canada: Association for Computing Machinery, 2007, pp. 17–24. ISBN: 9781595937438. DOI: 10.1145/1277548.1277552. URL: https://doi.org/10.1145/1277548.1277552.

[9] Brice Boyer et al. “Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm”. *ISSAC'2009, Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, Seoul, Korea*. Ed. by John P. May. New York: ACM Press, July 2009, pp. 135–143. DOI: 10.1145/1576702.1576713.

[10] Richard P. Brent and Paul Zimmermann. *Modern computer arithmetic*. Vol. 18. Cambridge monographs on applied and computational mathematics. Cambridge, UK: Cambridge University Press, 2011. ISBN: 0-521-19469-5 (hardcover). DOI: 10.1017/CBO9780511921698.

[11] Nader H. Bshouty. “On the additive complexity of 2 × 2 matrix multiplication”. *Information Processing Letters* 56.6 (Dec. 1995), pp. 329–335. DOI: 10.1016/0020-0190(95)00176-X.

[12] Harry Buhrman et al. "Computing with a full memory: catalytic space". *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*. Ed. by David B. Shmoys. ACM, 2014, pp. 857–866. DOI: 10.1145/2591796.2591874.

[13] David G. Cantor and Erich Kaltofen. "On Fast Multiplication of Polynomials over Arbitrary Algebras". *Acta Informatica* 28.7 (1991), pp. 693–701. DOI: 10.1007/BF01178683. URL: https://doi.org/10.1007/BF01178683.

[14] Shaoshi Chen, ed. *ISSAC'2024, Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, Raleigh, NC-USA*. Raleigh, NC, USA: ACM Press, July 2024.

[15] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". *Mathematics of computation* 19.90 (1965), pp. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

[16] Nicholas Coxon. "An in-place truncated Fourier transform". *Journal of Symbolic Computation* 110 (2022), pp. 66–80. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2021.10.002.

[17] Craig C. Douglas et al. "GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm". en. *Journal of Computational Physics* 110.1 (Jan. 1994), pp. 1–10. ISSN: 00219991. DOI: 10.1006/jcph.1994.1001. (Visited on 01/16/2025).

[18] J-G. Dumas et al. *PLinOpt, a collection of C++ routines handling linear & bilinear programs.* 8.76 kSLOC. v3.2, 209c1c0, Jan. 2024. URL: %7Bhttps://github.com/jgdumas/plinopt%7D.

[19] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. "Dense Linear Algebra over Prime Fields". *ACM Transactions on Mathematical Software* 35.3 (Nov. 2008), pp. 1–42. DOI: 10.1145/1391989.1391992. URL: http://hal.archives-ouvertes.fr/hal-00018223.

[20] Jean-Guillaume Dumas and Bruno Grenet. "In-place accumulation of fast multiplication formulae". *ISSAC'2024, Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, Raleigh, NC-USA*. Ed. by Shaoshi Chen. Raleigh, NC, USA: ACM Press, July 2024, pp. 16–25. DOI: 10.1145/3666000.3669671. URL: https://hal.science/hal-04167499.

[21] Jean-Guillaume Dumas and Bruno Grenet. "In-place fast polynomial modular remainder". *ISSAC'2024, Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, Raleigh, NC-USA*. Ed. by Shaoshi Chen. Raleigh, NC, USA: ACM Press, July 2024, pp. 26–35. DOI: 10.1145/3666000.3669672. URL: https://hal.science/hal-03979016.

[22] Jean-Guillaume Dumas and Clément Pernet. "Symmetric indefinite elimination revealing the rank profile matrix". *ISSAC'2018, Proceedings of the 2018 International Symposium on Symbolic and Algebraic Computation, New York, USA*. Ed. by Carlos Arreche. New York, USA: ACM Press, July 2018, pp. 151–158. DOI: 10.1145/3208976.3209019. URL: https://hal.archives-ouvertes.fr/hal-01704793.

[23] Jean-Guillaume Dumas, Clément Pernet, and Alexandre Sedoglavic. "Some fast algorithms multiplying a matrix by its adjoint". *Journal of Symbolic Computation* 115 (Mar. 2023), pp. 285–315. ISSN: 0747-7171. DOI: 10.1016/j.jsc.2022.08.009.

[24] Jean-Guillaume Dumas, Clément Pernet, and Alexandre Sedoglavic. "Strassen's algorithm is not optimally accurate". *ISSAC'2024, Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, Raleigh, NC-USA*. Ed. by Shaoshi Chen. Raleigh, NC, USA: ACM Press, July 2024, pp. 254–263. DOI: 10.1145/3666000.3669697. URL: https://hal.science/hal-04441653.

[25] Jean-Guillaume Dumas, Clément Pernet, and Ziad Sultan. "Simultaneous computation of the row and column rank profiles". *ISSAC'2013, Proceedings of the 2013 International Symposium on Symbolic and Algebraic Computation, Boston, USA*. Ed. by Manuel Kauers. New York: ACM Press, June 2013, pp. 181–188. DOI: 10.1145/2465506.2465517. URL: http://hal.archives-ouvertes.fr/hal-00778136.

[26] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. 3rd. Cambridge University Press, 2013. ISBN: 978-1-107-03903-2. DOI: 10.1017/CBO9781139856065.

[27] Pascal Giorgi. *Efficient algorithms and implementation in exact linear algebra*. Habilitation thesis, University of Montpellier, France. 2019. URL: https://tel.archives-ouvertes.fr/tel-02360023.

[28] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. "Fast in-place algorithms for polynomial operations: division, evaluation, interpolation". *ISSAC'2020, Proceedings of the 2020 International Symposium on Symbolic and Algebraic Computation, Kalamata, Greece*. Ed. by Ioannis Z. Emiris, Lihong Zhi, and Anton Leykin. Kalamata, Greece: ACM Press, July 2020, pp. 210–217. DOI: 10.1145/3373207.3404061. URL: https://doi.org/10.1145/3373207.3404061.

[29] Pascal Giorgi, Bruno Grenet, and Daniel S. Roche. "Generic Reductions for In-place Polynomial Multiplication". *ISSAC'2019, Proceedings of the 2019 International Symposium on Symbolic and Algebraic Computation, Beijing, China*. Ed. by James H. Davenport et al. Beijing, China: ACM Press, July 2019, pp. 187–194. DOI: 10.1145/3326229.3326249. URL: https://doi.org/10.1145/3326229.3326249.

[30] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008. ISBN: 9780521884730. DOI: 10.1017/CBO9780511804106.

[31] Oded Goldreich. "On the Cook-Mertz Tree Evaluation Procedure". *Electronic Colloquium on Computational Complexity* TR24-109 (2024). URL: https://eccc.weizmann.ac.il/report/2024/109.

[32] Gene H. Golub and Charles F. van Loan. *Matrix computations*. third. Johns Hopkins Studies in the Mathematical Sciences. Baltimore, MD, USA: The Johns Hopkins University Press, 1996. ISBN: 0-8018-5413-X, 0-8018-5414-8. DOI: 10.56021/9781421407944.

[33] Bruno Grenet and Ilya Volkovich. "One (more) line on the most Ancient Algorithm in History". *Symposium on Simplicity in Algorithms (SOSA)*. 2020, pp. 15–17. DOI: 10.1137/1.9781611976014.3.

[34] Hans-Friedich de Groote. "On varieties of optimal algorithms for the computation of bilinear mappings II. Optimal algorithms for 2 × 2-matrix multiplication". *Theoretical Computer Science* 7.2 (1978), pp. 127–148. DOI: 10.1016/0304-3975(78)90045-2.

[35] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. "The Middle Product Algorithm I". *Applicable Algebra in Engineering, Communication and Computing* 14.6 (Feb. 2004), pp. 415–438. ISSN: 0938-1279, 1432-0622. DOI: 10.1007/s00200-003-0144-2. (Visited on 10/06/2016).

[36] David Harvey and Joris van der Hoeven. "Polynomial Multiplication over Finite Fields in Time O(nlog n)". *Journal of the ACM* 69.2 (2022), 12:1–12:40. DOI: 10.1145/3505584. URL: https://doi.org/10.1145/3505584.

[37] David Harvey and Daniel S. Roche. "An In-Place Truncated Fourier Transform and Applications to Polynomial Multiplication". *ISSAC'2010, Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation, Munich, Germany*. Ed. by Wolfram Koepf. New York: ACM Press, July 2010, pp. 325–329. ISBN: 978-1-4503-0150-3. DOI: 10.1145/1837934.1837996.

[38] Joris van der Hoeven. "The Truncated Fourier Transform and Applications". *ISSAC'2004, Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*. Ed. by Jaime Gutierrez. New York: ACM Press, July 2004, pp. 290–296. DOI: 10.1145/1005285.1005327.

[39] John E. Hopcroft and Jean E. Musinski. "Duality Applied to the Complexity of Matrix Multiplication and Other Bilinear Forms". *SIAM J. Comput.* 2.3 (1973), pp. 159–173. DOI: 10.1137/0202013. URL: https://doi.org/10.1137/0202013.

[40] Steven Huss-Lederman et al. "Implementation of Strassen's Algorithm for Matrix Multiplication". *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*. Ed. by ACM. http://doi.acm.org/10.1145/369028.369096. New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: ACM Press and IEEE Computer Society Press, 1996, 32–es. ISBN: 0-89791-854-1.

[41] Michael Kaminski, David G. Kirkpatrick, and Nader H. Bshouty. "Addition Requirements for Matrix and Transposed Matrix Products". *Journal of Algorithms* 9.3 (1988), pp. 354–364. DOI: 10.1016/0196-6774(88)90026-0.

[42] Anatolii Karatsuba and Yuri P. Ofman. "Multiplication of multidigit numbers on automata". *Soviet physics doklady*. Vol. 7. 1963, pp. 595–596.

[43] Elaye Karstadt and Oded Schwartz. "Matrix Multiplication, a Little Faster". *J. ACM* 67.1 (2020), 1:1–1:31. DOI: 10.1145/3364504.

[44] Manuel Kauers, ed. *ISSAC'2013, Proceedings of the 2013 International Symposium on Symbolic and Algebraic Computation, Boston, USA*. New York: ACM Press, June 2013.

[45] Wolfram Koepf, ed. *ISSAC'2010, Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation, Munich, Germany*. New York: ACM Press, July 2010. ISBN: 978-1-4503-0150-3.

[46] Roman E. Maeder. "Storage Allocation for the Karatsuba Integer Multiplication Algorithm". *Design and Implementation of Symbolic Computation Systems*. Ed. by Alfonso Miola. Vol. 722. Berlin/Heidelberg: Springer-Verlag, 1993, pp. 59–65. ISBN: 978-3-540-57235-0. DOI: 10.1007/BFb0013168. (Visited on 07/16/2019).

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

[48] Michael Monagan. "In-place arithmetic for polynomials over Zn". *Design and Implementation of Symbolic Computation Systems*. Ed. by John Fitch. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 22–34. ISBN: 978-3-540-48031-0. DOI: 10.1007/3-540-57272-4_21.

[49] Victor Y. Pan. *Structured Matrices and Polynomials: Unified Superfast Algorithms*. Boston, MA, USA: Birkhäuser, 2001, pp. xxiv+278. ISBN: 0-8176-4240-4. DOI: 10.1007/978-1-4612-0129-8.

[50] 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.

[51] Daniel S. Roche. "Space-and Time-Efficient Polynomial Multiplication". *ISSAC'2009, Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, Seoul, Korea*. Ed. by John P. May. New York: ACM Press, July 2009, pp. 295–302. DOI: 10.1145/1576702.1576743.

[52] Arnold Schönhage and Volker Strassen. "Schnelle Multiplikation großer Zahlen". *Computing* 7.3 (1971), pp. 281–292. ISSN: 1436-5057. DOI: 10.1007/BF02242355. (Visited on 02/26/2019).

[53] Volker Strassen. "Gaussian elimination is not optimal". English. *Numerische Mathematik* 13 (1969), pp. 354–356. DOI: 10.1007/BF02165411.

[54] Emmanuel Thomé. *Karatsuba Multiplication With Temporary Space*. Sept. 2002. URL: https://hal.inria.fr/hal-02396734.

[55] Shmuel Winograd. "La complexité des calculs numériques". *La Recherche* 8 (1977), pp. 956–963.

[56] Shmuel Winograd. "On multiplication of 2×2 matrices". *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388. ISSN: 0024-3795. DOI: 10.1016/0024-3795(71)90009-7.