Fast in-place algorithms for polynomial operations: division, evaluation, interpolation

Pascal Giorgi, Bruno Grenet, Daniel S. Roche

Fast in-place algorithms for polynomial operations: division, evaluation, interpolation

Pascal Giorgi
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
pascal.giorgi@lirmm.fr

Bruno Grenet
LIRMM, Univ. Montpellier, CNRS
Montpellier, France
bruno.grenet@lirmm.fr

Daniel S. Roche
United States Naval Academy
Annapolis, Maryland, U.S.A.
roche@usna.edu

February 25, 2020

Abstract

We consider space-saving versions of several important operations on univariate polynomials, namely power series inversion and division, division with remainder, multi-point evaluation, and interpolation. Now-classical results show that such problems can be solved in (nearly) the same asymptotic time as fast polynomial multiplication. However, these reductions, even when applied to an in-place variant of fast polynomial multiplication, yield algorithms which require at least a linear amount of extra space for intermediate results. We demonstrate new in-place algorithms for the aforementioned polynomial computations which require only constant extra space and achieve the same asymptotic running time as their out-of-place counterparts. We also provide a precise complexity analysis so that all constants are made explicit, parameterized by the space usage of the underlying multiplication algorithms.

1 Introduction

1.1 Background and motivation

Computations with dense univariate polynomials or truncated power series over a finite ring are of central importance in computer algebra and symbolic computation. Since the discovery of subquadratic ("fast") multiplication algorithms [10, 4, 17, 9, 3], a major research task was to reduce many other polynomial computations to the cost of polynomial multiplication.

This project has been largely successful, starting with symbolic Newton iteration for fast inversion and division with remainder [12], product tree algorithms for multi-point evaluation and interpolation [13], the "half-GCD" fast Euclidean algorithm [16], and many more related important problems [2, 5]. Not only are these problems important in their own right, but they also form the basis for many more, such as polynomial factorization, multivariate and/or sparse polynomial arithmetic, structured matrix computations, and further applications in areas such as coding theory and public-key cryptography.

But the use of fast arithmetic frequently comes at the expense of requiring extra *temporary space* to perform the computation. This can make a difference in practice, from the small scale where embedded systems engineers seek to minimize hardware circuitry, to the medium scale where a space-inefficient algorithm can exceed the boundaries of (some level of) cache and cause expensive cache misses, to the large scale where main memory may simply not be sufficient to hold the intermediate values.

In a streaming model, where the output must be written only once, in order, explicit time-space tradeoffs prove that fast multiplication algorithms will always require up to linear extra space. And indeed, all sub-quadratic polynomial multiplication algorithms we are aware of — in their original formuation — require linear extra space [10, 4, 17, 9, 3].

However, if we treat the output space as pre-allocated random-access memory, allowing values in output registers to be both read and written multiple times, then improvements are possible. In-place quadratic-time algorithms for polynomial arithmetic are described in [14]. A series of recent results provide explicit algorithms and reductions from arbitrary fast multiplication routines which have the same *asymptotic* running time, but use only constant extra space [18, 15, 8, 6]. That is, these algorithms trade a *constant* increase in the running time for a *linear* reduction in the amount of extra space.

So far, these results are limited to multiplication routines and related computations such as middle and short product. Applying in-place multiplication algorithms directly to other problems, such as those considered in this paper, does not immediately yield an in-place algorithm for the desired application problem.

1.2 Our work

	Time	**Space**	**Reference**
Power series inversion	$(\lambda_m + \lambda_s)M(n)$	$\frac{1}{2}\max(c_m, c_s + 1)n$	[7, Alg. `MP-inv`]
at precision n	$\lambda_m \mathsf{M}(n) \log_{\frac{c_m+2}{c_m+1}}(n)$	$O(1)$	Theorem 2.3
	$(\lambda_m + \frac{3}{2}\lambda_s)\mathsf{M}(n)$	$\frac{c_m+1}{2}n$	[7, Alg. `MP-div-KM`]
Power series division	$\lambda_m \mathsf{M}(n) \log_{\frac{c_m+3}{c_m+2}}(n)$	$O(1)$	Theorem 2.5
at precision n	$O(\mathsf{M}(n))$	αn, for any $\alpha > 0$	Remark 2.6
	$\left(\lambda_m(\frac{c+1}{2} + \frac{1}{c}) + \lambda_s(1 + \frac{1}{c})\right)\mathsf{M}(n)$	$O(1)^{\ddagger}$	Remark 2.7
Euclidean division	$(\lambda_m + \frac{3}{2}\lambda_s)\mathsf{M}(m) + \lambda_s \mathsf{M}(n)$	$\max(\frac{c_m+1}{2}m - n, c_s n)$	standard algorithm
of polynomials	$2\lambda_s \mathsf{M}(m) + (\lambda_m + \lambda_s)\mathsf{M}(n)$	$(1 + \max(\frac{c_m}{2}, \frac{c_s+1}{2}, c_s))n$	$\lceil \frac{m}{n} \rceil$ balanced div. (precomp)
in sizes $(m+n-1, n)$	$\left(\lambda_m(\frac{c+1}{2} + \frac{1}{c}) + \lambda_s(2 + \frac{1}{c})\right)\mathsf{M}(m)$	$O(1)$	Theorem 2.8
multipoint evaluation	$3/2\mathsf{M}(n)\log(n)$	$n\log(n)$	[2]
size-n polynomial on n points	$7/2\mathsf{M}(n)\log(n)$	n	[19], Lemma 3.1
	$(4 + 2\lambda_s/\log(\frac{c_s+3}{c_s+2}))\mathsf{M}(n)\log(n)$	$O(1)$	Theorem 3.4
interpolation	$5/2\mathsf{M}(n)\log(n)$	$n\log(n)$	[2]
size-n polynomial on n points	$5\mathsf{M}(n)\log(n)$	$2n$	[5, 19], Lemma 3.3
	$\simeq 105\mathsf{M}(n)\log(n)$	$O(1)$	Theorem 3.6

Table 1: Summary of complexity analyses, omitting non-dominant terms and assuming $c_f \leq c_s \leq c_m$. We use $c = c_m + 3$. For $O(1)^{\ddagger}$ space, the memory model is changed such that the input dividend can be overwritten.

In this paper, we present new in-place algorithms for power series inversion, polynomial division with remainder, multi-point evaluation, and interpolation. These algorithms are *fast* because their running time is only a constant time larger than the fastest known out-of-place algorithms, parameterized by the cost of dense polynomial multiplication.

By "in-place", we mean precisely that our algorithms can work using the output space plus a constant number of extra memory locations, for any input size. In this, we assume that a single memory location or register may contain either an element of the coefficient ring, or a pointer to the input or output space (i.e., an index from 0 up to the input size plus output size).

For all five problems, we present in-place variants which have nearly the same asymptotic running time as their fastest out-of-place counterparts. The power series inversion and division algorithms incur an extra $\log(n)$ overhead in computational cost when composed with a quasi-linear multiplication algorithm, while the polynomial division, evaluation, and interpolation algorithms have exactly the same asymptotic runtime as the fastest known algorithm for the same problem.

Our reductions essentially trade a small amount of extra runtime for a significant decrease in space usage. We make this tradeoff explicit by also providing precise leading-term constants in the running time calculations.

A summary of the complexities of previous approaches, as well as our new in-place algorithms, is provided in Table 1. We emphasize that the main novelty of our algorithms is that they require no extra space; the constant difference in running time may give some idea of how they would compare in practical situations.

1.3 Notation

As usual, we denote by $\mathsf{M}(n)$ a bound on the number of operations in $\mathbb{K}$ to multiply two size-n polynomials, and we assume classically that $\alpha\mathsf{M}(n) \leq \mathsf{M}(\alpha n)$ for any constant $\alpha \geq 1$.

All known multiplication algorithms have at most a linear space complexity. Nevertheless, several results managed to reduce this space complexity at the expense of a slight increase in the time complexity [18, 15, 8, 6]. To provide tight complexity analysis, we consider that multiplication algorithms have a time complexity $\lambda_f\mathsf{M}(n)$ while using $c_f n$ extra space for some constants $\lambda_f \geq 1$ and $c_f \geq 0$.

Let us recall that the middle product of a size-$(m+n-1)$ polynomial $F \in \mathbb{K}[X]$ and a size-n polynomial $G \in \mathbb{K}[X]$ is the size-m polynomial defined as $\mathsf{MP}(F,G) = (FG \operatorname{div} X^{n-1}) \bmod X^m$. We denote by $\lambda_m\mathsf{M}(n)$ and $c_m n$ the time and space complexities of the middle product of size $(2n-1,n)$. Then, a middle product in size $(m+n-1,n)$ where $m < n$ can be computed with $\lceil \frac{n}{m} \rceil \lambda_m \mathsf{M}(m)$ operations in $\mathbb{K}$ and $(c_m+1)m$ extra space. Similarly, the short product of two size-n polynomials $F,G \in \mathbb{K}[X]$ is defined as $\mathsf{SP}(F,G) = FG \bmod X^n$ and we denote by $\lambda_s\mathsf{M}(n)$ and $c_s n$ its time and space complexities.

On the one hand, the most time-efficient algorithms achieve $\lambda_f = \lambda_m = \lambda_s = 1$ while c_f, c_m, c_s are usually between 2 and 4, using the *Transposition principle* [7, 2] for $\lambda_m = \lambda_f$. On the other hand, the authors recently proposed new space-efficient algorithms reaching $c_f = 0$, $c_m = 1$ and $c_s = 0$ while λ_f, λ_m and λ_s remain constants [6].

Writing $F = \sum_{i=0}^{d} f_i X^i \in \mathbb{K}[X]$, we will use $\operatorname{rev}(F) \in \mathbb{K}[X]$ to denote the reverse polynomial of F, that is, $\operatorname{rev}(F) = X^d F(1/X)$, whose computation does not involve any operations in $\mathbb{K}$. Note that we will use abusively the notation $F_{[a..b[}$ to refer to the chunk of F that is the polynomial $\sum_{i=a}^{b-1} f_i X^i$, and the notation $F_{[a]}$ for the coefficient f_a. Considering our storage, the notation $F_{[a..b[}$ will also serve to refer to some specific registers associated to F.

2 Inversion and divisions

In this section, we present in-place algorithms for the inversion and the division of power series as well as the Euclidean division of polynomials. As a first step, we investigate the space complexity from the literature for these computations.

2.1 Space complexity of classical algorithms

Power series inversion Power series inversion is usually computed through Newton iteration: If G is the inverse of F at precision k then $H = G + (1 - GF)G \mod X^{2k}$ is the inverse of F at precision $2k$. This allows to compute F^{-1} at precision n using $O(\mathsf{M}(n))$ operations in $\mathbb{K}$, see [5, Chapter 9]. As noticed in [7, Alg. `MP-inv`] only the coefficients of degree k to $2k-1$ of H are needed. Thus, assuming that $G_{[0..k[} = F^{-1} \bmod X^k$, one Newton iteration computes k new coefficients of F^{-1} into $G_{[k..2k]}$ as

$$G_{[k..2k[} = -\mathsf{SP}(\mathsf{MP}(F_{[1..2k[}, G_{[0..k[}), G_{[0..k[}). \tag{1}$$

The time complexity is then $(\lambda_m + \lambda_s)\mathsf{M}(n)$ for an inversion at precision n. For space complexity, the most consuming part is the last iteration of size $\frac{n}{2}$. It needs $\max(c_m, c_s + 1)\frac{n}{2}$ extra registers: One can store temporarily the middle product in $G_{[\frac{n}{2}..n[}$ using $c_m\frac{n}{2}$ extra registers, then move it to $\frac{n}{2}$ extra registers and compute the short product using $c_s\frac{n}{2}$ registers.

Power series division Let $F, G \in \mathbb{K}[[X]]$, the fast approach to compute $F/G \bmod X^n$ is to first invert G at precision n and then to multiply the result by F. The complexity is given by one inversion and one short product at precision n. Actually, Karp and Markstein remarked in [11] that the last iteration can directly compute F/G. Applying this trick, [7] shows that the complexity becomes $(\lambda_m + \frac{3}{2}\lambda_s)\mathsf{M}(n)$. Further details on these constants together with an historical report can be found in [1]. The main difference with inversion is the storage of the short product of size $\frac{n}{2}$, yielding a space complexity of $\max(c_m + 1, c_s + 1)\frac{n}{2}$.

Euclidean division of polynomials Given two polynomials A, B of respective size $m + n - 1$ and n, the fast Euclidean division computes the quotient $A \operatorname{div} B$ as $\operatorname{rev}(\operatorname{rev}(A)/\operatorname{rev}(B))$ viewed as power series at precision m [5, Chapter 9]. The remainder R is retrieved with a size-n short product, yielding a total time complexity of $(\lambda_m + \frac{3}{2}\lambda_s)\mathsf{M}(m) + \lambda_s\mathsf{M}(n)$. Since the remainder size is not determined by the input size we assume that we are given a maximal output space of size $n - 1$. As this space remains free when computing the quotient, this step requires $\frac{1}{2}\max(c_m + 1, c_s + 1)m - n + 1$ extra space, while computing the remainder needs $c_s n$.

As a first result, when $m \leq n$, using space-efficient multiplication is enough to obtain an in-place $O(\mathsf{M}(n))$ Euclidean division. Indeed, the output space is enough to compute the *small* quotient, while the remainder can be computed in-place [6].

When $m > n$, the space complexity becomes $O(m - n)$. In that case, the Euclidean division of A by B can also be computed by $\lceil \frac{m}{n} \rceil$ *balanced* Euclidean divisions of polynomials of size $2n - 1$ by B. It actually corresponds to a variation of the *long division algorithm*, in which each step computes n new coefficients of the quotient. To save some time, one can precompute the inverse of $\operatorname{rev}(B)$ at precision n, which gives a time complexity $(\lambda_m + \lambda_s)\mathsf{M}(n) + \frac{m}{n}2\lambda_s\mathsf{M}(n) \leq 2\lambda_s\mathsf{M}(m) + (\lambda_m + \lambda_s)\mathsf{M}(n)$ and space complexity $(1 + \max(\frac{c_m}{2}, \frac{c_s+1}{2}, c_s))n$.

Finally, one may consider to only compute the quotient or the remainder. Computing quotient only is equivalent to power series division. For the computation of the remainder, it is not yet know how to compute it without the quotient. In that case, we shall consider space usage for the computation and the storage of the quotient. When m is large compared to n, one may notice that relying on balanced divisions does not require to retain the whole quotient, but only its n latest computed coefficients. In that case the space complexity only increases by n.

Using also the fact that we can always a perform a middle product via two short products, we state the following result formally because it will be useful later on.

Lemma 2.1. *Given $A \in \mathbb{K}[X]_{<m}$ and monic $B \in \mathbb{K}[X]_{<n}$, and provided n registers for the output, the remainder $A \bmod B$ can be computed using $2\lambda_s \mathsf{M}(m)+3\lambda_s \mathsf{M}(n)+O(m+n)$ operations in $\mathbb{K}$ and $(c_s+2)n$ extra registers.*

2.2 In-place power series inversion

We notice that during the first Newton iterations, only few coefficients of the inverse have been already written. The output space thus contains lots of free registers, and the standard algorithm can use them as working space. In the last iterations, the number of free registers becomes too small to perform a standard iteration. Our idea is then to *slow down* the computation: instead of still doubling the number of coefficients computed at each iteration, the algorithm computes less and less coefficients at each iteration, in order to be able to use the free output space as working space. We denote these two phases as acceleration and deceleration phases.

The following easy lemma generalizes Newton iteration to compute only $\ell \leq k$ new coefficients from an inverse at precision k.

Lemma 2.2. *Let F be a power series and $G_{[0..k[}$ contain its inverse at precision k. Then for $0 < \ell \leq k$, if we compute*

$$G_{[k..k+\ell[} = -\mathsf{SP}\left(\mathsf{MP}\left(F_{[1..k+\ell[}, G_{[0..k[}\right), G_{[0..\ell[}\right) \tag{2}$$

then $G_{[0..k+\ell[}$ contains the inverse of F at precision $k+\ell$.

Algorithm 1 is an in-place fast inversion algorithm. Accelerating and decelerating phases correspond to $\ell = k$ and $\ell < k$.

Algorithm 1 In-Place Fast Power Series Inversion

```
Input: $F \in \mathbb{K}[X]$ of size $n$, such that $F_{[0]}$ is invertible;
Output: $G \in \mathbb{K}[X]$ of size $n$, such that $FG = 1 \mod X^n$.
Required: $\mathsf{MP}$ and $\mathsf{SP}$ alg. using extra space $\leq c_m n$ and $\leq c_s n$.
1: $G_{[0]} \leftarrow F_{[0]}^{-1}$
2: $k \leftarrow 1, \quad \ell \leftarrow 1$
3: while $\ell > 0$ do
4:     $G_{[n-\ell..n[} \leftarrow \mathsf{MP}(F_{[1..k+\ell[}, G_{[0..k[})$          ▷ WS: $G_{[k..n-\ell[}$
5:     $G_{[k..k+\ell[} \leftarrow \mathsf{SP}(G_{[0..\ell[}, -G_{[n-\ell..n[})$          ▷ WS: $G_{[k+\ell..n-\ell[}$
6:     $k \leftarrow k+\ell$
7:     $\ell \leftarrow \min\left(k, \left\lfloor \frac{n-k}{c} \right\rfloor\right)$ where $c = 2+\max(c_m, c_s)$
8: $G_{[k..n[} \leftarrow \mathsf{SP}(G_{[0..n-k[}, -\mathsf{MP}(F_{[1..n[}, G_{[0..k[}))$          ▷ $O(1)$ space
```

Theorem 2.3. *Algorithm 1 is correct. It uses $O(1)$ space, and either $\lambda_m \mathsf{M}(n) \log_{\frac{c_m+2}{c_m+1}}(n) + O(\mathsf{M}(n))$ operations in $\mathbb{K}$ when $\mathsf{M}(n)$ is quasi-linear, or $O(\mathsf{M}(n))$ operations in $\mathbb{K}$ when $\mathsf{M}(n) = n^{1+\gamma}$, $0 < \gamma \leq 1$.*

Proof. Steps 4 and 5, and Step 8, correspond to Equation (2) and they compute ℓ new coefficients of G when k of them are already written in the output, whence Lemma 2.2 implies the correctness.

Step 4 needs $(c_m+2)\ell$ free registers for its computation and its storage. Then $(c_s+2)\ell$ free registers are needed to compute $\mathsf{SP}(G_{[0..\ell[}i, G_{[n-\ell..n[})$ using ℓ registers for $G_{[n-\ell..n[}$ and $(c_s+1)\ell$ registers for the short product computation and its result. To ensure that this computation can be done in-place, we thus need $c\ell \leq n-k$. Since at most k new coefficients can be computed, the maximal number of new coefficients in each step is $\ell = \min\left(k, \left\lfloor \frac{n-k}{c} \right\rfloor\right)$.

Each iteration of the algorithm costs $O(\mathsf{M}(k))$ operations in $\mathbb{K}$ since the middle product in Step 4 amounts to $O(\lceil \frac{k}{\ell} \rceil \mathsf{M}(\ell))$ while the short product of Step 5 is $O(\mathsf{M}(\ell))$. The accelerating phase stops when $k > \frac{n-k}{c+1}$, that is $k > \frac{n}{c+2}$, and it costs $\sum_{i=0}^{\lfloor \log \frac{n}{c+2} \rfloor} \mathsf{M}(2^i) = O(\mathsf{M}(n))$. During the decelerating phase, each iteration computes a constant fraction of the remaining coefficients. Hence, this phase lasts for $\delta = \log_{\frac{c}{c-1}} n$ steps.

Let ℓ_i and k_i denote the values of ℓ and k at the i-th iteration of the deceleration phase and $t_i = n - k_i$. Then one iteration of the deceleration phase costs one middle product in sizes $(n - t_i + \lfloor \frac{t_i}{c} \rfloor - 1, n - t_i)$ and one short product in size $\lfloor \frac{t_i}{c} \rfloor$. The total cost of all the short products amounts to $\sum_i \mathsf{M}(t_i) = O(\mathsf{M}(n))$ since $\sum_i t_i \leq cn$. The cost of the middle product at the i-th step is

$$\lambda_m \left\lceil (n - t_i) / \left\lfloor \frac{t_i}{c} \right\rfloor \right\rceil \mathsf{M}\left(\left\lfloor \frac{t_i}{c} \right\rfloor \right) = \lambda_m \mathsf{M}(n) + O(n).$$

Therefore, the cost coming from all the middle products is at most $\lambda_m \mathsf{M}(n) \log_{\frac{c}{c-1}}(n) + O(\mathsf{M}(n))$. Since the middle products dominate and $c_m \geq 1$, we can choose the in-place short products of [6]. This implies that $c = c_m + 2$ and that the algorithm has cost $\lambda_m \mathsf{M}(n) \log_{\frac{c_m+2}{c_m+1}}(n) + O(\mathsf{M}(n))$.

Assuming now that $\mathsf{M}(n) = n^{1+\gamma}$ with $0 < \gamma \leq 1$, we can further reduce this bound. The cost of each iteration is then $O(\lceil \frac{n-t_i}{\ell_i} \rceil \ell_i^{1+\gamma})$. Since $\ell_0 \leq n$, we easily show that $\ell_i < n \left(\frac{c-1}{c} \right)^i + c$ and this implies

$$\sum_{i=1}^{\delta} \left\lceil \frac{n - t_i}{\ell_i} \right\rceil \ell_i^{1+\gamma} \leq n \sum_{i=1}^{\delta} \ell_i^{\gamma} \leq n \sum_{i=1}^{\delta} \left(n \left(\frac{c-1}{c} \right)^i + c \right)^{\gamma}.$$

Since $0 < \gamma \leq 1$, we have $(\alpha + \beta)^\gamma \leq \alpha^\gamma + \beta^\gamma$ for any $\alpha, \beta > 0$, and

the complexity is $n^{1+\gamma} \sum_{i=1}^{\delta} \left(\frac{c-1}{c} \right)^{i\gamma} + O(n \log n) = O(\mathsf{M}(n))$. □

2.3 In-place division of power series

Division of power series can be implemented easily as an inversion followed by a product. Yet, using in-place algorithms for these two steps is not enough to obtain an in-place division algorithm since the intermediate result must be stored. Karp and Markstein's trick, that includes the dividend in the last iteration of Newton iteration [11], cannot be used directly in our case since we replace the very last iteration by several ones. We thus need to build our in-place algorithm on the following generalization of their method.

Lemma 2.4. *Let F and G be two power series, G invertible, and $Q_{[0..k[}$ contain their quotient at precision k. Then for $0 < \ell \leq k$, if we compute*

$$Q_{[k..k+\ell[} = \mathsf{SP}\left(G^{-1}_{[0..\ell[}, F_{[k..k+\ell[} - \mathsf{MP}(G_{[1..k+\ell[}, Q_{[0..k[}) \right)$$

then $Q_{[0..k+\ell[}$ contains their quotient at precision $k + \ell$.

Proof. Let us write $F/G = Q_k + X^k Q_\ell + O(X^{k+\ell})$. We aim to prove that $Q_\ell = G^{-1} \times ((F - GQ_k) \operatorname{div} X^k) \bmod X^\ell$. By definition, $F \equiv G(Q_k + X^k Q_\ell) \bmod X^{k+\ell}$, hence $F - GQ_k$ has valuation at least k and $(F - GQ_k) \operatorname{div} X^k = GQ_\ell \bmod X^\ell$. Therefore, $Q_\ell = (G^{-1} \times ((F - GQ_k) \operatorname{div} X^k)) \bmod X^\ell$. Finally, since only the coefficients of degree k to $k + \ell - 1$ of GQ_k are needed, they can be computed as $\mathsf{MP}(G_{[1..k+\ell[}, Q_{[0..k[})$. □

Algorithm 2 is an in-place power series division algorithm based on Lemma 2.4, choosing at each step the appropriate value of ℓ so that all computations can be performed in place.

Algorithm 2 In-Place Power Series Division

```
Input: $F, G \in \mathbb{K}[X]$ of size $n$, such that $G_{[0]}$ is invertible;
Output: $Q \in \mathbb{K}[X]$ of size $n$, such that $F/G = Q \mod X^n$.
Required: MP, SP, Inv alg. using extra space $\leq c_m n,\ c_s n,\ c_i n$.
1:  $k \leftarrow \lfloor n/\max(c_i + 1, c_s + 2)\rfloor$
2:  $Q_{[n-k..n[} \leftarrow \mathrm{rev}(\mathsf{Inv}(G_{[0..k[}))$                  ▷ WS: $Q_{[0..n-k[}$
3:  $Q_{[0..k[} \leftarrow \mathsf{SP}(F_{[0..k[}, \mathrm{rev}(Q_{[n-k..n[}))$          ▷ WS: $Q_{[k..n-k[}$
4:  $\ell \leftarrow \lfloor (n-k)/(3 + \max(c_m, c_s))\rfloor$
5:  while $\ell > 0$ do
6:      $Q_{[n-2\ell..n-\ell[} \leftarrow \mathsf{MP}(G_{[1..k+\ell[}, Q_{[0..k[})$        ▷ WS: $Q_{[k..n-2\ell[}$
7:      $Q_{[n-2\ell..n-\ell[} \leftarrow F_{[k..k+\ell[} - Q_{[n-2\ell..n-\ell[}$
8:      let us define $Q_l^* = \mathrm{rev}(Q_{[n-\ell..n[})$
            $Q_{[k..k+\ell[} \leftarrow \mathsf{SP}(Q_{[n-2\ell..n-\ell[}, Q_l^*)$        ▷ WS: $Q_{[k+\ell..n-2\ell[}$
9:      $k \leftarrow k + \ell$
10:     $\ell \leftarrow \lfloor (n-k)/(3 + \max(c_m, c_s))\rfloor$
11: $tmp \leftarrow F_{[k..n[} - \mathsf{MP}(G_{[1..n[}, Q_{[0..k[})$               ▷ constant space
12: $Q_{[k..n[} \leftarrow \mathsf{SP}(tmp, \mathrm{rev}(Q_{[k..n[}))$                 ▷ constant space
```

Theorem 2.5. *Algorithm 2 is correct. It uses $O(1)$ space, and either $\lambda_m \mathsf{M}(n)\log_{\frac{c_m+3}{c_m+2}}(n) + O(\mathsf{M}(n))$ operations in $\mathbb{K}$ when $\mathsf{M}(n)$ is quasi-linear or $O(\mathsf{M}(n))$ operations in $\mathbb{K}$ when $\mathsf{M}(n) = O(n^{1+\gamma})$, $0 < \gamma \leq 1$.*

Proof. The correctness follows from Lemma 2.4.

Note that the inverse of G is required at each step, but with less and less precision. Hence, it is computed only once at Step 2 for a maximal initial value k and its unneeded coefficients are progressively erased. For simplicity of the presentation, we store its value in reversed order as the coefficients $n-k$ to n of the output space Q.

Since $c \geq 2$ in the main loop, $\ell \leq (n-k)/2$ and it implies that $k+\ell \leq n-\ell$. Thus, exactly ℓ coefficients of the inverse remains at the end of each loop which is sufficient to run the algorithm.

The available space is specified for each computation in the description of the algorithm as WS. Step 2 requires space $c_i k$ while the free space has size $n-k$: since $k \leq \frac{n}{c_i+1}$, the free space is large enough. Similarly, the next step requires space $c_s k$ while the free space has size $n-2k$, and $k \leq \frac{n}{c_s+2}$. Step 6 needs $(c_m+1)\ell$ space and the free space has size $n-k-2\ell$, and Step 8 requires $c_s\ell$ space while the free space has size $n-k-3\ell$. Since $\ell \leq \frac{n-k}{3+\max(c_m,c_s)}$, these computations can also be performed in place.

The time complexity of this algorithm is basically the same as the in-place inversion algorithm, that is $O(\mathsf{M}(n)\log n)$ in general, and $O(\mathsf{M}(n))$ if $\mathsf{M}(n)$ is not quasi-linear. Indeed, the only difference is Step 7 which adds $O(\ell)$ operations in $\mathbb{K}$ at each iteration, thus only impacting the complexity with a negligible term $O(n\log n)$. The more precise bound given in Theorem 2.3 also holds for that algorithm, using the appropriate value of c. □

The proofs of the following remarks can be found in Appendix A.

Remark 2.6. *Algorithm 2 can be easily modified to improve the complexity to $O(M(n))$ operations in $\mathbb{K}$ when a linear amount of extra space is available, say αn registers for some $\alpha \in \mathbb{R}_+$.*

Remark 2.7. *If it can erase its dividend, Algorithm 2 can be modified to improve its complexity $\left(\lambda_m(\frac{c+1}{2} + \frac{1}{c}) + \lambda_s(1 + \frac{1}{c})\right)\mathsf{M}(n) + O(n)$ operations in $\mathbb{K}$, still using $O(1)$ extra space.*

2.4 In-place Euclidean division of polynomials

If A is a size-$(m+n-1)$ polynomial and B a size-n polynomial, one can compute their quotient Q of size m in-place using Algorithm 2, with $O((\mathsf{M}(m)\log m))$ operations in $\mathbb{K}$. When Q is known, the remainder R satisfying $R = A{-}BQ$, can then be computed in-place using $O(\mathsf{M}(n))$ operations in $\mathbb{K}$ as it requires a single short product and some substractions. As previously mentioned, the exact size of the remainder is not determined by the size of the inputs. Note though that the suggested algorithm can still work in-place if a tighter space $r < n$ for R is given, assuming $\deg R < n$. The cost for computing R in that case becomes $O(\mathsf{M}(r))$.

Altogether, we get in-place algorithms for the computation of the quotient of two polynomials in time $O(\mathsf{M}(m)\log m)$, or the quotient and size-r remainder in time $O(\mathsf{M}(m)\log m + \mathsf{M}(r))$. As suggested in Section 2.1 and in Remark 2.6, this complexity becomes $O(\mathsf{M}(m)+\mathsf{M}(r))$ whenever $m \le n$. Indeed, in that case the remainder space can be used to speed-up the quotient computation. We shall mention that computing only the remainder remains a harder problem as we cannot account on the space of the quotient while it is required for the computation. As of today, only the classical quadratic long division algorithm allows such an in-place computation.

We now provide a new in-place algorithm for computing both the quotient and the remainder that achieves a complexity of $O(\mathsf{M}(m)+\mathsf{M}(n))$ operation in $\mathbb{K}$ when $m \ge n$.

Algorithm 3 In-Place Euclidean Division

```
Input: $A, B \in \mathbb{K}[X]$ of sizes $(m+n, n)$, $m \ge n$, such that $B_{[0]} \neq 0$;
Output: $Q, R \in \mathbb{K}[X]$ of sizes $(m+1, n-1)$ such that $A = BQ + R$;
Required: In-place DivErase$(F, G, n)$ computing $F/G \bmod X^n$ while erasing $F$; In-place $\mathsf{SP}$;
    For simplicity, $H$ is a size-$n$ polynomial such that $H_{[0..n-1[}$ is $R$ and $H_{[n-1]}$ is an extra register
 1: $H \leftarrow A_{[m..m+n[}$
 2: $k \leftarrow m+1$
 3: while $k > n$ do
 4:     $Q_{[k-n..k[} \leftarrow \mathrm{rev}(\textsc{DivErase}(\mathrm{rev}(H), \mathrm{rev}(B), n))$
 5:     $H_{[0..n-1[} \leftarrow \mathsf{SP}(Q_{[k-n..k-1[}, B_{[0..n-1[})$
 6:     $H_{[1..n[} \leftarrow A_{[k-n..k-1[} - H_{[0..n-1[}$
 7:     $H_{[0]} \leftarrow A_{[k-n-1]}$
 8:     $k \leftarrow k-n$
 9: $Q_{[0..k[} \leftarrow \mathrm{rev}(\textsc{DivErase}(\mathrm{rev}(H_{[n-k..n[}), \mathrm{rev}(B_{[n-k..n[})))$
10: $H_{[0..n-1[} \leftarrow \mathsf{SP}(Q_{[0..n-1[}, B_{[0..n-1[})$
11: $H_{[0..n-1[} \leftarrow A_{[0..n-1[} - H_{[0..n-1[}$
12: return $(Q, H_{[0..n-1[})$
```

Theorem 2.8. *Algorithm 3 is correct. It uses $O(1)$ extra space and $\big(\lambda_m(\frac{c+1}{2}+\frac{1}{c})+\lambda_s(2+\frac{1}{c})\big)\mathsf{M}(m)+O(m\log n)$ operations in $\mathbb{K}$ where $c = \max(c_m+3, c_s+2)$.*

Proof. Algorithm 3 is an adaptation of the classical *long division algorithm*, recalled in Section 2.1, where chunks of the quotient are computed iteratively *via* Euclidean division of size $(2n-1, n)$. The main difficulty is that the update of the dividend cannot be done on the input. Since we compute only chunks of size n from the quotient, the update of the dividend affects only $n-1$ coefficients. Therefore, it is possible to use the space of R for storing these new coefficients. As we need to consider n coefficients from the dividend to get a new chunck, we add the missing coefficient from A and consider the polynomial H as our new dividend.

By Remark 2.7, Step 4 can be done in-place while erasing H.

Note that erasing H is not a problem as it is not from the original input. It is thus immediate that our algorithm is in-place. For the complexity, Step 4 and 5 dominate the cost.

Using the exact complexity for Step 4 given in Remark 2.7, one can deduce easily that Algorithm 3 requires $\left(\lambda_m(\frac{c+1}{2}+\frac{1}{c})+\lambda_s(2+\frac{1}{c})\right)\mathsf{M}(m)+O(m\log n)$ operations in $\mathbb{K}$.

□

Using time-efficient products with $\lambda_m = \lambda_s = 1$, $c_m = 4$ and $c_s = 3$ yields a complexity $\simeq 6.29\mathsf{M}(m)$, which is roughly 1.57 times slower than the most time-efficient out-of-place algorithm for Euclidean division.

3 Multipoint evaluation and interpolation

In this section, we present in-place algorithms for the two related problems of multipoint evaluation and interpolation. We first review both classical algorithms and their space-efficient variants on which we base our own in-place variants.

3.1 Space complexity of classical algorithms

Multipoint evaluation Given n elements $a_1, \ldots, a_n$ of $\mathbb{K}$ and a size-n polynomial $F \in \mathbb{K}[X]$, multipoint evaluation aims to computing the n values $F(a_1), \ldots, F(a_n)$. While the naive approach using Horner scheme leads to a quadratic complexity, the fast approach of [13] reaches a quasi-linear complexity $O(\mathsf{M}(n)\log(n))$ using a divide-and-conquer approach and the fact that $F(a_i) = F \bmod (X - a_i)$. As proposed in [2] this complexity can be sharpened to $(\lambda_m + \frac{1}{2}\lambda_f)\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ using the transposition principle and working on the transposed problem of multipoint evaluation.

The fast algorithms are based on building the so-called *subproduct tree* [5, Chapter 10] whose leaves contain the $(X - a_i)$'s and whose root contains the polynomial $\prod_{i=1}^{n}(X - a_i)$. We notice that the computation of this tree already requires $O(\mathsf{M}(n)\log n)$ operations, and one needs $O(n\log n)$ space to store it: More precisely, the tree contains 2^i degree-$n/2^i$ monic polynomials at level i, and can be stored in exactly $n\log n$ registers if n is a power of two. The fast algorithms then require $n\log(n) + O(n)$ registers as work space.

Here, because the space complexity constants c_f, c_m, c_s do not appear in the leading term $n\log(n)$ of space usage, we can always choose the fastest underlying multiplication routines, so the computational cost for this approach is simply $\frac{3}{2}\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$.

As remarked in [19], one can easily derive a fast variant that uses only $O(n)$ extra space. In particular, [19, Lemma 2.1] shows that the evaluation of a size-n polynomial F on k points $a_1, \ldots, a_k$ with $k \le n$ can be done at a cost $O(\mathsf{M}(k)(\frac{n}{k} + \log(k)))$ with $O(k)$ extra space.

We begin with the *balanced case* is when $n = k$, i.e., the number of evaluation points is equal to the number size of F. The algorithm proceeds as follows, following the general structure laid out in [19, Lemma 2.1]. The idea is to group the points in $\lceil\log(n)\rceil$ groups of $\lfloor n/\log(n)\rfloor$ points each, and to use standard multipoint evaluation on each group, by first reducing F modulo the root of the corresponding subproduct tree.

The complexity analysis of this approach is given in the following lemma. Observe that here too, the constants λ_s, c_s, etc., do not enter in since we can always use the fastest out-of-place subroutines without affecting the $O(n)$ term in the space usage.

Lemma 3.1. *Given $F \in \mathbb{K}[X]_{<n}$ and $a_1, \ldots, a_n \in \mathbb{K}$, one can compute $F(a_1), \ldots, F(a_n)$ using $\frac{7}{2}\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $\mathbb{K}$ and $n + O(\frac{n}{\log(n)})$ extra registers.*

Proof. Computing each subproduct tree on $O(n/\log(n))$ points can be done in time $\frac{1}{2}\mathsf{M}(n/\log(n))\log(n) \leq \frac{1}{2}\mathsf{M}(n)$ and space $n + O(n/\log(n))$. The root of this tree is a polynomial of degree at most $n/\log(n)$. Each reduction of F modulo such a polynomial takes time $2\mathsf{M}(n) + O(n/\log(n))$ and space $O(n/\log(n))$ using the balanced Euclidean division algorithm from Section 2.1. Each multi-point evaluation of the reduced polynomial on $n/\log(n)$ points, using the pre-computed subproduct tree, takes $\mathsf{M}(n/\log(n))\log(n) + O(\mathsf{M}(n/\log(n)))$ operations in $\mathbb{K}$ and $O(n/\log(n))$ extra space [2].

All information except the evaluations from the last step — which are written directly to the output space — may be discarded before the next iteration begins. Therefore the total time and space complexity are as stated. □

When the number of evaluation points k is large compared to the size n of the polynomial F being evaluated, we can simply repeat the approach of Lemma 3.1 $\lceil k/n \rceil$ times. But when $k \leq n$ the situation is more complicated, because the output space is smaller. Specifically, we compute the degree-k polynomial M at the root of the product tree, reduce F modulo M and perform balanced k-point evaluation of $F \bmod M$.

Lemma 3.2. *Given $F \in \mathbb{K}[X]_{<n}$ and $a_1, \ldots, a_k \in \mathbb{K}$, one can compute $F(a_1), \ldots, F(a_k)$ using $2\lambda_s\mathsf{M}(n) + 4\mathsf{M}(k)\log(k) + O(n + \mathsf{M}(k)\operatorname{loglog}(k))$ operations in $\mathbb{K}$ and $(c_s + 2)k + O(k/\log(k))$ extra registers.*

Proof. Computing M using a product tree proceeds in two phases. For the bottom levels of the tree, we use the fastest out-of-place full multiplication algorithm with time $\mathsf{M}(t)$ and space $O(t)$. Then, only for the top $\operatorname{loglog}(n)$ levels, we switch to an in-place full product algorithm from [6], which has time $O(\mathsf{M}(t))$ but only $O(1)$ extra space. The result is that M can be computed using $\frac{1}{2}\mathsf{M}(k)\log(k) + O(\mathsf{M}(k)\operatorname{loglog}(k))$ operations in $\mathbb{K}$ and $k + O(k/\log(k))$ registers.

Then, we need to reduce F modulo M. Utilizing the size-k output space, by Lemma 2.1, this is accomplished using $2\lambda_s\mathsf{M}(n) + O(n + \mathsf{M}(k))$ time and $(c_s + 2)k$ extra registers.

Adding the cost of the k-point evaluation given by Lemma 3.1 completes the proof. □

Interpolation Interpolation is the inverse operation of multipoint evaluation, that is, to reconstruct a size-n polynomial F from its evaluations on n distinct points $F(a_1), \ldots, F(a_n)$. The classic approach using Lagrange's interpolation formula has a quadratic complexity [5, Chapter 5] while the fast approach of [13] has quasi-linear time complexity $O(\mathsf{M}(n)\log(n))$. We first briefly recall this fast algorithm.

Let $M(X) = \prod_{i=1}^{n}(X - a_i)$ and M' its derivative. Noting that $\frac{M}{X-a_i}(a_i) = M'(a_i)$ for $1 \leq i \leq n$, we have

$$F(X) = M(X)\sum_{i=1}^{n}\frac{F(a_i)/M'(a_i)}{X - a_i}. \tag{3}$$

Hence the fast algorithm of [13] consists in computing $M'(X)$ and its evaluation on each a_i through multipoint evaluation, and then to sum the n fractions using a divide-and-conquer strategy. The numerator of the result is then F by Equation (3).

Performing this rational fraction sum in a binary tree fashion uses $\frac{3}{2}\lambda_f\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $\mathbb{K}$ and requires $(3 + \frac{1}{2}c_f)n$ registers, including the space for the output.

However, if the subproduct tree over the a_i's is already computed, this gives all the denominators in the rational fraction sum. This leads to the fastest interpolation algorithm, using the textbook method of [5] with the multi-point evaluation of [2].

Because the same subproduct tree is used for evaluating M' and for the rational fraction sum, the total computational cost is only $\frac{5}{2}\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$, while the space is dominated by the size of this subproduct tree, $n\log(n) + O(n)$ registers.

A more space-efficient approach uses instead the $O(n)$-space multi-point evaluation of [19] as described above, but suffers more in running time because the subproduct tree must be essentially recomputed on the first and last steps. The total running time is $(2\lambda_f + \frac{7}{2})\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$, using $(2 + \frac{1}{2}c_f)n + O(n/\log(n))$ registers.

This $O(n)$-space approach can be further improved in two ways: first by again grouping the interpolation points and re-using the smaller subproduct trees for each group, and secondly by using an in-place full multiplication algorithm from [6] to combine the results of each group in the rational function summation. A detailed description of the resulting algorithm, along with a proof of the following lemma, can be found in Appendix B.

Lemma 3.3. *Given $a_1, \dots, a_n \in \mathbb{K}$ and $y_1, \dots, y_n \in \mathbb{K}$, one can compute $F \in \mathbb{K}[X]_{<n}$ such that $F(a_i) = y_i$ for $1 \le i \le n$ using $5\mathsf{M}(n)\log(n) + O(\mathsf{M}(n)\operatorname{loglog}(n))$ operations in $\mathbb{K}$ and $2n + O(n/\log(n))$ extra registers.*

3.2 In-place multipoint evaluation

In order to derive an in-place algorithm we make repeated use of the unbalanced multi-point evaluation with linear space from [19] and Lemma 3.2 to compute only k evaluations of the polynomial F among the n original points. The strategy is to set k as a fraction of n to ensure that $n - k$ is large enough to serve as extra space. Applying this strategy on smaller and smaller values of k leads to Algorithm 4, which is an in-place algorithm with the same asymptotic time complexity $O(\mathsf{M}(n)\log(n))$ as out-of-place fast multipoint evaluation.

Algorithm 4 In-place multipoint evaluation

```
Input: $F \in \mathbb{K}[X]$ of size $n$ and $(a_1, \dots, a_n) \in \mathbb{K}^n$;
Output: $R = (F(a_1), \dots, F(a_n))$
Required: EVAL of space complexity $\le (c_s + 2)k$ as in Lemma 3.2
1: $s \leftarrow 0$,   $k \leftarrow \lfloor n/(c_s + 3) \rfloor$
2: while $k > 0$ do
3:     $R_{[s..s+k[} \leftarrow$ EVAL$(F, a_s, \dots, a_{s+k})$        ▷ WS: $R_{[s+k..n[}$
4:     $s \leftarrow s + k$
5:     $k \leftarrow \left\lfloor \frac{n-s}{c_s+3} \right\rfloor$
6: $R_{[s..n[} \leftarrow$ EVAL$(F, a_s, \dots, a_n)$        ▷ constant space
```

Theorem 3.4. *Algorithm 4 is correct. It uses $O(1)$ extra space and $\left(4 + 2\lambda_s / \log(1 + \frac{1}{c_s+2})\right)\mathsf{M}(n)\log(n) + O(\mathsf{M}(n)\log\log n)$ operations in $\mathbb{K}$.*

Proof. The correctness is obvious as soon as EVAL is correct. By the choice of k and from the extra space bound of EVAL from Lemma 3.2, Step 3 has sufficient work space, and therefore the entire algorithm is in-place.

The sequence $k_i = \frac{(c_s+2)^{i-1}}{(c_s+3)^i} n$, for $i = 1, 2, \dots$, gives the values of k in each iteration. Then $\sum_i k_i \le n$ and the loop terminates after at most $\ell \log(n)$ iterations, where $\ell \le 1/\log(1 + \frac{1}{c_s+2})$.

Applying Lemma 3.2, the cost of the entire algorithm is therefore dominated by $\sum_{1 \le i \le \ell} (2\lambda_s \mathsf{M}(n) + 4\mathsf{M}(k_i)\log(k_i))$, which is at most $(2\lambda_s \ell + 4)\mathsf{M}(n)\log(n)$. □

Using a time-efficient short product with $\lambda_s = 1$ and $c_s = 3$ yields a complexity $\simeq 11.61\mathsf{M}(n)\log n$, which is roughly 7.74 times slower than the most time-efficient out-of-place algorithm for multi-point evaluation.

3.3 In-place interpolation

Let $(a_1, y_1), \dots, (a_n, y_n)$ be n pairs of evaluations, with the a_i's pairwise distinct. The goal of interpolation is to compute the unique polynomial $F \in \mathbb{K}[X]_{<n}$ such that $F(a_i) = y_i$ for $1 \le i \le n$. As before, we will be able to derive an in-place algorithm for interpolation by computing the result on smaller and smaller chunks.

Our first aim is to provide a variant of polynomial interpolation that computes $F \bmod X^k$ using $O(k)$ extra space. Without loss of generality, we assume that k divides n. For $i = 1$ to n/k, let $T_i = \prod_{j=1+k(i-1)}^{ki}(X - a_j)$ and $S_i = M/T_i$ where $M = \prod_{i=1}^{n}(X - a_i)$. Note that $S_i = \prod_{j \neq i} T_j$. One can rewrite Equation (3) as

$$\begin{aligned} F(X) &= M(X)\sum_{i=1}^{n/k}\sum_{j=1+k(i-1)}^{ki}\frac{F(a_j)}{M'(a_j)}\frac{1}{(X-a_j)} \\ &= M(X)\sum_{i=1}^{n/k}\frac{N_i(X)}{T_i(X)} = \sum_{i=1}^{n/k}N_i(X)S_i(X) \end{aligned} \tag{4}$$

for some size-k polynomials $N_1, \dots, N_{n/k}$. One may remark that the latter equality can also be viewed as an instance of the chinese remainder theorem where $N_i = F/S_i \bmod T_i$ (see [5, Chapter 5]). Since we want the first k terms of the polynomial F, we only need to compute

$$F \bmod X^k = \sum_{i=1}^{n/k} N_i(S_i \bmod X^k) \bmod X^k. \tag{5}$$

One can observe that $M'(a_j) = (S_i \bmod T_i)(a_j)T_i'(a_j)$ for $k(i-1) < j \le ki$. Therefore, Equation (4) implies that N_i is the unique size-k polynomial satisfying $N_i(a_j) = (F/S_i \bmod T_i)(a_j)$. Therefore, N_i can be computed using interpolation, by first computing $S_i \bmod T_i$, evaluating it at the a_j's, performing k divisions in $\mathbb{K}$ to get each $N_i(a_j)$ and finally interpolating N_i.

Our second aim is to generalize the previous approach when some initial coefficients of F are known. Writing $F = G + X^sH$ where G is known, we want to compute $H \bmod X^k$ from some evaluations of F. Since H has size at most $(n-s)$, only $(n-s)$ evaluation points are needed. Therefore, using Equation 4 with $M = \prod_{i=1}^{n-s}(X - a_i)$, we can write

$$H(X) = M(X)\sum_{i=1}^{(n-s)/k}\sum_{j=1+k(i-1)}^{ki}\frac{F(a_j) - G(a_j)}{a_j^s M'(a_j)}\frac{1}{(X-a_j)}. \tag{6}$$

This implies that $H \bmod X^k$ can be computed using the same approach described above by replacing $F(a_j)$ with $H(a_j) = (F(a_j) - G(a_j))/a_j^s$. We shall remark that the $H(a_j)$'s can be computed using multipoint evaluation and fast exponentation. Algorithm 5 fully describes this approach.

Lemma 3.5. *Algorithm 5 is correct. It requires* $6k + O(k/\log k)$ *extra space and it uses* $\left(\frac{1}{2}(\frac{n-s}{k})^2 + \frac{23}{2}\frac{n-s}{k}\right)\mathsf{M}(k)\log(k) + (n-s)\log(s) + O((\frac{n-s}{k})^2\mathsf{M}(k)\log\log k)$ *operations in* $\mathbb{K}$.

Proof. The correctness follows from the above discussion. In particular, note that the polynomials S_i^k and S_i^T at Steps 6 and 7 equal $S_i \bmod X^k$ and $S_i \bmod T_i$ respectively. Furthermore, $z_j = G(a_{j+k(i-1)})$ since $G(a_{j+k(i-1)}) = (G \bmod T_i)(a_{j+k(i-1)})$. Hence, Step 12 correctly computes the polynomial N_i and the result follows from Equations (5) and (6).

From the discussion in Section 3.1, we can compute each T_i in $1/2\mathsf{M}(k)\log(k) + O(M(k)\log\log k)$ operations in $\mathbb{K}$ and k extra space. Step 9 requires some care as we can share some computation

Algorithm 5 Partial interpolation (PARTINTERPOL)

```
Input: $G \in \mathbb{K}[X]_{<s}$ and $(y_1, \dots, y_{n-s})$, $(a_1, \dots, a_{n-s})$ in $\mathbb{K}^{n-s}$ ; an integer $k \le n-s$
Output: $H \bmod X^k$ where $F = G + X^s H \in \mathbb{K}[X]_{<n}$ is the unique size-$n$ polynomial s.t. $F(a_i) = y_i$ for
    $1 \le i \le n-s$
 1: for $i = 1$ to $(n-s)/k$ do
 2:     $S_i^k \leftarrow 1$, $S_i^T \leftarrow 1$
 3:     $T_i \leftarrow \prod_{j=1+k(i-1)}^{ki}(X - a_j)$                          ▷ Fast divide-and-conquer
 4:     for $j = 1$ to $(n-s)/k$, $j \neq i$ do
 5:         $T_j \leftarrow \prod_{t=1+k(j-1)}^{kj}(X - a_t)$                      ▷ Fast divide-and-conquer
 6:         $S_i^k \leftarrow S_i^k \times T_j \bmod X^k$                             ▷ $S_i^k = S_i \bmod X^k$
 7:         $S_i^T \leftarrow S_i^T \times T_j \bmod T_i$                              ▷ $S_i^T = S_i \bmod T_i$
 8:     $G^T \leftarrow G \bmod T_i$
 9:     $(b_1, \dots, b_k) \leftarrow$ EVAL($S_i^T, a_{1+k(i-1)}, \dots, a_{ki}$)
        $(z_1, \dots, z_k) \leftarrow$ EVAL($G^T, a_{1+k(i-1)}, \dots, a_{ki}$)
10:     for $j = 1$ to $k$ do
11:         $b_j \leftarrow (y_{j+k(i-1)} - z_j)/(a^s_{j+k(i-1)} b_j)$
12:     $N_i \leftarrow$ INTERPOL($(z_1, \dots, z_k), (b_1, \dots, b_k)$)
13:     $H_{[0..k[} \leftarrow H_{[0..k[} + N_i S_i^k \bmod X^k$
```

among the two *equal-size* evaluations. Indeed, the subproduct trees induced by this computation are identical and thus can be computed only once. Using Lemma 3.1, this amounts to $\frac{13}{2}\mathsf{M}(k)\log(k) + O(\mathsf{M}(k))$ operations in $\mathbb{K}$ using $k + O(k/\log k)$ extra space. Step 12 can be done in $5\mathsf{M}(k)\log(k) + O(M(k)\operatorname{loglog} k)$ operations in $\mathbb{K}$ and $2k + O(k/\log k)$ extra space using Lemma 3.3. Taking into account the $n-s$ exponentations a_j^s, and that other steps have a complexity in $O(\mathsf{M}(k))$, the cost of the algorithm is

$$\left(\frac{1}{2}\left(\frac{n-s}{k}\right)^2 + \frac{23}{2}\frac{n-s}{k}\right)\mathsf{M}(k)\log(k) + (n-s)\log(s) + O(\left(\frac{n-s}{k}\right)^2 \mathsf{M}(k)\operatorname{loglog} k).$$

We show that $6k + O(k/\log k)$ extra registers are enough to implement this algorithm. At Step 7, the polynomials T_i, T_j, S_i^k, S_i^T must be stored in memory. The computation involved at this step requires only $2k$ extra registers as $S_i^T \times T_j \bmod T_i$ can be computed with an in-place full product (stored in the extra registers) followed by an in-place division with remainder using the registers of S_i^T and T_j for the quotient and remainder storage. Using the same technique Step 8 requires only k extra space as for Steps 2 to 6. At Step 9, we need $3k$ registers to store G_T, S_i^T, S_i^k and $2k$ registers to store $(b_1, \dots, b_k)$ and $(z_1, \dots, z_k)$, plus $k + O(k/\log k)$ extra register for the computation. At Step 12 we can re-use the space of G^T, S_i^T for N_i and the extra space of the computation which implies the claim. □

We can now provide our in-place variant for fast interpolation.

Theorem 3.6. *Algorithm 6 is correct. It uses at most* $\frac{1}{2}(\phi^2 + 23\phi)\mathsf{M}(n)\log n + O(\mathsf{M}(n)\operatorname{loglog} n)$ *operations in* $\mathbb{K}$ *and* $O(1)$ *extra space, where* $\phi = 1 + c_{pi}$.

Proof. The correctness is clear from the correctness of Algorithm PARTINTERPOL. To ensure that the algorithm uses $O(1)$ extra space we notice that at Step 6, $F_{[s+k..n[}$ can be used as work space. Therefore, as soon as $c_{pi}k \le n-s-k$, that is $k \le \frac{n-s}{c_{pi}+1}$, this free space is enough to run PARTINTERPOL. Note

Algorithm 6 In-Place interpolation

```
Input: $(y_1, \dots, y_n)$ and $(a_1, \dots, a_n)$ of size $n$ such that $a_i, y_i \in \mathbb{K}$;
Output: $F \in \mathbb{K}[X]$ of size $n$, such that $F(a_i) = y_i$ for $0 \le i \le n$.
Required: PartInterpol with space complexity $\le c_{pi} k$
s ← 0
while $s < n$ do
    $k \leftarrow \left\lfloor \frac{n-s}{c_{pi}+1} \right\rfloor$
    if $k = 0$ then $k \leftarrow n - s$
    $Y, A \leftarrow (y_1, \dots, y_{n-s}), (a_1, \dots, a_{n-s})$
    $F_{[s..s+k[} \leftarrow$ PartInterpol$(F_{[0..s[}, Y, A, k)$
    $s \leftarrow s + k$
```

that when $k = 0$, $n - s < c_{pi} + 1$ is a constant, which means that the final computation can be done with $O(1)$ extra space.

Let us denote $k_1, k_2, \dots, k_t$ and $s_1, s_2, \dots, s_t$ all the values of k and s taken during the course of the algorithm.

Since $s_i = \sum_{j=1}^{i} k_j \le n$ with $s_0 = 0$, we have $k_i \le \lambda n(1-\lambda)^{i-1}$, and $s_i \ge n(1-(1-\lambda)^i)$ where $\lambda = \frac{1}{c_{pi}+1}$. Therefore, the complexity $T(n)$ of the algorithm satisfies

$$T(n) \le \sum_{i=1}^{t} \left(\frac{\phi^2}{2} + \frac{23\phi}{2} \right) \mathsf{M}(k_i) \log(k_i) + \sum_{i=1}^{t} (n - s_{i-1}) \log(s_{i-1}) \\ + O((\phi^2 \mathsf{M}(k_i) \operatorname{loglog} k_i)$$

since $\frac{n-s_{i-1}}{k_i} \le \phi = c_{pi} + 1$ by definition of k_i. Moreover,

$$\sum_{i=1}^{t} \mathsf{M}(k_i) \log(k_i) \le \mathsf{M}\left(\sum_{i=1}^{t} k_i \right) \log(n) \le \mathsf{M}(n) \log(n).$$

By definition of s_i, we have $n - s_i \le n(1-\lambda)^i$ which gives

$$\sum_{i=1}^{t} (n - s_{i-1}) \log(s_{i-1}) \le n \log(n) \sum_{i=1}^{t} (1-\lambda)^i \le (c_{pi} + 1) n \log n.$$

This concludes the proof. □

Since $c_{pi} < 6 + \epsilon$ for any $\epsilon > 0$, the complexity can be approximated to $105\mathsf{M}(n)\log(n)$, which is 42 times slower than the fastest interpolation algorithm (see Table 1).

References

[1] D. Bernstein. Fast multiplication and its applications. *Mathematical Sciences Research Institute Publications*, 44:325–384, 2008.

[2] A. Bostan, G. Lecerf, and E. Schost. Tellegen's principle into practice. In *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, ISSAC '03, pages 37–44. ACM, 2003.

[3] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[4] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, May 1966.

[5] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3rd edition, 2013.

[6] P. Giorgi, B. Grenet, and D. S. Roche. Generic reductions for in-place polynomial multiplication. In *International Symposium on Symbolic and Algebraic Computation*, ISSAC'19, 2019.

[7] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm I. *Applicable Algebra in Engineering, Communication and Computing*, 14(6):415–438, Mar 2004.

[8] D. Harvey and D. S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 325–329. ACM, 2010.

[9] D. Harvey and J. van der Hoeven. Polynomial multiplication over finite fields in time O(n log n). preprint, 2019.

[10] A. Karatsuba and Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Soviet Physics-Doklady*, 7:595–596, 1963.

[11] A. H. Karp and P. Markstein. High-precision division and square root. *ACM Transactions on Mathematical Software*, 23(4):561–589, 1997.

[12] H. T. Kung. On computing reciprocals of power series. *Numerische Mathematik*, 22(5):341–348, 1974.

[13] R. Moenck and A. Borodin. Fast modular transforms via division. In *13th Annual Symposium on Switching and Automata Theory (swat 1972)*, pages 90–96, Oct 1972.

[14] M. Monagan. In-place arithmetic for polynomials over Zn. In *Design and Implementation of Symbolic Computation Systems*, volume 721, pages 22–34. Springer, 1993.

[15] D. S. Roche. Space- and time-efficient polynomial multiplication. In *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, pages 295–302. ACM, 2009.

[16] A. Schönhage. Probabilistic computation of integer polynomial gcds. *Journal of Algorithms*, 9(3):365–371, 1988.

[17] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[18] E. Thomé. Karatsuba multiplication with temporary space of size ≤ n. online, 2002.

[19] J. von zur Gathen and V. Shoup. Computing frobenius maps and factoring polynomials. *computational complexity*, 2(3):187–224, Sep 1992.

A Proofs of remarks on Algorithm 2

Remark 2.6. *Algorithm 2 can be easily modified to improve the complexity to $O(M(n))$ operations in $\mathbb{K}$ when a linear amount of extra space is available, say αn registers for some $\alpha \in \mathbb{R}_+$.*

Indeed, in that specific case, the value of k at Step 1 can be replaced by $\min(n, \lfloor(1+\alpha)n/c\rfloor)$ and the value of ℓ at Step 10 by $\min(n-k, \lfloor((1+\alpha)n-k)/c\rfloor)$. The same proof of Theorem 2.5 shows in that case that all computations can be performed in the free space of Q plus the extra αn registers. As a result, ℓ is more than a fraction of the remaining coefficients to compute. In particular, the loop stops when $n-k \le \lfloor((1+\alpha)n-k)/c\rfloor$, that is $n-k \le \frac{\alpha n}{c-1}$. In terms of time complexity, this means that the number of iterations becomes a constant. In other words, allowing αn extra space makes the time complexity decrease to $O(\mathsf{M}(n))$.

Remark 2.7. *If it can erase its dividend, Algorithm 2 can be modified to improve its complexity $\big(\lambda_m(\frac{c+1}{2}+\frac{1}{c})+\lambda_s(1+\frac{1}{c})\big)\mathsf{M}(n)+O(n)$ operations in $\mathbb{K}$, still using $O(1)$ extra space.*

Indeed, it is sufficient to notice that $F_{[0..k[}$ in the main loop of Algorithm 2 is not accessed anymore and it can thus be erased. Then, in Step 7 one can write the result directly in $F_{[k..k+\ell[}$. Therefore, the amount of free space for the computations at Step 6 and 8 become $n-2\ell$ instead of $n-k-2\ell$ and $n-k-3\ell$ respectively.

This means that ℓ can always be chosen as large as $\lfloor \frac{n}{c} \rfloor$ where $c = \max(c_m+3, c_s+2)$. To simplify the analysis, we note that Steps 2 and 3 can also use this value of c (rather than the one computed at Step 1). Therefore, the number of loop iterations is at most c which is constant and it implies the cost $O(\mathsf{M}(n))$. Note that one shall modify the algorithm to return the result after the last loop as Steps 11 and 12 will never be done in that case. To provide a finer complexity analysis, first notice that the sum of the input sizes of all the short products during the algorithm is n, so that they contribute to $\lambda_s\mathsf{M}(n)$ operations in $\mathbb{K}$. Step 2 requires $(\lambda_s+\lambda_m)\mathsf{M}(\lfloor \frac{n}{c} \rfloor)$ operations in $\mathbb{K}$. At the i-th iteration of the loop, Step 6 requires $i\lambda_m\mathsf{M}(\lfloor \frac{n}{c} \rfloor)$ operations in $\mathbb{K}$ since $k = i\lfloor \frac{n}{c} \rfloor$ at that iteration and $\ell = \lfloor \frac{n}{c} \rfloor$ and Step 7 requires $\lfloor \frac{n}{c} \rfloor$ operations. Therefore, the exact complexity is

$$\lambda_s\mathsf{M}(n)+(\lambda_s+\lambda_m)\mathsf{M}\left(\left\lfloor \frac{n}{c} \right\rfloor\right)+\sum_{i=1}^{c}\left(i\lambda_m\mathsf{M}\left(\left\lfloor \frac{n}{c} \right\rfloor\right)+\left\lfloor \frac{n}{c} \right\rfloor\right)$$

which is $\big(\lambda_m(\frac{c+1}{2}+\frac{1}{c})+\lambda_s(1+\frac{1}{c})\big)\mathsf{M}(n)+O(n)$ operations in $\mathbb{K}$.

B Interpolation with linear space

The algorithm proceeds as:

1. Run the subproduct tree algorithm for each group of $n/\log(n)$ interpolation points, saving only the roots of each subtree $M_1, \ldots, M_{\lceil\log(n)\rceil}$, using fast out-of-place full multiplications.
2. Run the subproduct tree algorithm over these M_i's to compute the root M, using in-place full multiplications from [6], discarding other nodes in the tree.
3. Compute the derivative M' in place.
4. Compute the remainders $M' \bmod M_i$ for $1 \le i \le \lceil\log(n)\rceil$, using the balanced (with precomputation) algorithm described in Section 2.1. The size-n polynomial M' may now be discarded.

5. For each group i, compute the subproduct tree over its $n/\log(n)$ points. Use this to perform multi-point evaluation of $M' \bmod M_i$ over the $n/\log(n)$ points of that group only, and then compute the partial sum of (3) for that group's points. Discard the subproduct tree but save the rational function partial sum for each group.

6. Combine the rational functions for the $\lceil \log(n) \rceil$ groups using a divide-and-conquer strategy, employing again the in-place full multiplications from [6].

The following lemma gives the complexity of this linear-space interpolation algorithm.

Lemma 3.3. *Given* $a_1, \ldots, a_n \in \mathbb{K}$ *and* $y_1, \ldots, y_n \in \mathbb{K}$, *one can compute* $F \in \mathbb{K}[X]_{<n}$ *such that* $F(a_i) = y_i$ *for* $1 \le i \le n$ *using* $5\mathsf{M}(n)\log(n) + O(\mathsf{M}(n)\operatorname{loglog}(n))$ *operations in* $\mathbb{K}$ *and* $2n + O(n/\log(n))$ *extra registers.*

Proof. Steps (1) and (5) collectively involve, for each group, 2 subproduct tree computations, one multi-point evaluation, and one rational function summation over each group, for a total of $3\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ time.

Step (4) also dominates the time complexity, contributing another $2\mathsf{M}(n)\log(n) + O(\mathsf{M}(n))$ operations in $\mathbb{K}$.

In steps (2) and (5), the expensive in-place multiplications are used only for the top $\lceil \operatorname{loglog}(n) \rceil$ levels of the entire subproduct tree, so this contributes only $O(\mathsf{M}(n)\operatorname{loglog}(n))$.

For the space, note that the size-n output space may be used during all steps until the last to store intermediate results. □