Short SLPs for sparse polynomial maps*†

JORIS VAN DER HOEVEN[a], GRÉGOIRE LECERF[b], ARNAUD MINONDO[c]

Laboratoire d'informatique de l'École polytechnique (LIX, UMR 7161)
CNRS, École polytechnique, Institut Polytechnique de Paris
1, rue Honoré d'Estienne d'Orves
91120 Palaiseau, France

a. Email: vdhoeven@lix.polytechnique.fr
b. Email: lecerf@lix.polytechnique.fr
c. Email: minondo@lix.polytechnique.fr

Preliminary version of May 1, 2026

Given a finite number of sparse polynomials, we present a new algorithm, along with its C++ implementation, to compute an efficient straight-line program which is able to simultaneously evaluate these polynomials at a given point.

1. INTRODUCTION

Let $\mathbb{A}$ be an effective ring. In other words, elements of $\mathbb{A}$ can be represented on a computer and we have algorithms for all ring operations. Let $x = (x_1, \ldots, x_n)$ be a system of coordinates and let $x^e := x_1^{e_1} \cdots x_n^{e_n}$ for any $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$. A *sparse polynomial* in x over $\mathbb{A}$ is an expression of the form

$$f = \sum_{1 \leqslant i \leqslant t} c_i x^{e_i}, \tag{1}$$

with $c_i \in \mathbb{A}$ and $e_i \in \mathbb{N}^n$ for $i = 1, \ldots, t$. We write $\operatorname{Supp} f := \{e_i : 1 \leqslant i \leqslant t, c_i \neq 0\}$ for the *support* of f. Given a vector $f = (f_1, \ldots, f_m)$ of sparse polynomials, we may consider the polynomial map

$$\begin{aligned} \mathbb{A}^n &\longrightarrow \mathbb{A}^m \\ a = (a_1, \ldots, a_n) &\longmapsto f(a) = (f_1(a), \ldots, f_m(a)). \end{aligned}$$

A natural question is how to evaluate this map as efficiently as possible. Since such sparse polynomial maps only involve additions, multiplications, and possibly subtractions, it is also natural to search for an evaluation algorithm in the form of a straight-line program (SLP). We recall that an SLP is a sequence of arithmetic instructions, without any branchings or loops; see the definition in [5] for instance, or the one from [15] for modern implementation aspects. The goal of this paper is to efficiently compute a short SLP for the evaluation of a sparse polynomial map.

*. Grégoire Lecerf and Arnaud Minondo have been supported by the French ANR-22-CE48-0016 NODE project. Joris van der Hoeven has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171).
Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

†. This article has been written using GNU TEXMACS [14, 16].

Whereas the size of an SLP is easily measured by the number of operations in $\mathbb{A}$, there are several natural measures for the size of a sparse polynomial (1): its number of terms $t_f := t$, its expression size

$$s_f := t + \sum_{1 \leqslant i \leqslant t, 1 \leqslant j \leqslant n, (e_i)_j \neq 0} 1,$$

or its bit size

$$b_f := t + \sum_{1 \leqslant i \leqslant t, 1 \leqslant j \leqslant n, (e_i)_j \neq 0} (\lfloor \log_2 (e_i)_j \rfloor + 1).$$

(For simplicity, we always regard coefficients in $\mathbb{A}$ as having unit size.) More generally, for a vector $f = (f_1, \ldots, f_m)$ of sparse polynomials, we define $t_f := t_{f_1} + \cdots + t_{f_m}$, $s_f := s_{f_1} + \cdots + s_{f_m}$, and $b_f := b_{f_1} + \cdots + b_{f_m}$. In this case, one might also consider size measures that take into account common monomials x^e or terms $c\,x^e$ in distinct polynomials $f_i \neq f_j$.

It is not hard to see that $O(b_f)$ operations always suffice for the evaluation of f (using binary powering), whereas $\geqslant 2\,(t_f - 1)$ binary sums and products are generally necessary for dense polynomials [19]. This lower bound is reached when $\operatorname{Supp} f = \{0, \ldots, d_1\} \times \cdots \times \{0, \ldots, d_n\}$ by using a multivariate Horner scheme.

Finding an SLP of optimal size is a very difficult problem (see recent advances in [17]), due to the fact that small SLPs like $(x_1 + 2\,x_2 - x_3)^{2^{10}} - (x_1 + x_2^2 - 4\,x_3)^{2^9}$ may become very large when expanding them as sparse polynomials; see also Remark 2 below. Our goal in this paper is *not* to design an algorithm with an optimal or provingly good complexity in all cases. We will rather describe an easy to implement algorithm that performs well in practice, as illustrated on both random polynomials and a database with more than 600 examples. It turns out that the computed SLP is often of size $O(t_f)$, whereas the computation of the SLP always takes at most $\tilde{O}(\tilde{n}_f^2\, t_f)$ operations, where $\tilde{n}_f$ is the largest number of variables that occur in a single term of f. We also note that the degrees of most polynomials in our database are modest.

In section 2 we start with a review of existing approaches. Already in the case when the input polynomials f_k are simple powers or monomials, it is an interesting question how to efficiently find a short SLP for their (joint) evaluation. Classical algorithms for these cases are due to Brauer [4], Straus [23], Yao [24], and Pippenger [20, 21, 22]; we refer to [2] for a nice survey. For the general case, the most common implementation strategy is based on multivariate Horner schemes [6, 7]; see also sections 2.4 and 3.3. But more dedicated bilinear schemes [13, section 4.4.3] or expensive partial factorization schemes [18] have also been studied: see sections 2.5 and 2.6.

In section 3 we present our new algorithm. It takes advantages of various ingredients. We first reduce to the case when all terms $c\,x^e$ occurring in the f_k are simple products (with $c = 1$ and $e_i \in \{0, 1\}$ for $i = 1, \ldots, n$). We next rely on a fast greedy method for the efficient evaluation of multiple products. We pursue with a simple greedy strategy to factor out some of the variables. The resulting SLP may finally be further simplified using common subexpression elimination and combining some of the additions and multiplications into "fused multiply-add" (FMA) instructions, which can be used to quickly evaluate the final SLP on modern hardware.

In our last section 4 we report on our implementation of the algorithm of section 3 inside JIL [1, 15] and show that it is well suited for sparse polynomials that are typically encountered in computer algebra. We complete our implementation by combining efficient strategies to handle polynomials across the full spectrum from sparse to dense.

2. SURVEY OF KNOWN STRATEGIES

In this section, we start with a review of known methods for evaluating monomials and polynomials.

2.1. Power trees

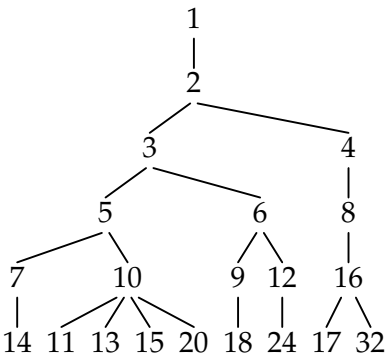
One important special case that has been studied extensively in the literature concerns the computations of powers $f = x^k$ with $k \in \mathbb{N}$. Brauer gave the first algorithm to compute x^k using $\lg k + (1+o(1)) \lg k / \lg\lg k$ multiplications [4], where $\lg k := \lceil \log_2 k + 1 \rceil$. The idea is to compute the 2^b-adic expansion

$$k = \sum_{i \leqslant \ell} k_i 2^{bi}$$

of k for $b = \lceil \lg\lg k\,(1-\epsilon(k)) \rceil$. Here $\epsilon(k)$ stands for a suitable function with $\epsilon(k) < 1$ for all k and which tends very slowly to zero. One may for instance take $\epsilon(k) := 1 / (2 \lg\lg\lg k)$. Then computing x^i for $i = 1, \ldots, 2^b - 1$ takes time $o(\lg k / \lg\lg k)$. After that, setting $K_j := \sum_{i \geqslant j} k_i 2^{bi}$, computing $x^{K_j} = x^{k_i} \cdot (x^{K_{j+1}})^{2^b}$ in terms of $x^{K_{j+1}}$ requires b squarings and one multiplication. Since $\ell \sim \lg k / \lg\lg k$, the stated complexity bound follows. Erdős proved that this bound is essentially optimal [11].

In general, the computation of x^k using only multiplications can be modeled by an *addition chain* $k_1, \ldots, k_\ell$ for k. This means that $k = k_\ell$, where $k_1 < \cdots < k_\ell$ are positive integers such that $k_1 = 1$ and for each $i > 1$, there exist $i_1 \leqslant i_2 < i$ with $k_i = k_{i_1} + k_{i_2}$. In practice, we may compute addition chains of small length for all integers until k as follows. We start with the trivial addition chain for 1. Assuming that we have computed addition chains for all integers below k, we determine the smallest $k' < k$ such that the addition chain $k'_1, \ldots, k'_{\ell'}$ for k' is of smallest length with $k - k' \in \{k'_1, \ldots, k'_{\ell'}\}$. Then sorting $k'_1, \ldots, k'_{\ell'}, k - k'$ yields an addition chain for k.

For each k, we actually only need to store the corresponding k' in a table in order to efficiently recover the addition chain. The algorithm can also be represented by a labeled tree [8, Section 4.6.3] in which the node labeled by k has the node labeled by k' as its parent:



For instance, 1, 2, 3, 6, 9, 18 is an addition chain for 18. In our implementation, we precompute this table (or tree) up to a certain threshold. For k above the threshold we resort to a 2^b-adic method as above for a suitable value of b (see also below).

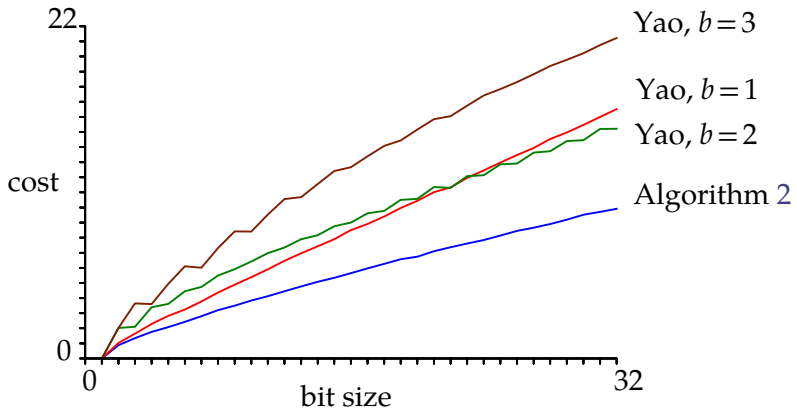


Figure 1. Computation of several powers using Yao's method and Algorithm 2.

2.2. Multiple powers

The 2^b-adic approach from the previous subsection generalizes to the computation of multiple powers instead of a single one. It was first shown by Yao [24] that m powers $f_1 = x^{k_1}, \ldots, f_m = x^{k_m}$ can be computed in time

$$\lg k + m\,(1+o(1))\lg k / \lg\lg k,$$

where $k = \max(k_1, \ldots, k_m)$. Yao's method consists of first computing $x, x^2, \ldots, x^{2^\ell}$ with $\ell := \max(\lfloor \log_2 k_1 \rfloor, \ldots, \lfloor \log_2 k_m \rfloor)$ and then to call the following algorithm m times for the computation of x^{k_i} for $i = 1, \ldots, m$:

Algorithm 1
Input: x in a multiplicative monoid $\mathcal{M}$, $k \in \mathbb{N}$ such that $k \geqslant 2^{982}$ and $x, x^2, \ldots, x^{2^{\lfloor \log_2 k \rfloor}}$.
Output: x^k.

1. Let $b := \lceil \log_2\log_2 k - 3\log_2\log_2\log_2 k \rceil$, $B := 2^b$ and let $k = \sum_{0 \leqslant i \leqslant t} k_i B^i$ stand for the B-adic expansion of k, with $k_i \in \{0, \ldots, B-1\}$.
2. For $j = 1, \ldots, B-1$ compute $y_j := \prod_{0 \leqslant i \leqslant t, k_i = j} x^{B^i}$.
3. Return $\prod_{1 \leqslant j < B} y_j^j$.

The condition $k \geqslant 2^{982}$ is used to ensure that $b \geqslant 1$, but steps 2 and 3 can still be applied to lower values of k, by taking $b := 1$ or $b := 2$ instead of the value from step 1. If $b = 1$, then the average number of bits of k is about $\frac{\log_2 k}{2}$, so Algorithm 1 performs about $\frac{\log_2 k}{2} - 1$ products. If $b = 2$, then the average number of digits k_i equal to a given $j = 1, 2, 3$ is roughly $\frac{\log_2 k}{8}$, so the number of products is $3\left(\frac{\log_2 k}{8} - 1\right)$ in step 2, and then 5 in step 3. Consequently, choosing $b = 2$ is advantageous over using $b = 1$ only when

$$3\left(\frac{\log_2 k}{8} - 1\right) + 5 < \frac{\log_2 k}{2} - 1,$$

which means $k \geqslant 2^{24}$. The number of products performed by Yao's method for $b = 1, 2, 3$ to compute m powers with random exponents in the range $[2^{k-1}, \ldots, 2^k - 1]$ for $k = 1, \ldots, 32$ is shown in Figure 1. The abscissa represents k, and the ordinate displays the number of multiplications divided by m (averaged over 100 runs).

The original condition $k \geqslant 2^{982}$ and the weakened condition $k \geqslant 2^{24}$ show that Yao's method is mostly of theoretical interest. For practical purposes, we therefore designed Algorithm 2 below, based on 2^b-adic expansions. It incorporates opportunistic optimizations whenever possible. As shown in Figure 1, this algorithm improves upon Yao's method for the exponent sizes that we are interested in.

Algorithm 2
Input: $k_1, \ldots, k_m \in \mathbb{N}$ with $1 \leqslant k_1 < \cdots < k_m$.
Output: $a_1, \ldots, a_\ell \in \mathbb{N}$ is an addition chain with $\{k_1, \ldots, k_m\} \subseteq \{a_1, \ldots, a_\ell\}$.

1. If $m = 1$ and $k_m < 2^{10}$, then use the algorithm from section 2.1.
2. Let $b := \lceil \lg \lg k_m \, (1 - \epsilon(k_m)) \rceil$.
3. Apply the algorithm recursively to the sorted elements of
$$\{2, \ldots, 2^{b-1}\} \cup \{k_i \operatorname{rem} 2^b : 1 \leqslant i \leqslant m\} \setminus \{0\}.$$
Let $r_1, \ldots, r_\rho$ be the result.
4. Apply the algorithm recursively to the sorted elements of $\{k_i \operatorname{quo} 2^b : 1 \leqslant i \leqslant m\} \setminus \{0\}$. Let $q_1, \ldots, q_\chi$ be the result.
5. Sort the list $r_1, \ldots, r_\rho, 2^b q_1, \ldots, 2^b q_\chi, k_1, \ldots, k_m$, remove multiple entries, and return the result.

Example 1. Taking $(k_1, k_2, k_3) = (6, 17, 35)$, Algorithm 2 returns the addition chain $1, 2, 3, 4, 6, 8, 16, 17, 32, 35$. An optimal addition chain is $1, 2, 4, 6, 8, 16, 17, 34, 35$.

Remark 2. Given $k_1, \ldots, k_m$, the problem of finding the absolute shortest addition chain $a_1, \ldots, a_\ell$ with $\{k_1, \ldots, k_m\} \subseteq \{a_1, \ldots, a_\ell\}$ is known to be NP-complete [10].

2.3. Multiple power products

The 2^b-adic approach can also be used for the computation of power products. This was already realized by Straus [23], who showed that $f = x_1^{k_1} \cdots x_n^{k_n}$ can be computed using $\lg k + (n + o(1)) \lg k / \lg \lg k$ multiplications, where $k = \max(k_1, \ldots, k_n)$, and provided that $n = O(1)$. The similarity between the complexities of Straus' and Yao's algorithms is not accidental, since both problems are equivalent due to a suitable application of the transposition principle. For more details and historical references, we refer to Bernstein's survey [2].

Another interesting observation due to Straus is that, for the computation of a single power product $x_1^{k_1} \cdots x_n^{k_n}$, the naive strategy to compute and multiply $x_1^{k_1}, \ldots, x_n^{k_n}$ is typically not the fastest. In our implementation, for small exponents, we found it simpler to simultaneously replace all powers x_i^e that occur in our vector $f = (f_1, \ldots, f_m)$ of sparse polynomials by new variables. For each variable x_i, this requires us to call Algorithm 2 for the list of all exponents e such that x_i^e occurs in f. Consequently, we do not benefit from the additional improvements that Straus provides for large exponents; however, we describe an alternate practical strategy in section 3.2.

In fact, Pippenger also proposed an algorithm [20, 21, 22] for evaluating multiple power products $f_1 = x_1^{k_{1,1}} \cdots x_n^{k_{1,n}}, \ldots, f_m = x_1^{k_{m,1}} \cdots x_n^{k_{m,n}}$. This algorithm is more elaborate to implement and we again refer to Bernstein's survey [2] for an overview. In section 3.4, we will present an efficient and easily implementable alternative for a key step in Pippenger's algorithm for the case when $k_{i,j} \in \{0, 1\}$ for all i, j.

2.4. Multivariate Horner schemes

Let us now return to the problem of computing an SLP for evaluating a single sparse polynomial (1). In computer algebra systems, sparse polynomials are often represented recursively as polynomials in a single variable x_i, whose coefficients are sparse polynomials in the remaining variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. This allows us to evaluate each of the univariate polynomials in this representation using Horner's method (or using a variant of Horner's method if the univariate polynomials are themselves represented in a sparse manner).

Another option is to decompose $f = c + x_1 f_1 + \cdots + x_n f_n$ with $c \in \mathbb{A}$ and where some of the terms may be zero and left out. We may then recursively expand $f_1, \ldots, f_n$ in a similar manner. Of course, we may also impose additional constraints on $f_1, \ldots, f_n$. For instance, if x_i is our main expansion variable as above, then we may require $f_1, \ldots, f_{i-1}, f_{i+1}, \ldots, f_n$ not to depend on x_i.

Such multivariate Horner schemes have been studied in several works [6, 7]. One major issue concerns the choice of the main expansion variable x_i and the subsequent choices of the main expansion variables for all recursive coefficients.

In lucky cases, typically when the support of f is rather dense, multivariate Horner schemes may return an SLP of size $O(t_f)$. However, on many examples from our data base, the size of the computed SLP is closer to b_f. Multivariate Horner schemes nonetheless remain a good "naive way" to evaluate sparse polynomials, since they are easy to implement and always outperform brute force evaluation of the expression (1).

2.5. Bilinear schemes

Consider a sparse polynomial $f = \sum_{i=1}^{t} c_i x^{e_i}$ with support $S := \{e_i : 1 \leqslant i \leqslant t\} \subseteq \mathbb{N}^n$. If f has a "dense flavor", then it is often possible to subdivide the set of variables $\{x_1, \ldots, x_n\}$ into two subsets, say $V_1 = \{x_1, \ldots, x_k\}$ and $V_2 = \{x_{k+1}, \ldots, x_n\}$, in such a way that the projected supports

$$\begin{aligned} S_1 &:= \{(\alpha_1, \ldots, \alpha_k) : \alpha \in S\} \\ S_2 &:= \{(\alpha_{k+1}, \ldots, \alpha_n) : \alpha \in S\} \end{aligned}$$

are significantly smaller than S. For instance, if f is a generic polynomial in $n = 2k$ variables of total degree d, so that $S = \{(\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n : \alpha_1 + \cdots + \alpha_n \leqslant d\}$, then $|S| = \binom{n+d}{n}$ and $|S_1| = |S_2| = \binom{k+d}{k}$. If n is fixed and $d \to \infty$, then this yields $|S| \sim d^n / n!$ and

$$|S_1| = |S_2| \sim d^k / k! \ll |S|.$$

In this favorable situation, we may first compute the power products $x_1^{\alpha_1} \cdots x_k^{\alpha_k}$ for all $\alpha \in S_1$, as well as the power products $x_{k+1}^{\beta_1} \cdots x_n^{\beta_{n-k}}$ for all $\beta \in S_2$, and then deduce all power products x^{e_i} for $i = 1, \ldots, t$ using only t additional multiplications. Assuming that $|S_1| + |S_2| = o(t)$, the entire evaluation of f can then be done using $(1 + o(1))\, t$ multiplications and t FMA instructions; Here an FMA instruction is of the form $\pm a\, b \pm c$. One may view this method as re-interpreting f as a bilinear polynomial in the variables $x_1^{\alpha_1} \cdots x_k^{\alpha_k}$ and $x_{k+1}^{\beta_1} \cdots x_n^{\beta_{n-k}}$.

Remark 3. Although this scheme is not expected to be efficient for general purposes, it becomes very attractive whenever the ring operations in $\mathbb{A}$ can be performed in a suitable FFT model or using modular arithmetic. In that case, the transforms (FFT transforms or modular reductions) of the coefficients c_i can be precomputed. For the evaluation of f, we then compute the power products $x_1^{\alpha_1} \cdots x_k^{\alpha_k}$ and $x_{k+1}^{\beta_1} \cdots x_n^{\beta_{n-k}}$ as above, along with their transforms. We next do the multiplications $(x_1^{\alpha_1} \cdots x_k^{\alpha_k}) \cdot \left(x_{k+1}^{\beta_1} \cdots x_n^{\beta_{n-k}}\right)$ in the transformed model, as well as all the FMAs. We finally transform the result back to the final value in $\mathbb{A}$. The bulk of the computation then reduces to t multiplications and t FMAs in the transformed model. This technique was proposed in [13, section 4.4.3].

Example 4. Assume that $\mathbb{A} = \mathbb{K}[z]/(z^r)$ is a ring of power series over a field $\mathbb{K}$ and assume that the c_i are actually in $\mathbb{K}$. Assuming that $\mathbb{K}$ contains a primitive root of unity ω of smooth order $\rho \geqslant 2r$, then we may use FFTs of length ρ to transform elements of $\mathbb{A}$ into vectors in $\mathbb{K}^\rho$. If the c_i are in $\mathbb{A}$, then we rather need an ω of smooth order $\rho \geqslant 3r$ instead. If we subdivide our set of variables into $\kappa > 2$ instead of two sets, then similar ideas still apply, but we need an ω of smooth order $\rho \geqslant (\kappa+1)\, r$. Similar remarks apply for modular arithmetic.

2.6. Partial factorizations

The smallest SLP that evaluates a given sparse polynomial may be much smaller than the input polynomial. One example is the polynomial

$$f = (x_1 + 2\,x_2 - x_3)^{2^{10}} - (x_1 + x_2^2 - 4\,x_3)^{2^9} \tag{2}$$

from the introduction, where $(x_1 + 2\,x_2 - x_3)^{2^{10}}$ and $(x_1 + x_2^2 - 4\,x_3)^{2^9}$ can be computed using repeated squarings. Unfortunately, finding such SLPs from their expanded representations is very hard in general. Nonetheless, it is often possible to find at least some partial factorizations that may help to speed up the evaluation.

The problem of factoring sparse polynomials in the traditional mathematical sense has been studied extensively. We refer to [9] for some recent algorithms and historical references. However, the kind of "factorizations" as in (2) that we are really after are not exclusively multiplicative, but may involve both additions and multiplications. In fact, the multivariate Horner schemes that we discussed in section 2.4 can be regarded as easy-to-compute additive-multiplicative factorizations, by recursively factoring out single variables. We will also use this kind of factorizations in section 3.3 below.

In [18], a reasonably efficient algorithm has been presented for finding "syntactic factorizations", in which none of the terms of the expanded products and sums overlap. For instance, $(a+b)\,(c+d) + a\,(a+b)$ is a syntactical factorization of $a^2 + a\,b + a\,c + a\,d + b\,c + b\,d$, but $(a+b)\,(c+d) + (a+c)\,(b+d)$ is not a syntactical factorization of $a\,b + a\,c + 2\,a\,d + 2\,b\,c + b\,d + c\,d$, because of the two "overlapping" terms $a\,d$ and $b\,c$. The algorithm from [18] works well for applications in which syntactic factorizations are likely to occur, such as input polynomials that are themselves the expanded result of some algebraic computation (in [18] a generic resultant was used as the running example).

In this paper, we focus on SLP transforms which can be computed in quasi-linear time or almost, so we regard the computation of more clever additive-multiplicative factorizations as a more or less independent problem to which we plan to return in future work.

3. THE NEW ALGORITHM

Let $x = (x_1, \ldots, x_n)$ be the input variables. Consider a vector $f = (f_1, \ldots, f_m)$ of sparse polynomials with

$$f_i = \sum_{1 \leqslant j \leqslant t_i} c_{i,j} x^{e_{i,j}}, \qquad i = 1, \ldots, m.$$

In this section, we present our main algorithm to compute an SLP for the efficient evaluation of $f : \mathbb{K}^n \longrightarrow \mathbb{K}^m$ as a polynomial map. In brief, our algorithm proceeds as follows:

1. All powers of individual variables are evaluated and replaced with new variables; coefficients are also replaced by new variables. Consequently, each term of the polynomial is expressed as a product of distinct variables.
2. Horner-type factorizations are applied to the polynomials obtained in step 1.
3. We collect all remaining products of distinct variables that need to be evaluated and use a greedy algorithm to evaluate them simultaneously.
4. The final straight-line program is then simplified and optimized for the target hardware.

3.1. Replacing coefficients and powers by new variables

The first step of our algorithm is a reduction to the case where $c_{i,j} = 1$ and $e_{i,j} \in \{0, 1\}$ for all i, j. We do this by first introducing new variables $x_{n+1}, x_{n+2}, \ldots, x_m$ for all constants $c_{i,j} \neq 1$. Whenever a constant occurs multiple times, we understand that we introduce a single variable to represent all these identical constants.

For each variable x_k, we next collect the set $\mathscr{E}_k = \{(e_{i,j})_k : 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant t_i\}$ of all non-zero exponents α such that x_k^α occurs in one of the terms of one of the f_i. All sets $\mathscr{E}_k$ can be determined jointly using a hash table and one linear pass. Whenever $\mathscr{E}_k \neq \emptyset$, we next use Algorithm 2 to compute an addition chain for the sorted list of elements of $\mathscr{E}_k$. This addition chain gives rise to an SLP of the same length minus one for the computation of x_k^α for all $\alpha \in \mathscr{E}_k$. For each x_k^α with $\alpha \neq 1$, we introduce a new variable $x_{m+1}, x_{m+2}, \ldots$, and use the latter SLPs to compute these new variables as a function of $x_1, \ldots, x_n$.

Example 5. As our running example, consider $n = 5$ and

$$\begin{aligned} f_1 &:= 2x_1x_3^3x_4^5 + x_2x_3^3x_4^7x_5^2 + 2x_3^4x_4^8x_5 \\ f_2 &:= 2x_2x_3^3x_4^5 + x_1x_3^3x_4^7x_5^2 + 2x_1x_3^3x_4^7x_5^3 + 3x_1x_3^4x_4^6x_5. \end{aligned} \tag{3}$$

We first introduce new variables $x_6 := 2$ and $x_7 := 3$ for the constants. We next need to introduce new variables for all non-trivial powers. For the variable x_3, we need to compute x_3^3 and x_3^4, which we do using Algorithm 2. This yields $x_8 := x_3 \cdot x_3$, $x_9 := x_3 \cdot x_8$, $x_{10} := x_8 \cdot x_8$, so $x_3^3 =: x_9$ and $x_3^4 =: x_{10}$. We next need to compute x_4^5, x_4^6, x_4^7, and x_4^8, which is done as follows: $x_{11} := x_4 \cdot x_4$, $x_{12} = x_{11} \cdot x_{11}$, $x_{13} := x_4 \cdot x_{12}$, $x_{14} := x_4 \cdot x_{13}$, $x_{15} := x_4 \cdot x_{14}$, $x_{16} := x_{12} \cdot x_{12}$. We finally need to compute x_5^2 and x_5^3, which is done using $x_{17} := x_5 \cdot x_5$, $x_{18} := x_5 \cdot x_{17}$. After the introduction of these new variables, our sparse polynomials are rewritten as

$$\begin{aligned} f_1 &:= x_1x_6x_9x_{13} + x_2x_9x_{15}x_{17} + x_5x_6x_{10}x_{16} \\ f_2 &:= x_2x_6x_9x_{13} + x_1x_9x_{15}x_{17} + x_1x_6x_9x_{15}x_{18} + x_1x_5x_7x_{10}x_{14}. \end{aligned} \tag{4}$$

3.2. Binary expansion of exponents

We have just shown how to reduce our evaluation problem to the case where all input polynomials

$$f = \sum_{1 \leqslant i \leqslant t} c_i x_1^{e_{i,1}} \cdots x_n^{e_{i,n}}$$

are sums of products of distinct variables. Instead of introducting a distinct variable for each individual power x_i^j, it is also possible to further factor such powers as products of powers of the form $x_i^{2^k}$. For this, we compute the 2-adic expansions of the exponents

$$e_{i,j} = \sum_{k \geqslant 0} e_{i,j,k} 2^k.$$

Setting $y_{j,k} := x_j^{2^k}$, we then have

$$f = \sum_{1 \leqslant i \leqslant t} c_i \prod_{j,k \geqslant 0} y_{j,k}^{e_{i,j,k}}. \tag{5}$$

As in the previous subsection, we introduce new variables for the coefficients $c_i \neq 1$, whereas the evaluation of the $y_{j,k}$ is done directly using binary exponentiation instead of Algorithm 2.

3.3. Factoring out variables in a greedy manner

After applying one of the two rewriting algorithms described in the previous subsections, the input polynomials $f_1, \ldots, f_m$ become sums of products of variables. Although finding the best possible partial factorizations is a hard problem in general, we still can go for "low hanging fruit", by recursively factoring out single variables x_i from the polynomials f_k as long as possible.

More precisely, for each polynomial f_k, let x_i be a variable such that the number of terms $\nu_{k,i}$ that depend on x_i is maximal. If $\nu_{k,i} > 1$ and $\nu_{k,i} \geqslant \epsilon\, t_i$ for some small number $\epsilon > 0$ (say $\epsilon := 1/10$), then we decompose $f_k = f_k' x_i + f_k''$ and replace f_k by the two polynomials f_k' and f_k'' in the list $f_1, \ldots, f_m$ (or just by f_k' if $f_k'' = 0$). We keep doing this until no such x_i exists. Due to the threshold $\epsilon > 0$, this algorithm runs in quasi-linear time.

Example 6. Applying this strategy to the polynomials (4) leads to the factorizations

$$\begin{aligned} f_1 &:= (x_1 x_6 x_{13} + x_2 x_{15} x_{17})\, x_9 + x_5 x_6 x_{10} x_{16} \\ f_2 &:= ((x_2 x_{13} + x_1 x_{15} x_{18})\, x_6 + x_1 x_{15} x_{17})\, x_9 + x_1 x_5 x_7 x_{10} x_{14}. \end{aligned}$$

The algorithm of the next subsection will compute the values of $x_1 x_6 x_{13}$, $x_2 x_{15} x_{17}$, $x_5 x_6 x_{10} x_{16}$, $x_2 x_{13}$, $x_1 x_{15} x_{18}$, $x_1 x_{15} x_{17}$, $x_1 x_5 x_7 x_{10} x_{14}$ into the variables x_{25}, x_{30}, x_{28}, x_{32}, x_{26}, x_{23}, x_{31}; see Example 9. In order to compute f_1 and f_2 it remains to perform the following instructions:

$$\begin{aligned} x_{33} &:= x_{25} + x_{30} \\ x_{34} &:= \mathrm{fma}(x_9, x_{33}, x_{28}) \\ x_{35} &:= x_{32} + x_{26} \\ x_{36} &:= \mathrm{fma}(x_6, x_{35}, x_{23}) \\ x_{37} &:= \mathrm{fma}(x_9, x_{36}, x_{31}). \end{aligned}$$

After this, we have $f_1 = x_{34}$ and $f_2 = x_{37}$. Here $\mathrm{fma}(a, b, c) := a\, b + c$.

3.4. Greedy evaluation of a list of products

In this subsection, we turn to the core subalgorithm. As input, we have a list $\mathscr{S}_1,\dots,\mathscr{S}_\ell$ of pairwise distinct subsets of $\{x_1,\dots,x_n\}$. As output, we produce an SLP that takes $x_1,\dots,x_n$ as input and that computes $\prod_{v\in\mathscr{S}_i} v$ for $i=1,\dots,\ell$. Without loss of generality we may assume that $n=O(\sum_{1\leqslant k\leqslant \ell}|\mathscr{S}_k|)$.

For the construction of our SLP we use a simple greedy approach: as long as the list $\mathscr{S}_1,\dots,\mathscr{S}_\ell$ contains two entries $\mathscr{S}_k$ with $|\mathscr{S}_k|\geqslant 2$, we determine variables x_i, x_j for which $|\{1\leqslant k\leqslant \ell : \{x_i,x_j\}\subseteq\mathscr{S}_k\}|$ is maximal. We then introduce a new variable $x_{n+1}:=x_i\cdot x_j$ and replace $\{x_i,x_j\}$ by $\{x_{n+1}\}$ in the subsets $\mathscr{S}_1,\dots,\mathscr{S}_\ell$ and increase n by one. As soon as $|\mathscr{S}_k|=1$ for all k, the products $\prod_{v\in\mathscr{S}_k} v$ all become trivial. The algorithm runs as follows.

Algorithm 3
Input: pairwise distinct subsets $\mathscr{S}_1,\dots,\mathscr{S}_\ell$ of $\{x_1,\dots,x_n\}$.
Output: an SLP for the computation of $\prod_{v\in\mathscr{S}_i} v$ for $i=1,\dots,\ell$.

1. Start a new SLP P.
2. Create a binary search tree T that associates to each pair (x_i,x_j) with $i<j$ and $\{x_i,x_j\}\subseteq\mathscr{S}_k$ for some k, the set $T[x_i,x_j]$ of all k with $\{x_i,x_j\}\subseteq\mathscr{S}_k$.
3. Create a heap H with triples $(N_{i,j},x_i,x_j)$, where $N_{i,j}=|T[x_i,x_j]|$, ordered via $N_{i,j}$.
4. As long as the heap H is non-empty, do the following:
   a. Extract and remove the highest triple $(N_{i,j},x_i,x_j)$ from H.
   b. If $N_{i,j}>|T[x_i,x_j]|$, then continue the loop (with the next highest triple from H).
   c. Create a new variable x_{n+1} and append the instruction $x_{n+1}:=x_i\cdot x_j$ to P.
   d. For all $k\in T[x_i,x_j]$, update $\mathscr{S}_k:=\mathscr{S}_k\cup\{x_{n+1}\}\setminus\{x_i,x_j\}$, update the entries $T[x_{i'},x_{j'}]$ with $x_{i'},x_{j'}\in\mathscr{S}_k$ and such that at least one among $x_{i'}$ and $x_{j'}$ is in $\{x_i,x_j,x_{n+1}\}$, and insert all new triples $(N_{i',j'},x_{i'},x_{j'})$ into H.
   e. Update $n:=n+1$.
5. For $k=1,\dots,l$, append the naive evaluation of $\prod_{v\in\mathscr{S}_i} v$ to P, and use this product as the k-th output value of P.
6. Return P.

PROPOSITION 7. *Algorithm 3 is correct and runs in time $\tilde{O}(\sum_{1\leqslant k\leqslant \ell}|\mathscr{S}_k|^2)$. The SLP in return performs at most $\sum_{1\leqslant k\leqslant \ell}(|\mathscr{S}_k|-1)$ products.*

Proof. Let $S=\sum_{1\leqslant k\leqslant \ell}|\mathscr{S}_k|$. Steps 2 and 3 perform $O((\sum_{1\leqslant k\leqslant \ell}|\mathscr{S}_k|^2)\log S)$ operations on integers of bit size $O(\log n)=O(\log S)$. If $N_{i,j}>|T[x_i,x_j]|$ holds in step 4b, then the triple is no longer valid. Let m denote the value of n at the end of the algorithm.

The bit cost of each update caused by $\mathscr{S}_k$ in step 4d is $O(|\mathscr{S}_k|\log S\log m)$. Moreover, each $\mathscr{S}_k$ can be updated at most $|\mathscr{S}_k|$ times. Consequently, $\log m=O(\log S)$. Hence, the total cost is $\tilde{O}(\sum_{1\leqslant k\leqslant \ell}|\mathscr{S}_k|^2)$. □

Remark 8. In practice a hash table is used for T in Algorithm 3. Assuming perfect hashing the asymptotic complexity is the same.

Example 9. Continuing our Example 6, let us take

$$\begin{array}{lll} \mathscr{S}_1 := \{x_1, x_6, x_{13}\} & & \mathscr{S}_4 := \{x_2, x_{13}\} \\ \mathscr{S}_2 := \{x_2, x_{15}, x_{17}\} & & \mathscr{S}_5 := \{x_1, x_{15}, x_{17}\} \\ \mathscr{S}_3 := \{x_5, x_6, x_{10}, x_{16}\} & & \mathscr{S}_6 := \{x_1, x_{15}, x_{18}\} \\ & & \mathscr{S}_7 := \{x_1, x_5, x_7, x_{10}, x_{14}\}. \end{array} \tag{6}$$

The two variables x_{15} and x_{17} appear jointly in two of the $\mathscr{S}_k$. The algorithm starts by picking $(2, x_{15}, x_{17})$ in step 4a, so we append $x_{19} := x_{15} \cdot x_{17}$ to the SLP, and also update $\mathscr{S}_2 := \{x_2, x_{19}\}$, $\mathscr{S}_5 := \{x_1, x_{19}\}$, and the table T. We next pick $(2, x_5, x_{10})$ in step 4a, append $x_{20} := x_5 \cdot x_{10}$ to the SLP, and also update $\mathscr{S}_3 := \{x_6, x_{10}, x_{20}\}$, $\mathscr{S}_7 := \{x_1, x_7, x_{14}, x_{20}\}$, and the table T. We next pick $(2, x_1, x_{15})$ from the heap, but since $T[x_1, x_{15}]$ has been updated to $T[x_1, x_{15}] = 1$, we ignore this pick in step 4b and directly continue with the next triple on the heap. From this point on, we only retrieve triples $(N_{i,j}, x_i, x_j)$ with $N_{i,j} = 1$ and further append the following instructions to the SLP (while indicating their values in terms of $x_1, \ldots, x_{18}$ in grey):

$$\begin{array}{llll} x_{21} := x_1 \cdot x_6 & = x_1 x_6 & x_{27} := x_{14} \cdot x_{20} & = x_5 x_{10} x_{14} \\ x_{22} := x_1 \cdot x_{18} & = x_1 x_{18} & x_{28} := x_{16} \cdot x_{24} & = x_5 x_6 x_{10} x_{16} \\ x_{23} := x_1 \cdot x_{19} & = x_1 x_{15} x_{17} & x_{29} := x_1 \cdot x_{27} & = x_1 x_5 x_{10} x_{14} \\ x_{24} := x_6 \cdot x_{20} & = x_5 x_6 x_{10} & x_{30} := x_2 \cdot x_{19} & = x_2 x_{15} x_{17} \\ x_{25} := x_{13} \cdot x_{21} & = x_1 x_6 x_{13} & x_{31} := x_7 \cdot x_{29} & = x_1 x_5 x_7 x_{10} x_{14} \\ x_{26} := x_{15} \cdot x_{22} & = x_1 x_{15} x_{18} & x_{32} := x_2 \cdot x_{13} & = x_2 x_{13} \end{array}$$

The output variables are x_{25}, x_{30}, x_{28}, x_{32}, x_{23}, x_{26}, x_{31}.

3.5. Simplification and polishing

As our final step, we try to combine as many additions with multiplications as possible into FMA instructions, and run common subexpression elimination on the result. Optionally, we may reschedule some of the instructions and try to reduce the number of variables that are being used.

Example 10. For our running Example 5, it is possible to group two pairs of additions and multiplications into FMA instructions. We may also rewrite multiplications $x_i := x_j \cdot x_j$ as squarings $x_i := x_j^2$ (which is more efficient for certain rings $\mathbb{A}$). On the other hand, common subexpression elimination leads to no further simplifications. The final SLP is:

$$\begin{array}{lll} \mathrm{in}(x_1, \ldots, x_5) & & \\ x_{34} := 2 & x_{14} := x_{10}^2 & x_{24} := x_{12} \cdot x_{18} \\ x_{35} := 3 & x_{15} := x_5^2 & x_{25} := x_{14} \cdot x_{22} \\ x_6 := x_3^2 & x_{16} := x_5 \cdot x_{15} & x_{26} := x_1 \cdot x_{24} \\ x_7 := x_3 \cdot x_6 & x_{17} := x_{13} \cdot x_{15} & x_{27} := x_2 \cdot x_{17} \\ x_8 := x_7^2 & x_{18} := x_5 \cdot x_8 & x_{28} := x_{26} \cdot x_{35} \\ x_9 := x_4^2 & x_{19} := x_1 \cdot x_{34} & x_{29} := \mathrm{fma}(x_{11}, x_{19}, x_{27}) \\ x_{10} := x_9^2 & x_{20} := x_1 \cdot x_{16} & x_{30} := \mathrm{fma}(x_7, x_{29}, x_{25}) \\ x_{11} := x_4 \cdot x_{10} & x_{21} := x_1 \cdot x_{17} & x_{31} := \mathrm{fma}(x_2, x_{11}, x_{23}) \\ x_{12} := x_4 \cdot x_{11} & x_{22} := x_{18} \cdot x_{34} & x_{32} := \mathrm{fma}(x_{34}, x_{31}, x_{21}) \\ x_{13} := x_4 \cdot x_{12} & x_{23} := x_{13} \cdot x_{20} & x_{33} := \mathrm{fma}(x_7, x_{32}, x_{28}) \\ & & \mathrm{out}(x_{30}, x_{33}) \end{array}$$

4. IMPLEMENTATION

We implemented our new algorithm within the JIL library [1, 15], GIT version d390808a9b9834f64a73bb823d7745cd806c0dc0. JIL is a free C++ software library for HPC computations with SLPs. It provides a convenient interface for automatic differentiation, common sub-expression elimination, lifting, transposition, and many other features. It also includes a JIT (Just In Time) compiler dedicated to SLPs. This compiler can generate executable machine code directly in memory, which is useful when the same SLP needs to be evaluated $N \gg 1$ multiple times efficiently. Typical applications include numerical integration and polynomial system solving.

The JIT compiler of JIL is one or two orders of magnitude faster than general-purpose compilers such as GCC or CLANG. Consequently, JIT compilation becomes advantageous even for relatively small values of N like $N \approx 1000$. In addition, if several competing SLPs are available for the same task, then we can generate machine code for each of them and empirically select the best SLP.

The source code of our evaluation can be found in the file `src/slp/slp_spol.cpp` of JIL. Examples are in `check/slp/slp_spol_check.cpp` and benchmarks in `bench/slp/slp_spol_bench.cpp`.

4.1. Implemented strategies

Currently, our implementation supports 32-bit integer exponents represented by sparse vectors. The SLPs produced by our software include the following operations: negation, binary addition, subtraction, and possibly negated multiplication $\pm a\,b$, as well as ternary FMA instructions $\pm a\,b \pm c$. Our goal is to minimize the length L (i.e. the number of operations) of the generated SLP. Below we analyze the obtained results for the following strategies.

Sparse. This strategy corresponds to our new polynomial evaluation algorithm from section 3, while using the algorithm from section 3.1 for the reduction to the case of sums of powers of distinct variables.

Exponent expansion. This is a variant of the "sparse" strategy in which we use the algorithm from section 3.2 instead of the one from section 3.1.

Horner. Here, the input polynomial f is considered in $\mathbb{A}[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n][x_i]$, where i maximizes the number s of distinct exponents $r_1 < \cdots < r_s$ of x_i among the non-zero terms of f:

$$f = \sum_{1 \leqslant j \leqslant s} g_j(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)\, x_i^{r_j}. \tag{7}$$

We apply this Horner strategy recursively to evaluate all the $g_j(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_n)$. Then the univariate Horner scheme is applied to obtain the value of f using

$$f = x_i^{r_1}(g_1(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) + x_i^{r_2-r_1}(g_2(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) + \cdots)).$$

The powers $x_i^{r_1}, x_i^{r_2-r_1}, \ldots, x_i^{r_s-r_{s-1}}$ are computed using Algorithm 2.

Greedy Horner. We have implemented an improved variant of the strategy presented in [7, section 3]: selecting the variable x_i as in the above "Horner" strategy, we write

$$f = g_0(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) + g_1(x_1, \ldots, x_n)\, x_i^r,$$

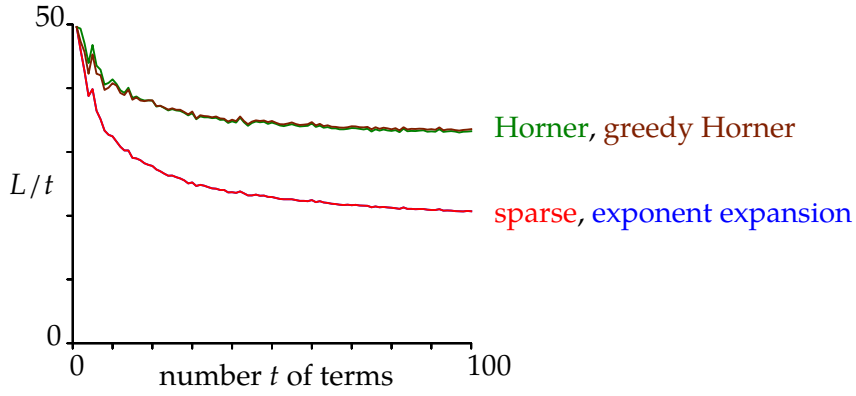


Figure 2. Ratio L/t as a function of t, where L is the length of the SLP generated using different strategies, for random polynomials with t terms in 100 variables and exponents in $\{0,1,2\}$.

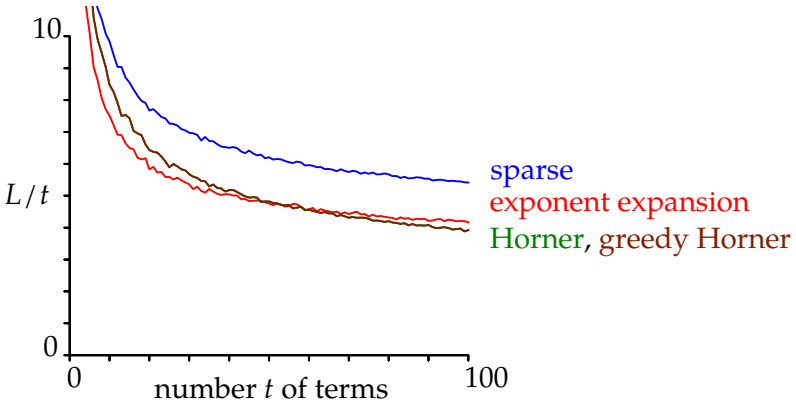


Figure 3. Ratio L/t as a function of t, where L is the length of the SLP generated using different strategies, for random univariate polynomials of degree $\leqslant 10^6$ with t terms.

where g_0 contains all the terms of f that are not multiple of x_i and g_1 is not divisible by x_i. The polynomials g_0 and g_1 are evaluated recursively. We collect all powers x_i^r from all recursive evaluations upfront and evaluate them using Algorithm 2.

4.2. Sparse random polynomials

The strategies from section 3 were designed specifically for polynomials with few terms of small degree. In particular, we considered the case of random polynomials with t terms in $n = 100$ variables and exponents in $\{0,1,2\}$. Figure 2 displays the length L of the generated SLP divided by t as a function of t. The cost is averaged over 10 random samples. The costs of the "sparse" and "exponent expansion" strategies are actual identical in this special case. Those of "Horner" and "greedy Horner" are almost the same.

Figure 3 displays similar cost measures with $n = 1$ and exponents $\leqslant 10^6$ constructed randomly as follows: we first select a bit size s in $[1, 20]$ at random and then pick a random number uniformly in $[2^{s-1}, \min(2^s, 10^6)]$. We observe that the "sparse" strategy is ineffective. With fewer than about 50 terms, the "exponent expansion" strategy performs best; thereafter, the "Horner" strategy (which coincides with the "greedy Horner" strategy) becomes more efficient. With a few variables (e.g., 2, 3, 4), the comparisons are similar, with larger thresholds (50 becomes approximately 200 for $n = 2$).

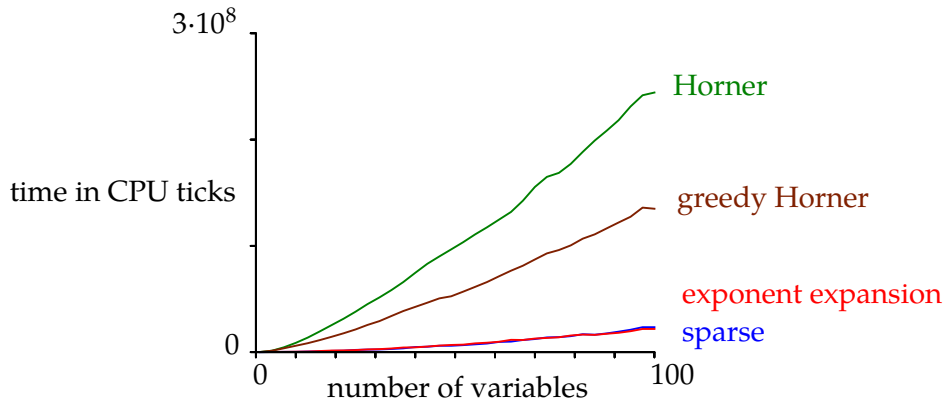


Figure 4. Times for builing SLPs of random polynomials with 100 terms and coefficients in $\{0,1,2\}$.

Figure 4 displays the time needed to build the SLPs using the four strategies: we construct polynomials in n variables with at most 100 terms and random exponents in $\{0,1,2\}$. Coefficients are random 64 bit integers. The cost is averaged over 10 random samples. We observe that the Horner strategies are considerably less efficient than the sparse strategies when the number of variables is large.

4.3. Dense polynomials

The "Horner" and "greedy Horner" strategies coincide for univariate polynomials and are usually the most efficient. The "exponent expansion" method is only slightly slower, while the "sparse" strategy is suboptimal in this situation. Figure 5 shows averaged ratios L/t as a function of t (for 10 random samples), this time for random bivariate dense polynomials of partial degrees $\leqslant 31$. The behavior is roughly similar to what we observed in the univariate case. For trivariate polynomials of partial degrees $\leqslant 9$, the four strategies have closer efficiencies.

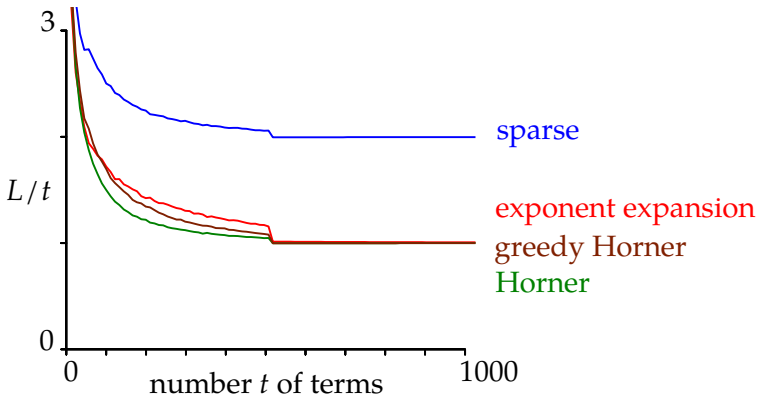


Figure 5. Ratio L/t as a function of t, where L is the length of the SLP generated using different strategies, for random bivariate polynomials of partial degrees $\leqslant 31$.

4.4. Our final "combined" strategy

Considering the relative performance of the four strategies described above, and noting that the construction of an SLP via the Horner methods tends to become more expensive for sparse polynomials, we retained a "combined" default strategy for the evaluation of

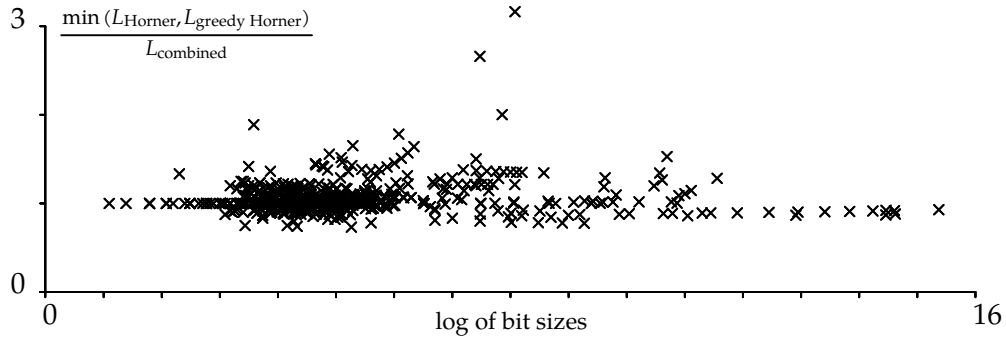


Figure 6. Relative performance of the "combined" strategy across the database.

sparse polynomials in JIL. More precisely, our implementation selects the shortest SLP among the ones produced via the following strategies:

1. The first strategy is "exponent expansion".
2. The second strategy performs a single step of the "Horner" strategy (i.e. with respect to a single variable), followed by the "exponent expansion" strategy (where we evaluate all the g_j of equation (7) simultaneously).
3. The third strategy performs two steps of the "Horner" strategy, followed by the "exponent expansion" strategy: after the first step of the "Horner" strategy and we evaluate all the g_j of (7) simultaneously using the second strategy. All the g_j are considered as univariate polynomials in the same variable x_j, where j maximizes the number of distinct exponents of x_j among the union of the non-zero terms of $g_1, \ldots, g_s$. Of course, we only apply this third strategy when the second strategy produces a shorter SLP than the first one.

This "combined" strategy is almost always best for sparse multivariate polynomials. For denser polynomial, it still performs well, since the density is typically captured by one and sometimes two of the variables. The most expensive step of computing the SLP using the "combined" strategy is Algorithm 3. The other steps run in quasi-linear time.

To benchmark the efficiency of our "combined" strategy, we collected 611 polynomial systems with integer coefficients primarily from the SYMBOLICDATA project [12] and the MSOLVE example suite [3]. These systems are available at `https://sourcesup.renater.fr/projects/jil-examples`. We measured the lengths L_{combined}, L_{Horner}, and $L_{\text{greedy Horner}}$ of the SLPs produced by the "combined", "Horner", and "greedy Horner" strategies. In Figure 6, each polynomial system is represented by a cross: the abscissa shows the logarithm of the bit size of the system, and the ordinate shows $\min(L_{\text{Horner}}, L_{\text{greedy Horner}}) / L_{\text{combined}}$.

After a closer examination of the timings, we observed that relatively dense homogeneous bivariate polynomials are better suited for the "exponent expansion" method than for Horner's method. On the other hand, the points with low ordinates in Figure 6 often correspond to systems with a very specific structure, for example those labeled `Singular_rcyclic` in the database. These rare cases are where the "greedy Horner" method outperforms "combined". Of course, in our software implementation, the user can select the strategy and even invoke an intensive search for optimal combinations. We are also investigating variants of our algorithms that use more aggressive factorizations in the same vein as [18]. We plan to detail this work in a future paper.

BIBLIOGRAPHY

[1] A. Ahlbäck, J. van der Hoeven, and G. Lecerf. JIL: a high performance library for straight-line programs. https://sourcesup.renater.fr/projects/jil, 2025.

[2] D. J. Bernstein. Pippenger's exponentiation algorithm. Available at https://cr.yp.to/papers/pippenger.pdf, 2002.

[3] J. Berthomieu, Ch. Eder, and M. Safey El Din. Msolve: a library for solving polynomial systems. In *2021 International Symposium on Symbolic and Algebraic Computation*, ISSAC'21, pages 51–58. ACM.

[4] A. Brauer. On addition chains. *Bull. Amer. Math. Soc.*, 45(10):736–739, 1939.

[5] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag, 1997.

[6] J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Math. Comp.*, 54(231–243), 1990.

[7] M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004.

[8] D.E. Knuth . *The art of computing programming Vol.1, fundamental algorithms*. Addison-Wesley, 3rd edition, 1997.

[9] A. Demin and J. van der Hoeven. Factoring sparse polynomials fast. *J. Complexity*, 88:101934, 2025.

[10] P. Downey, B. Leong, and R. Sethi. Computing sequences with addition chains. *SIAM J. Comput.*, 10(3):638–646, 1981.

[11] P. Erdős. Remarks on number theory III. On addition chains. *Acta Arith.*, 6(1):77–81, 1960.

[12] H.-G. Gräbe. The SymbolicData project. 2020. https://symbolicdata.github.io.

[13] J. van der Hoeven. Calcul analytique. In *Journées Nationales de Calcul Formel. 14 – 18 Novembre 2011*, volume 1 of *Les cours du CIRM*, pages 1–85. CIRM, 2011. https://www.numdam.org/articles/10.5802/ccirm.16/.

[14] J. van der Hoeven. *The Jolly Writer. Your Guide to GNU TeXmacs*. Scypress, 2020.

[15] J. van der Hoeven and G. Lecerf. Towards a library for straight-line programs. *Appl. Algebra Eng. Commun. Comput.*, 37:331–387, 2026.

[16] J. van der Hoeven et al. GNU TeXmacs. https://www.texmacs.org, 1998.

[17] R. Ilango. Constant depth formula and partial function versions of MCSP are hard. *SIAM J. Comput.*, 53(6):FOCS20–317, 2022.

[18] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Efficient evaluation of large polynomials. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software – ICMS 2010. Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010, Proceedings*, volume 6327 of *Lect. Notes Comput. Sci.*, pages 342–353. Springer, Berlin, Heidelberg, 2010.

[19] V. Ya Pan. Methods of computing values of polynomials. *Russ. Math. Surv.*, 21:105–136, 1966.

[20] N. Pippenger. On the evaluation of powers and related problems. In *17th Annual Symposium on Foundations of Computer Science (SFCS 1976)*, pages 258–263. IEEE Computer Society, 1976.

[21] N. Pippenger. The minimum number of edges in graphs with prescribed paths. *Math. Syst. Theory*, 12(1):325–330, 1978.

[22] N. Pippenger. On the evaluation of powers and monomials. *SIAM J. Comput.*, 9(2):230–250, 1980.

[23] E. G. Straus. Addition chains of vectors (problem 5125). *Am. Math. Mon.*, 71(7):806–808, 1964.

[24] A. C.-C. Yao. On the evaluation of powers. *SIAM J. Comput.*, 5(1):100–103, 1976.