A Note on Multilinear Polynomial Evaluation

Bryan R. Gillespie*

Department of Mathematics
Colorado State University
Fort Collins, CO, U.S.A.

Updated February 23, 2023

In the following we fix a field $\mathbb{F}$, which may be thought of as a familiar example such as the real numbers or the integers modulo a prime without sacrificing generality. Recall that a **polynomial** p in variables $x_1, \dots, x_k$ over the field $\mathbb{F}$ is a formal sum of the form

$$p(x_1, \dots, x_k) = \sum_{\alpha \in \mathbb{N}^k} c_\alpha x^\alpha,$$

where each **monomial** x^α, $\alpha = (\alpha_1, \dots, \alpha_k)$ represents the product $x_1^{\alpha_1} \dots x_k^{\alpha_k}$, and the **coefficients** $c_\alpha \in \mathbb{F}$ are nonzero for only finitely many exponent vectors α. The collection of these polynomials is a **polynomial ring** denoted $\mathbb{F}[x_1, \dots, x_k]$.

If $k = 1$ then p is called a **univariate** polynomial with **degree** equal to the largest d such that $c_{(d)}$ is nonzero, or $-\infty$ if p is the zero polynomial with all coefficients equal to zero.

If $k > 1$ then p is called a **k-variate** or **multivariate** polynomial with **total degree** equal to the largest d such that c_α is nonzero for some tuple α whose components sum to d, or $-\infty$ if p is the zero polynomial. The degree of a multivariate polynomial in terms of a variable x_j is defined as the largest d such that c_α is nonzero for some tuple α whose j-th component is d, or $-\infty$ if p is the zero polynomial.

The **evaluation** of a polynomial p at a point $a = (a_1, \dots, a_k) \in \mathbb{F}^k$ is the value $p(a) \in \mathbb{F}$ obtained by substituting a_i for each variable x_i in the formal sum of p and evaluating the resulting algebraic expression. This is well-defined because p has only finitely many nonzero coefficients.

In this note, we will focus on a particular class of polynomials which are useful in many computational applications, the *multilinear* polynomials.

1 Multilinear Polynomials

A polynomial $p \in R = \mathbb{F}[x_1, \dots, x_k]$ is called **multilinear** if for each j, the polynomial has degree at most 1 as a polynomial in x_j. The space $\mathrm{ML}(R)$ of multilinear polynomials in R

*Email: `Bryan.Gillespie@colostate.edu`

is a 2^k-dimensional $\mathbb{F}$-vector subspace of R which is spanned by the square-free monomials $x^\alpha, \alpha \in \{0,1\}^k$. In particular, a canonical representation for a multilinear polynomial is the collection of its coefficients when (uniquely) written as a sum of square-free monomials. This representation is called the **monomial representation** of p, which we denote by $\textsc{Monom}(p)$.

We will discuss several algorithms for efficiently evaluating multilinear polynomials. As a warm-up, we begin with the most straightforward algorithm, specialized to the monomial representation. Subsequently, we will extend our scope to a parametrized class of bases for $\mathrm{ML}(R)$ for which our algorithms will work equally well, which will include both the monomial basis representation and the important *Lagrange basis representation* as special cases.

Algorithm 1 Evaluating a multilinear polynomial p with monomial representation

```
function EVAL(c ← MONOM(p), r : F^k) → F
    ACC : F ← 0
    for α ∈ {0,1}^k do
        PROD ← c_α · ∏_{i:α_i=1} r_i
        ACC ← ACC + PROD
    return ACC
```

Algorithm 1 simply evaluates each monomial term of p in sequence, and adds the result to an accumulator to collect the final sum. A simple count shows that the algorithm uses $k2^{k-1}$ field multiplications and 2^k field additions, so for a polynomial with $n = 2^k$ coefficients, the algorithm runs in $O(n \log n)$ time and constant space. We will wait to fill in more details until we present Algorithm 2, which generalizes the above.

We now define a general class of vector space bases for the space $\mathrm{ML}(R)$ of multilinear polynomials, each instance of which provides a way to represent multilinear polynomials by recording the vector coefficients in terms of the basis. The building blocks for these bases are pairs of linear polynomials $a+bx, c+dx$ satisfying the condition $ad-bc \neq 0$. This condition ensures that the two polynomials are linearly independent over $\mathbb{F}$, and in particular this allows us to represent the polynomials 1 and x as a linear combination:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} a+bx \\ c+dx \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} 1 \\ x \end{pmatrix} = \begin{pmatrix} 1 \\ x \end{pmatrix}.$$

For each $i \in \{1, \dots, k\}$, let $a_i, b_i, c_i, d_i \in \mathbb{F}$ with $a_i d_i - b_i c_i \neq 0$, and for $\alpha \in \{0,1\}^k$, define the basis polynomial B_α by

$$B_\alpha(\mathbf{x}) \coloneqq \prod_{i=1}^{k} \big((1-\alpha_i)(a_i + b_i x_i) + \alpha_i(c_i + d_i x_i)\big). \tag{1}$$

In other words, B_α is a product of univariate linear polynomials, one for each variable x_i, where a value of 0 for α_i selects the polynomial $a_i + b_i x_i$, and a value of 1 for α_i selects the polynomial $c_i + d_i x_i$. We can the show the following.

Theorem 1. *The collection of polynomials $\big(B_\alpha\big)_{\alpha \in \{0,1\}^k}$ is an $\mathbb{F}$-basis of the space of multilinear polynomials in $\mathbb{F}[x_1, \dots, x_k]$.*

Proof. From the form of Equation 1, it is clear that each of the polynomials B_α is multilinear. It is thus sufficient to represent an arbitrary square-free monomial x^β as a linear combination of the polynomials B_α. For $j \in \{0, \ldots, k\}$ and $\alpha \in \{0,1\}^j$, let

$$B_\alpha^{(j)} \coloneqq \prod_{i=1}^{j} \big((1-\alpha_i)(a_i + b_i x_i) + \alpha_i(c_i + d_i x_i)\big) \prod_{i=j+1}^{k} x_i^{\beta_i}.$$

We see that each polynomial $B_\alpha^{(k)}$ is just the basis polynomial B_α, and that the single polynomial $B_{()}^{(0)}$ is the target monomial x^β. We will show that for $1 \le j \le k$, the polynomials $B_\alpha^{(j-1)}$ can be expressed as a linear combination of the polynomials $B_\alpha^{(j)}$, from which the result follows.

As mentioned above, because $a_j d_j - b_j c_j \neq 0$, there are coefficients $u_j, v_j \in \mathbb{F}$ such that $u_j(a_j x_j + b_j) + v_j(c_j x_j + d_j) = x^{\beta_j}$. Then, writing αb for the concatenation of α with $b \in \{0,1\}$, we can see that

$$B_\alpha^{(j-1)} = u_j B_{\alpha 0}^{(j)} + v_j B_{\alpha 1}^{(j)}.$$

This expression holds for arbitrary $\alpha \in \{0,1\}^{j-1}$, so the result follows. □

We will call a polynomial basis $B = (B_\alpha)$ as defined above a **multiaffine basis** of the space of multilinear polynomials. In particular, each multiaffine basis B gives a unique representation for multilinear polynomials by listing the coefficients when written as a linear combination of the elements of B. Given a basis B, we write $\textsc{Repr}(p, B)$ for the B-coefficient representation of a multilinear polynomial p.

The multiaffine basis polynomials include as a special case the standard monomial basis, by setting

$$\begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \text{for each } i.$$

Another important class of bases are the **Lagrange bases**, which represent polynomials in terms of their *evaluations* at a fixed structured collection of points. The **binary Lagrange basis polynomials** $(L_\alpha)_{\alpha \in \{0,1\}^k}$ are given by

$$L_\alpha(x_1, \ldots, x_k) \coloneqq \prod_{i=1}^{k} \big((1-\alpha_i)(1-x_i) + \alpha_i x_i\big),$$

which in particular is also a special case of the multiaffine basis polynomials, with

$$\begin{pmatrix} a_i & b_i \\ c_i & d_i \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \quad \text{for each } i.$$

The Lagrange basis polynomial L_α evaluates to 1 at the point α, and to 0 at every other binary input. In particular, this means that a multilinear polynomial $p = \sum_{\beta \in \{0,1\}^k} c_\beta L_\beta$ evaluates to the coefficient c_α for each binary input α. We write $\textsc{Lagrange}(p)$ to represent the collection $\textsc{Repr}(p, L)$ of coefficients representing a multilinear polynomial p in this basis, called the (binary) **Lagrange representation** of p.

2 Streaming Evaluation

Next we will discuss approaches for evaluating multilinear polynomials which can be used when the coefficients are **streamed**, meaning that they may only be accessed one at a time, and in an order which is not controlled by the algorithm. We begin with the direct generalization of Algorithm 1 to multilinear polynomials represented in terms of a multiaffine basis.

Algorithm 2 Evaluating a multilinear polynomial p over multiaffine basis polynomials B

1 **function** $\textsc{Eval}(\mathbf{c} \leftarrow \textsc{Repr}(p, B),\ \mathbf{r} : \mathbb{F}^k) \rightarrow \mathbb{F}$
2     $\mathbf{s} : \mathbb{F}^k \leftarrow (a_i + b_i r_i \text{ for } i \in [1 \mathinner{..} k])$ ▷ Evaluate B-basis product terms
3     $\mathbf{t} : \mathbb{F}^k \leftarrow (c_i + d_i r_i \text{ for } i \in [1 \mathinner{..} k])$

4     $\textsc{acc} : \mathbb{F} \leftarrow 0$ ▷ Main evaluation loop
5   **for** $\alpha \in \{0, 1\}^k$ **do**
6         $\textsc{prod} \leftarrow c_\alpha \cdot \prod_{i:\alpha_i=0} s_i \cdot \prod_{i:\alpha_i=1} t_i$
7         $\textsc{acc} \leftarrow \textsc{acc} + \textsc{prod}$
8   **return** $\textsc{acc}$

Theorem 2. *Algorithm 2 computes the evaluation of a k-variate multilinear polynomial, represented as $n = 2^k$ coefficients over a multiaffine basis B, and permits streaming input of these coefficients in arbitrary order. The algorithm uses $O(n \log n)$ time and $O(\log n)$ space, and in particular requires $k \cdot 2^k + O(k)$ field multiplications and $2^k + O(k)$ field additions.*

Proof. Correctness is clear since the algorithm simply accumulates the appropriate linear combination of the basis polynomials B_α to represent $p(\mathbf{r})$. This sum does not depend on the order in which the indices α are visited because addition is commutative, so the coefficients may be streamed in arbitrary order.

A product of $j \geq 1$ field elements may be naively computed using $j - 1$ field multiplications, so line 6 computes $\textsc{prod}$ using exactly k multiplication operations per iteration of the loop, yielding $k \cdot 2^k$ multiplications in total. The loop also requires one addition operation per iteration, giving 2^k additions in total. Finally, $2k$ additions and multiplications are used by the initialization operations on lines 2 and 3, yielding the desired totals.

The computation time is dominated by the $k \cdot 2^k = n \log n$ field multiplications, and the space usage is dominated by the $2k = 2 \log n$ field elements used to store the product terms of the basis polynomials B_α. This yields the desired time and space bounds. □

Remark 3. Algorithm 2 approximately doubles the number of multiplication operations needed when compared to Algorithm 1, which is specialized for the monomial basis, because each iteration of the main loop must take a product of $k + 1$ terms, as opposed to the only on average $k/2 + 1$ terms needed for the monomial basis. This factor of 2 can be regained at the cost of some complexity with the following modifications:

- Precompute:
  - An index set $I \leftarrow [i : s_i \neq 0]$

- An initial product term $p_0 \leftarrow \prod_{i \in I} s_i \cdot \prod_{i \notin I} t_i$
- Monomial terms $\mathbf{m} \leftarrow (s_i^{-1} t_i : i \in I)$

- Skip the main loop whenever $\alpha_i = 0$ for any $i \notin I$.

- Compute PROD as the product: $c_\alpha \cdot p_0 \cdot \prod_{i \in I : \alpha_i = 1} m_i$.

After implementing the above, the algorithm requires at most $(k+2)2^{k-1} + 4k$ field multiplications and k field inversions, which reduces the multiplicative order of magnitude to that of the simplified monomial algorithm.

Next we present an alternate approach that makes a space-time tradeoff to speed up the computation. In the following, we will write $\textsc{Int}(\alpha)$ for the integer $n = \sum_{i=1}^{k} \alpha_i 2^{i-1}$ corresponding to a binary tuple $\alpha \in \{0,1\}^k$, and $\textsc{Bin}_k(n)$ for the inverse operation.

Algorithm 3 Evaluating a multilinear polynomial p over multiaffine basis polynomials B, making a space-time tradeoff

function $\textsc{Eval}(\mathbf{c} \leftarrow \textsc{Repr}(p, B),\ \mathbf{r} : \mathbb{F}^k) \to \mathbb{F}$
    $\mathbf{s} : \mathbb{F}^k \leftarrow (a_i + b_i r_i$ for $i \in [1 \mathinner{..} k])$ ▷ Evaluate B-basis product terms
    $\mathbf{t} : \mathbb{F}^k \leftarrow (c_i + d_i r_i$ for $i \in [1 \mathinner{..} k])$

    BASIS : $\textsc{Array}_{2^k}(\mathbb{F}) \leftarrow [1, -, \cdots, -]$ ▷ Precompute B-basis polynomials at $\mathbf{r}$
    for $i \leftarrow [1 \mathinner{..} k]$ **do**
        for $j \leftarrow [0 \mathinner{..} 2^{i-1})$ **do**
            PREV $\leftarrow$ BASIS$[j]$
            BASIS$[j] \leftarrow s_j \cdot$ PREV
            BASIS$[j + 2^{i-1}] \leftarrow t_j \cdot$ PREV

    ACC : $\mathbb{F} \leftarrow 0$ ▷ Main evaluation loop
    for $\alpha \in \{0,1\}^k$ **do**
        PROD $\leftarrow c_\alpha \cdot$ BASIS$[\textsc{Int}(\alpha)]$
        ACC $\leftarrow$ ACC + PROD
    return ACC

Theorem 4. *Algorithm 3 computes the evaluation of a k-variate multilinear polynomial, represented as $n = 2^k$ coefficients over a multiaffine basis B, and permits streaming input of these coefficients in arbitrary order. The algorithm uses $O(n)$ time and $O(n)$ space, and in particular requires $3 \cdot 2^k + O(k)$ field multiplications, $2^k + O(k)$ field additions, and $2^k + O(k)$ field elements of memory.*

Proof. Algorithm 3 works exactly like Algorithm 2 except that it precomputes and stores in memory the evaluations of the B-basis polynomials at input $\mathbf{r}$. The main evaluation loop again is a simple sum taken over the indices α, so the polynomial coefficients may be streamed in arbitrary order.

Correctness of the algorithm thus follows once we argue that: after running the loop starting on Line 5, the array BASIS holds the product $\prod_{i:\alpha_i=0} s_i \cdot \prod_{i:\alpha_i=1} t_i$ at index $\text{INT}(\alpha)$ for each $\alpha \in \{0,1\}^k$. To this end, let $B^{(i)}$ be the i-variate multilinear basis consisting of the 2^i choices of products of the B-basis product terms for $x_1, \ldots, x_i$. Then $B^{(i-1)}_\alpha$ may be used to construct $B^{(i)}_{\alpha 0}$ and $B^{(i)}_{\alpha 1}$ by

$$B^{(i)}_{\alpha 0} = (a_i + b_i x_i) B^{(i-1)}_\alpha,$$
$$B^{(i)}_{\alpha 1} = (c_i + d_i x_i) B^{(i-1)}_\alpha.$$

In particular, iteration i of the loop computes the entries of $B^{(i)}$ evaluated at $(r_1, \ldots, r_i)$ using the entries of $B^{(i-1)}$ evaluated at $(r_1, \ldots, r_{i-1})$. Thus by the end of the loop, all of the terms of the basis $B = B^{(k)}$ have been computed.

For a tally of arithmetic operations, the initialization of $\mathbf{s}$ and $\mathbf{t}$ requires $2k$ field additions and multiplications, and the main evaluation loop requires 2^k field additions and multiplications. When precomputing the B-basis polynomials, the inner loop repeats 2^{i-1} times for each $i = 1, \ldots, k$, for a total of $2^k - 1$ repetitions, giving $2 \cdot 2^k - 2$ field multiplications. This gives $3 \cdot 2^k + 2k - 2 = 3 \cdot 2^k + O(k)$ field multiplications and $2^k + 2k = 2^k + O(k)$ field additions in total, as required.

The space used by the algorithm is dominated by the BASIS array, which uses 2^k field elements. An additional $2k + O(1)$ field elements of space are required to store $\mathbf{s}, \mathbf{t}$ and other incidental variables, giving the desired $O(k)$ error bound.

The $O(n)$ time and space bounds follow immediately from the above. □

Remark 5. As was the case for Algorithm 2, the implementation of Algorithm 3 may be modified to reduce the number of multiplication operations needed to compute the B-basis polynomials by about half, at the cost of some additional bookkeeping to keep track of zero product terms. (See Remark 3.) This reduces the number of required field multiplications to $2 \cdot 2^k + O(k)$.

As an additional observation, notice that the approaches of Algorithms 2 and 3 may be combined to produce an algorithm with characteristics somewhere between the two. Specifically, a new algorithm EVAL_j may execute the precomputation loop of Algorithm 3 only up to iteration $j < k$, and then compute the remaining product terms of the B-basis polynomials for indices $j+1, \ldots, k$ on the fly during the main evaluation loop. This results in a space requirement of $2^j + O(k)$ and a time requirement of $(k-j)2^k + O(2^j)$.

Setting j dynamically based on the value of k can in fact produce algorithm variants with distinct asymptotic performance. For instance, letting $j = k/2$ makes use of $O(\sqrt{n})$ space to gain a speed-up of a factor of around 2 compared to Algorithm 2. Letting $j = k - \log k$ results in asymptotic runtime of $O(n \log \log n)$ and asymptotic space usage of $O(n/\log n)$.

2.1 Non-streaming Evaluation

In the following we introduce an algorithm which evaluates a multilinear polynomial with better performance than achieved by any of the previous approaches. This is accomplished by abandoning the requirement that the coefficients of the polynomial are streamable, requiring instead that they are accessible in a prescribed traversal order. Note that random access to the polynomial coefficients is sufficient for this purpose.

Algorithm 4 traverses the indices $\alpha \in \{0,1\}^k$ in lexicographic order using a recursive construction, building up the corresponding B-basis polynomial evaluations term by term throughout the traversal, and storing in memory only the currently required sequence of prefix product terms. This has the effect of both minimizing the number of field multiplications needed for evaluation, and using only logarithmic stack space for the recursive calls.

Algorithm 4 Evaluating a multilinear polynomial p over multiaffine basis polynomials B using recursive traversal of the index space

---

Precondition: Coefficients c_α allow sequential access in lexicographic order by index

1 **function** $\textsc{Eval}(\mathbf{c} \leftarrow \textsc{Repr}(p, B),\ \mathbf{r} : \mathbb{F}^k) \to \mathbb{F}$
2     $\mathbf{s} : \mathbb{F}^k \leftarrow (a_i + b_i r_i \text{ for } i \in [1 \mathinner{..} k])$ ▷ Evaluate B-basis product terms
3     $\mathbf{t} : \mathbb{F}^k \leftarrow (c_i + d_i r_i \text{ for } i \in [1 \mathinner{..} k])$

4   **function** $\textsc{PrefixEval}(\alpha : \{0,1\}^*) \to \mathbb{F}$ ▷ Find sum of partial terms for prefix α
5         $i \leftarrow \textsc{Len}(\alpha)$
6   **if** $i = k$ **then**
7   **return** c_α
8   **else**
9   **return** $s_{i+1} \cdot \textsc{PrefixEval}(\alpha 0) + t_{i+1} \cdot \textsc{PrefixEval}(\alpha 1)$

10     $\alpha : \{0,1\}^* \leftarrow \varepsilon$ ▷ Main recursive call
11   **return** $\textsc{PrefixEval}(\alpha)$

Theorem 6. *Algorithm 4 computes the evaluation of a k-variate multilinear polynomial, represented as $n = 2^k$ coefficients over a multiaffine basis B. The algorithm uses $O(n)$ time and $O(\log n)$ space, and in particular requires $2 \cdot 2^k + O(k)$ field multiplications and $2^k + O(k)$ field additions.*

Proof. For $\alpha \in \{0,1\}^*$ of length $i \le k$, we argue by induction on i that $\textsc{PrefixEval}(\alpha)$ computes the sum

$$\sum_{\alpha' = \alpha\beta \in \{0,1\}^k} c_{\alpha'} \prod_{j=i+1}^{k} \big((1 - \alpha'_j) s_j + \alpha'_j t_j\big). \tag{2}$$

For $i = k$, this is clear because $\textsc{PrefixEval}$ returns c_α, which is equal to the desired sum. If $i < k$ and the expression is assumed valid for prefixes of length $i + 1$, then we see that $\textsc{PrefixEval}(\alpha)$ is given by

$$s_{i+1} \sum_{\alpha' = \alpha 0 \beta} c_{\alpha'} \prod_{j=i+2}^{k} \big((1 - \alpha'_j) s_j + \alpha'_j t_j\big) + t_{i+1} \sum_{\alpha' = \alpha 1 \beta} c_{\alpha'} \prod_{j=i+2}^{k} \big((1 - \alpha'_j) s_j + \alpha'_j t_j\big),$$

which can be simplified to Equation 2. When α is the empty sequence, this expression is equal to $p(\mathbf{r})$ evaluated in terms of the B-basis polynomials, so Lines 10 and 11 produce the desired result.

Each recursive call to PrefixEval increases the length of the prefix α by 1, so since the function makes two recursive calls and terminates at length k, the outer call to PrefixEval(ε) results in $2^{k+1} - 1$ calls in total, of which 2^k satisfy $\text{Len}(\alpha) = k$ and execute the base case, and $2^k - 1$ execute the recursive case. Each recursive case requires two field multiplications and one field addition, so the total needed for the call to PrefixEval is $2 \cdot 2^k - 2$ multiplications and $2^k - 1$ additions. The preprocessing of the product terms $\mathbf{s}$ and $\mathbf{t}$ additionally requires $2k$ multiplications and additions, yielding a final count of $2 \cdot 2^k + 2k - 2 = 2 \cdot 2^k + O(k)$ multiplications and $2^k + 2k - 1 = 2^k + O(k)$ additions.

The desired $O(n)$ time bound follows from the counts of arithmetic operations. For space usage, note that the recursive calls to PrefixEval go to depth at most $k + 1$, so the space usage (e.g. on the stack) of these recursive calls is a constant times the maximum depth. Together with the $2k$ field elements of space used to store $\mathbf{s}$ and $\mathbf{t}$, this yields the desired space bound of $O(k) = O(\log n)$.

To see that the algorithm uses the polynomial coefficients sequentially in lexicographic order by index, note that the recursive calls to PrefixEval append 0 to the prefix before appending 1. This means that if two indices α and α' differ first at index j, with $\alpha_j = 0$ and $\alpha'_j = 1$, then c_α is visited in the recursive call to PrefixEval($\alpha_1 \cdots \alpha_{j-1}0$), while $c_{\alpha'}$ is visited after this in the recursive call PrefixEval($\alpha_1 \cdots \alpha_{j-1}1$). $\square$

Remark 7. Similarly to previous algorithms in this note, an approach like that outlined in Remark 3 allows Algorithm 4 to be modified to use only one multiplication operation per recursive call to PrefixEval. This reduces the number of field multiplications to $2^k + O(k)$.

Additionally, the traversal order used by Algorithm 4 to produce the basis polynomial evaluations is not a hard requirement. An alternate approach would be to traverse the indices in the order of a *Gray Code*, which enumerates the length k binary strings in such a way that each is obtained from the last by inverting a single bit. This approach thus has the potential advantage of only writing to a constant number of variables during the main loop, but also adds some additional complexity to the implementation.