berkeDet: A Dual Iterative Algorithm for Exact Determinant Computation via Block Replication and Index Mapping

Berke Gülmen[1] Meryem Gülmen[1] Ömer Gülmen[1,*]

26 February 2026

[1]Independent Researcher
[*]Corresponding author: gulmenberkemeryem@gmail.com
Website: https://detsignper.org

Abstract

We introduce **berkeDet**, a dual iterative algorithm that computes the determinant of an $n \times n$ matrix via the Leibniz formula using two independent subroutines: *meryemSign*, which generates the complete list of permutation signs in lexicographic order, and *meryemPer*, which generates the corresponding permutations. Both subroutines operate by block replication and index-mapping operations, avoiding the per-step overhead inherent in classical approaches.

In the standard lexicographic implementation of the Leibniz formula, computing the sign of each permutation via inversion counting costs $O(n^2)$ per permutation, and advancing to the next permutation via the naive lexicographic generator requires an $O(n \log n)$ suffix sort. Even the well-known Narayana Pandita optimization, which replaces the sort with a suffix reversal, still requires per-step scanning and swapping operations. By contrast, *meryemSign* generates all $n!$ signs in $\Theta(n!)$ total time—an amortized cost of $O(1)$ per sign—and *meryemPer* generates all permutations in $O(n \cdot n!)$ total time without any sorting, reversal, or pivot scanning at any step. The combined berkeDet algorithm therefore runs in $O(n \cdot n!)$ time, improving the per-permutation cost by a factor of n over the classical lexicographic approach. We prove correctness by induction and verify the algorithm computationally for matrices up to order 8.

Keywords: determinant, Leibniz formula, permutation, lexicographic order, sign function, iterative algorithm
MSC2020 Classifications: 05A05, 15A15

1 Introduction

The determinant of a square matrix is a fundamental quantity in linear algebra with applications throughout science and engineering. It determines whether a matrix is invertible, quantifies the volume scaling factor of the associated linear transformation, and indicates whether the transformation preserves or reverses orientation.

Computing the determinant efficiently is a classical problem. For practical purposes, Gaussian elimination and LU decomposition provide $O(n^3)$ algorithms that are universally preferred for large matrices. However, the *Leibniz formula*

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma(i)} \tag{1}$$

remains important for exact symbolic computation, theoretical analysis, and pedagogical exposition, as it expresses the determinant directly as a sum over all $n!$ permutations of $\{1, 2, \ldots, n\}$.

A direct implementation of the Leibniz formula requires (a) enumerating all permutations and (b) determining the sign of each. In the standard lexicographic approach, the naive method of advancing from one permutation to the next requires sorting the suffix, which costs $O(n \log n)$ per step. The well-known Narayana Pandita optimization replaces this sort with a suffix reversal (exploiting the fact that the suffix is always in descending order), reducing the per-step cost to $O(n)$. However, this optimization is non-obvious and still requires scanning for a pivot element, performing a swap, and reversing a suffix at each step. Computing the sign of each permutation independently via inversion counting costs $O(n^2)$ per permutation. The total cost of the standard lexicographic Leibniz implementation is therefore $O(n^2 \cdot n!)$.

In this paper we introduce **berkeDet**, a dual iterative algorithm that reduces this cost to $O(n \cdot n!)$ by replacing both per-permutation sign computation and per-step permutation advancement with global block replication constructions. The algorithm comprises two subroutines:

- **meryemSign**(n): generates the complete list of $n!$ Leibniz signs in lexicographic order using iterated block replication with sign flips, in $\Theta(n!)$ total time.
- **meryemPer**(n): generates all $n!$ permutations of $\{1, \ldots, n\}$ in lexicographic order via layer-wise block replication and index mapping, in $O(n \cdot n!)$ total time, without any sorting, reversal, or pivot scanning at any step.

The key insight is that both the sign sequence and the permutation list for S_n in lexicographic order have recursive block structures that can be constructed from the corresponding structures for S_{n-1} using only replication and index remapping. This eliminates per-step computation entirely: instead of computing each permutation and its sign from the previous one, both are built globally by assembling blocks.

2 Preliminaries

Definition 1. An $m \times n$ *matrix* $\mathbf{A}$ is a rectangular array $\mathbf{A} = (a_{ij})$ with $1 \le i \le m$ and $1 \le j \le n$. When $m = n$, the matrix is *square* of order n.

Definition 2. The *symmetric group* S_n is the set of all bijections $\sigma\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$. The *sign* (or *signature*) of a permutation σ is

$$\operatorname{sgn}(\sigma) = (-1)^{|\{(i,j):1 \le i < j \le n,\, \sigma(i) > \sigma(j)\}|},$$

where the exponent counts the number of *inversions* of σ.

Definition 3. The *determinant* of a square matrix $\mathbf{A}$ of order n is given by the Leibniz formula (1).

3 Related Work

The problem of iteratively generating permutations has been studied extensively. We review the main approaches and compare their per-step costs, with particular attention to whether each method requires sorting.

3.1 Lexicographic Generators

Naive lexicographic generation. The most straightforward approach to generating the next permutation in lexicographic order involves finding the rightmost position where the sequence can be increased, placing the next-larger element there, and *sorting* the remaining suffix into ascending order. This suffix sort costs $O(n \log n)$ per step and $O(n \log n \cdot n!)$ in total.

Narayana Pandita's algorithm (14th century, rediscovered in modern form as the "next lexicographic permutation" algorithm, and implemented as `std::next_permutation` in C++) optimizes the naive approach by observing that the suffix to be sorted is always in descending order, so sorting can be replaced by a simple reversal. Each step thus requires: (1) scanning from the right to find the *pivot*—the rightmost position i where $\sigma(i) < \sigma(i+1)$; (2) scanning to find the smallest element in the suffix larger than $\sigma(i)$; (3) swapping; and (4) reversing the suffix. The per-step cost is $O(n)$ in the worst case, yielding $O(n \cdot n!)$ total. This is a non-trivial optimization that requires recognizing and exploiting a structural property of the suffix.

meryemPer (this work) achieves $O(n \cdot n!)$ total cost through an entirely different mechanism. Rather than computing each permutation from the previous one via scanning, swapping, and reversing, meryemPer builds the complete list by layer-wise block replication and index mapping. *No sorting, no reversal, no pivot scanning, and no suffix manipulation appear at any step.* The construction is purely compositional: permutations of $\{1, \ldots, n\}$ are assembled from permutations of $\{1, \ldots, n-1\}$ by prepending each possible first element and remapping indices. This structural difference makes the algorithm transparent and eliminates the need for the non-obvious optimization insight that Narayana Pandita's method relies on.

3.2 Non-Lexicographic Generators

The Steinhaus–Johnson–Trotter (SJT) algorithm generates permutations by adjacent transpositions, which has the elegant property that the sign flips at every step, allowing $O(1)$ sign tracking. However, SJT does *not* produce permutations in lexicographic order.

Heap's algorithm generates all permutations via a recursive swap structure in $O(1)$ amortized time per permutation. Like SJT, it does not produce lexicographic order, and sign tracking requires additional bookkeeping.

3.3 Sign Computation

Inversion counting, the standard method for computing $\text{sgn}(\sigma)$ for a given permutation, costs $O(n^2)$ by direct enumeration of pairs, or $O(n \log n)$ via merge sort. When applied to each of $n!$ permutations independently, this yields a total sign computation cost of $O(n^2 \cdot n!)$ or $O(n \log n \cdot n!)$.

meryemSign (this work) eliminates per-permutation sign computation entirely. It generates all $n!$ signs in $\Theta(n!)$ total time—an amortized cost of $O(1)$ per sign—by exploiting the recursive block structure of the sign sequence in lexicographic order.

3.4 Summary

The standard lexicographic implementation of the Leibniz formula combines a lexicographic permutation generator with per-permutation inversion counting, yielding at least $O(n^2 \cdot n!)$ total cost (or $O(n \log n \cdot n!)$ with merge-sort-based sign computation). The SJT-based approach achieves $O(n \cdot n!)$ total but sacrifices lexicographic order.

The present work shows that lexicographic order and efficient generation are not mutually exclusive. berkeDet achieves $O(n \cdot n!)$ total—matching SJT's cost—while preserving lexicographic order, through a fundamentally different construction that avoids sorting, reversal, scanning, and per-permutation sign computation.

4 The berkeDet Algorithm

4.1 Overview

Given an $n \times n$ matrix $\mathbf{A}$, berkeDet computes $\det(\mathbf{A})$ as follows:

1. Compute $\mathbf{s} = \text{meryemSign}(n)$, the list of $n!$ signs.
2. Compute $\mathbf{P} = \text{meryemPer}(n)$, the list of $n!$ permutations.

Table 1: Comparison of Leibniz-formula implementations. Here T_{perm} is the cost per permutation step, T_{sign} is the cost per sign computation, and "Lex. order" indicates whether permutations are produced in lexicographic order. "Sort-free" indicates whether the permutation generator avoids sorting or suffix reversal operations.

Method	T_{perm}	T_{sign}	**Total**	**Lex.**	**Sort-free**
Naive lex. + inversion count	$O(n \log n)$	$O(n^2)$	$O(n^2 \cdot n!)$	Yes	No
Narayana + inversion count	$O(n)$	$O(n^2)$	$O(n^2 \cdot n!)$	Yes	No†
Narayana + merge-sort sign	$O(n)$	$O(n \log n)$	$O(n \log n \cdot n!)$	Yes	No†
SJT + sign flip	$O(n)$	$O(1)$	$O(n \cdot n!)$	No	Yes
Heap + bookkeeping	$O(1)$ am.	$O(1)$ am.	$O(n!)$	No	Yes
berkeDet (this work)	$O(n)$ am.	$O(1)$ am.	$O(n \cdot n!)$	**Yes**	**Yes**

†Narayana Pandita replaces sorting with suffix reversal, which is a non-trivial optimization; the method still requires per-step pivot scanning, swapping, and reversal.

3. For each $i = 1, \ldots, n!$, compute $\text{pair}(i) = s_i \cdot \prod_{j=1}^{n} a_{j,\,P_i(j)}$.
4. Return $\det(\mathbf{A}) = \sum_{i=1}^{n!} \text{pair}(i)$.

Since steps 1 and 2 produce exactly the data required by the Leibniz formula in matched order, correctness follows immediately from the correctness of the two subroutines.

4.2 meryemSign: Sign Generation by Block Replication

Theorem 1. *The following iterative procedure produces the list of* $\text{sgn}(\sigma)$ *for all* $\sigma \in S_n$ *in lexicographic order:*

1. *Initialize* $b \leftarrow [+1]$.
2. *For* $e = 1, 2, \ldots, n-1$*:*
   (a) *Set* $c \leftarrow b$ *(replicate the current list).*
   (b) *For* $d = 0, 1, \ldots, e-1$*:*
   - *If* d *is even, let* $f \leftarrow -c$ *(negate every element of the replica).*
   - *If* d *is odd, let* $f \leftarrow c$ *(keep the replica unchanged).*
   - *Append* f *to* b.

After completion, b *contains* $n!$ *entries, and* $b[i] = \text{sgn}(\sigma_i)$ *where* σ_i *is the* i*-th permutation in lexicographic order.*

Proof. We proceed by induction on n.

Base case ($n = 1$): S_1 contains only the identity permutation, with sign $+1$. The algorithm initializes $b = [+1]$ and performs no iterations, so the output is correct.

Inductive step: Assume that after processing layers 1 through $n-1$, the list b contains the correct signs for all $(n-1)!$ permutations of S_{n-1} in lexicographic order. At layer $e = n-1$ (which extends the list from S_{n-1} to S_n), the algorithm produces $e = n-1$ additional blocks, each of size $(n-1)!$.

Consider the permutations of $\{1, \ldots, n\}$ grouped by their first element. The group with first element k (for $k = 1, 2, \ldots, n$) consists of $(n-1)!$ permutations whose remaining elements form permutations of $\{1, \ldots, n\} \setminus \{k\}$.

The sign relationship is: if $\sigma = (k, \tau_1, \tau_2, \ldots, \tau_{n-1})$, then moving k to the first position from its natural position requires $k-1$ adjacent transpositions, so

$$\text{sgn}(\sigma) = (-1)^{k-1} \cdot \text{sgn}(\tau),$$

where τ is the corresponding permutation of the remaining $n-1$ elements in their natural relabelling.

The algorithm starts with the block for $k = 1$ (the existing list b, unchanged since $(-1)^0 = +1$), then appends blocks for $k = 2, 3, \ldots, n$. For $k = 2$ (i.e., $d = 0$, even), it negates the signs (since $(-1)^1 = -1$). For $k = 3$ ($d = 1$, odd), it keeps them unchanged (since $(-1)^2 = +1$). In general, block d corresponds to $k = d + 2$, and the sign factor $(-1)^{k-1} = (-1)^{d+1}$ matches the algorithm's rule: negate when d is even, copy when d is odd. This completes the induction. □

Proposition 2 (Complexity of meryemSign)**.** *The total number of operations performed by meryemSign is* $\Theta(n!)$. *The amortized cost per sign is* $O(1)$.

Proof. At layer e, the algorithm replicates and possibly negates e blocks, each of size $e!/e = (e-1)!$, for a total of $e \cdot (e-1)! = e!$ operations. Summing over all layers:

$$\sum_{e=1}^{n-1} e! = \Theta(n!),$$

since the last term $(n-1)!$ dominates (the ratio of the sum to $(n-1)!$ converges to $e \approx 2.718$). Hence the total cost is $\Theta(n!)$, and the amortized cost per sign is $\Theta(n!)/n! = O(1)$. □

4.3 meryemPer: Permutation Generation by Block Replication and Index Mapping

Theorem 3. *The following iterative procedure generates all* $n!$ *permutations of* $\{1, \ldots, n\}$ *in lexicographic order:*

1. *Initialize* $a \leftarrow [[1]]$.
2. *For each layer* $b = 2, 3, \ldots, n$*:*
   (a) *Let* $c = [1, 2, \ldots, b]$.
   (b) *Initialize* $d \leftarrow$ *empty list.*
   (c) *For each* $f \in c$ *(in increasing order):*
      i. *Form* $g = c \setminus \{f\}$ *(the remaining elements in order).*
      ii. *For each permutation* $h \in a$*:*
         - *Compute* $j = [g[h[1] - 1],\ g[h[2] - 1],\ \ldots,\ g[h[b-1] - 1]]$.
         - *Append* $[f] \circ j$ *to* d.
   (d) *Set* $a \leftarrow d$.

Proof. By induction on n.

Base case ($n = 1$): $a = [[1]]$, which is the single permutation of $\{1\}$. Correct.

Inductive step: Assume a contains all $(n-1)!$ permutations of $\{1, \ldots, n-1\}$ in lexicographic order. At layer $b = n$, the algorithm iterates over each possible first element f from 1 to n in increasing order. For each f, the remaining elements $g = \{1, \ldots, n\} \setminus \{f\}$ are listed in increasing order.

The index-mapping step $j = [g[h[k]-1]]_{k=1}^{n-1}$ applies the permutation pattern h (a permutation of abstract positions $\{1, \ldots, n-1\}$) to the concrete remaining elements g. Since h ranges over all $(n-1)!$ permutations in lexicographic order (by hypothesis) and the mapping from positions to elements is order-preserving, the resulting permutations of g are also in lexicographic order.

Since f increases from 1 to n, and for each f the suffixes are lexicographically ordered, the full output is in lexicographic order. □

Remark 1 (Structural difference from classical generators). The standard lexicographic permutation generator (whether in its naive form with suffix sorting or in the optimized Narayana Pandita form with suffix reversal) operates *incrementally*: each permutation is derived from the previous one by modifying it in place. This requires per-step operations including scanning for a pivot, swapping elements, and sorting or reversing a suffix. meryemPer operates by a fundamentally different mechanism: it builds the entire permutation list *compositionally* from the

previous layer via block replication and index remapping. No sorting, reversal, scanning, or in-place modification occurs at any step. The algorithm does not need to "discover" that the suffix is in descending order (the key insight behind Narayana Pandita's optimization)—it avoids the need for such an observation entirely.

Proposition 4 (Complexity of meryemPer)**.** *The total number of operations performed by meryemPer is* $O(n \cdot n!)$*. The amortized cost per permutation is* $O(n)$*.*

Proof. At layer b, the algorithm generates $b!$ permutations, each of length b. The index-mapping step for each permutation costs $O(b)$. The total cost at layer b is therefore $O(b \cdot b!)$. Summing:

$$\sum_{b=2}^{n} b \cdot b! = O(n \cdot n!),$$

since the last term $n \cdot n!$ dominates. The amortized cost per permutation is $O(n \cdot n!)/n! = O(n)$. □

4.4 Combined Complexity

Corollary 5. *berkeDet computes* $\det(\mathbf{A})$ *for an* $n \times n$ *matrix in* $O(n \cdot n!)$ *time and* $O(n \cdot n!)$ *space.*

Proof. meryemSign runs in $\Theta(n!)$, meryemPer in $O(n \cdot n!)$, and the summation step computes $n!$ products of n terms each, costing $O(n \cdot n!)$. The dominant term is $O(n \cdot n!)$. Space is $O(n \cdot n!)$ for storing the permutation list. □

Remark 2. The standard lexicographic implementation using naive suffix sorting and inversion counting has total complexity $O(n^2 \cdot n!)$. Even with the Narayana Pandita optimization (suffix reversal instead of sorting) and merge-sort-based sign computation, the total is $O(n \log n \cdot n!)$. berkeDet improves upon both by achieving $O(n \cdot n!)$. This matches the total complexity of the non-lexicographic SJT-based approach, but unlike SJT, berkeDet produces permutations in lexicographic order and does so without any sorting, reversal, or per-step scanning operations.

5 Implementation

We provide Python implementations for reference.

5.1 meryemSign

```
def meryemSign(n: int) -> list[int]:
    """Generate signs for all n! permutations in lex order.
    Returns list of +1 and -1 values."""
    b = [1]  # start with +1
    for e in range(1, n):
        c = b[:]
        for d in range(e):
            if d % 2 == 0:
                f = [-x for x in c]
            else:
                f = c[:]
            b.extend(f)
    return b
```

5.2 meryemPer

```
def meryemPer(n: int) -> list[list[int]]:
    """Generate all n! permutations of {1,...,n} in lex order.
    No sorting, reversal, or pivot scanning at any step."""
    a = [[1]]
    for b in range(2, n + 1):
        c = list(range(1, b + 1))
        d = []
        for f in c:
            g = [e for e in c if e != f]
            for h in a:
                j = [g[i - 1] for i in h]
                d.append([f] + j)
        a = d
    return a
```

5.3 berkeDet

```
def berkeDet(matrix: list[list[float]]) -> float:
    """Compute determinant via Leibniz formula using
    meryemSign and meryemPer."""
    n = len(matrix)
    signs = meryemSign(n)
    perms = meryemPer(n)
    det = 0
    for sign, perm in zip(signs, perms):
        product = 1
        for i in range(n):
            product *= matrix[i][perm[i] - 1]
        det += sign * product
    return det
```

6 Computational Verification

We verified the correctness of berkeDet against NumPy's `numpy.linalg.det` for randomly generated integer matrices of orders $n = 1$ through $n = 8$. In all cases the results agreed to within floating-point tolerance ($< 10^{-9}$ relative error). Selected examples:

- $n = 1$: $\mathbf{A} = (7)$, berkeDet $= 7$. ✓
- $n = 2$: $\mathbf{A} = \left(\begin{smallmatrix} 3 & 8 \\ 4 & 6 \end{smallmatrix}\right)$, berkeDet $= 3 \cdot 6 - 8 \cdot 4 = -14$. ✓
- $n = 3$: Random 3×3 integer matrix, both methods return -306. ✓
- $n = 4$: Identity matrix, berkeDet $= 1$. ✓
- $n = 5$ through $n = 8$: Random integer matrices, agreement confirmed. ✓

7 Discussion

7.1 Structural Insight

The core observation behind meryemSign is that the sign sequence for lexicographically ordered permutations has a recursive block structure. When extending from S_{n-1} to S_n, the n blocks of

$(n-1)!$ signs corresponding to first elements $1, 2, \ldots, n$ follow the pattern

$$\underbrace{+\mathbf{s}}_{k=1}, \quad \underbrace{-\mathbf{s}}_{k=2}, \quad \underbrace{+\mathbf{s}}_{k=3}, \quad \underbrace{-\mathbf{s}}_{k=4}, \quad \ldots$$

where $\mathbf{s}$ is the sign list for S_{n-1} and the alternation arises from the factor $(-1)^{k-1}$. This structure makes the entire sign list computable by pure block replication with no arithmetic beyond sign flips.

Similarly, meryemPer exploits the observation that permutations of $\{1, \ldots, n\}$ with a fixed first element f correspond bijectively to permutations of the remaining elements, and this correspondence is captured by a simple index remapping using the previous layer's output. This compositional structure avoids the need for any in-place modification, scanning, or suffix manipulation.

7.2 The Advantage of Sort-Free Construction

A key property distinguishing meryemPer from all classical lexicographic generators is the complete absence of sorting. The naive lexicographic generator requires an $O(n \log n)$ suffix sort per step. The Narayana Pandita optimization avoids explicit sorting by recognizing that the suffix is always in descending order, allowing it to be reversed in $O(n)$. However, this is a non-trivial insight that must be discovered and verified. meryemPer achieves $O(n)$ per permutation without requiring this insight: the lexicographic order emerges naturally from the compositional block structure. This makes the algorithm both transparent and efficient.

7.3 When Is Lexicographic Order Useful?

While the SJT algorithm achieves the same $O(n \cdot n!)$ total complexity, it does not produce permutations in lexicographic order. Lexicographic order is useful in several contexts:

- Deterministic testing and debugging, where reproducible enumeration order is valuable.
- Symbolic algebra systems that expect canonical ordering.
- Educational settings, where the natural ordering aids comprehension.
- Parallel computation, where lexicographic ranges can be cleanly partitioned.

7.4 Scope and Applicability

Like all Leibniz-based methods, berkeDet has factorial time complexity and is therefore intended for exact determinant computation of small to moderate matrices, symbolic algebra, and educational contexts—not as a replacement for $O(n^3)$ numerical methods such as LU decomposition. The contribution of this work is to provide the most efficient known implementation of the Leibniz formula in lexicographic order, achieving $O(n \cdot n!)$ total cost by eliminating per-permutation sign computation entirely and generating permutations without sorting or suffix manipulation.

8 Conclusion

We have presented berkeDet, a dual iterative algorithm for computing the determinant via the Leibniz formula. The algorithm decomposes the computation into two independent subroutines—meryemSign and meryemPer—each based on block replication and index mapping. The key contributions are twofold: (1) meryemSign eliminates per-permutation sign computation by generating all $n!$ signs in $\Theta(n!)$ total time through block replication with sign flips, and (2) meryemPer generates all $n!$ permutations in lexicographic order in $O(n \cdot n!)$ total time without any sorting, reversal, or pivot scanning, achieving its efficiency through compositional construction rather than in-place optimization. The combined algorithm runs in $O(n \cdot n!)$ time, matching the complexity

of non-lexicographic methods such as Steinhaus–Johnson–Trotter while preserving lexicographic order—a combination no previous method achieved.

Future work includes formal benchmarking against alternative implementations, extension to symbolic matrix computation, and exploration of whether the block structure can be exploited for partial or streaming evaluation of the Leibniz sum.

Visual illustrations of the meryemPer pattern for orders $n = 2$ through $n = 9$ are provided in the supplementary presentation.

Acknowledgments

The writing of this manuscript was assisted by AI tools for LaTeX formatting, structural organization, and exposition. The mathematical content, algorithms, and proofs are the original work of the authors.