Smith Normal Form (SNF)

BENGISU DURU GÖKSU

SELIN BARDAKCI

ABDULLAH SAGLAM

MEHMET HAYRULLAH ÖZKUL

Outline

1. Introduction to Smith Normal Form (SNF)
2. Classical SNF and the Extended Euclidean Algorithm (EEA)
3. Recent Algorithmic Improvements for SNF
   - Modular SNF Algorithms
   - Fraction-Free SNF Algorithms
   - Divide-and-Conquer SNF Algorithms
   - Determinant-Based SNF Algorithms
4. SageMath Comparison and Validation
5. Smith Normal Form Algorithms Comperehensive Comparison
6. Execution Time vs Matrix Size of Smith Normal Form Algorithms
7. Key Takeaways Based on Our Evaluation
8. References

Introduction to Smith Normal Form (SNF)

$$S = UAV = \begin{bmatrix} s_1 & & & & & \\ & \ddots & & & & \\ & & s_r & & & \\ & & & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{bmatrix}$$

- Smith Normal Form (SNF) is a canonical diagonal form of integer matrices obtained using elementary row and column operations over the integers.
  It reveals the fundamental algebraic structure of a matrix and provides a unique representation through its diagonal entries, known as **invariant factors**.

- SNF expresses an integer matrix as $S = U A V$ where U and V are unimodular matrices and S is a diagonal matrix whose entries are invariant factors.
  These diagonal values uniquely characterize the algebraic structure of the matrix and remain unchanged under all valid integer row and column operations.

- SNF plays a central role in **symbolic computation**, particularly in solving linear Diophantine equations, classifying finitely generated abelian groups, and analyzing modules over principal ideal domains.

- Because it operates on exact integer arithmetic, efficient computation of SNF is an important problem in computational algebra.

Classical SNF and the Extended Euclidean Algorithm (EEA)

The classical Smith Normal Form algorithm transforms an integer matrix into a **diagonal form** using **elementary row and column operations**, which preserve the algebraic structure of the matrix. Its core computational tool is the **Extended Euclidean Algorithm (EEA)**, which computes:

- the **gcd (greatest common divisor)** of matrix entries
- the **Bézout coefficients**, which describe how to combine rows or columns using integers to eliminate entries

Key Steps

- **EEA:** Used to compute gcds and Bézout coefficients for reducing entries
- **Integer row and column operations:** Eliminate off-diagonal elements while preserving integer structure
- **Iterative reduction:** The matrix is repeatedly simplified until a diagonal form is reached

Modular SNF Algorithms

Main Approach

Solve many easy problems instead of one very hard problem.

Core Idea

- Instead of computing the Smith Normal Form directly over the integers, which causes rapid coefficient growth, the modular SNF algorithm works **modulo small prime numbers**.
- For each prime p, the matrix is analyzed modulo p, where arithmetic is fast and numbers remain small.
  These modular results capture partial information about the true Smith Normal Form.
- Finally, the the invariant factors are reconstructed using Chinese Remainder Theorem (CRT), from which the Smith Normal Form is obtained.

Why this works?

- Modular arithmetic avoids **coefficient explosion.**
- Computations are significantly faster for large matrices
- Independent modular computations allow parallelization
- The final result is mathematically guaranteed to be correct

Coefficient explosion is a problem where the numbers inside a matrix grow **very large very quickly** during intermediate steps of an algorithm, even if the final result is small.
This makes computations slow, memory-intensive, and sometimes practically impossible.
Modular algorithms avoid this by keeping all computations within small numeric ranges.

Walkthrough of Modular Smith Normal Form Algorithm

Given Matrix:

$$A = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

Our goal is to compute the Smith Normal Form of A **without large integer growth**, by performing all intermediate computations **modulo small prime numbers**.

Core idea

Instead of transforming the matrix directly over the integers, the modular SNF algorithm:

- performs the **same row and column operations**
- but **modulo a prime p**
- keeping all intermediate values small and efficient.

Walkthrough of Modular Smith Normal Form Algorithm

Step 1 — Reduce the matrix modulo *p*

- Choose a small prime (e.g. *p=5*):
- At this stage, all arithmetic is performed in $\mathbb{Z}/5\mathbb{Z}$.

$$A \bmod 5 = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix} \bmod 5 = \begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix}$$

Step 2 — Row operations (mod *p*)

- Elementary row operations are applied modulo *p*, analogously to the classical algorithm.
  No coefficient growth occurs, since all values remain bounded by *p*.

$$R_2 \leftarrow R_2 - 0 \cdot R_1$$

$$\begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix} \pmod 5$$

Step 3 — Column operations (mod p)

- The matrix is now diagonal modulo *p*.

$$C_2 \leftarrow C_2 - 3C_1$$

$$\begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \pmod 5$$

Step 4 — Intermediate modular diagonal form

- The resulting diagonal matrix represents an **intermediate modular normal form**, not the Smith Normal Form over $\mathbb{Z}$.
- The Smith Normal Form is defined over the integers, not modulo a prime.

$$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$$

Important observation

- Row and column operations commute with modular reduction.
- That is:

$$(UAV) \bmod p = (U \bmod p)(A \bmod p)(V \bmod p)$$

- Therefore, performing SNF steps modulo *p* preserves the algebraic structure of the transformation.

Walkthrough of Modular Smith Normal Form Algorithm

Step 5 — Repeat for different primes

- The same procedure is applied for several primes:

$$p_1, p_2, p_3, \dots$$

- Each prime provides **partial information** about the invariant factors of the true integer SNF.

Final Step — Chinese Remainder Theorem

- The Chinese Remainder Theorem is used to reconstruct the **integer determinantal divisors**, from which the invariant factors of the Smith Normal Form are obtained.

Note: The Chinese Remainder Theorem states that if an integer is known modulo several **pairwise coprime moduli**, then its exact value is **uniquely determined** modulo the product of those moduli.
In the modular SNF algorithm, each computation modulo a prime ppp reveals **partial information** about the invariant factors of the Smith Normal Form.
By performing the algorithm for multiple primes and combining the results using CRT, we can **reconstruct the exact integer Smith Normal Form**.
CRT guarantees that this reconstruction is **exact**, not an approximation.

Classical SNF vs Modular SNF

Classical (Euclidean) SNF

$$A = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

$$R_2 \leftarrow R_2 - R_1$$

$$\begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & 8 \\ 4 & 6 \end{pmatrix}$$

$$C_2 \leftarrow C_2 - C_1$$

$$\begin{pmatrix} 6 & 8 \\ 4 & 6 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & 2 \\ 4 & 2 \end{pmatrix}$$

$$R_1 \leftarrow R_1 - R_2$$

$$\begin{pmatrix} 6 & 2 \\ 4 & 2 \end{pmatrix} \longrightarrow \begin{pmatrix} 2 & 0 \\ 4 & 2 \end{pmatrix}$$

$$R_2 \leftarrow R_2 - 2R_1$$

$$\begin{pmatrix} 2 & 0 \\ 4 & 2 \end{pmatrix} \longrightarrow \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

$$\boxed{\mathrm{SNF}(A) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}}$$

↓

Modular SNF

$$A = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

$$A \bmod 5 = \begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix}$$

$$C_2 \leftarrow C_2 - 3C_1 \pmod 5$$

$$\begin{pmatrix} 1 & 3 \\ 0 & 4 \end{pmatrix} \to \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix} \pmod 5$$

$$\text{(intermediate)} \quad \mathrm{diag}(1,4) \pmod 5$$

$$\Delta_1 = \gcd(6,8,10,14) = 2$$

$$\Delta_2 = |\det(A)| = |6 \cdot 14 - 8 \cdot 10| = 4$$

$$d_1 = \Delta_1 = 2, \qquad d_2 = \Delta_2/\Delta_1 = 2$$

$$\mathrm{SNF}_{\mathbb{Z}}(A) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

Advantages of Modular Smith Normal Form

- Avoids coefficient explosion by keeping all intermediate values small
- Uses modular arithmetic, which is faster and more memory-efficient
- Allows independent computations for different primes
- Naturally supports parallel and distributed execution
- Scales well to large and sparse integer matrices
- Reduces risk of overflow and numerical instability
- Guarantees exact integer results via Chinese Remainder Theorem
- Widely used in modern computer algebra systems and large-scale applications

Fraction-Free SNF Algorithms

Core Logic of the Algorithm

In the **classical EEA-based SNF algorithm**, you repeatedly:

- Multiply one row by a number
- Subtract multiples of one row from another
- Normalize entries by division

The **fraction-free SNF algorithm does NOT change the goal.**

It still wants:

- the same diagonal SNF
- the same invariant factors
- the same unimodular row/column transformations

What it changes is **how divisions are handled**.

**This idea is related to fraction-free elimination techniques such as Bareiss-type methods, but we focus on the general concept rather than a specific algorithm.*

Walkthrough of Fraction Free SNF

$$A = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

$$\mathrm{SNF}(A) = \begin{pmatrix} d_1 & 0 \\ 0 & d_2 \end{pmatrix}, \qquad d_1 \mid d_2$$

- Instead of dividing during intermediate steps, fraction-free methods restructure computations using **integer multiplication and gcd operations**.
- Division is performed only when it is **guaranteed to be exact**, ensuring that all intermediate matrices contain **only integers**.
- This avoids large fractions, controls coefficient growth, and results in **more stable and efficient symbolic computation**.
- `new row =` $\dfrac{a \cdot r_1 - b \cdot r_2}{\gcd(a,b)} \in \mathbb{Z}$
- The numerator is formed using integer operations, and division by `gcd(a, b)` is exact by definition.

Classical SNF

$$A = \begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

$$R_2 \leftarrow R_2 - R_1$$

$$\begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & 8 \\ 4 & 6 \end{pmatrix}$$

$$C_2 \leftarrow C_2 - C_1$$

$$\begin{pmatrix} 6 & 8 \\ 4 & 6 \end{pmatrix} \longrightarrow \begin{pmatrix} 6 & 2 \\ 4 & 2 \end{pmatrix}$$

$$R_1 \leftarrow R_1 - R_2$$

$$\begin{pmatrix} 6 & 2 \\ 4 & 2 \end{pmatrix} \longrightarrow \begin{pmatrix} 2 & 0 \\ 4 & 2 \end{pmatrix}$$

$$R_2 \leftarrow R_2 - 2R_1$$

$$\begin{pmatrix} 2 & 0 \\ 4 & 2 \end{pmatrix} \longrightarrow \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

$$\boxed{\mathrm{SNF}(A) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}}$$

Fraction Free SNF

$$\begin{pmatrix} 6 & 8 \\ 10 & 14 \end{pmatrix}$$

$$R_2 \leftarrow \frac{6R_2 - 10R_1}{\gcd(6,10)}$$

$$\begin{pmatrix} 6 & 8 \\ 0 & 2 \end{pmatrix}$$

$$C_2 \leftarrow \frac{6C_2 - 8C_1}{\gcd(6,8)}$$

$$\begin{pmatrix} 6 & 0 \\ 0 & 6 \end{pmatrix}$$

$$d_1 = \gcd(6,8,10,14) = 2$$

$$d_2 = \frac{\det(A)}{d_1} = \frac{4}{2} = 2$$

$$\boxed{\mathrm{SNF}(A) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}}$$

Why "Fraction Free"? (Advantages)

The Fraction-Free strategy stands out by improving efficiency in symbolic computation through strict integer arithmetic, making it especially effective for exact linear algebra problems.

Avoidance of Fraction

Fraction-free algorithms eliminate the use of divisions that introduce rational numbers. By restructuring computations using integer multiplication and gcd operations, all intermediate values remain integers, preventing costly fraction growth.

Controlled Coefficient Growth

By avoiding fractions, fraction-free SNF algorithms limit the size of intermediate coefficients.
This results in smaller numbers throughout the computation, reducing the cost of arithmetic and gcd operations compared to classical methods.

Improved Practical Performance

Although the asymptotic complexity remains similar to classical SNF, fraction-free algorithms achieve significantly better runtime in practice.
Reduced bit-complexity and lower memory usage make them well-suited for large symbolic matrices.

Divide-and-Conquer SNF Algorithms

Core Logic of the Algorithm

- **Recursive Reduction:** Unlike traditional iterative methods, the algorithm processes the matrix by reducing its size step-by-step from (n x n) to (n-1) x (n-1).
- **Sub-Matrix Processing:** Once the first row and column are cleared (via Pivot operations), the remaining part (Sub-matrix) is treated as a **new, independent, and smaller problem**.
- **Algorithm Flow:** The function calls itself recursively to solve the sub-matrix. This strategy breaks down a complex matrix diagonalization problem into manageable, hierarchical steps.

Classic vs Modern:

- **Classic (Iterative)** :Element-based loops,complex statemanagement, prone tonumerical instability.
- **Modern (Recursive)**Sub-problem based,**dynamic pivot selection**,mathematically robust,and structurally clean.

Why "Divide and Conquer"? (Advantages)

The Divide and Conquer strategy stands out with the significant performance improvements it offers, especially in large-scale linear algebra problems.

Algorithmic Clarity:

It offers a clean and readable structure that strictly follows the "Divide & Conquer" paradigm, replacing complex nested loops with recursive function calls.

Robustness & Correctness:

Ensuring the critical **Divisibility Condition** $(d_i|d_{i+1})$ is much more reliable in this step-by-step process compared to block-based methods.

Dynamic Adaptation:

Since the **Pivot Selection** scans the entire active sub-matrix at every recursive step, it effectively minimizes coefficient growth and avoids numerical instability

Walkthrough of D&C Algorithm

Step 1: The Pivot Strategy (Targeting the Smallest)

- **Goal:** We need a "leader" for the current step.
- **Action:** We scan the entire matrix to find the non-zero entry with the **smallest absolute value**.
- **Why?** Smaller numbers act as better divisors (GCD) to eliminate other entries efficiently.
- **Result:** We swap rows and columns to move this "best pivot" to the top-left position (1,1).

Step 2: The Cleanup (Elimination via GCD)

- **Goal:** Isolate the pivot. The pivot wants to be alone in its row and column.
- **Action:** We use **Euclidean Division**.
  - Subtract multiples of the Pivot from other entries in the first column to make them zero.
  - Do the same for the first row.
- **Visual:** Imagine wiping the slate clean, leaving only the Pivot at (1,1).

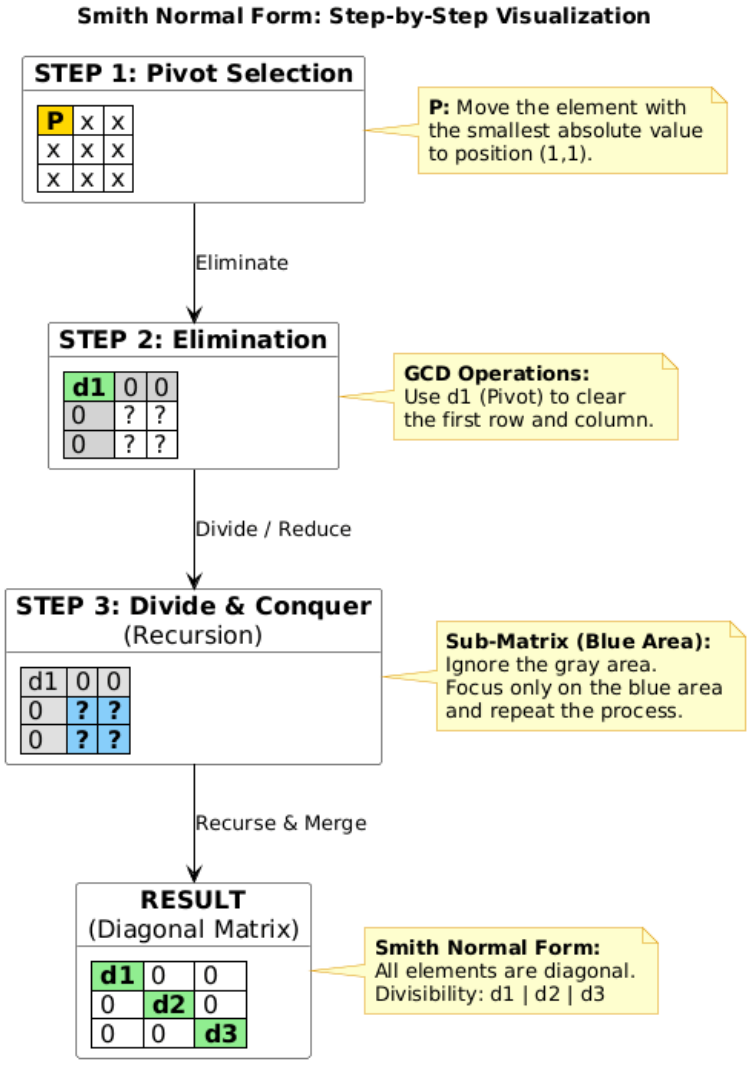
Step 3: The Recursive Leap (Divide and Conquer)

- **Goal:** Solve the rest of the problem without worrying about the cleared part.
- **Action:** We "ignore" the first row and first column.
- **The Logic:** We take the remaining bottom-right submatrix and treat it as a **brand new, smaller problem**.

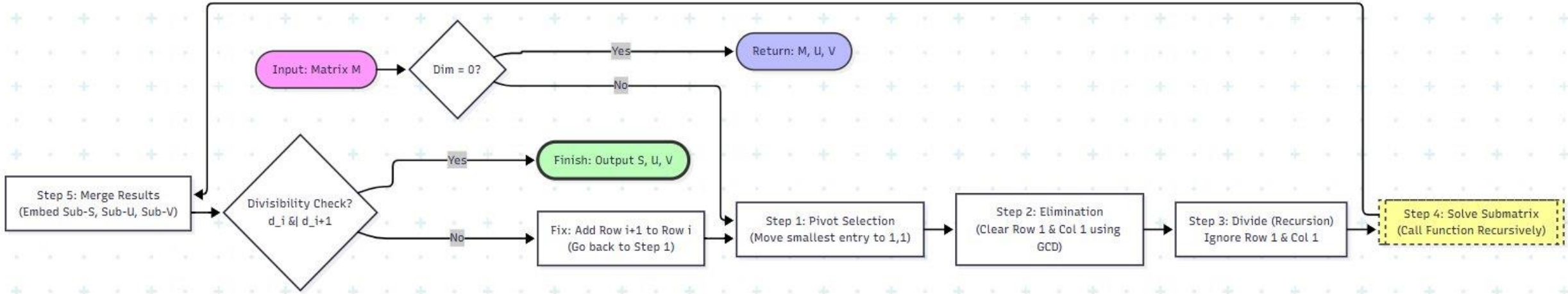
Step 4: The Final Check (Ensuring Hierarchy)

- **Goal:** The Smith Normal Form requires strict divisibility: $d_1 \mid d_2 \mid \dots \mid d_r$.
- **Action:** After the recursion returns, we check if our Pivot (d_1) divides the next diagonal entry (d_2).
- **Fix:** If it doesn't divide, we perform a simple row addition to mix them and restart the elimination process to fix the GCD.
- **Outcome:** A perfect diagonal matrix where every number divides the next one.

UML Diagram of D&C Algorithm



Flowchart of the D&C Algorithm



Determinant-Based SNF Algorithm

Introduction

The **determinant-based SNF algorithm** is the **most direct and theoretical method** to compute SNF. It is based entirely on **determinants of submatrices** and the **greatest common divisor (gcd)**.

The definition of SNF is based on **determinants of minors**:

- Determinants measure the "volume" or "strength" of linear independence
- All invariant factors are encoded in these determinants

The numbers that appear in the Smith Normal Form are determined by the **gcd of determinants of all minors** of the matrix.

Instead of transforming the matrix step by step, we:

- Look at all smaller submatrices
- Compute their determinants
- Use gcd operations to extract the SNF values

  So, SNF is obtained by **measuring**, not transforming.

Determinant-Based SNF Algorithm

Key Concepts

To truly understand the determinant-based SNF algorithm—the most direct and theoretical method for computing Smith Normal Form—the key concepts are the determinants of submatrices and the greatest common divisor (gcd), upon which the entire algorithm is built.

- **Minors**: A kxk minor is a submatrix formed by choosing; k rows and k columns.

  Example: 1x1, 2x2...

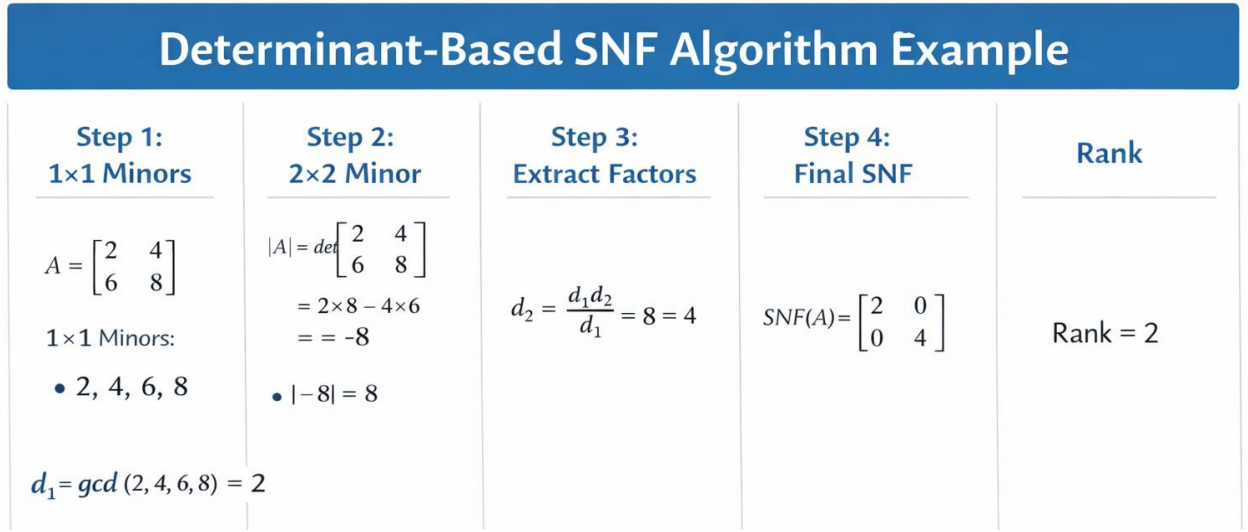
- **Minor Determinant:** Determinant of an kxk minor
- Δ_k: GCD of determinants of all kxk minors
- D_k: k'th diagonal entry of the Smith Normal Form

Determinant-Based SNF Algorithm

Determinant-Based SNF — Step-by-Step Algorithm

- Input
  - Let A be an integer matrix of size **n × m**
  - Set k=min(n,m), k>0
- Step 1 — 1×1 Minors – Start From 1x1 Matrix
  - Treat each matrix entry as a 1×1 minor
  - Compute: d1=gcd(all entries of A)
- Step 2 — i×i Minors – Compute up to k
  - For **i = 2, 3, … , k**:
    - Enumerate all i×i submatrices of A
    - Compute their determinants
    - Compute: Δi=gcd(all non-zero i×i determinants)
- Step 3 — Extract Invariant Factors
  - For **i ≥ 2**, compute: di=Δi / Δi−1
  - Use Δ1=d1
- Output

  SNF(A)=diag(d1,d2,…,dr)
  - The largest i with Δi≠0 gives the **rank**



Determinant-Based SNF Algorithm

Determinant-Based SNF —Advantages & Disadvantages

Advantages

- Conceptually **simple and definition-based**
- Directly uses **determinants and GCD operations**
- No need for row or column transformations
- Clearly reveals **invariant factors**
- Useful for **theoretical explanation and teaching**

Disadvantages

- **Computationally expensive** (many minors)
- Poor scalability for **large matrices**
- Determinant computation is costly
- Not practical for numerical or large-scale applications
- Mainly of **theoretical interest**

SageMath Comparison and Validation

What is SageMath?

- It is an **open-source, Python-based** software system designed for advanced mathematical computations, particularly in linear algebra and number theory.

Why did we choose it?

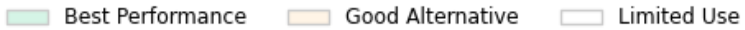
- **Independent Cross-Validation:** Used as a third-party verification tool to validate our Julia implementations beyond internal tests.
- **Academic Gold Standard:** Widely accepted and trusted in the mathematics research community as a reliable computational reference.
- **Online Accessibility:** Requires no installation (via `sagecell.sagemath.org`), making it ideal for instant and reproducible verification.

Validation Process

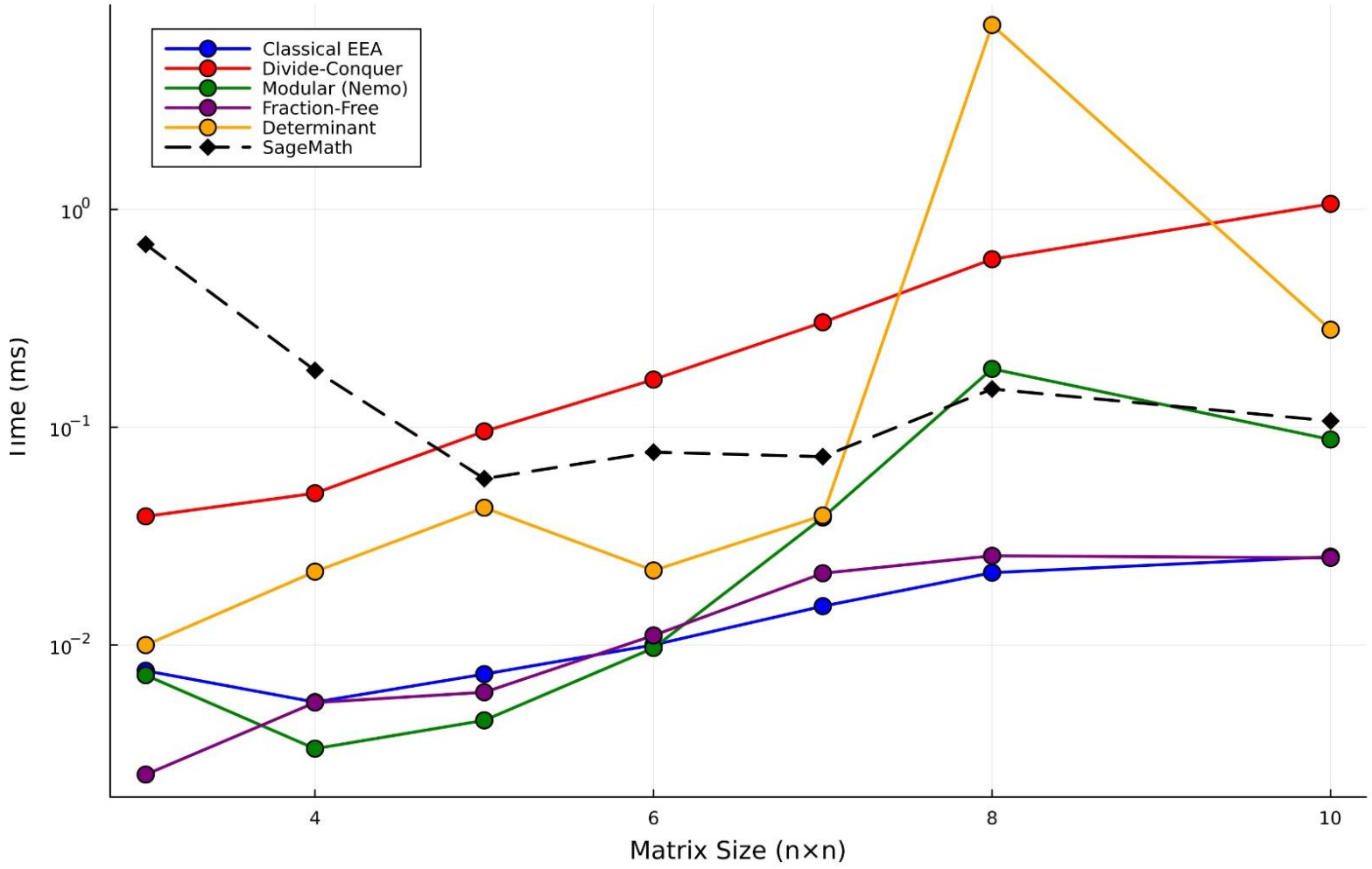
- **Identical Test Matrices:** We tested the exact same input matrices in both our Julia implementations and SageMath to ensure fair comparison.
- **Direct Comparison:** Elementary divisors were extracted and compared between the two systems to verify mathematical equivalence.
- **100% Match Rate:** All test cases produced identical results, confirming the correctness of our implementations.

Smith Normal Form Algorithms Comperehensive Comparison

Algorithm	Best Use Case	Strengths	Weaknesses	Overall Verdict
Classical EEA	Small matrices (≤5×5)	Very fast, low overhead	Coefficient explosion	□ Small-scale only
Divide-Conquer	Large matrices (theoretical)	Good asymptotic complexity	High recursion overhead	⚠ Niche use
Modular	Large matrices (≥6×6)	Scalable, exact, stable	Higher constant factors	□□ Best overall choice
Fraction-Free	Medium matrices (5×5–8×8)	Exact arithmetic, balanced	Some coefficient growth	⚠ Good alternative
Determinant-Based	Didactic / theoretical only	Conceptually simple	Combinatorial explosion	□ Not practical
SageMath	All sizes (adaptive)	Highly optimized CAS	Black-box heuristics	□□ CAS baseline



Execution Time vs Matrix Size of Smith Normal Form Algorithms



Key Takeaways Based On Our Evaluation

- **No single SNF algorithm is universally optimal**
  Performance strongly depends on matrix size, coefficient growth, and arithmetic strategy.
- **Modular SNF (Nemo / SageMath) achieves the best overall performance**
  Fastest and most scalable for medium and large matrices due to modular arithmetic and CRT reconstruction.
- **Classical EEA-based SNF is ideal only for small matrices**
  Very fast for small inputs, but suffers from coefficient explosion as size increases.
- **Fraction-free SNF provides numerical stability**
  Avoids divisions and preserves exact arithmetic, but scales worse than modular methods.
- **Divide-and-conquer SNF shows theoretical promise, practical overhead**
  Recursion and memory costs reduce real-world performance.
- **Determinant-based SNF is not computationally practical**
  Exponential growth of minors limits usage to theoretical or educational contexts.

Overall Recommendation

Modular SNF is the preferred choice for practical and large-scale computations, while classical and fraction-free methods remain useful for small-scale and validation tasks.

References

- Middeke, Johannes, and David J. Jeffrey. “Matrix Factoring by Fraction-Free Reduction.” Version 1, arXiv, 2016.
- Levandovskyy, Viktor, and Kristina Schindelar. “Fraction-Free Algorithm for the Computation of Diagonal Forms Matrices over Ore Domains Using Gröbner Bases.” Journal of Symbolic Computation, vol. 47, no. 10, Oct. 2012, pp. 1214–32.
- Storjohann, A. & Department of Computer Science, University of Waterloo, Ontario, Canada. (n.d.). Near optimal algorithms for computing Smith normal forms of integer matrices. In Technical Report CS [Report].
- Dumas, J.-G., 1, Saunders, B. D., 2, & Villard, G., 3. (2001). On efficient sparse integer matrix Smith normal form computations. In Journal of Symbolic Computation (Vols. 32–32, Issues 1–2, pp. 71–99).

THANK YOU FOR LISTENING!