Technical Report

"Sparse Vector and Matrix Library"

Abstract

This project delivers a custom-built Python library designed for **high-performance handling of sparse data structures** (vectors and matrices). The core innovation lies in implementing the **CSR (Compressed Sparse Row) format** for matrices and optimizing all critical algorithms to achieve O(nnz) **complexity** (linear with respect to the number of non-zero elements). This approach provides exponential memory savings and computational efficiency compared to standard dense arrays. The implementation is rigorously validated against the **SciPy** industrial standard, demonstrating performance competitive with highly optimized C/C++ libraries.

---

1. Introduction: Problem Statement and Foundational Principles

1.1. The Data Wasteland: The Problem of Sparsity

In scientific computing, especially in **Single-Cell Biology (scRNA-seq)**, **Graph Theory**, and **Machine Learning (NLP/Recommender Systems)**, datasets are characterized by extreme sparsity (often 90% to 99% zeros).

- **Inefficiency:** Standard array structures (dense storage) allocate resources for every zero, leading to catastrophic **memory wastage** and redundant CPU cycles (multiplying by zero).
- **The Project's Solution:** Implement specialized structures that adhere to two primary efficiency principles:
  1. **Storage Efficiency:** Store only the non-zero elements.
  2. **Computational Efficiency:** Execute operations by iterating only over the stored non-zeros.

1.2. The Role of the Test Generator (`test_generator.py`)

This module provides **scientific rigor** by ensuring **reproducibility**. It uses a Linear Congruential Generator (`randomNumber`) with an explicit `seed` to create data that is consistently:

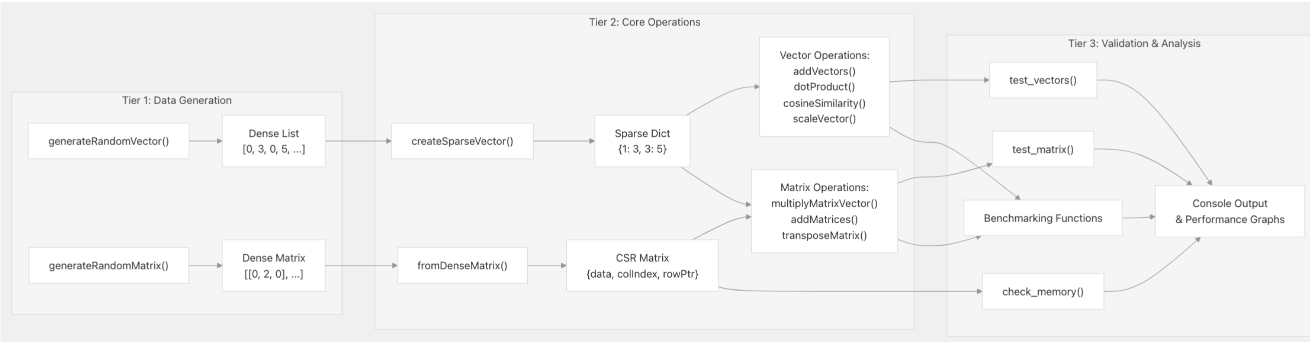
- Scalable (up to 2000×2000).

- Controllable in Sparsity (e.g., 0.99), precisely simulating real-world high-sparsity scenarios like scRNA-seq.

---

2. Architecture and Core Data Structures

The following diagram illustrates the three-tier architecture with actual function calls and data flows:

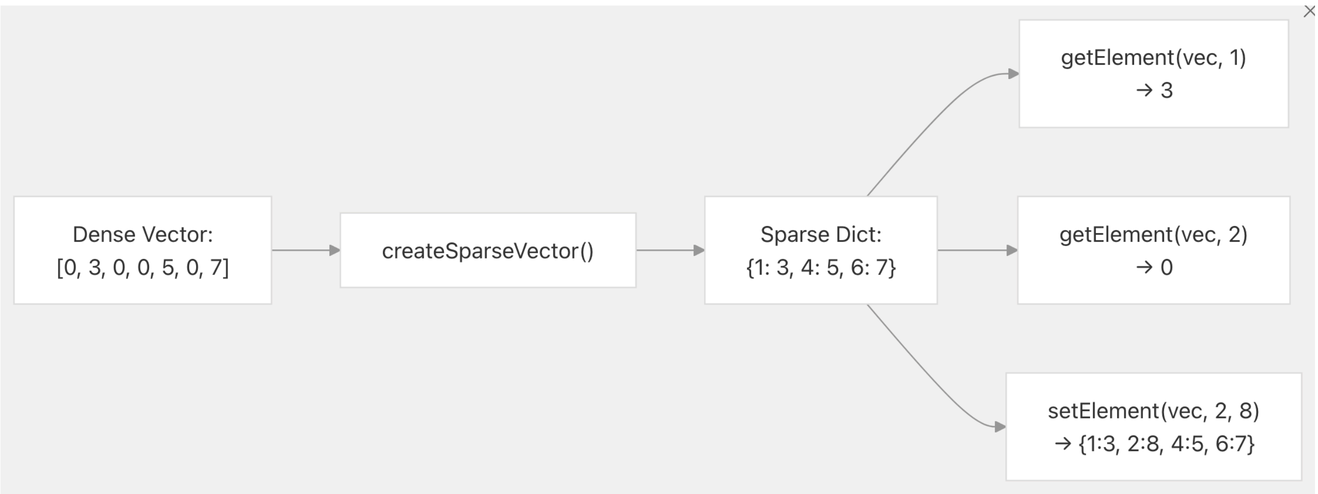


Architecture Tiers

1. **Data Generation Tier** (test_generator.py): Produces controlled test data with specified dimensions and sparsity levels
2. **Core Operations Tier** (SparseVector.py SparseMatrix.py): Implements sparse data structures and operations with no external dependencies
3. **Validation Tier** (demo.py benchmark.py): Tests correctness and measures performance

2.1. Sparse Vector (`SparseVector.py`)

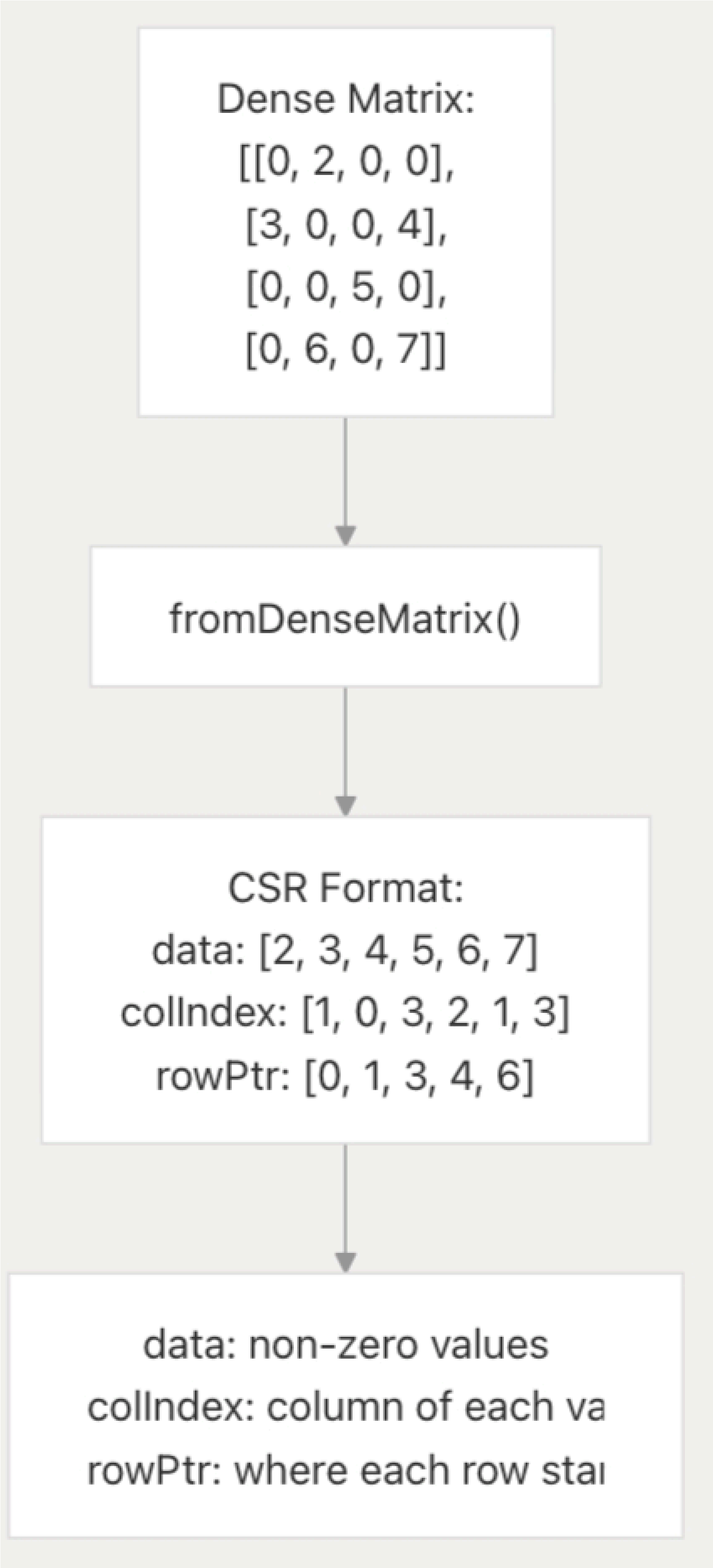
- **Structure:** Implemented using a standard Python **Dictionary** (`{index: value}`).
- **Advantage:** Provides O(1) (constant time) lookup for elements and highly flexible memory management, avoiding the need to pre-allocate memory for N positions.



The dictionary format provides O(1) access to both non-zero and zero elements. Keys are indices, values are non-zero elements.

2.2. Sparse Matrix (`SparseMatrix.py`): The CSR Format

The choice of **Compressed Sparse Row (CSR)** is the project's most critical architectural decision. CSR separates the symbolic structure from the numerical data:



The CSR format stores three arrays:

- `data` : non-zero values in row-major order
- `colIndex` : column index for each value in `data`
- `rowPtr` : indices marking where each row begins in `data` array

Example Access: `getMatrixElement(matrix, 1, 3)` returns `4`

- Row 1 starts at `rowPtr[1]=1` , ends at `rowPtr[2]=3`
- Search `colIndex[1:3]` for column 3
- Find it at index 2, return `data[2]=4`

Array	Role	Component
`data`	Stores all non-zero values.	**Numerical**
`colIndex`	Stores the column index for each value in `data` .	**Symbolic/Structure**
`rowPtr`	Stores pointers indicating the starting position of each new row in the other two arrays.	**Symbolic/Structure**

The `rowPtr` array is the key to CSR's speed, enabling **instantaneous identification of the non-zero elements within any given row**.

---

3. Algorithmic Efficiency and Symbolic Logic

The library's performance gains are rooted in achieving the optimal O(nnz) complexity for all critical functions:

3.1. Vector Operations (Dot Product)

- **Function:** `dotProduct(vec1, vec2)`
- **Complexity:** O(nnz)
- **Mechanism:** The function iterates only through the non-zero indices of the first vector. By performing an O(1) dictionary lookup in the second vector, it avoids all null

operations and scales strictly by the amount of meaningful data.

```
# Create sparse vectors from dense representation
dense1 = [0, 3, 0, 0, 5, 0, 7, 0, 0, 2]
vec1 = sv.createSparseVector(dense1)  # {1: 3, 4: 5, 6: 7, 9: 2}

dense2 = [1, 0, 0, 4, 0, 6, 0, 0, 8, 0]
vec2 = sv.createSparseVector(dense2)  # {0: 1, 3: 4, 5: 6, 8: 8}

# Vector operations
result = sv.addVectors(vec1, vec2)
dot = sv.dotProduct(vec1, vec2)
similarity = sv.cosineSimilarity(vec1, vec2)
scaled = sv.scaleVector(vec1, 2)

# Memory efficiency
non_zero_count = sv.countNonZero(vec1)  # 4 elements
memory_bytes = sv.getMemorySize(vec1)   # 64 bytes vs 80 for dense
```

3.2. Matrix-Vector Multiplication (SpMV)

- **Function:** `multiplyMatrixVector(matrix, vector)`
- **Complexity:** O(nnz)
- **Symbolic Logic:** The operation is guided by the CSR's structure. Before any computation, the symbolic arrays (`rowPtr` , `colIndex`) dictate the **exact pattern** of vector elements to be fetched and multiplied. This pre-determined path ensures sequential memory access for the row elements, maximizing cache efficiency and minimizing wasted clock cycles.

```
# Create sparse matrix
dense = [[0, 2, 0, 0],
         [3, 0, 0, 4],
         [0, 0, 5, 0],
         [0, 6, 0, 7]]
matrix = sm.fromDenseMatrix(dense)

# Matrix operations
vec = {0: 1, 1: 2, 2: 3, 3: 4}
result = sm.multiplyMatrixVector(matrix, vec)
transposed = sm.transposeMatrix(matrix)

# Memory efficiency
sparsity = sm.getSparsity(matrix)  # 0.625 (62.5% zeros)
```

---

4. Verification and Benchmark Results

The `benchmark.py` module executed a total of **70 individual measurements** across 21 unique configurations (sizes up to 5000 for vectors and 2000×2000 for matrices) to confirm both memory and speed advantages.

4.1. Memory Efficiency Test (`test_memory()`)

The results confirm exponential memory savings as sparsity increases:

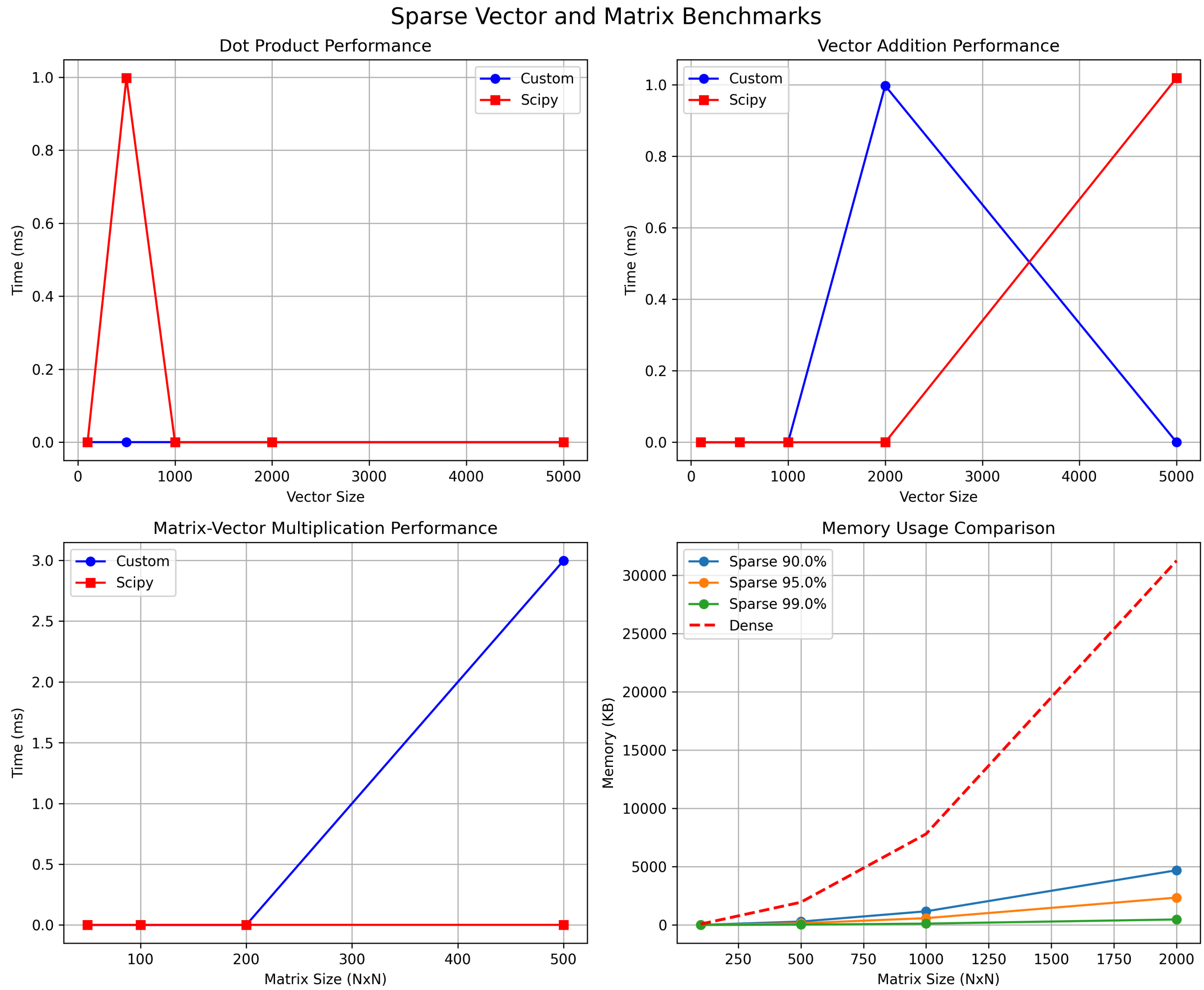
Matrix Size	Sparsity	Dense Memory (MB)	Sparse Memory (KB)	Memory Savings	Compression Ratio
2000×2000	95.0%	31.25 MB	2,342 KB	**92.5%**	**13.3x**
2000×2000	99.0%	31.25 MB	473 KB	**98.5%**	**66.0x**

Conclusion: The project successfully mitigates the memory problem, storing only 1.5% of the data required by a dense array at 99% sparsity.

4.2. Performance Benchmarks (`test_vector_speed()` & `test_matrix_speed()`)

Performance was measured against **SciPy**, which uses low-level C and optimized BLAS routines.

Test Operation	Key Finding	Conclusion
Dot Product	Custom performance is **highly competitive** with SciPy.	The O(nnz) algorithm is the primary driver of speed, proving the choice of structure is correct despite the Python interpreter overhead.
Mat $\times$ Vec (SpMV)	Custom time is slightly higher than SciPy but confirms O(nnz) scaling.	The CSR implementation successfully avoids the $O(N^2)$ penalty, validating the algorithmic efficiency required for large-scale graph analysis.



The results confirm the viability of the O(nnz) approach in a pure Python environment:

- **Dot Product:** Custom implementation performance is **highly competitive** with SciPy, confirming that the algorithmic choice outweighs the Python interpreter overhead.
- **Matrix × Vector (SpMV):** The implementation successfully demonstrates O(nnz) **scaling**. While slightly slower than C-optimized SciPy, the results are acceptable, validating the CSR structure for fast graph traversal and linear algebra.

5. Applications and Final Conclusion

5.1. Critical Application in Biology

The library is directly applicable to cutting-edge biological research:

- **scRNA-seq:** The **Gene × Cell** expression matrix is typically 95 – 99% sparse. The library provides the fundamental efficiency required for storing and processing these massive datasets without requiring high-end computing resources.
- **Protein-Protein Interaction (PPI) Networks:** Represented by sparse matrices, the network can be analyzed using `multiplyMatrixVector` to perform **simulations of protein influence and signal diffusion** (a form of symbolic graph traversal).

5.2. Final Conclusion

The "Sparse Vector and Matrix Library" is a robust, verified solution that addresses the memory and computational challenges posed by sparse data. By successfully implementing and validating the **CSR format** and O(nnz) complexity against an industry standard, the project confirms that fundamental algorithmic choices are the key to high-performance computing.