# Technical write-up

**Overview**

The data structure is implemented in Python. The `BlockDiagonalMatrix` class is designed to represent and perform efficient operations on matrices composed of smaller diagonal blocks. The goal is to leverage the inherent structure of such matrices to reduce both memory usage and computational complexity for common matrix operations like addition, multiplication, and inversion.

**Data Structure**

1. **Storage Efficiency**: Each block diagonal matrix is represented as a 3D NumPy array, `blocks`, where:
    - `blocks[i, j]` is a 1D array representing the diagonal elements of the `(i, j)`-th block in the matrix.
    - This reduces the memory required for storage since only the diagonal elements of each block are stored (not the zero elements).
2. **Matrix Dimensions**:
    - `n`: The number of blocks along one dimension.
    - `d`: The size of each diagonal block.
3. For an $nd \times nd$ matrix, there are $n \times n$ blocks, each of size $d \times d$.

**Implemented Methods**

1. **`__add__` Method (Matrix Addition)**
    - Performs element-wise addition of two block diagonal matrices.
    - Efficient because it directly adds the arrays representing diagonal matrix-blocks without expanding to full matrices.
2. **`__sub__` Method (Matrix Subtraction)**
    - Similar to `__add__`, but performs element-wise subtraction.
3. **`__mul__` Method (Dot Product)**
    - Computes the element-wise dot product of the corresponding blocks of two matrices.
    - Only diagonal elements need to be multiplied.
4. **`__matmul__` Method (Matrix Multiplication)**
    - Implements matrix multiplication using the `@` operator.
    - **Algorithm**:
        - Leverages NumPy's broadcasting capabilities by expanding the dimensions of the block matrices.
        - Multiplies the appropriate blocks together and sums over the common dimension.
    - This is more efficient than standard matrix multiplication due to the diagonal structure of each block.
5. **`inverse` Method (Matrix Inversion)**

- Efficiently computes the inverse of the block diagonal matrix using a divide-and-conquer strategy:
  - For each of the `d` diagonal positions, inverts a smaller submatrix (`n x n`).
  - Combines these inverses into the overall block diagonal matrix inverse.

6. **`numpy` Method**
   - Converts the block diagonal matrix representation back into a full NumPy array form.
   - Complexity: (O(n^2 \times d^2)), where it fills each block's diagonal elements into the full matrix.
7. **`__str__` and `_repr_pretty_` Methods**
   - Provides human-readable representations of the matrix for easy debugging and visualization.
   - The `__str__` method returns a string form of the full matrix, while `_repr_pretty_` improves the display in Jupyter notebooks.

**Efficiency Analysis**

1. **Space Efficiency**:
   - The use of a 3D NumPy array (`n x n x d`) to store only the diagonal elements of each block makes the class memory-efficient, especially for large matrices with many zeros.
2. **Time Complexity Improvements**:
   - Many operations that would traditionally require cubic time (like matrix multiplication and inversion) are reduced, thanks to the block diagonal structure.

**Comparison of time complexity**

The average required time to complete the matrix operations are compared with the full matrix operations using numpy. In case of small matrices, the difference is very negligible. However, the difference is more apparent for larger matrices. For this comparison, the used block matrix has a size of $n = 15$ and $d = 11$ making the full matrix size $165 \times 165$. The test is done using Jupyter notebook's %timeit magic using 7 runs and 20 loops.

| Operation | Implemented code | Numpy |
|---|---|---|
| Matrix addition | 8.64 µs ± 6.32 µs | 24.5 µs ± 7.64 µs |
| Matrix subtraction | 7.98 µs ± 11.1 | 21.8 µs ± 8.17 µs |
| Matrix dot product | 16.8 µs ± 9.72 µs | 30.2 µs ± 6.28 µs |
| Matrix multiplication | 190 µs ± 53.4 µs | 333 µs ± 92.6 µs |
| Matrix inversion | 1.29 ms ± 233 µs | 110 ms ± 18.5 ms |

**Library requirements**

numpy == 1.26.4

## Conclusion

The `BlockDiagonalMatrix` class successfully leverages NumPy's capabilities and the special properties of block diagonal matrices to provide efficient and scalable matrix operations. This design allows for fast computation, minimal memory usage, and is well-suited for applications dealing with large, structured sparse matrices.

By focusing on the inherent diagonal structure of the blocks, the implementation optimizes both space and time complexity, making it a valuable tool for high-performance computing tasks.

This write-up covers the primary aspects of your implementation, highlighting its design, efficiency, and applications. If you have any specific points or details you want to emphasize, let me know!