

<b>Matrix Ops - Problem Statement</b> .....	<b>1</b>
<b>Solution</b> .....	<b>1</b>
<b>Data Structure</b> .....	<b>1</b>
1. Block Structure .....	2
2. Diagonal Blocks.....	2
3. Sparsity .....	2
4. Example .....	2
<b>Algorithm</b> .....	<b>3</b>
1. Matrix Multiplication.....	3
2. Matrix Inversion.....	3
<b>Code</b> .....	<b>3</b>
<b>Output</b> .....	<b>4</b>

## Matrix Ops - Problem Statement

Consider a matrix  $A$  of size  $R^{nd \times nd}$ , where each non-overlapping  $d \times d$  block of the matrix,  $D_{ij}$ , is a diagonal matrix. So, the matrix consists of  $n^2$  such blocks. An example of such a matrix is shown below:

$$\begin{bmatrix} D_{11} & D_{12} & D_{13} & \cdots & D_{1n} \\ D_{21} & D_{22} & D_{23} & \cdots & D_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ D_{n1} & D_{n2} & D_{n3} & \cdots & D_{nn} \end{bmatrix}$$

Construct an efficient data structure to represent such matrices and devise algorithms to perform matrix operations, such as matrix multiplications and matrix inverse, on the data structure you designed. Provide a technical write-up of your solution along with associated code implementing your solution.

## Solution

### Data Structure

There are 3 points to consider in designing an efficient data structure to represent the large matrix A of size  $n \times n$  so that we only store the diagonal entries of the blocks and save memory:

### 1. Block Structure

The matrix is divided into smaller blocks, each of size  $d \times d$  times. Like a big grid (matrix) made up of smaller square grids (blocks).

### 2. Diagonal Blocks

Each of these smaller blocks  $D_{ij}$  is a diagonal matrix. A diagonal matrix is a matrix where all the numbers outside the diagonal are zero. For example, in below  $D_{ij}$  Matrix, only the diagonal entries (1, 2, 3) are non-zero.

$$D_{ij} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

### 3. Sparsity

Most entries in this large matrix A are zero because the blocks  $D_{ij}$  are either empty (zero everywhere) or diagonal. Storing a very large matrix with mostly zeroes waste a lot of memory as we store every single number. Similarly, performing operations like matrix multiplication or finding the inverse becomes very slow because we unnecessarily compute with those zeroes.

Instead, we can:

- Store only the important (non-zero) values.
- Perform operations more efficiently by focusing only on the diagonal blocks.

### 4. Example

For `BlockDiagonalMatrix(n, d)`, We only need to store the diagonal blocks of size  $d \times d$  since off-diagonal blocks are always zero. Each diagonal block itself is a diagonal matrix.

- List can be used to store  $n$  diagonal blocks.
- Each diagonal block  $D_{ii}$  can be stored as a 1D list of its diagonal elements (of size  $d$ ).
- For example, for  $n=3$  and  $d=2$ :

```
blocks = [  
    [a11, a12], # Diagonal of D_11
```

```

    [b21, b22], # Diagonal of D_22
    [c31, c32] # Diagonal of D_33
]

```

## Algorithm

### 1. Matrix Multiplication

To multiply two block diagonal matrices A and B, both of size  $nd \times nd$ :

1. **Input:**
  - Two BlockDiagonalMatrix objects, A and B, with n blocks each of size  $d \times d$ .
2. **Steps:**
  - Initialize an empty list C to store n diagonal blocks for the result.
  - For each diagonal block i (from 1 to n):
    - Multiply the diagonal elements of  $D_{ii}$  from A with the corresponding diagonal elements of  $D_{ii}$  from B.
    - Store the result as the i-th block in C.
3. **Output:**
  - Return a new BlockDiagonalMatrix with n blocks.

**Complexity:**  $O(nd)$ . since only n blocks with d elements each are processed. Complexity is reduced from being cubic to just linear because of the efficient data structure design.

### 2. Matrix Inversion

To invert a block diagonal matrix A:

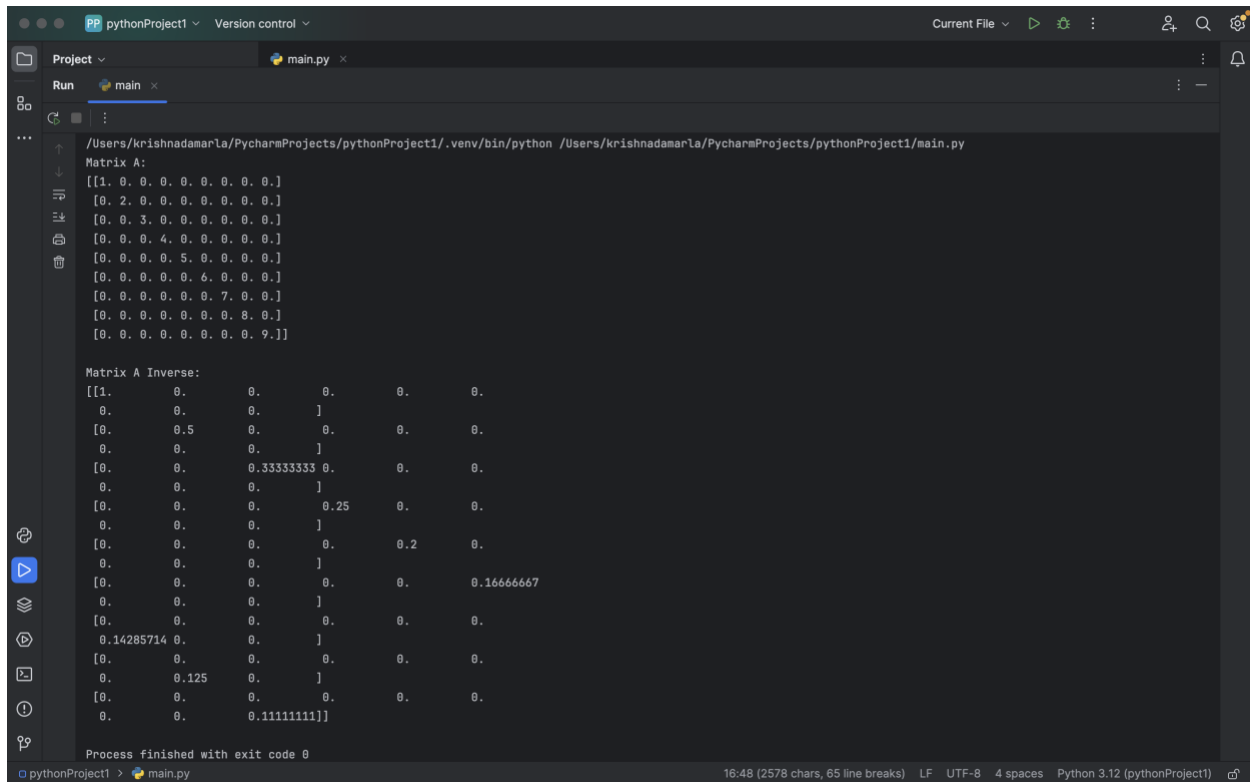
1. **Input:**
  - A BlockDiagonalMatrix object, A with n block.
2. **Steps:**
  - Initialize an empty list C to store n diagonal blocks for the result.
  - For each diagonal block i (from 1 to n):
    - Compute the reciprocal of each diagonal element of  $D_{ii}$ , ensuring none of the elements are zero.
    - Store the result as the i-th block in C.
3. **Output:**
  - Return a new BlockDiagonalMatrix representing the inverse.

**Complexity:**  $O(nd)$ . since only n blocks with d elements each are processed. Complexity is reduced from being cubic to just linear because of the efficient data structure design.

## Code

[https://github.com/i-krishna/Business-Analytics/blob/main/Data-Science/Python/matrix\\_multiply\\_inverse.py](https://github.com/i-krishna/Business-Analytics/blob/main/Data-Science/Python/matrix_multiply_inverse.py)

## Output



```
pythonProject1 Version control
Current File
main.py
Run main
/Users/krishnadamarla/PycharmProjects/pythonProject1/.venv/bin/python /Users/krishnadamarla/PycharmProjects/pythonProject1/main.py
Matrix A:
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 3. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 4. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 5. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 6. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 7. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 8. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 9. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Matrix A Inverse:
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0.5 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.33333333 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.25 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0.2 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.16666667]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0.14285714 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0.125 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.11111111 0. 0. 0. 0. 0.]

Process finished with exit code 0
pythonProject1 > main.py
16:48 (2578 chars, 65 line breaks) LF UTF-8 4 spaces Python 3.12 (pythonProject1)
```