HAL Matrix

Rishitosh Kumar Singh

December 11, 2020

1 Problem

Consider a matrix A of size $\mathbb{R}^{nd \times nd}$, 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.

2 Solution

Data Structure to represent matrix stated in the problem statement must be memory efficient and should be compatible with various matrix operations. It can be observed in fig. 1 that due to presence of block diagonal matrix D_{ij}, the final matrix A is a sparse matrix if, $d \neq 1$. So, instead of saving all $nd \times nd$ elements, only $n^2 d$ elements need to be saved, thus reducing memory requirement by factor of $\frac{1}{d}$.

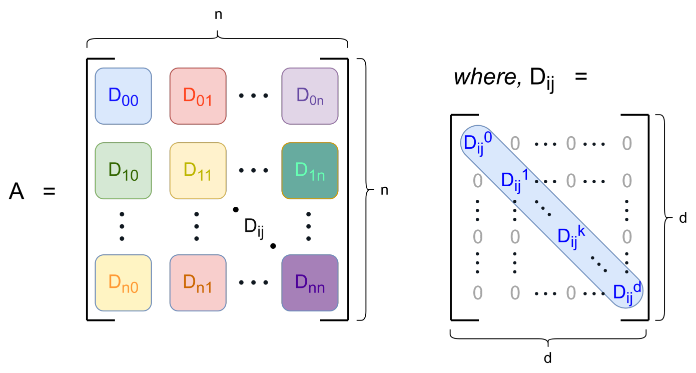


Figure 1: Matrix A

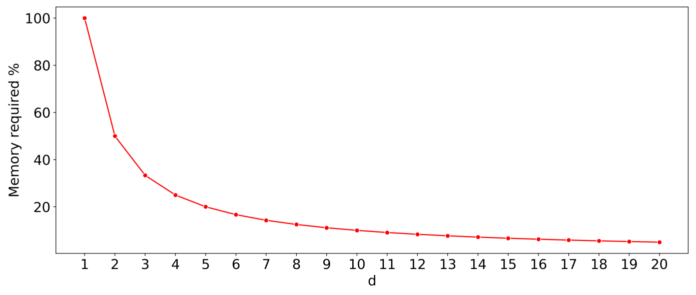


Figure 2: Memory Requirements of Hal Matrix (n=50)

Figure 2 shows the percentage of memory required if we increase d and keeping n constant when compared with NumPy arrays. From fig. 2 it can be observed that when $d = 1$, 100% memory is required, thus there will be no drop in memory use, and in this particular condition matrix is no more a sparse matrix.

It can also be observed, that elements that are 0, will never change after any matrix operation. So those elements can be ignored while applying matrix operation.

The best way to store these matrices of size $\mathbb{R}^{nd \times nd}$ is to store them in an array of shape $(n^2 \times d)$ and is represented in fig. 3, where each row stores diagonal elements of the block matrix.

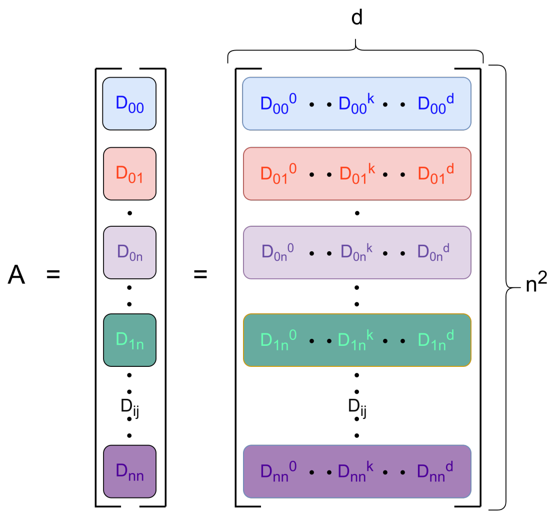
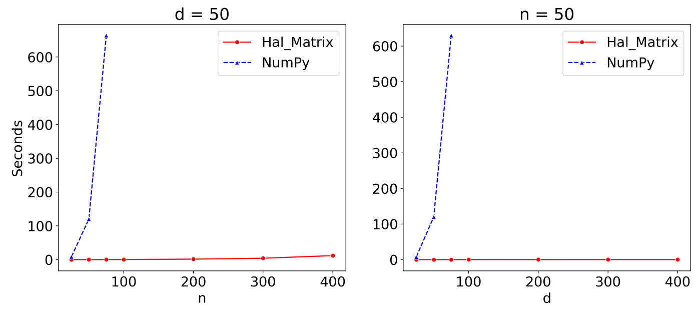
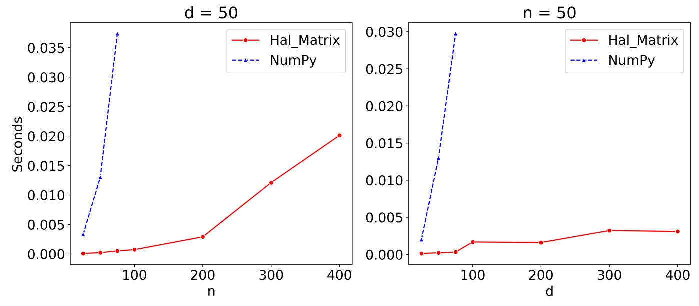


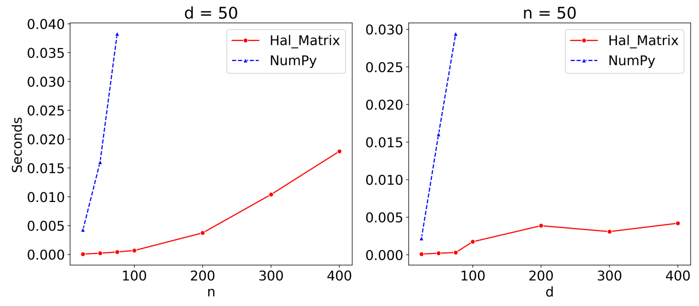
Figure 3: Matrix A representation



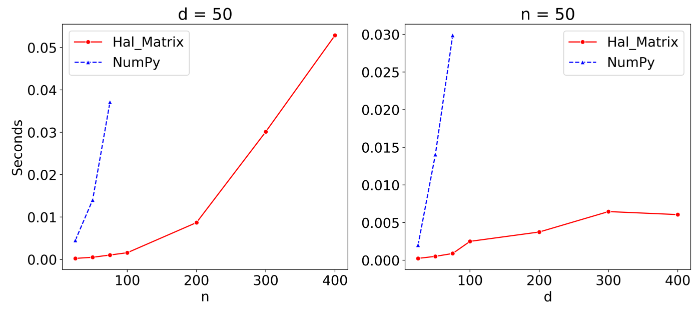
(a) Time taken to calculate dot product (in seconds)



(b) Time taken to add two matrices (in seconds)



(c) Time taken to perform element wise multiplication in two matrices (in seconds)



(d) Time taken to subtract one matrix from another (in seconds)

Figure 4: Comparison of NumPy arrays and proposed data structure to store matrix stated in problem statement

3 Matrix Operations

3.1 Dot Product

Algorithm 1: dot-product procedure

```
Input  : $x$ (First Hal_Matrix), $y$ (Second Hal_Matrix)
Output: $a$ (Output Hal_Matrix)
$a$ = Hal_Matrix($n = x.n, d = x.d$)
a.data ← list()
for $i \leftarrow 0$ to $x.n - 1$ do
    left = list()
    for $k \leftarrow i * x.n$ to $(i + 1) * x.n$ do
        left.append(x.data[k])
    end
    for $j \leftarrow 0$ to $x.n - 1$ do
        right ← list()
        for $k \leftarrow j$ to $(x.n)^2$, step $x.n$ do
            right.append(y.data[k])
        end
        res ← (left * right).sum(axis=0)
        a.append(res)
    end
end
```

The above algorithm is used to find the dot product of given two matrices and fig. 4a depicts the performance of the proposed data structure in comparison to NumPy arrays. NumPy arrays are fast as various operations are implemented in C language and avoid expensive "for" loops of python. The proposed data structure outperforms NumPy arrays when the matrix size is too large.

3.2 Inverse Operation

Gauss-Jordan Method is used to find the inverse of given hal_matrix. The algorithm used with the proposed data structure will return the inverse of hal_matrix if present or else it will raise NotInvertible exception.

4 Remarks

For complete source code of proposed data structure with algorithms checkout project repository