Homework 1

Saulo Mendes

December 2020

Hadamard, Kronecker and Khatri-Rao Products

*As the professor has allowed, the solutions in this work were implemented in Python. Each solution section will point out the respective substitutions.

Problem 1 For randomly generated matrices $\mathbf{A}$ and $\mathbf{B} \in \mathbb{C}^N \times N$, create an algorithm to compute the Hadamard Product $\mathbf{A} \odot \mathbf{B}$. Then, compare the run time of your algorithm with the operator .* of the *software* Octave/Matlab®. Plot the run time curve as a function of the number of rows/columns $N \in \{2, 4, 8, 16, 32, 64, 128\}$.

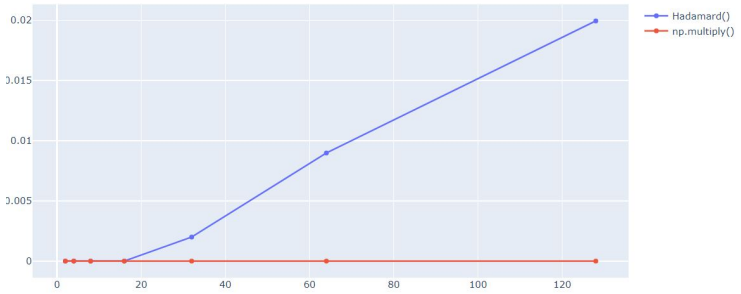
Solution: The custom implementation of the Hadamard product function performs element-wise multiplication on two square matrices by running through the indices i and j and multiplying the elements in place, as indicated in Fig. 1.

Figure 1: Hadamard Product custom implementation

```
def Hadamard(A, B):
    result = np.zeros(A.shape)

    for i in range(len(A)):
        for j in range(len(B)):
            result[i, j] = A[i, j]*B[i, j]

    return result
```

The function used for run time comparison was *multiply()*, from the Numpy package. The *multiply()* function performs element-wise multiplication on matrices, and it's equivalent to the .* operator from Octave/Matlab. When tested with matrices of $N \times N$ dimension for $N \in \{2, 4, 8, 16, 32, 64, 128\}$, we have the results indicated in Fig. 2. The pronounced increase in run time ih *Hadamard*() might be due to computational complexity $O(n^2)$ of the solution. The *np.multiply*() function must have a more optimized method.

Figure 2: Hadamard() and np.multiply() comparison



Problem 2 For randomly generated matrices $\mathbf{A}$ and $\mathbf{B} \in \mathbb{C}^N \times N$, create an algorithm to compute the Kronecker Product $\mathbf{A} \otimes \mathbf{B}$. Then, compare the run time of your algorithm with the operator *kron*($\mathbf{A}$, $\mathbf{B}$) of the *software* Octave/Matlab®. Plot the run time curve as a function of the number of rows/columns $N \in \{2, 4, 8, 16, 32, 64, 128\}$.

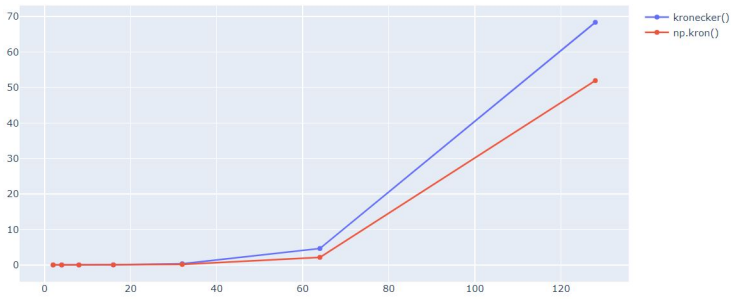
Solution: The custom Kronecker product function implements a loop through the rows of A and B. For a given row of A take a row of B, and then loop through the values of that row of A. For each value of that row, multiply it by the current row of B, append the result to a new row of the result matrix. Fig. 3 indicates the procedure.

Figure 3: Kronecker product custom implementation

```
def kronecker(A, B):
    sublist = []
    result = []

    for row_A in A:
        for row_B in B:
            # multiply each scalar element in row_A by row_B
            for col_A in row_A:
                for item in np.multiply(col_A, row_B):
                    sublist.append(item) # append row of kron matrix

            result.append(sublist)
            sublist = []

    return result
```

The function used for run time comparison was *kron()*, from the Numpy package, which performs the Kronecker product of two input matrices. As show in Fig. 4, we have a similar behavior between both methods, with *np.kron*() being slightly more optimized. This might be due to the fact the we are using explicit for loops in the custom implementation. The use of List Comprehensions could have made the margin between the results narrower.

Figure 4: Kronecker() and np.kron() comparison



Problem 3 For randomly generated matrices $\mathbf{A}$ and $\mathbf{B} \in \mathbb{C}^N \times N$, create an algorithm to compute the Khatri-Rao Product $\mathbf{A} \diamond \mathbf{B}$ according with the following prototype function:

$$\mathbf{R} = kr(\mathbf{A}, \mathbf{B})$$

Solution: The algorithm implemented for the Khatri-Rao product is indicated in Fig. 5. The approach used was to transpose both input matrices, perform Kronecker products on the respective sets of rows $[\mathbf{A}_1 \otimes \mathbf{B}_1 | \mathbf{A}_2 \otimes \mathbf{B}_2 | ... | \mathbf{A}_N \otimes \mathbf{B}_N]$, and then transpose the resulting matrix back to the correct form for a Khatri-Rao product. The transposing approach was chosen because its simpler (in the programming and amount of code sense) to loop through the rows in a matrix then through the columns.

Figure 5: Khatri-Rao product custom implementation

```python
def khatri_rao(A, B):
    # transposing matrices because looping through rows is simpler in Python
    A = A.T
    B = B.T

    result = []

    for Ai in range(0, len(A)):
        for Bi in range(0, len(B)):
            if Ai == Bi:
                # if index of row_A is equals to index of row_B, then append kron(row_A, row_B)
                result.append(np.kron(A[Ai], B[Bi]))

    return np.array(result).T # transposing back to column format
```