AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody obliczeniowe w nauce i technice

Laboratorium 2

Jan Gołda, WIET 2018/2019

1. Wprowadzenie

Laboratorium miało na celu zbadanie wydajności algorytmów i funkcji bibliotecznych służących do wykonywania operacji mnożenia macierzy.

Zadanie wykonano w Jupyter Notebook, pisząc w Pythonie i używając biblioteki do obliczeń naukowych o nazwie NumPy.

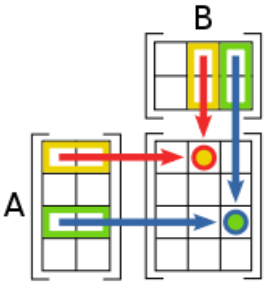
Wykorzystany kod oraz wyniki pomiarów są dostępne w repozytorium.

Dla czytelności pominięto sprawdzanie czy mnożone macierze mają odpowiednie rozmiary.

2. Naiwny algorytm mnożenia macierzy

Pierwszy algorytm dokonuje mnożenia macierzy w „ręczny” sposób, zgodnie z poniższym wzorem.

$$C_{i,j} = \sum_{r=1}^{m} A_{i,r} \cdot B_{r,j}$$



Wykonano dwie implementacje w celu porównania ich wydajności, jedną przy użyciu pętli for oraz drugą korzystającą z „list comprehension”.

2.1. Implementacja przy użyciu pętli for

```
def mul_list(A, B):
    C = [[0 for y in range(len(B[0]))] for x in range(len(A))]

    for x in range(len(A)):
        for y in range(len(B[0])):
            for i in range(len(A[0])):
                C[x][y] += A[x][i] * B[i][y]

    return C
```

2.2. Implementacja przy użyciu list comprehension

```
def mul_listcomprehension(A, B):
    return [
        [
            sum(
                A[x][i] * B[i][y]
                for i in range(len(A[0]))
            )
            for y in range(len(B[0]))
        ]
        for x in range(len(A))
    ]
```

3. Naiwny algorytm mnożenia macierzy działający na NumPy

Drugi algorytm nie różni się niczym od pierwszego, jest jednak zaimplementowany nie na wbudowanych listach tak jak poprzedni, lecz na macierzach – strukturach z biblioteki NumPy.

3.1. Implementacja algorytmu

```
def mul_matrix(A, B):
    C = np.zeros((A.shape[0], B.shape[1]))

    for x in range(C.shape[0]):
        for y in range(C.shape[1]):
            for i in range(A.shape[1]):
                C[x][y] += A[x][i] * B[i][y]

    return C
```

4. Biblioteczna funkcja mnożenia macierzy

Dla porównania wykorzystano funkcję z biblioteki NumPy o sygnaturze:

matrix.dot(b, out=None)

Co w kodzie sprowadza się do:

```
def mul_numpy(A, B):
    return A.dot(B)
```

5. Porównanie wydajności

Pomiarów wydajności dokonano przy użyciu funkcji **%timeit** dostępnej w Jupyter Notebook.

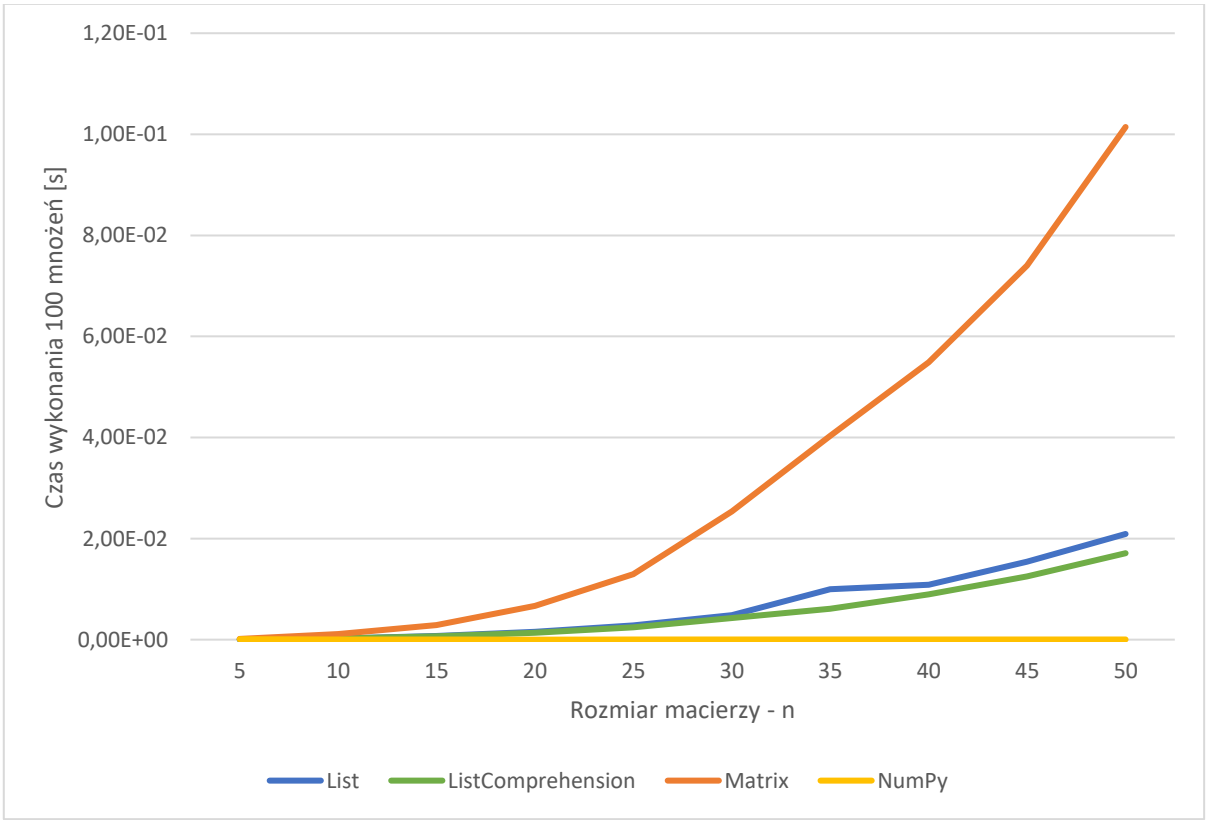
Pomiary wykonywane były dla dwóch losowo wygenerowanych macierzy o rozmiarze n na n.

Każdy pomiar wykonywany był poprzez stukrotne powtórzenie tej samej operacji mnożenia.

Każdy pomiar został powtórzony trzy razy, a wynik końcowy jest średnią z tych trzech pomiarów.

5.1. Wyniki pomiarów

n	List	List comprehension	Matrix	NumPy
5	3,62 E-05	3,87 E-05	1,47 E-04	9,86 E-05
10	2,20 E-04	2,06 E-04	1,14 E-03	2,33 E-06
15	7,13 E-04	6,67 E-04	2,89 E-03	2,29 E-06
20	1,53 E-03	1,37 E-03	6,68 E-03	2,21 E-06
25	2,83 E-03	2,43 E-03	1,29 E-02	2,84 E-06
30	4,87 E-03	4,31 E-03	2,54 E-02	3,47 E-06
35	9,99 E-03	6,12 E-03	4,03 E-02	1,02 E-05
40	1,09 E-02	8,96 E-03	5,49 E-02	5,92 E-06
45	1,54 E-02	1,25 E-02	7,40 E-02	7,78 E-06
50	2,09 E-02	1,71 E-02	1,01 E-01	8,40 E-06



5.2. Wnioski

Najważniejszym spostrzeżeniem jest to że funkcja biblioteczna jest zdecydowanie szybsza, co nie jest dziwne, gdyż w jej przypadku obliczenia są wykonywane według całkiem innego algorytmu, o dużo lepszej złożoności.

Porównanie pozostałych trzech metod (wszystkie implementują naiwny algorytm) pozwala wyciągnąć dwa ciekawe wnioski:

- Operowanie na macierzach z biblioteki NumPy jest bardzo wolne. Prawdopodobnie jest to spowodowane tym, że nie są one zoptymalizowane pod kątem dostępu do konkretnych pól, a raczej pod kątem wykonywania bardziej zaawansowanych operacji dostępnych w bibliotece.
- Używanie list comprehension jest trochę szybsze niż używanie pętli for. Jest to spowodowane tym, że używając pętli tak czy siak musimy wcześniej zainicjalizować tablicę zerowymi wartościami, czego nie musimy robić w przypadku list comprehension.