Zadanie 1: Mnożenie macierzy

- Jakub Karbowski
- Jakub Szymczak

Metoda rekurencyjna Binet'a

Polega na podziale macierzy na 4 równe bloki i rekurencyjnym wywoływaniu aż do przypadku granicznego, którym jest mnożenie skalarów.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

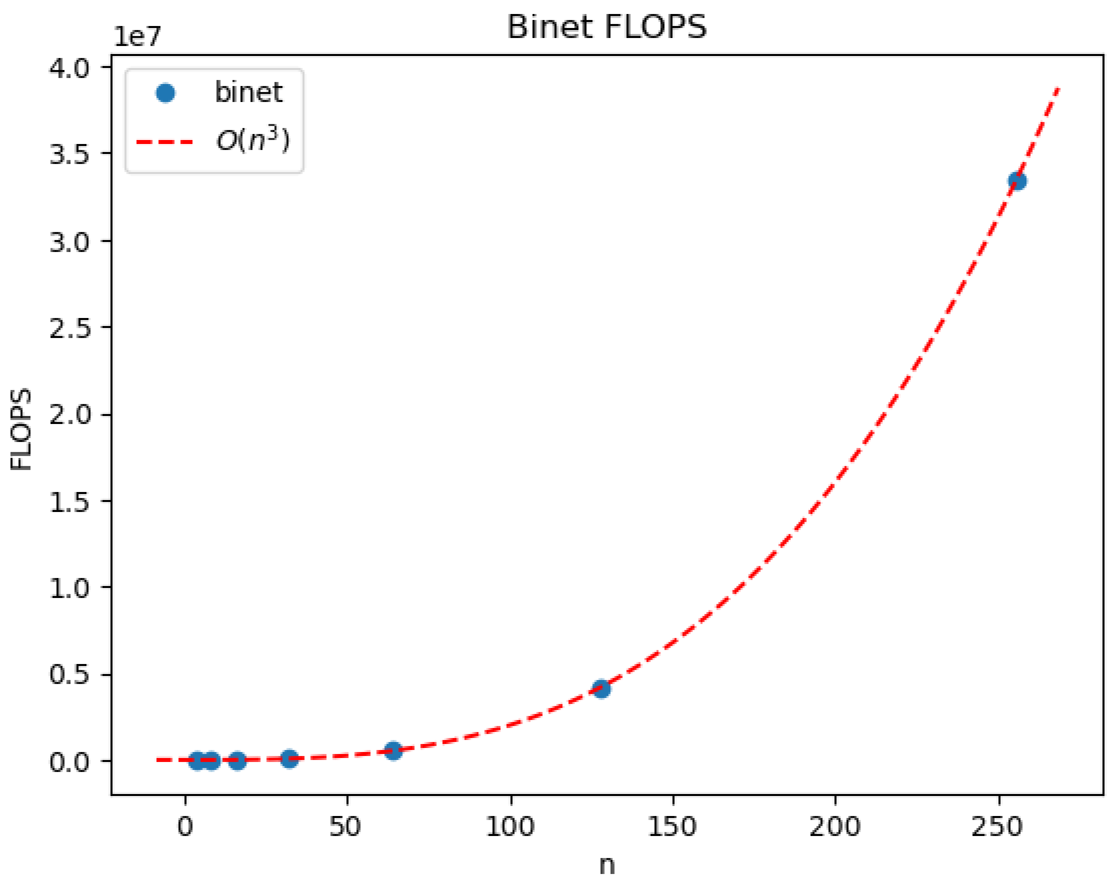
$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{12} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

```python
def binet_mul(a: Mat, b: Mat) -> Mat:
    assert len(a) == len(a[0]) == len(b) == len(b[0])
    n = len(a)

    if n == 1:
        # base case scalar multiplication
        return [[a[0][0] * b[0][0]]]
    else:
        assert n % 2 == 0
        # block size
        m = n // 2

        # input blocks
        a11 = mat_slice(a, 0, m, 0, m)
        a12 = mat_slice(a, 0, m, m, n)
        a21 = mat_slice(a, m, n, 0, m)
        a22 = mat_slice(a, m, n, m, n)

        b11 = mat_slice(b, 0, m, 0, m)
        b12 = mat_slice(b, 0, m, m, n)
        b21 = mat_slice(b, m, n, 0, m)
        b22 = mat_slice(b, m, n, m, n)

        # recursive calls
        c11 = mat_add(binet_mul(a11, b11), binet_mul(a12, b21))
        c12 = mat_add(binet_mul(a11, b12), binet_mul(a12, b22))
        c21 = mat_add(binet_mul(a21, b11), binet_mul(a22, b21))
        c22 = mat_add(binet_mul(a21, b12), binet_mul(a22, b22))

        # combine blocks
        c = mat_zeros(n, n)
        for i in range(m):
            for j in range(m):
                c[i][j] = c11[i][j]
                c[i][j + m] = c12[i][j]
```

```
            c[i + m][j] = c21[i][j]
            c[i + m][j + m] = c22[i][j]

    return c
```

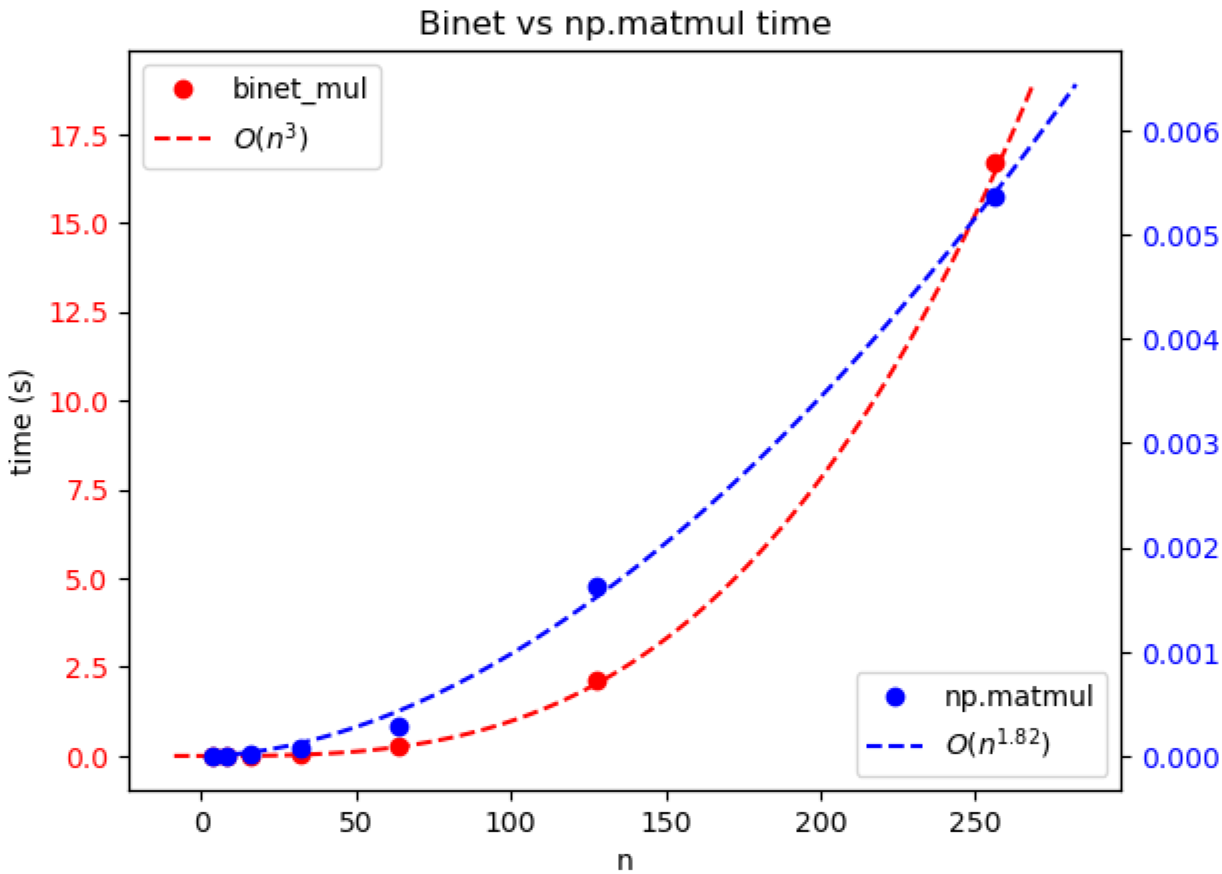
Wydajność

Zmierzono liczbę operacji zmiennoprzecinkowych `+ *` w zależności od rozmiaru macierzy. Do pomiarów wpasowano wielomian $y = ax^3$.



Zmierzono również czas, który został porównany z implementacją `numpy.matmul`.

```
/var/folders/sd/yrr8ypm52gv1wbxvgg_8f1b40000gn/T/ipykernel_21371/3762522458.py:21: RuntimeWarnin
g: invalid value encountered in power
  return a * x**b
```



Algorytm Binet'a ma złożoność $O(n^3)$, a algorytm stosowany w `numpy` ma mniejszą złożoność. Do pomiarów czasów `numpy` została wpasowana złożoność $O(n^{1.82})$, co wydaje się błędne. Może to być efekt stosowanych heurystyk lub po prostu błąd pomiaru czasu.

Metoda Strassen'a