Algorytmy macierzowe
Sprawozdanie z laboratorium nr. 3

Szymon Nowak-Trzos, Dominik Adamczyk

9 grudnia 2023

1 Treść zadania

Zadanie polegało na zaimplementowaniu algorytmu stratnej kompresji macierzy z wykorzystaniem $\boldsymbol{TruncatedSVD(A, \delta, b)}$, oraz zaimplementowania funkcji rysującej macierze.

2 Wstęp

Zgodnie z treścią, zaimplementowany został wskazany algorytm. Całe zadanie zostało napisane w języku Python 3.10. Testy przeprowadzono na urządzeniu z cpu Intel Core i5-1135g7 o bazowej częstotliwości 2.4GHz.

3 Użyte rozwiązania

Do budowy drzewa zaimplementowana została klasa Node:

```python
class Node:
    def __init__(self):
        self.rank = None
        self.size = None
        self.singular_values = None
        self.U = None
        self.V = None
        self.sons = None
```

Dodatkowo zaimplementowane zostały funkcje pomocnicze do dzielenia macierzy na części oraz sklejania macierzy z tych części.

```python
def matrix_partition(A):
    n = A.shape[0] // 2
    m = A.shape[1] // 2
    A11 = A[:n, :m]
    A12 = A[:n, m:]
    A21 = A[n:, :m]
    A22 = A[n:, m:]
    return A11, A12, A21, A22

def matrix_repartition(A11, A12, A21, A22):
    C = np.block([[A11, A12],
                  [A21, A22]])
    return C
```

Algorytmy zostały zaimplementowane w osobnych funkcjach. Do mnożenia macierzy użyto mnożenia z biblioteki numpy w celach przyśpieszenia obliczeń.

4 Algorytm kompresji macierzy

Poniżej prezentujemy opisy i kody implementowanych algorytmów. Kod Python'a służy również za pseudokod.

4.1 Truncated SVD

Jako pierwszy został zaimplementowany algorytm częściowej dekompozycji SVD.

Algorytm ten wykorzystuje $\boldsymbol{np.linalg.svd(A)}$ do obliczenia pełnej dekompozycji SVD, a następnie ucina macierze zgodnie z $\boldsymbol{\delta}$ oraz $\boldsymbol{b}$.

```
def truncated_SVD(A, delta, b):
    b = b-1
    U, s, V = np.linalg.svd(A)
    i = find_s_index_for_delta(s, delta)
    idx = min(i, b, len(s))
    return U[:, :idx + 1], s[:idx+1], V[:idx + 1, :]
```

4.2 Kompresja macierzy

Poniższy algorytm tworzy węzeł w drzewie ze skompresowaną macierzą funkcją $\boldsymbol{truncated\_SVD()}$.

```
def compress_matrix(A, delta, b):
    if not np.any(A):
        v = Node()
        v.rank = 0
        v.size = A.shape
        return v
    U, s, V = truncated_SVD(A, delta, b)
    rank = matrix_rank(s)
    v = Node()
    v.rank = rank
    v.U = U * s
    v.V = V
    v.size = A.shape
    return v
```

4.3 Tworzenie drzewa

Jeśli spełniony jest **warunek dopuszczalności** to kompresujemy węzeł drzewa, jeśli nie to rozbijamy macierz węzła na cztery podmacierze i tworzymy tam rekurencyjnie drzewa.

```
def create_tree(A, r, e):
    r = bound(min(A.shape[0], A.shape[1]), 0, r+1)
    U, s, V = truncated_SVD(A, e, r + 1)
    if s[-1] < e or U.shape[0] <= r:
        v = compress_matrix(A, e, r)
    else:
        v = Node()
        A11, A12, A21, A22 = matrix_partition(A)
        v.sons = [create_tree(A11, r, e),
                  create_tree(A12, r, e),
                  create_tree(A21, r, e),
                  create_tree(A22, r, e)]
    return v
```

4.4 Dekompresja macierzy

Rekurencyjnie przechodzimy drzewo i dla każdego liścia przemnażamy $\boldsymbol{U}$ i $\boldsymbol{V}$.

```
def recover_matrix(v):
    if v.sons:
        m = matrix_repartition(recover_matrix(v.sons[0]), recover_matrix(v.sons[1]),
                               recover_matrix(v.sons[2]), recover_matrix(v.sons[3]))
    elif v.rank == 0:
        m = np.zeros(v.size)
    else:
        m = v.U @ v.V
    return m
```

4.5 Rysowanie drzewa oraz macierzy

Reprezentacje skompresowanej macierzy uzyskujemy przez ułożenie $\boldsymbol{U}$ i $\boldsymbol{V}$ w macierzy zer.

Aby uzyskać reprezentację całego drzewa przechodzimy po nim rekurencyjnie i, tak samo jak przy dekompresji macierzy, składamy dużą macierz z mniejszych.

```
def U_V_to_array(U, V):
    n = U.shape[0]
    m = U.shape[1]
    repr = np.zeros((n, n))
    repr[:n, :m] = U
    repr[:m, :n] = V
    return repr

def tree_to_repr(v):
    if v.sons:
        m = matrix_repartition(tree_to_repr(v.sons[0]), tree_to_repr(v.sons[1]),
                               tree_to_repr(v.sons[2]), tree_to_repr(v.sons[3]))
    elif v.rank == 0:
        m = np.zeros(v.size)
    else:
        m = U_V_to_array(v.U, v.V)
    return m

def show_array(repr, zeros=False):
    if zeros:
        plt.spy((repr != 0).astype(int))
    else:
        plt.spy(repr)
    plt.grid(False)
    plt.show()
```

5 Wykresy

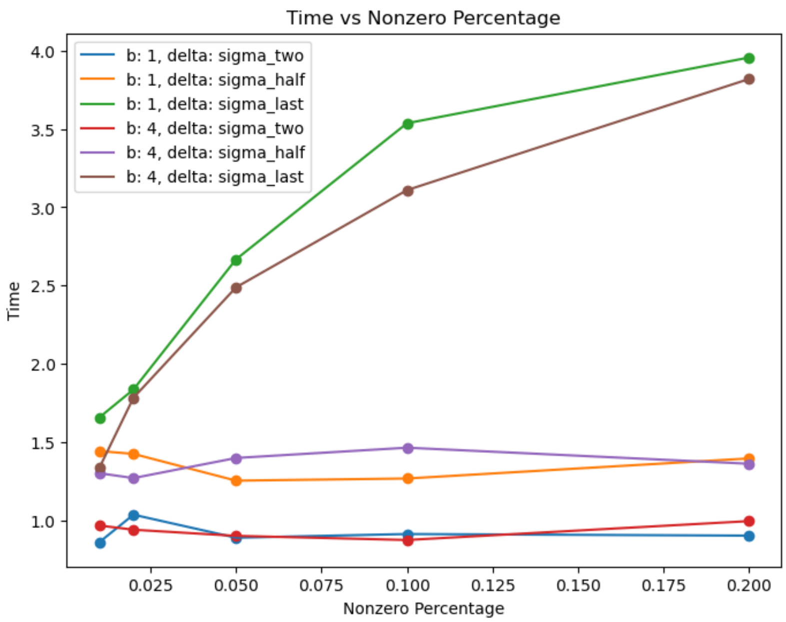
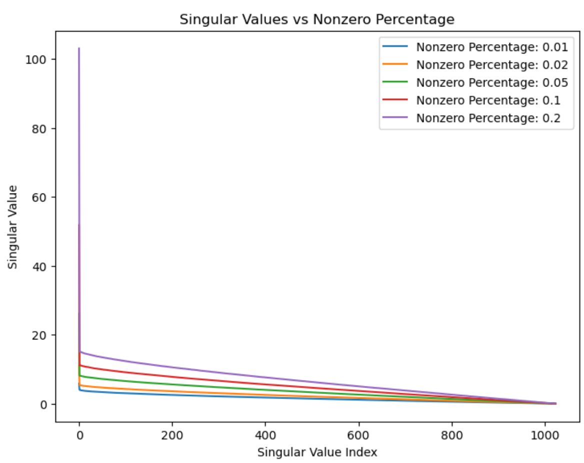
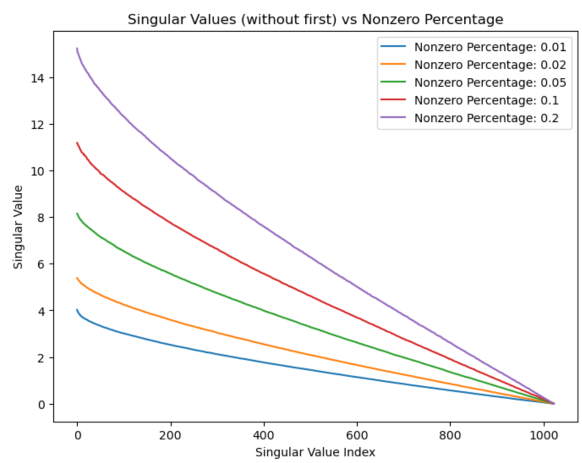


Tabela 1: Wykres czasów algorytmu



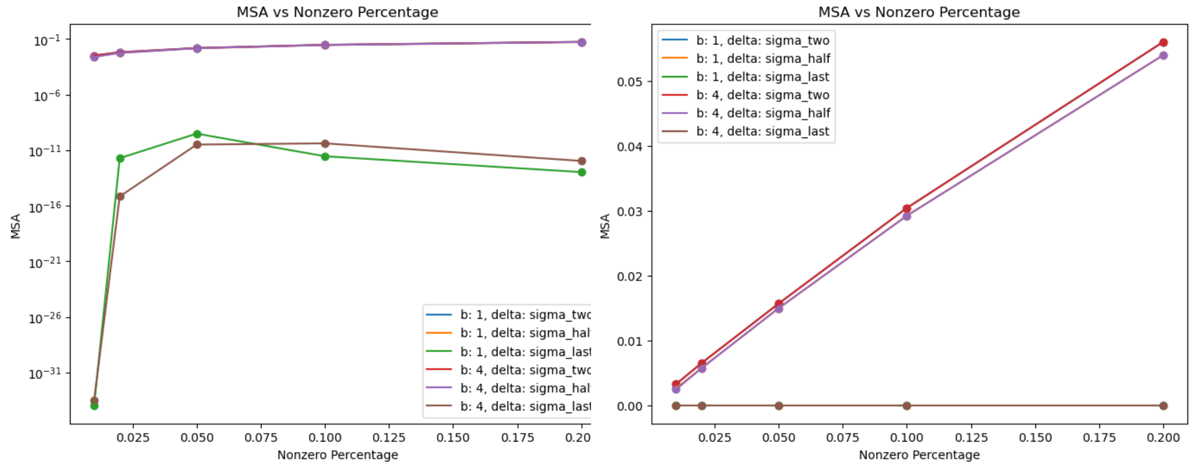
(a) Wartości osobliwe



(b) Wartości osobliwe - bez pierwszej

Czasy intuicyjnie zwiększają się ze zwiększającym się zakresem wartości osobliwych. Najdłużej wykonuje się algorytm dla $\delta = \sigma_{last}$. Nie ma dużej różnicy pomiędzy algorymtem dla $b = 1$ i $b = 4$.

Pierwsza wartość osobliwa jest zawsze znacząco większa od pozostałych.



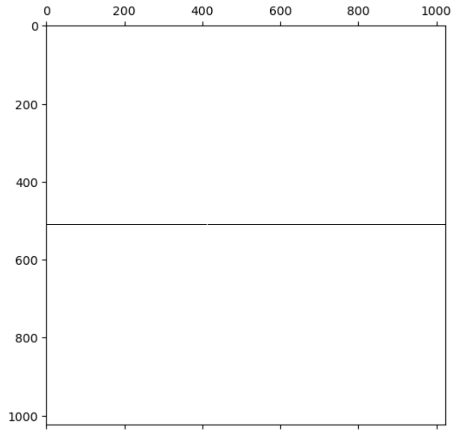
(a) Błędy średniokwadratowe - skala logarytmiczna

(b) Błędy średniokwadratowe

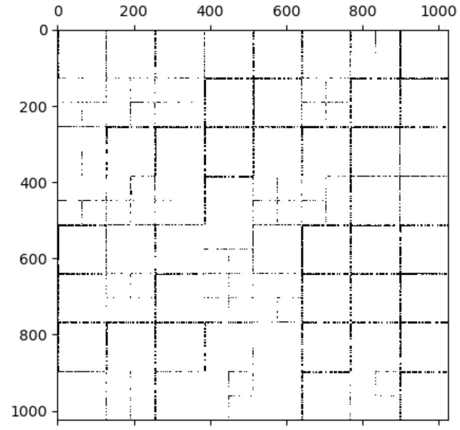
Rysunek 3: Wykresy

Na wykresach widzimy, że błąd średniokwadratowy jest najmniejszy dla $\delta = \sigma_{last}$ - jest to logiczne, bo prawie nic nie wyrzucamy z oryginalnej dekompozycji SVD. Nie ma zauważalnej różnicy między $\delta = \sigma_{two}$ oraz $\delta = \sigma_{half}$. Widać, że błąd dla nich wzrasta liniowo wraz ze wzrostem ilości niezerowych elementów macierzy.

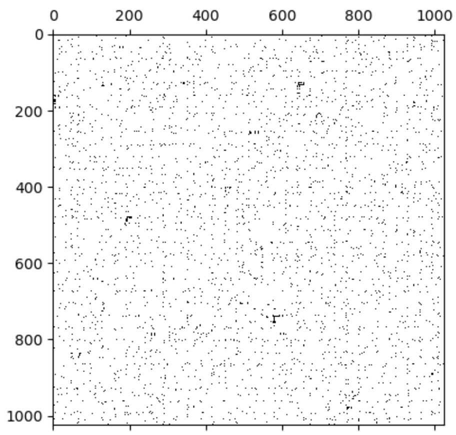
6 H macierze



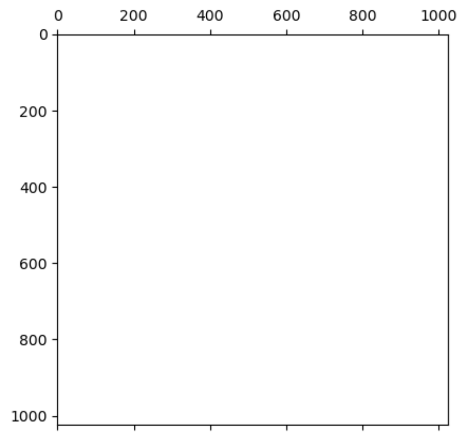
(a) $b = 1, \delta = \sigma_{two}$



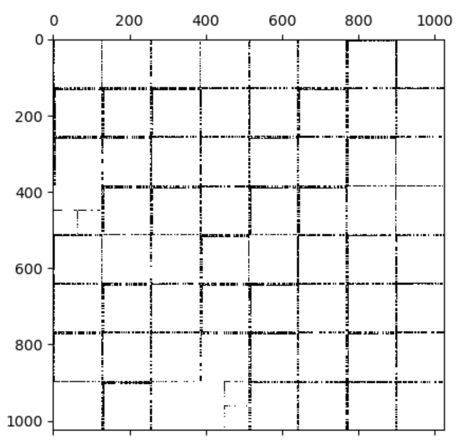
(b) $b = 1, \delta = \sigma_{half}$



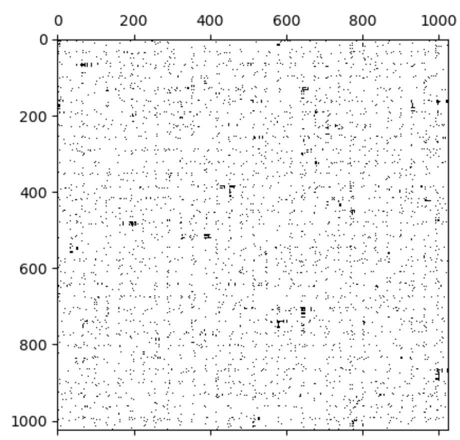
(c) $b = 1, \delta = \sigma_{last}$



(d) $b = 4, \delta = \sigma_{two}$



(e) $b = 4, \delta = \sigma_{half}$



(f) $b = 4, \delta = \sigma_{last}$

Rysunek 4: H macierze