Algorytmy macierzowe

Laboratorium 3

Sprawozdanie

Łukasz Stępień, Szymon Urbański

1. Temat zadania

Laboratorium polegało na zaimplementowaniu i przetestowaniu rekurencyjnego algorytmu kompresji macierzy z wykorzystaniem częściowego SVD dla wybranych parametrów.

2. Rozwiązanie

Zaimplementowano rekurencyjną kompresje macierzy.

- **Pseudokod algorytmu:**
  a) Tworzenie drzewa kompresji

```
create_tree(t_min, t_max, s_min, s_max, r, ε):
    [U, D, V] = truncatedSVD(A[t_min:t_max, s_min:s_max], r+1)
    if D[r+1] < ε then
        v = compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r)
    else
        create node v
        v.append(create_tree(t_min, t_new_max, s_min, s_new_max, r, ε)
        v.append(create_tree(t_min, t_new_max, s_new_max, s_max, r, ε)
        v.append(create_tree(t_new_max, t_max, s_min, s_new_max, r, ε)
        v.append(create_tree(t_new_max, t_max, s_new_max, s_max, r, ε)
    return v
```

b) Kompresja SVD

```
compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r):
    create node v
    if A[t_min:t_max, s_min:s_max] is zeros then
        v.U <- zeros(t_max - t_min, s_max - s_min)
        v.V <- zeros(t_max - t_min, s_max - s_min)
    else
        v.size <- (t_min, t_max, s_min, s_max)
        v.rank <- r
```

```
        v.singular_values <- D[0 : r + 1]
        v.U = U[*, 0 : r + 1]
        v.V = diag(D[0: r + 1]) * V[0 : r + 1, *]
    return v
```

c) Dekonstrukcja macierzy z macierzy skompresowanej

```
decompress(B, v):
    if v has no children then:
        B[v.size] = v.U * v.V
    else
        for child in v.children:
            B = decompress(B, child)
    return B
```

- **Najważniejsze fragmenty kodu:**

Poniżej przedstawiono fragmenty kodu realizujące powyższe pseudokody oraz fragment implementujący rysowacz.

```python
class Node:

    Szuum02
    def __init__(self, size, rank=0):
        self.children = []
        self.parent = None
        self.size = size
        self.rank = rank
        self.sv = []
        self.U = None
        self.V = None

    Szuum02
    def append(self, node):
        self.children.append(node)
        node.parent = self
```

Kod 1. Implementacja klasy Node

```
def create_tree(t_min, t_max, s_min, s_max, r, eps):
    U, D, V = randomized_svd(
        A[t_min:t_max, s_min:s_max], n_components=r+1, random_state=0)
    if len(D) <= r or D[r] < eps:
        v = compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r)

    else:
        v = Node((t_min, t_max, s_min, s_max))
        t_new_max = t_min + (t_max - t_min) // 2
        s_new_max = s_min + (s_max - s_min) // 2

        v.append(create_tree(t_min, t_new_max, s_min, s_new_max, r, eps))
        v.append(create_tree(t_min, t_new_max, s_new_max, s_max, r, eps))
        v.append(create_tree(t_new_max, t_max, s_min, s_new_max, r, eps))
        v.append(create_tree(t_new_max, t_max, s_new_max, s_max, r, eps))
    return v
```

Kod 2. Implementacja tworzenia drzewa.

```
def compress_matrix(t_min, t_max, s_min, s_max, U, D, V, r):
    if np.all(A[t_min:t_max, s_min:s_max] == 0):
        v = Node((t_min, t_max, s_min, s_max))
        v.U = np.zeros((t_max - t_min, s_max - s_min))
        v.V = np.zeros((t_max - t_min, s_max - s_min))
        return v
    v = Node((t_min, t_max, s_min, s_max), r)
    v.sv = D[:r+1]
    v.U = U[:, :r + 1]
    v.V = np.diag(D[:r + 1]) @ V[:r + 1, :]
    return v
```

Kod 3. Implementacja kompresji SVD macierzy

```
def decompress(B, v):
    if v.children == []:
        B[v.size[0]: v.size[1], v.size[2]: v.size[3]] = v.U @ v.V
        return B
    for child in v.children:
        B = decompress(B, child)
    return B
```

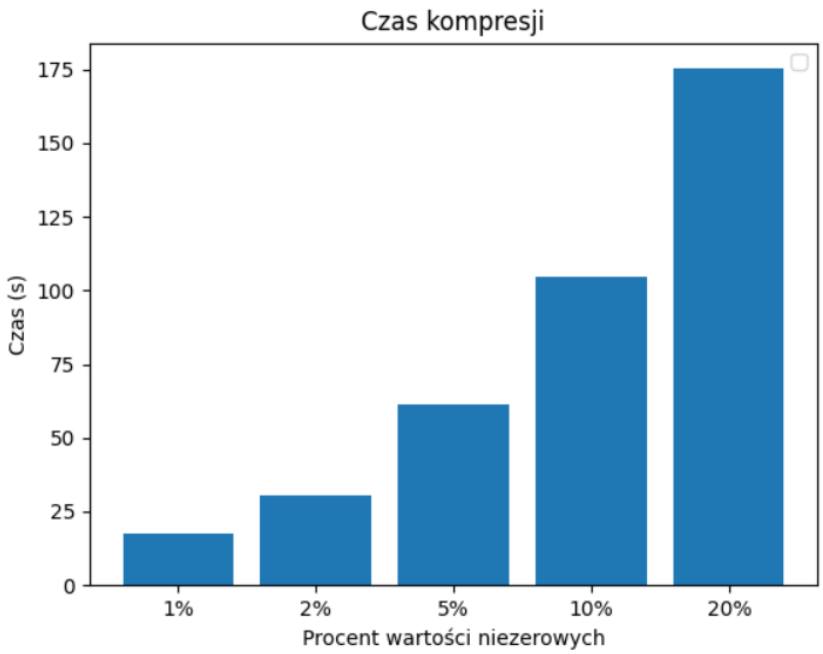
Kod 4. Implementacja dekonstrukcji macierzy

```
def draw_cross_with_square(left_top, right_bottom, ax):
    center_x = (left_top[0] + right_bottom[0]) / 2
    center_y = (left_top[1] + right_bottom[1]) / 2
    width = right_bottom[0] - left_top[0]
    height = right_bottom[1] - left_top[1]

    rect = patches.Rectangle(left_top, width, height,
                             linewidth=0.5, edgecolor='black', facecolor='none')
    ax.add_patch(rect)

    cross_size = min(width, height)

    ax.plot([center_x - cross_size / 2, center_x + cross_size / 2],
            [center_y, center_y], color='black', linewidth=0.5)
    ax.plot([center_x, center_x], [center_y - cross_size / 2,
            center_y + cross_size / 2], color='black', linewidth=0.5)


new *
def draw_black(left_top, right_bottom, ax):
    width = right_bottom[0] - left_top[0]
    height = right_bottom[1] - left_top[1]

    rect = patches.Rectangle(left_top, width*0.2, height,
                             linewidth=0.5, color='black', facecolor='none')
    ax.add_patch(rect)

    rect = patches.Rectangle(right_bottom, -width, -height*0.2,
                             linewidth=0.5, color='black', facecolor='none')
    ax.add_patch(rect)
```

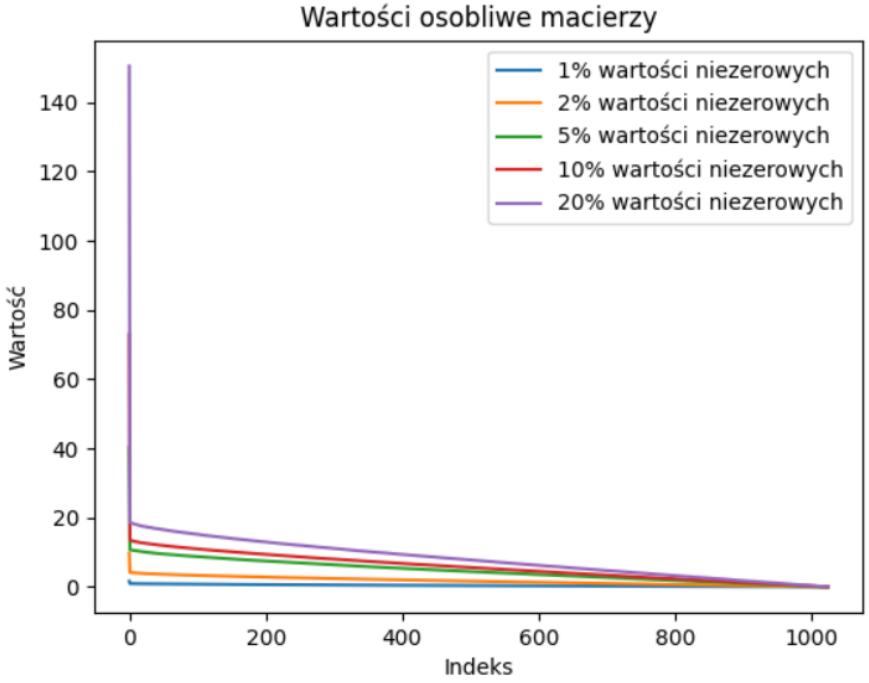
Kod 5. Implementacja rysowacza

3. Wyniki

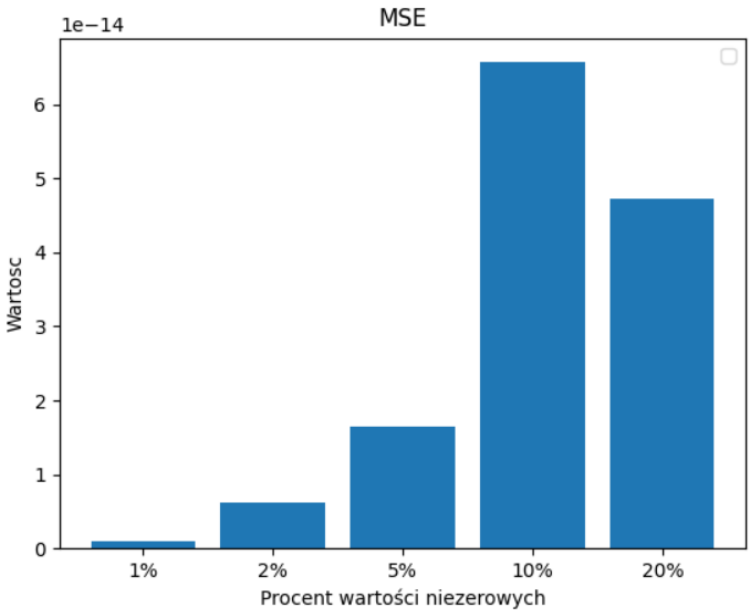
Algorytm przetestowano dla 5 macierzy 2^k x 2^k dla k=10, odpowiednio zawierających 1,2,5,10,20 procent wartości niezerowych. Do funkcji przekazywano maksymalny rząd równy 1 oraz za najmniejszą wartość osobliwą przyjmowano tą ostatnią (czyli rzeczywiście najmniejszą). Dla każdej z macierzy podano czas kompresji, uruchomiono SVD i znaleziono wartości osobliwe, dokonano dekonstrukcji i obliczenia MSE. Wyniki przedstawiono na wykresach.



Wykres 3.1 Czas kompresji w zależności od ilości wartości niezerowych macierzy



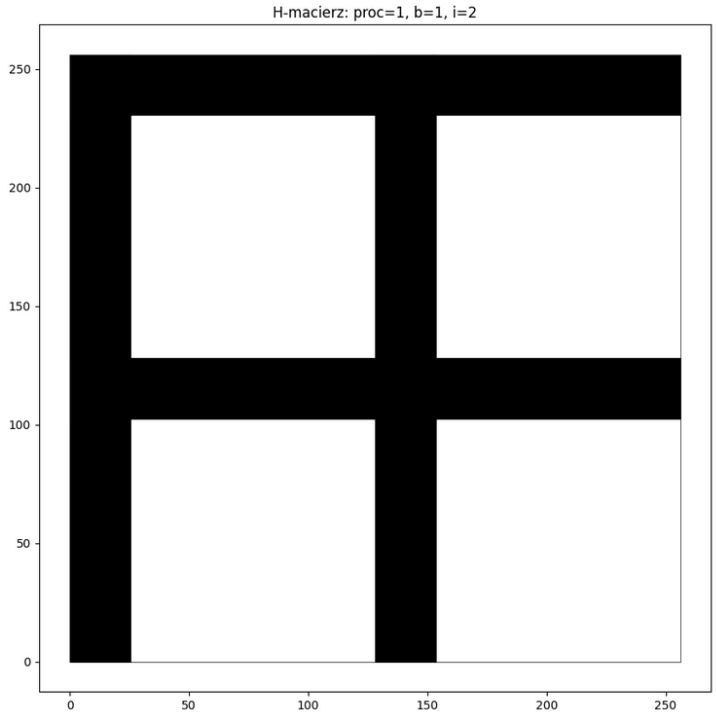
Wykres 3.2 Wartości osobliwe poszczególnych macierzy



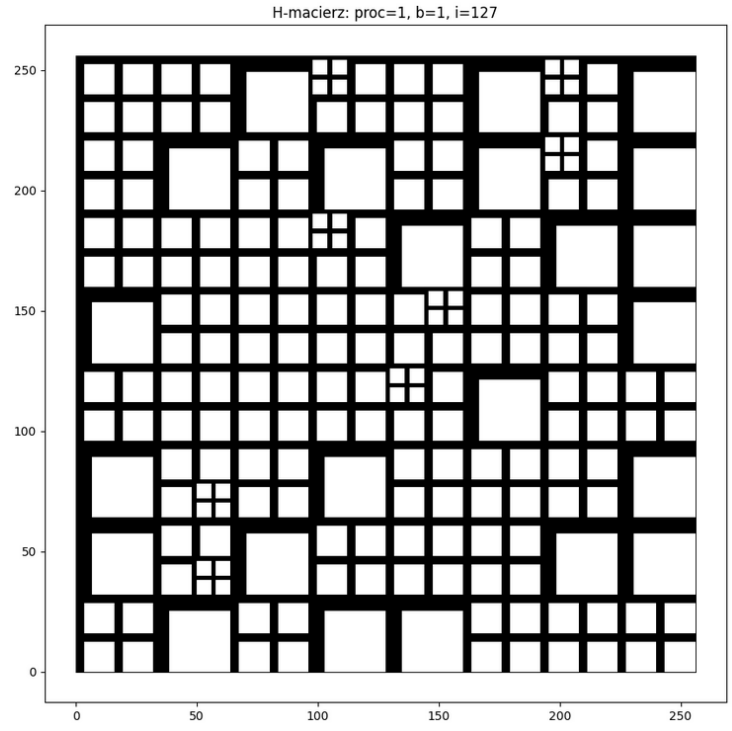
Wykres 3.3 Błąd dekonstrukcji macierzy dla poszczególnych przypadków

Na poniższych wykresach przedstawiono narysowane rekurencyjne dekompozycje macierzy, w zależności od obranego maksymalnego rzędu macierzy i minimalnej wartości osobliwej. Testy przeprowadzono dla macierzy o wymiarach 2^8 x 2^8 w celu uzyskanie przejrzystości wyniku.

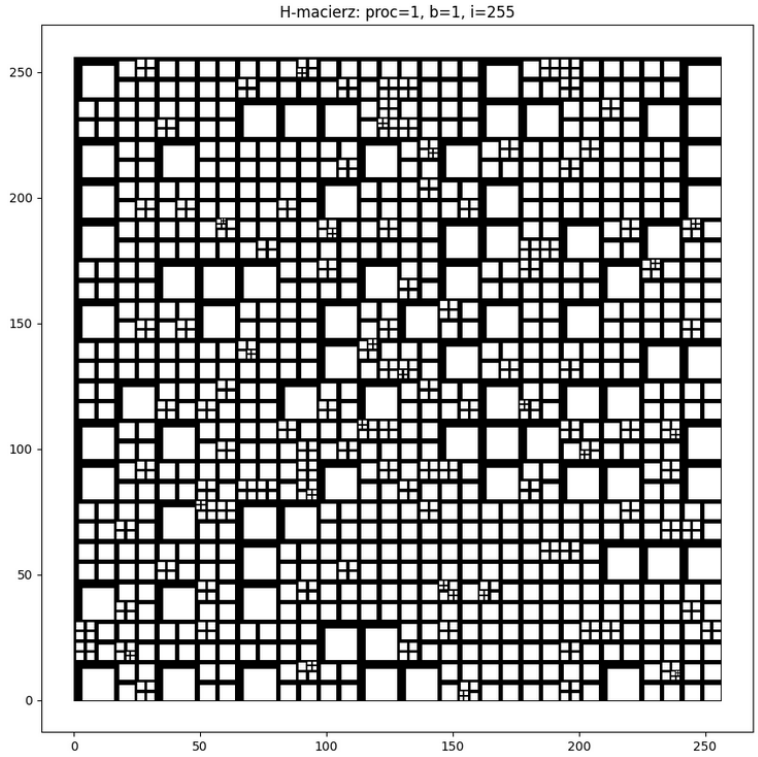
1) Ilość wartości niezerowych: 1%
    - Maksymalny rząd: 1, minimalna wartość osobliwa: σ_2



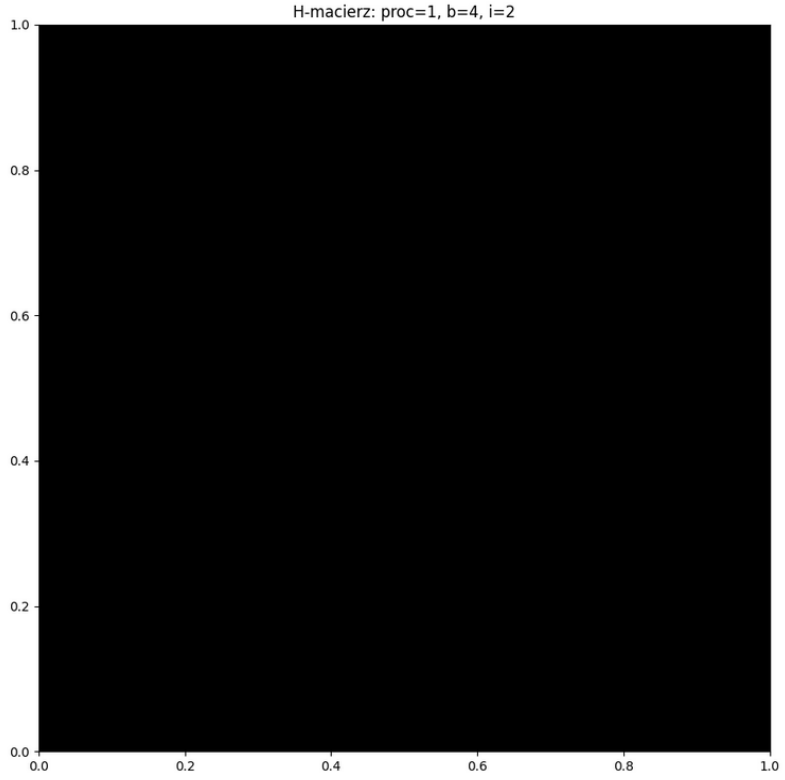
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^7}



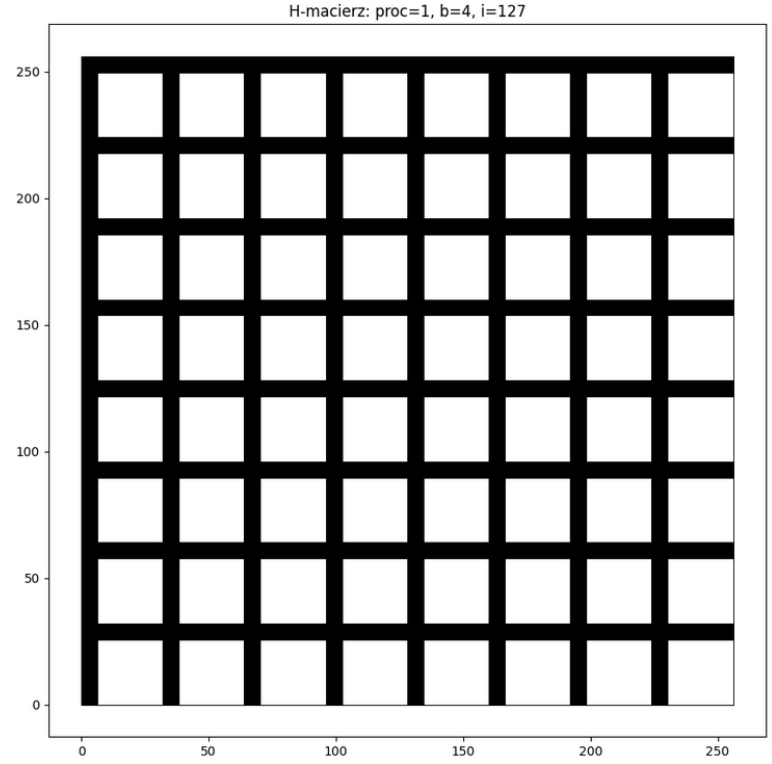
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^8}



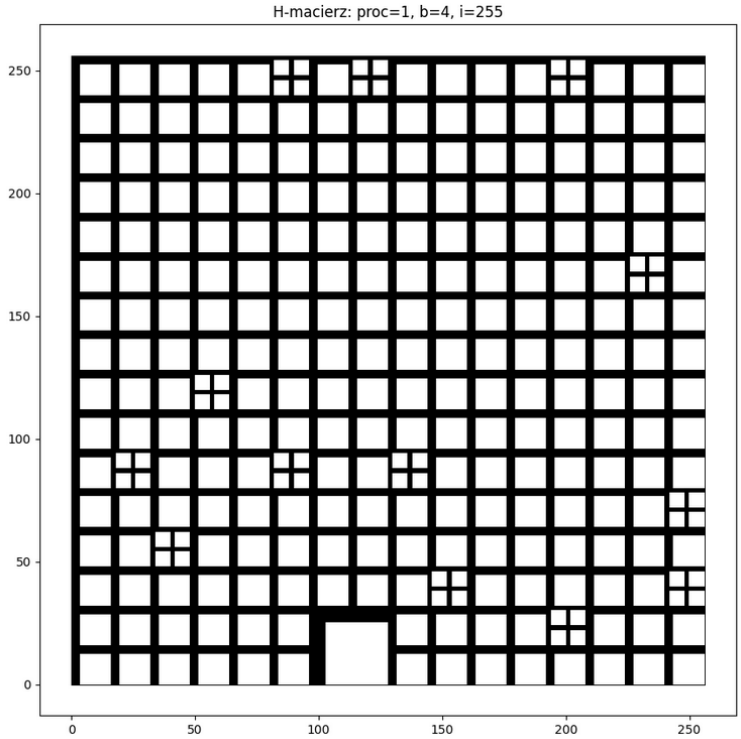
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_2



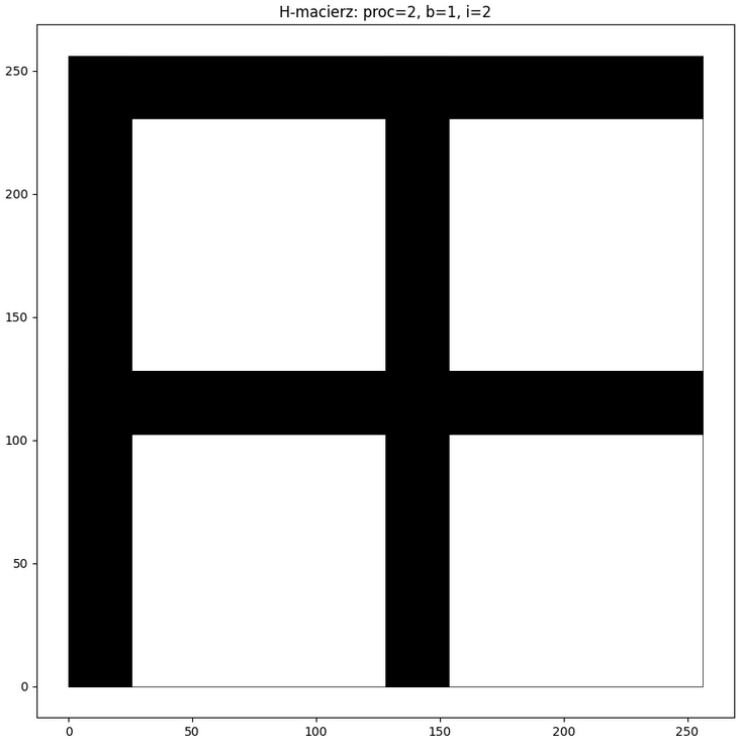
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_{2^7}



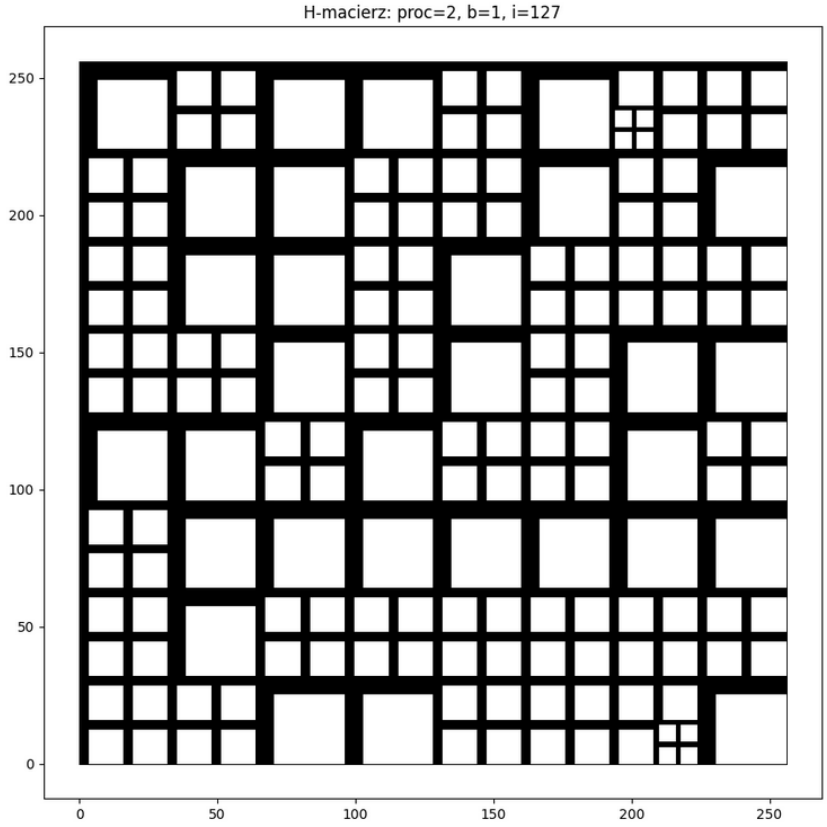
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_8



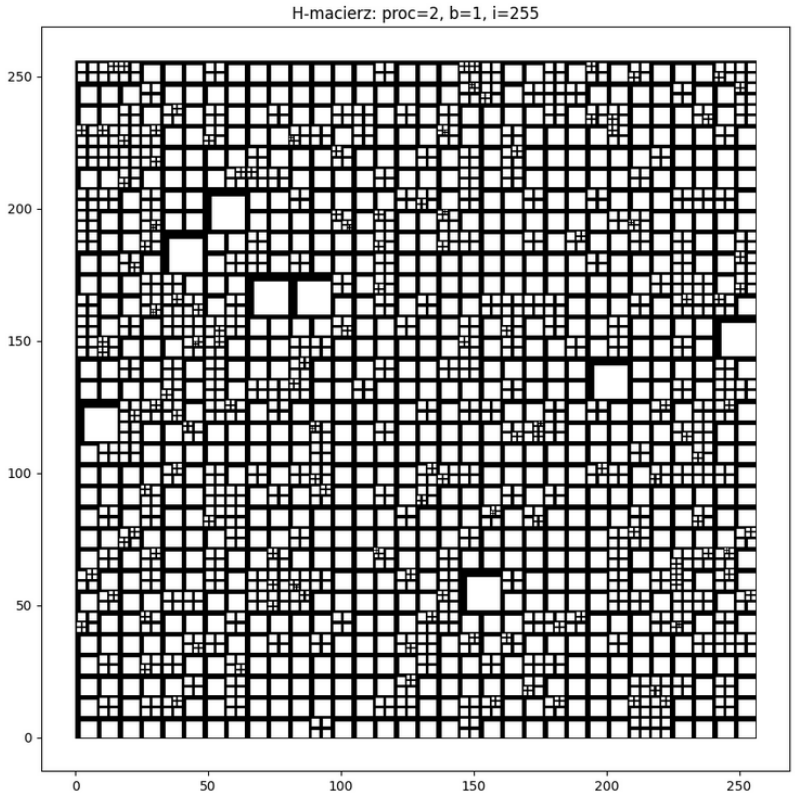
2) Ilość wartości niezerowych: 2%
   - Maksymalny rząd: 1, minimalna wartość osobliwa: σ_2



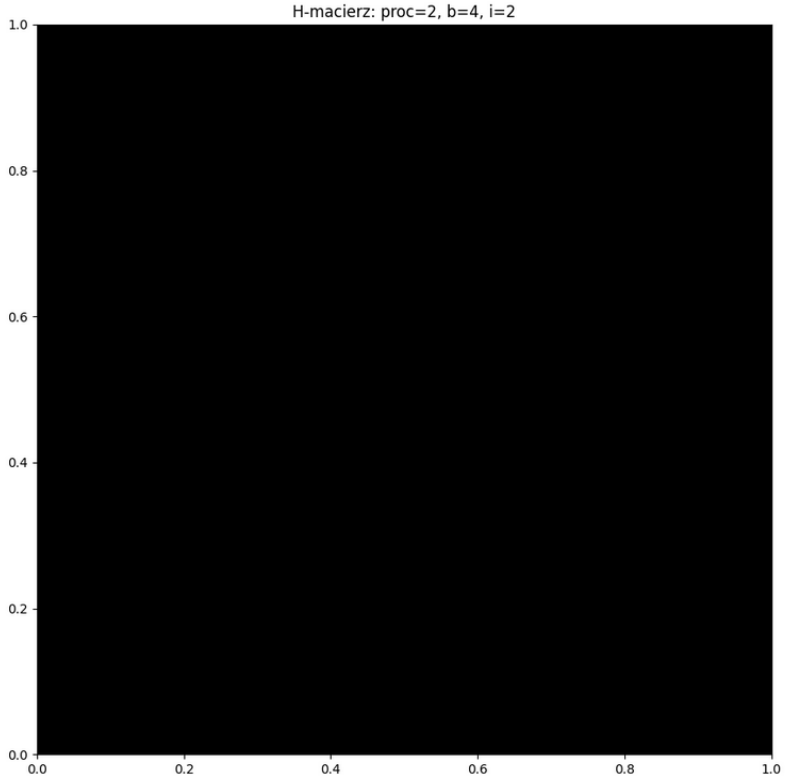
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^7}



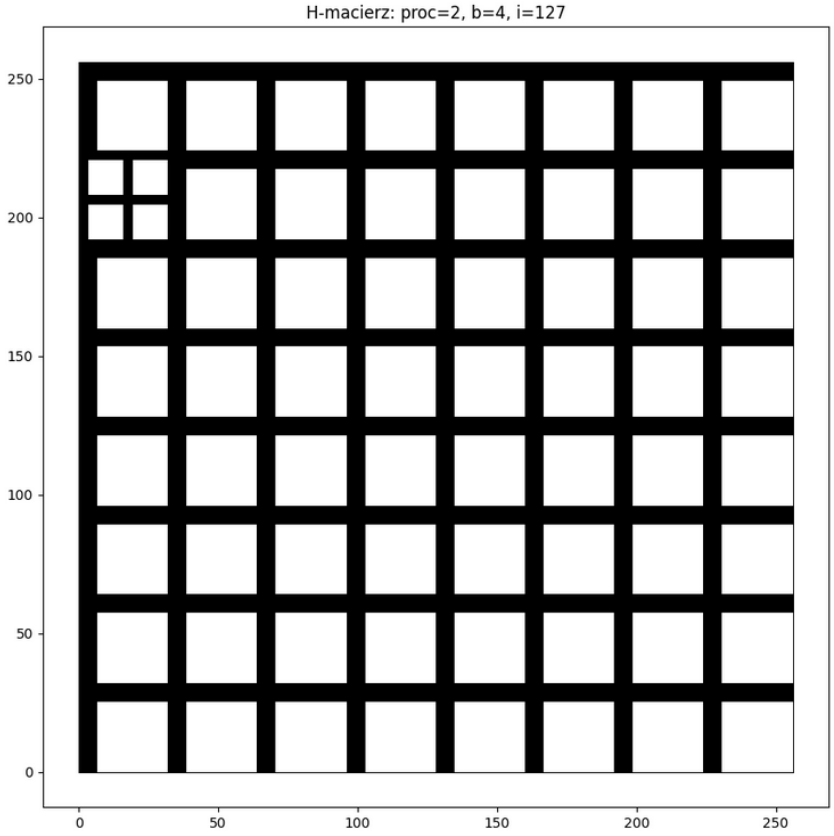
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^8}



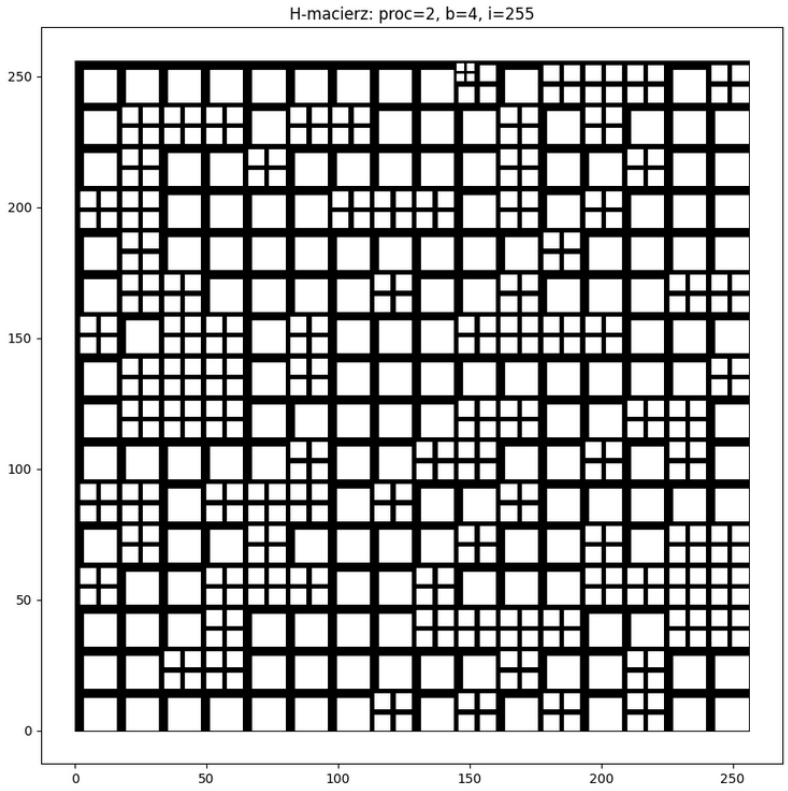
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_2



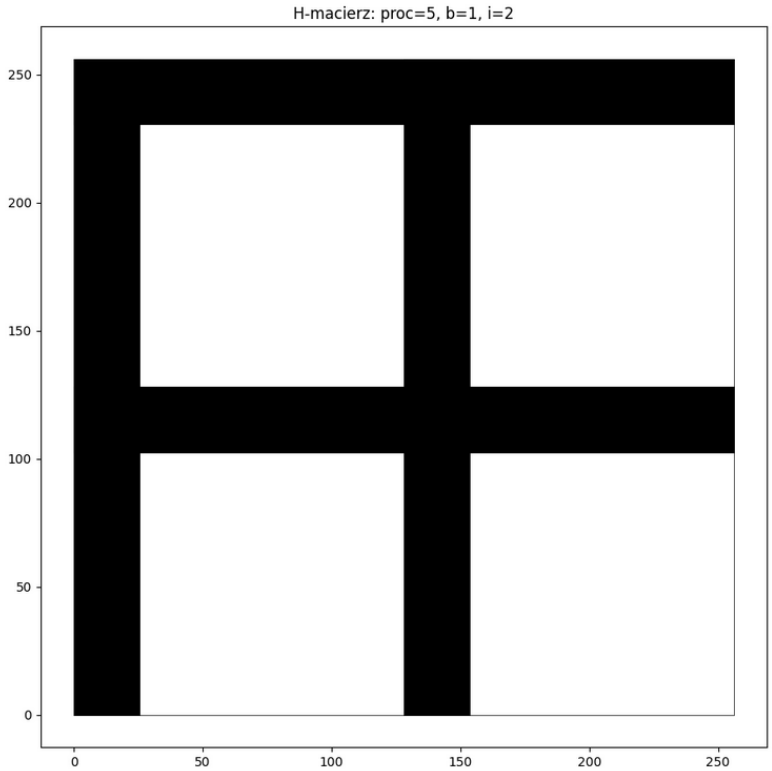
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_{2^7}



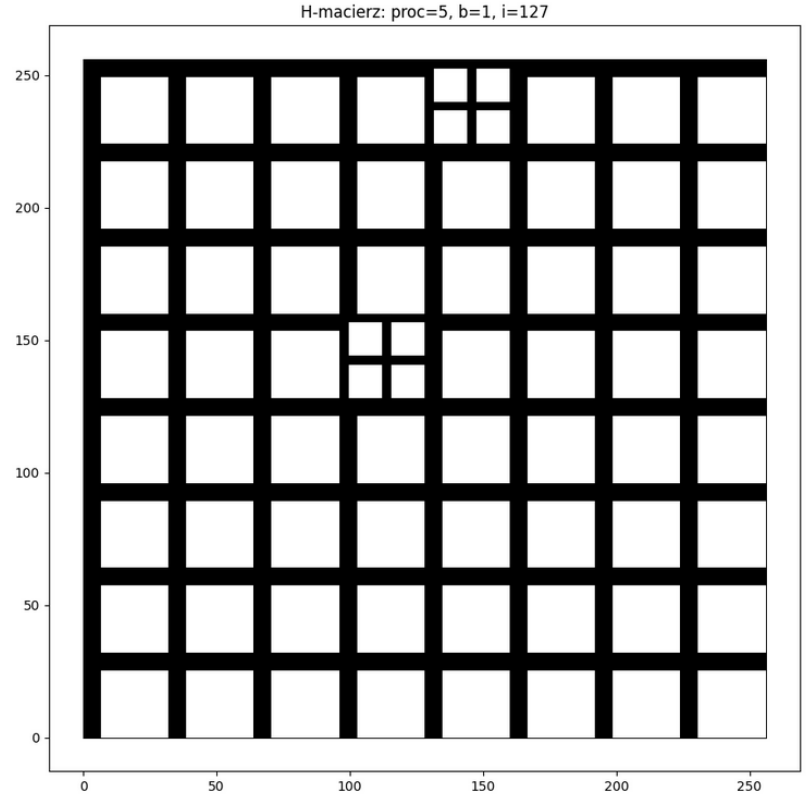
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_8



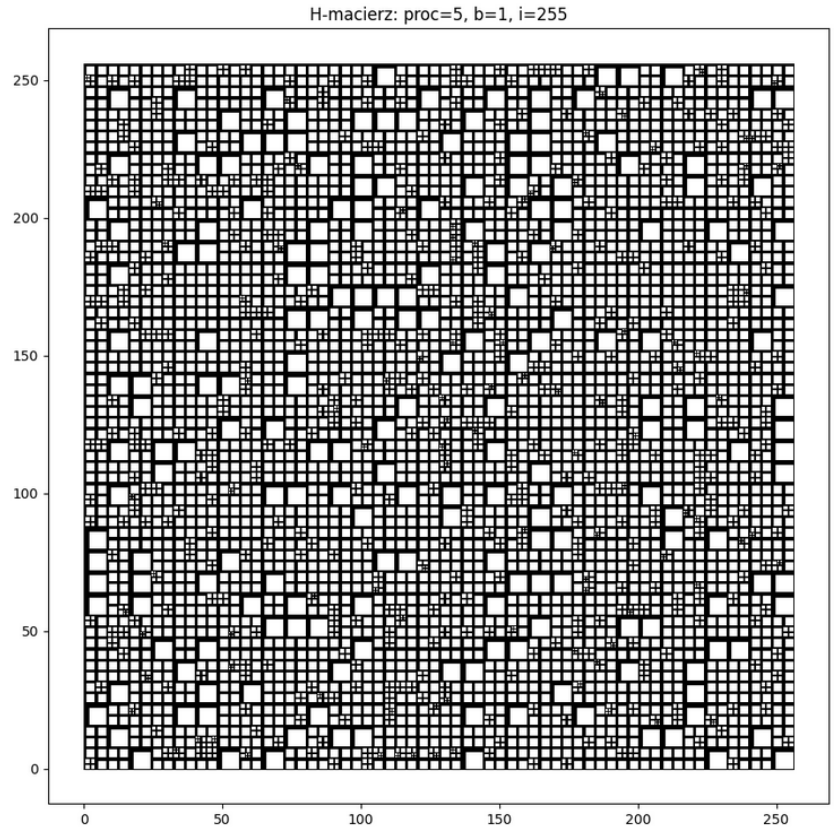
3) Ilość wartości niezerowych: 5%
   - Maksymalny rząd: 1, minimalna wartość osobliwa: σ_2



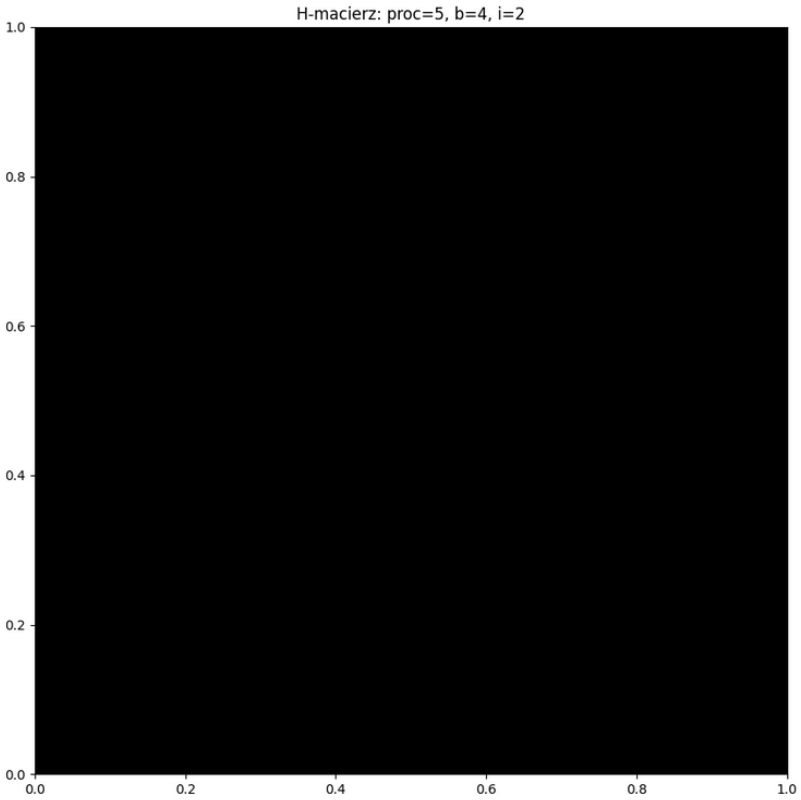
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^7}



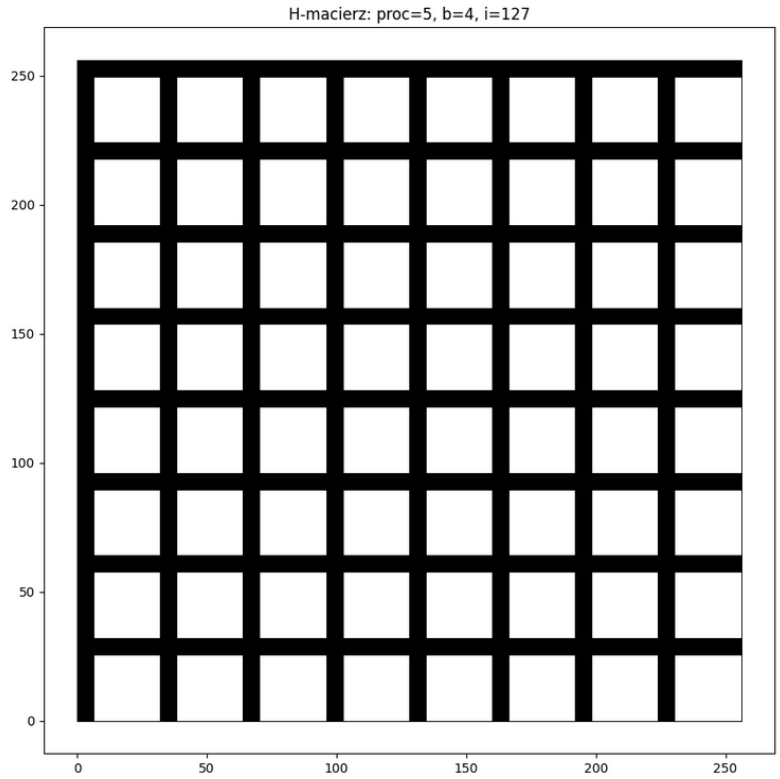
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^8}



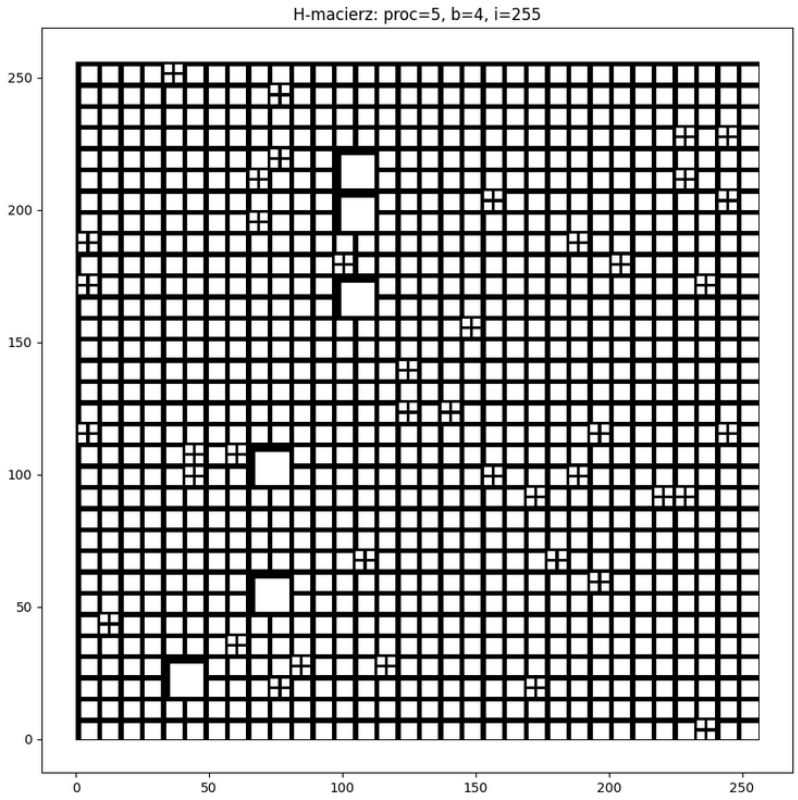
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_2



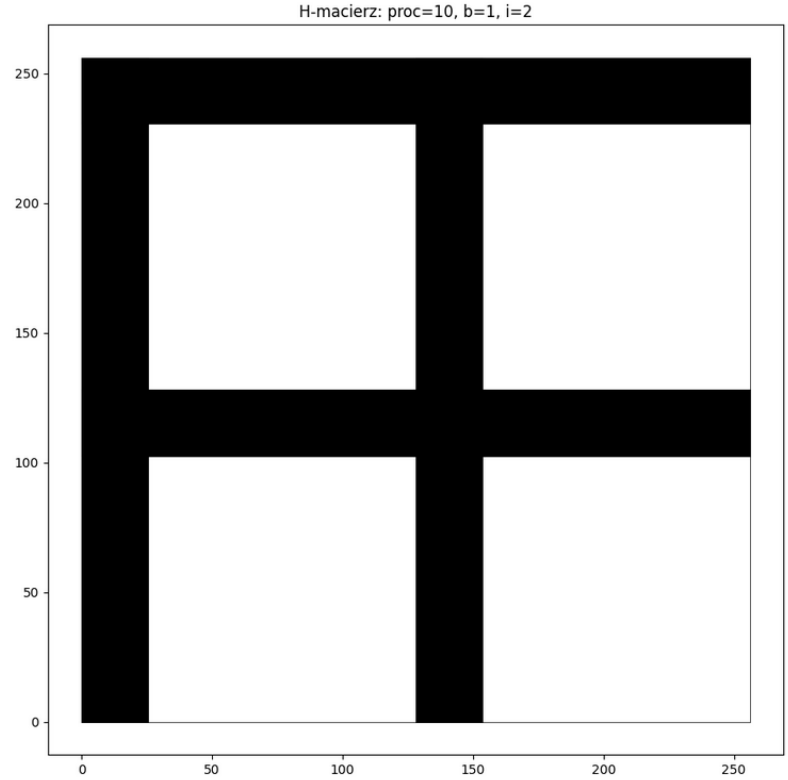
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_{2^7}



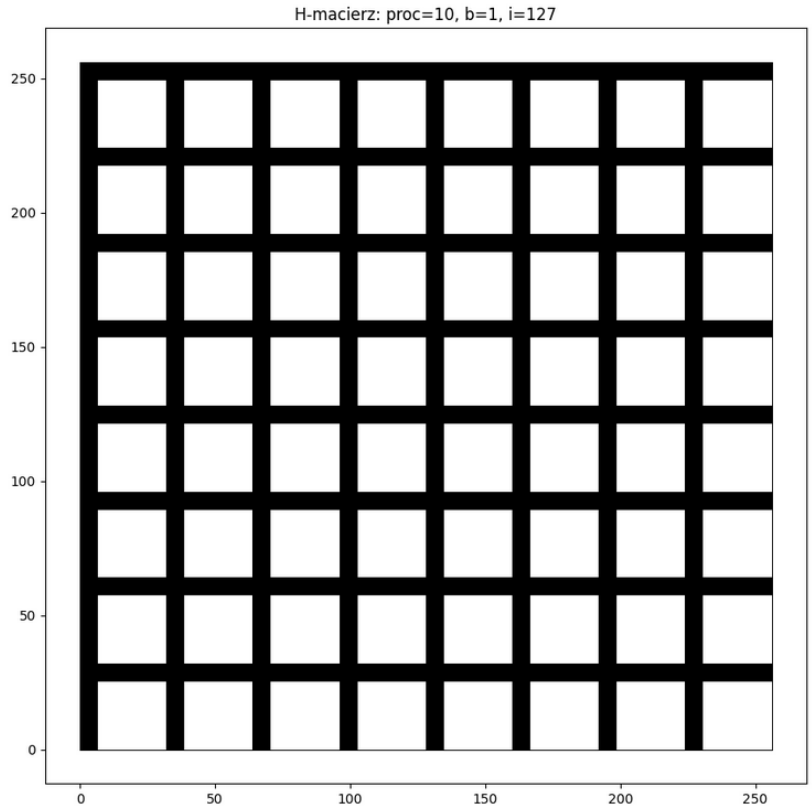
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_8



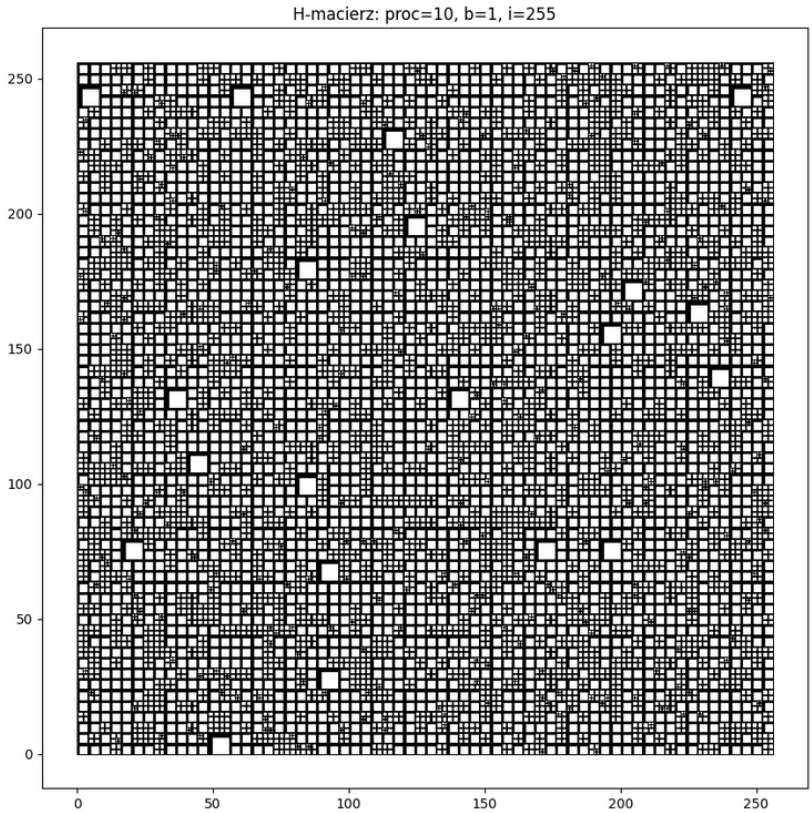
4) Ilość wartości niezerowych: 10%
   - Maksymalny rząd: 1, minimalna wartość osobliwa: σ_2



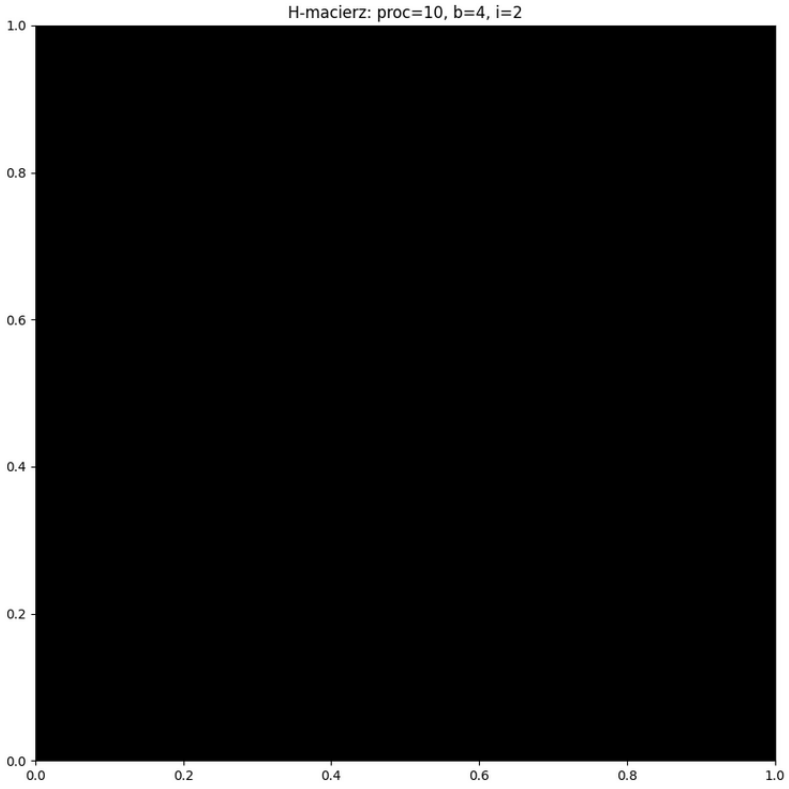
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^7}



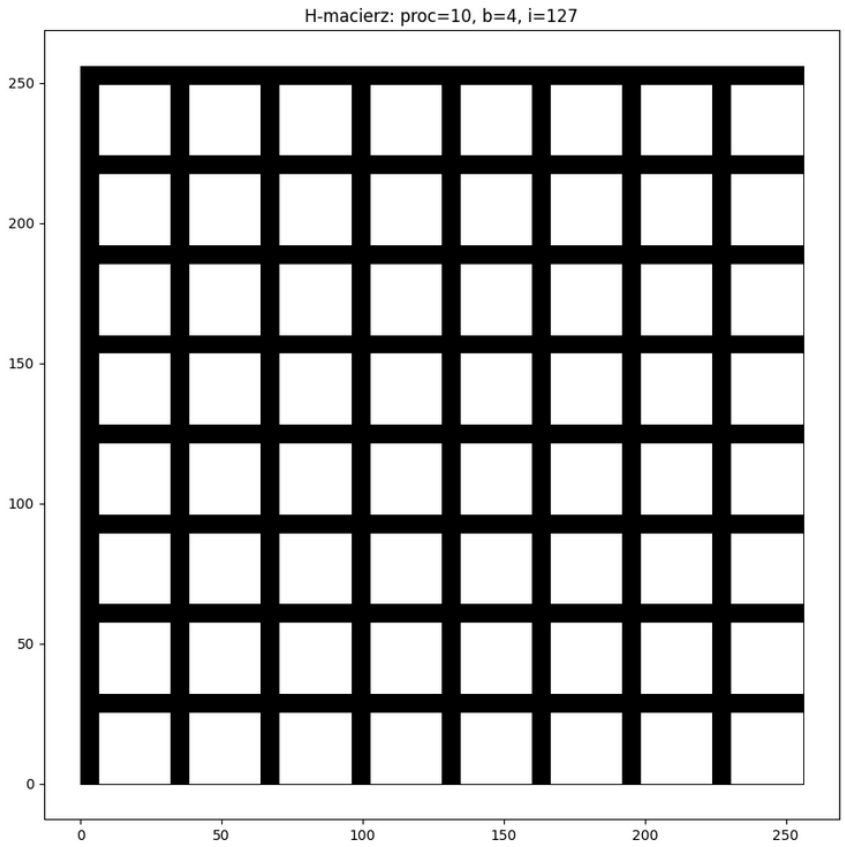
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^8}



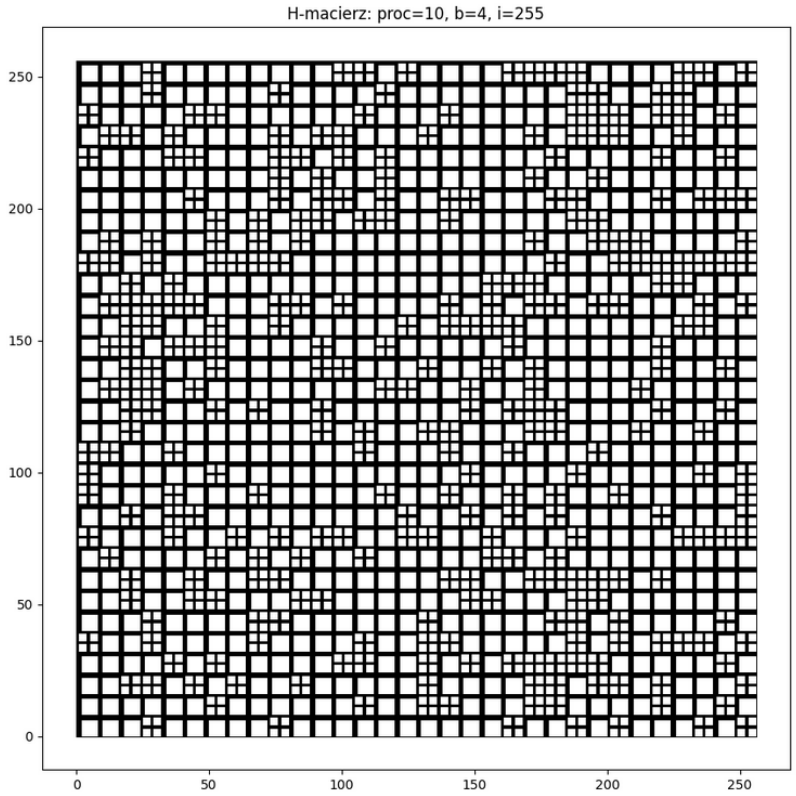
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_2



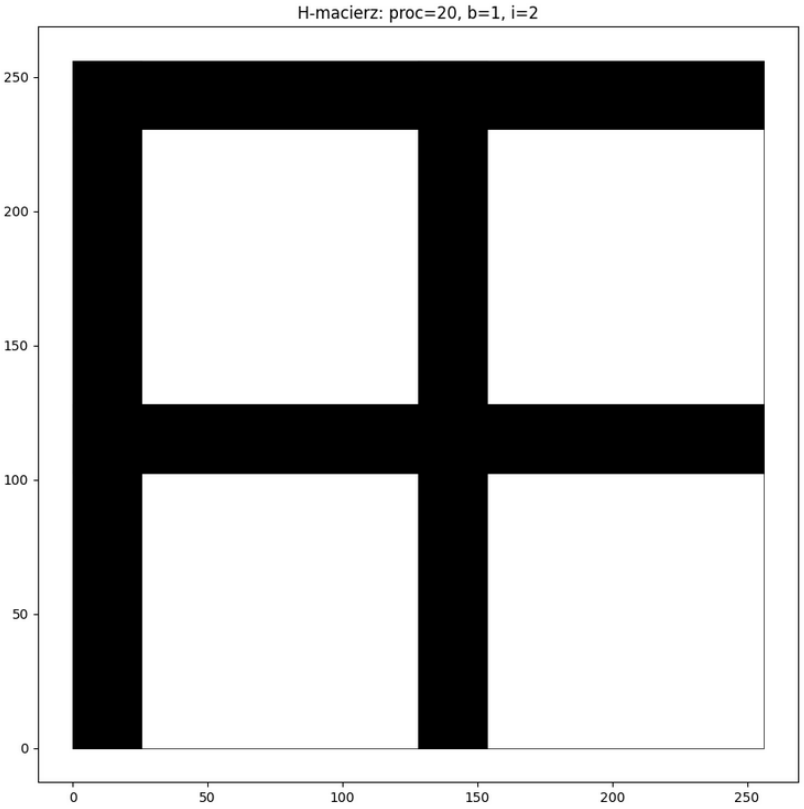
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_{2^7}



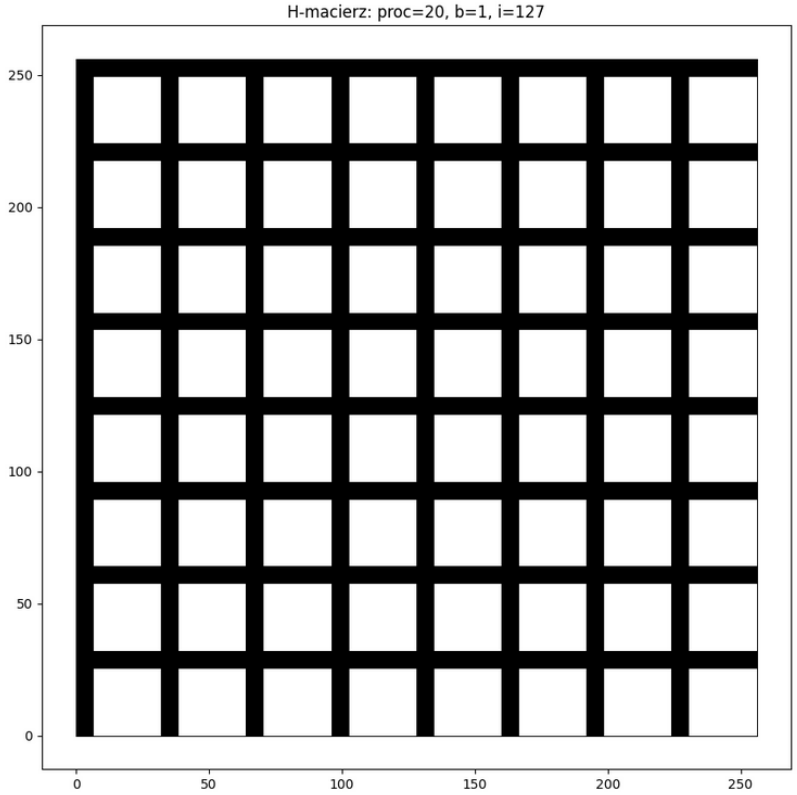
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_8



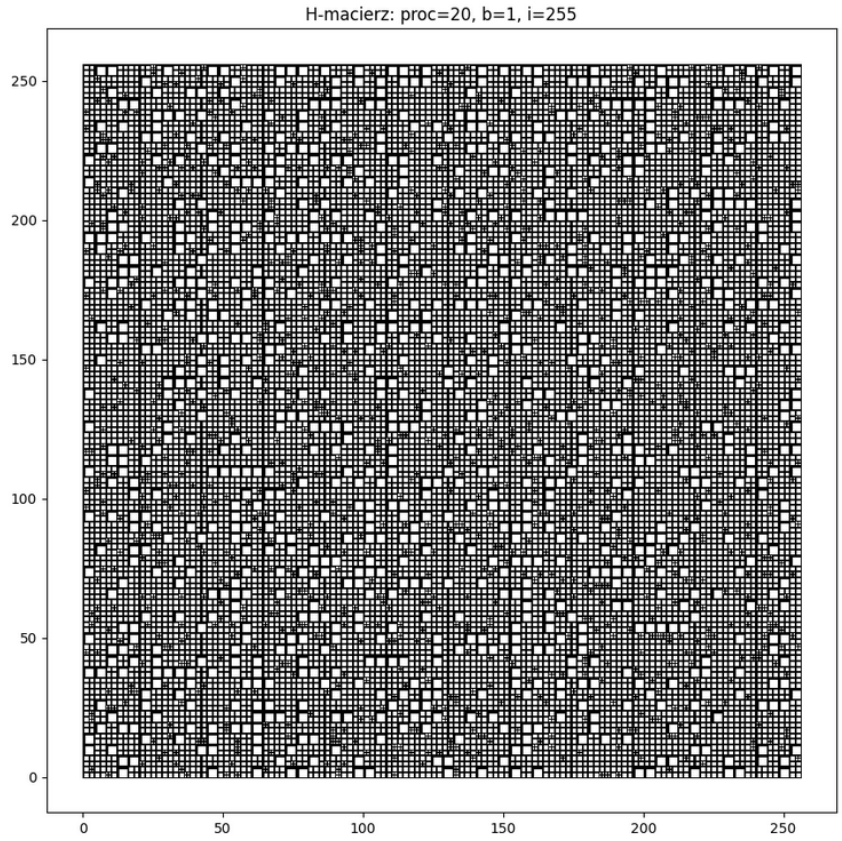
5) Ilość wartości niezerowych: 20%
   - Maksymalny rząd: 1, minimalna wartość osobliwa: σ_2



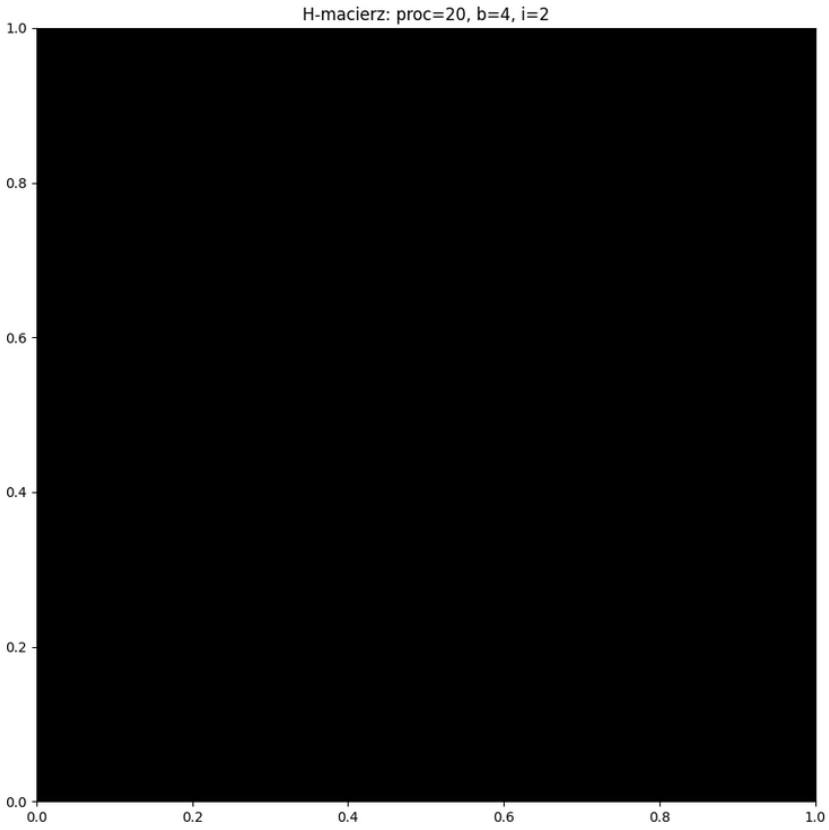
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^7}



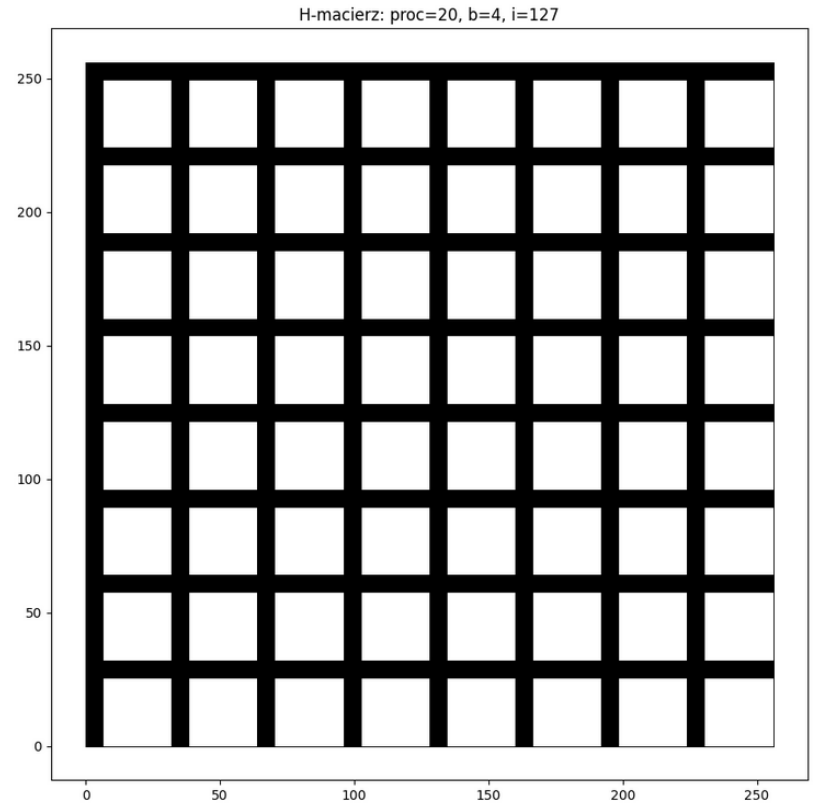
- Maksymalny rząd: 1, minimalna wartość osobliwa: σ_{2^8}



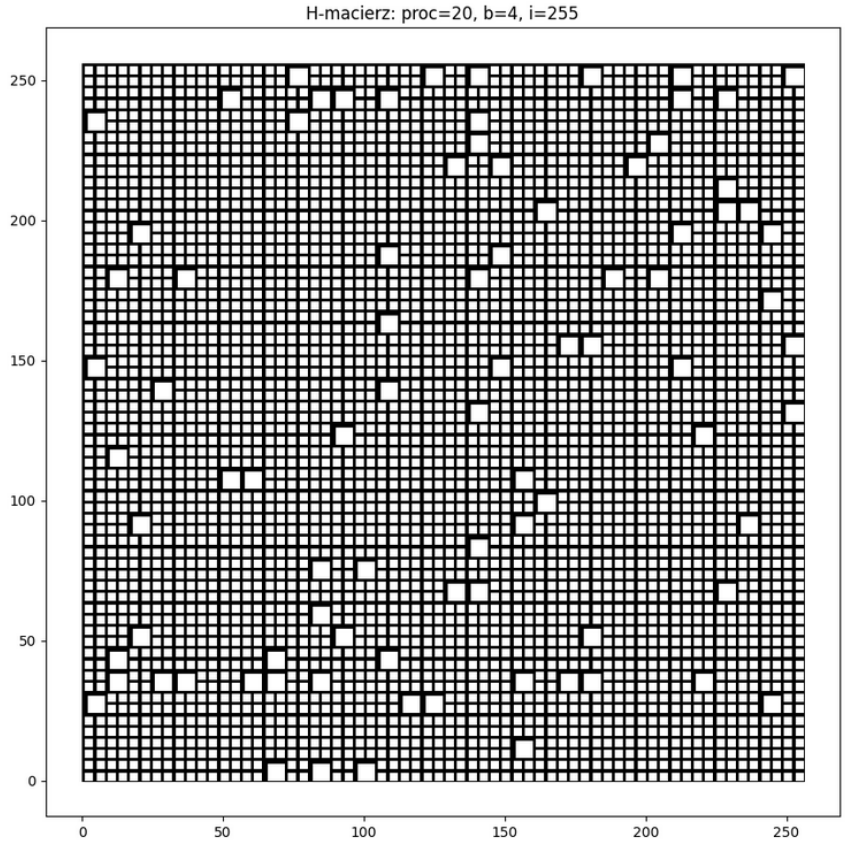
- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_2



- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_{2^7}



- Maksymalny rząd: 4, minimalna wartość osobliwa: σ_8



4. Wnioski

Analizując wykres 3.1 można stwierdzić, że czas działania rekurencyjnej kompresji SVD rośnie wraz z ilością niezerowych wartości (pomimo faktu że macierz ma takie same rozmiary). Jest to spowodowane tym, że w momencie kompresji sprawdzane jest, czy macierz nie jest zerowa. Jeżeli tak, to jej kompresja też będzie składała się z samych zer, więc pomija się obliczenia związane z mnożeniem macierzy (co w oczywisty sposób skraca czas działania algorytmu).

Biorąc pod uwagę dane z wykresu 3.2 można stwierdzić, że w każdym przypadku wartości osobliwe maleją oraz w pewnym momencie są bardzo bliskie zeru. To dlatego można czasem używać stratnej kompresji macierzy, ponieważ niewielkie wartość osobliwa nie wpłynie w znaczącym stopniu na błąd. Należy tu jednak pamiętać, że błąd ten może się kumulować i przy większych rozmiarach macierzy może on już być bardziej znaczący.

Na podstawie wykresu 3.3 można stwierdzić, że zaimplementowany algorytm zadziałał poprawnie. Błąd kwadratowy jest rzędu 10^{-14}, a co za tym idzie można powiedzieć, że uzyskane wyniki są prawidłowe.

Porównując uzyskane rysunki podczas kompresji macierzy można zauważyć, że w każdym przypadku ilość rekurencyjnych wywołań rośnie wraz ze wzrostem minimalnej wartości osobliwej (dopuszczalnego błędu), ale maleje wraz ze wzrostem rządu macierzy. Dodatkowo dla większej ilości niezerowych wartości liczba wywołań rekurencyjnych także wzrasta, co tłumaczy również fakt dłuższego czasu działania. Warto również podkreślić, że rysunki te nie są symetryczne, a co za tym idzie zależą one od ułożenia wartości w macierzy.

5. Bibliografia

- wykład z przedmiotu „Algorytmy macierzowe” przygotowany przez prof. dr hab. Macieja Paszyńskiego
- https://en.wikipedia.org/wiki/Singular_value_decomposition