

# Hierarchiczna kompresja macierzy

Maciej Borowiec

7 grudnia 2025

## 1 Przebieg algorytmu

Algorytm polega na stworzeniu drzewa, które reprezentuje skompresowaną macierz - każdy liść reprezentuje pewien region macierzy.

### 1.1 `create_tree`

Funkcja `create_tree` rekurencyjnie tworzy drzewo reprezentujące skompresowaną macierz. Na wejściu przyjmuje macierz wejściową  $A$ , region macierzy aktualnie obsługiwany ( $r_{start}, r_{end}, c_{start}, c_{end}$ ) oraz wskaźniki  $r$  i  $\epsilon$  oznaczające rząd kompresji i jej próg. Funkcja zwraca korzeń utworzonego drzewa. Do kompresji liści drzewa używa funkcji `compress_matrix`, która została opisana w kolejnym podpunkcie.

- Obliczamy  $U, d, V = \text{rSVD}(A[r_{start} : r_{end}, c_{start} : c_{end}], r + 1)$ , gdzie  $U$  i  $V$  to macierze, a  $d$  to wektor wartości osobliwych.
- Jeśli ostatnia wartość osobliwa jest mniejsza od progu  $\epsilon$  to zwracamy liść `compress_matrix`( $A, r_{start}, r_{end}, c_{start}, c_{end}, U[:, :-1], d[:, :-1], V[:, :-1, :]$ ).
- W przeciwnym wypadku:
  - Tworzymy nowy wierzchołek drzewa  $v$ .
  - Dzielimy obsługiwany fragment macierzy na podmacierze, poprzez obliczenie wartości  $r_{half}$  i  $c_{half}$ :

$$r_{half} = \frac{r_{start} + r_{end}}{2}$$

$$c_{half} = \frac{c_{start} + c_{end}}{2}$$

- Dodajemy dzieci utworzone przez rekurencyjne wywołanie funkcji `create_tree` na utworzonych podmacierzach do wierzchołka  $v$ :

```
v.add_child(create_tree(A, r_start, r_half, c_start, c_half, r, ε))
```

```
v.add_child(create_tree(A, r_start, r_half, c_half, c_end, r, ε))
```

```

v.add_child(create_tree(A, rhalf, rend, cstart, chalf, r, ε))
v.add_child(create_tree(A, rhalf, rend, chalf, cend, r, ε))

```

– Zwracamy wierzchołek  $v$ .

## 1.2 compress\_matrix

Funkcja **compress\_matrix** dokonuje kompresji liścia utworzonego drzewa. Na wejściu przyjmuje macierz wejściową  $\mathbf{A}$ , region macierzy aktualnie obsługiwany ( $r_{start}, r_{end}, c_{start}, c_{end}$ ), trójkę  $U, d, V$  - wynik **rSVD** na obsługiwanej części macierzy oraz rząd kompresji  $r$ . Funkcja zwraca skompresowany liść.

- Tworzymy liść  $v$ .
- Ustawiamy region liścia:

$$v.region = (r_{start}, r_{end}, c_{start}, c_{end})$$

- Jeśli region zawiera same zera, to ustawiamy rząd na 0:

$$v.rank = 0$$

- W przeciwnym wypadku ustawiamy rząd na  $r$  i obliczamy wartości tego regionu macierzy, używając  $U, d$  oraz  $V$ :

$$v.rank = r$$

$$v.matrix = U \cdot \text{diag}(d) \cdot V$$

- Zwracamy liść  $v$ .

## 2 Kod

```
1 # class for nodes of tree
2 class Node:
3     def __init__(self):
4         self.rank = None
5         self.region = None
6         self.matrix = None
7         self.children = set()
8
9     def add_child(self, v):
10        self.children.add(v)
```

Listing 1: Klasa wierzchołka

```
1 # leaf handler
2 def compress_matrix(A, r_start, r_end, c_start, c_end, U, d, V, compression):
3     v = Node()
4     v.region = (r_start, r_end, c_start, c_end)
5     if np.all(A[r_start:r_end, c_start:c_end] < EPS):
6         v.rank = 0
7     else:
8         v.rank = compression
9         v.matrix = U@np.diag(d)@V
10    return v
```

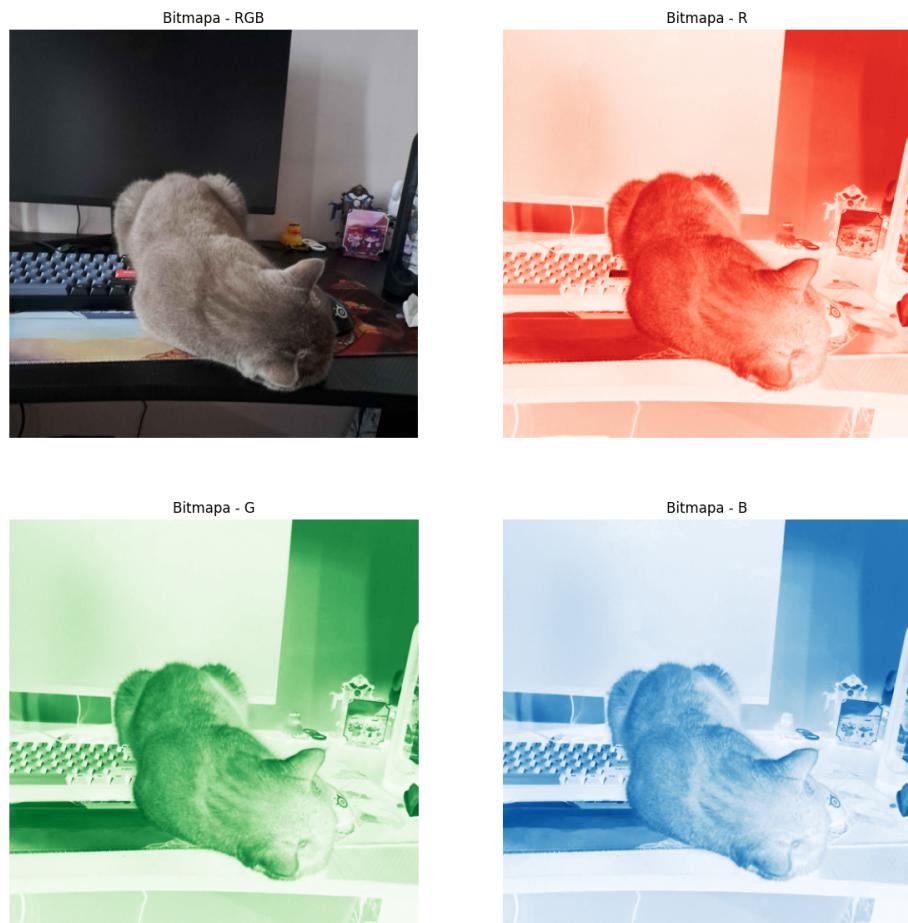
Listing 2: Funkcja compress\_matrix

```
1 # create tree
2 def create_tree(A, r_start, r_end, c_start, c_end, compression, threshold):
3     # handle cases where compression >= size
4     if r_end - r_start <= compression or c_end - c_start <= compression:
5         U, d, V = r_svd(A[r_start:r_end, c_start:c_end],
6                           min(compression, r_end-r_start, c_end-c_start))
7         return compress_matrix(A, r_start, r_end, c_start, c_end,
8                               U, d, V, compression)
9
10    U, d, V = r_svd(A[r_start:r_end, c_start:c_end], compression+1)
11    if d[-1] < threshold:
12        v = compress_matrix(A, r_start, r_end, c_start, c_end,
13                             U[:, :-1], d[:-1], V[:-1, :], compression)
14    else:
15        v = Node()
16        r_half = (r_start + r_end) // 2
17        c_half = (c_start + c_end) // 2
18        v.add_child(create_tree(A, r_start, r_half, c_start, c_half,
19                               compression, threshold))
20        v.add_child(create_tree(A, r_start, r_half, c_half, c_end,
21                               compression, threshold))
22        v.add_child(create_tree(A, r_half, r_end, c_start, c_half,
23                               compression, threshold))
24        v.add_child(create_tree(A, r_half, r_end, c_half, c_end,
25                               compression, threshold))
26    return v
```

Listing 3: Funkcja create\_tree

### 3 Nieskompresowana bitmapa

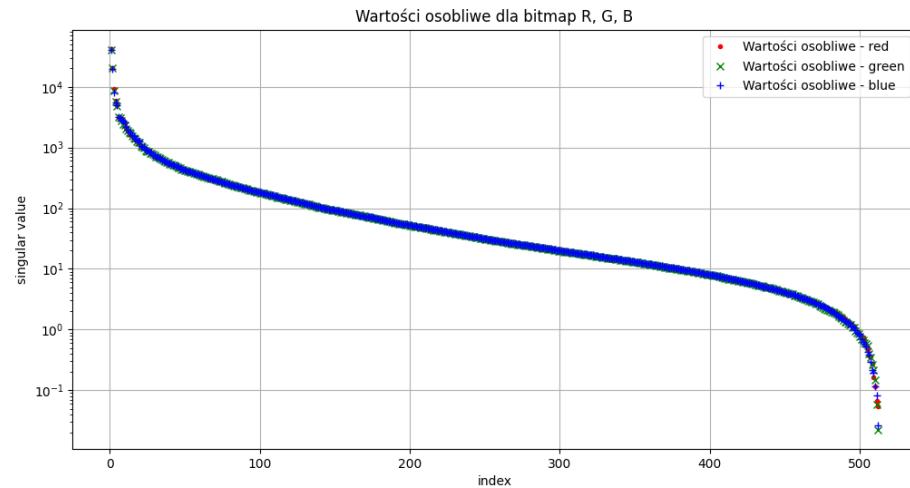
Poniżej została przedstawiona oryginalna bitmapa oraz bitmapy poszczególnych kolorów - R, G, B. Obraz przedstawia moją kotkę - Irmę :)



Rysunek 1: Nieskompresowany bitmapy

## 4 Wykres wartości osobliwych

Poniżej przedstawiono wykres wartości osobliwych dla macierzy R, G, B. Dla osi y użyto skali logarytmicznej.

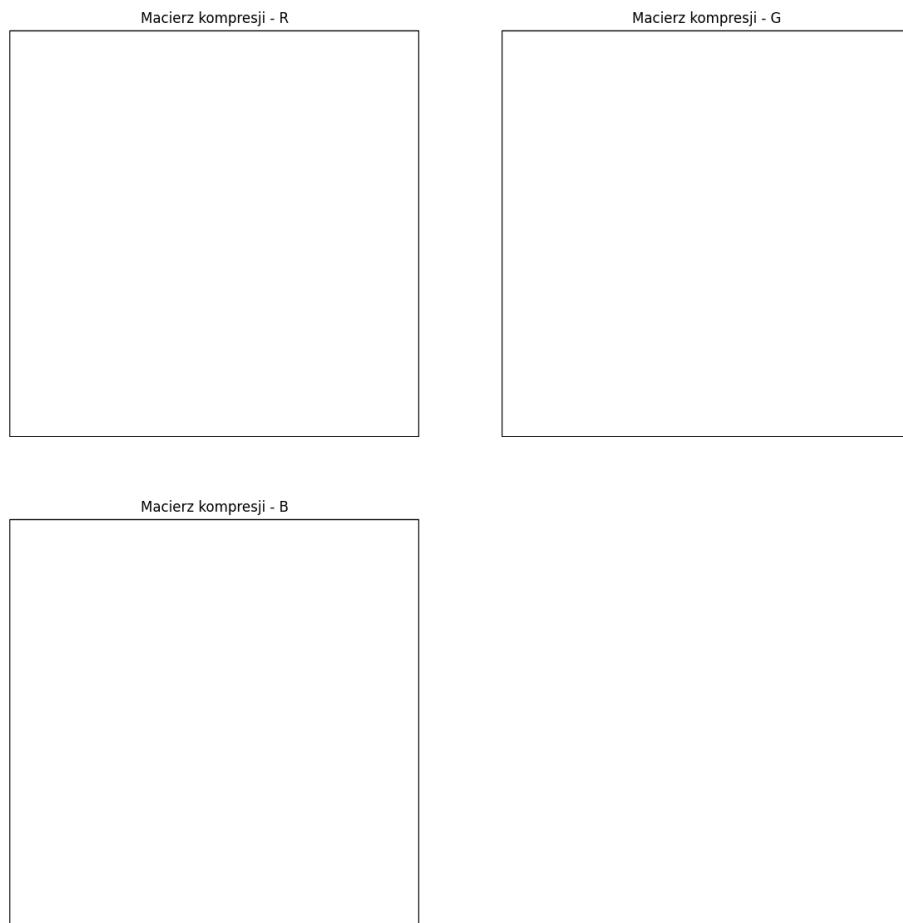


Rysunek 2: Wartości osobliwe macierzy R, G, B

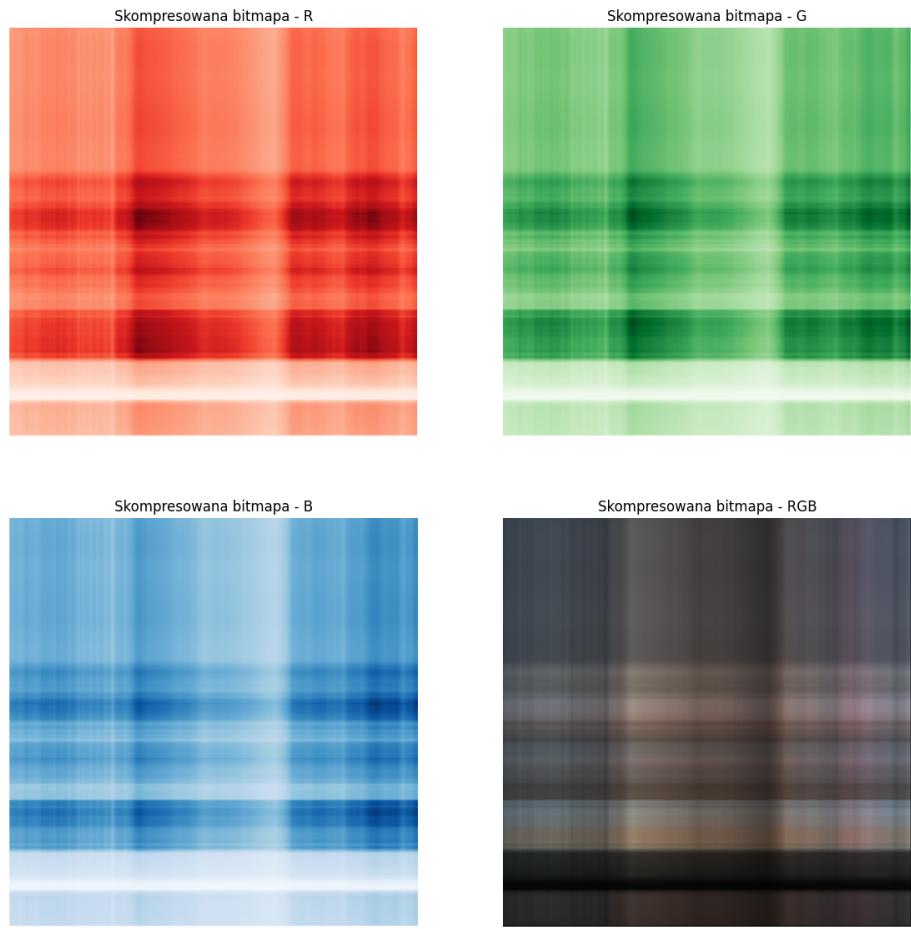
## 5 Wyniki

Poniżej przedstawiono macierze kompresji oraz skompresowane bitmapy dla różnych parametrów

**5.1**  $r = 1, \epsilon = \sigma_0$

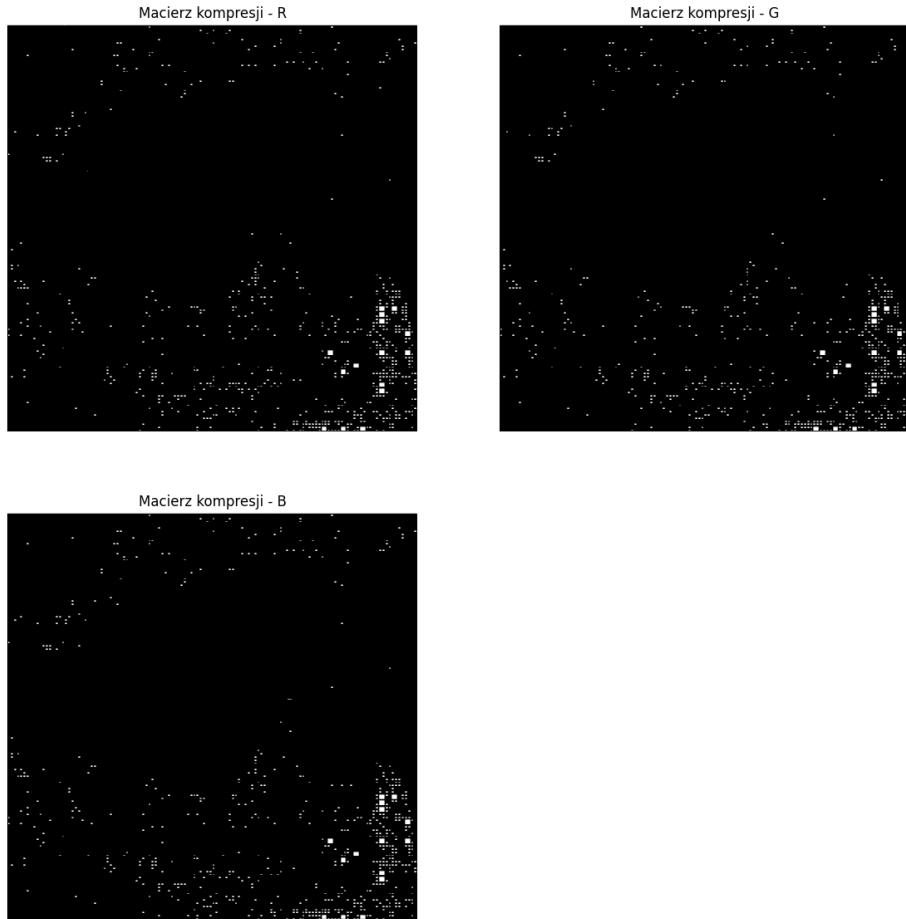


Rysunek 3: Macierze kompresji dla  $r = 1$  i  $\epsilon = \sigma_0$



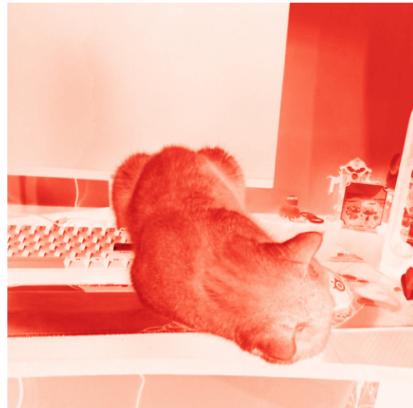
Rysunek 4: Bitmapy dla  $r = 1$  i  $\epsilon = \sigma_0$

## 5.2 $r = 1, \epsilon = \sigma_{n-1}$

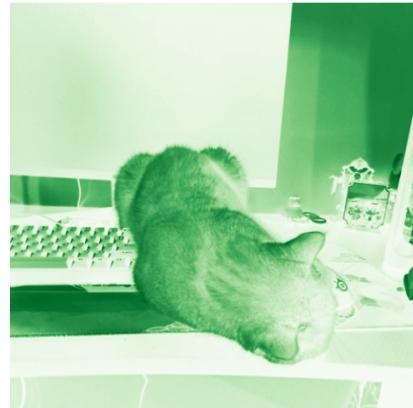


Rysunek 5: Macierze kompresji dla  $r = 1$  i  $\epsilon = \sigma_{n-1}$

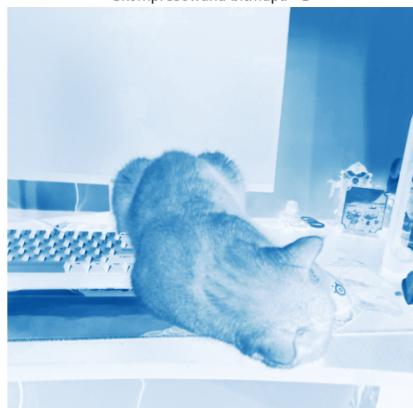
Skompresowana bitmapa - R



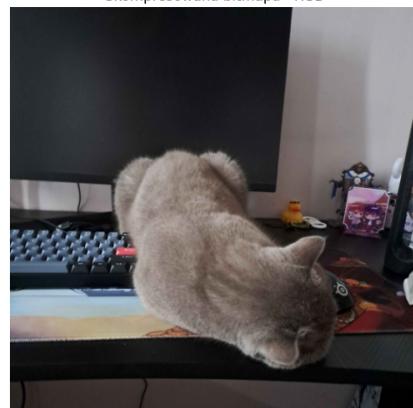
Skompresowana bitmapa - G



Skompresowana bitmapa - B

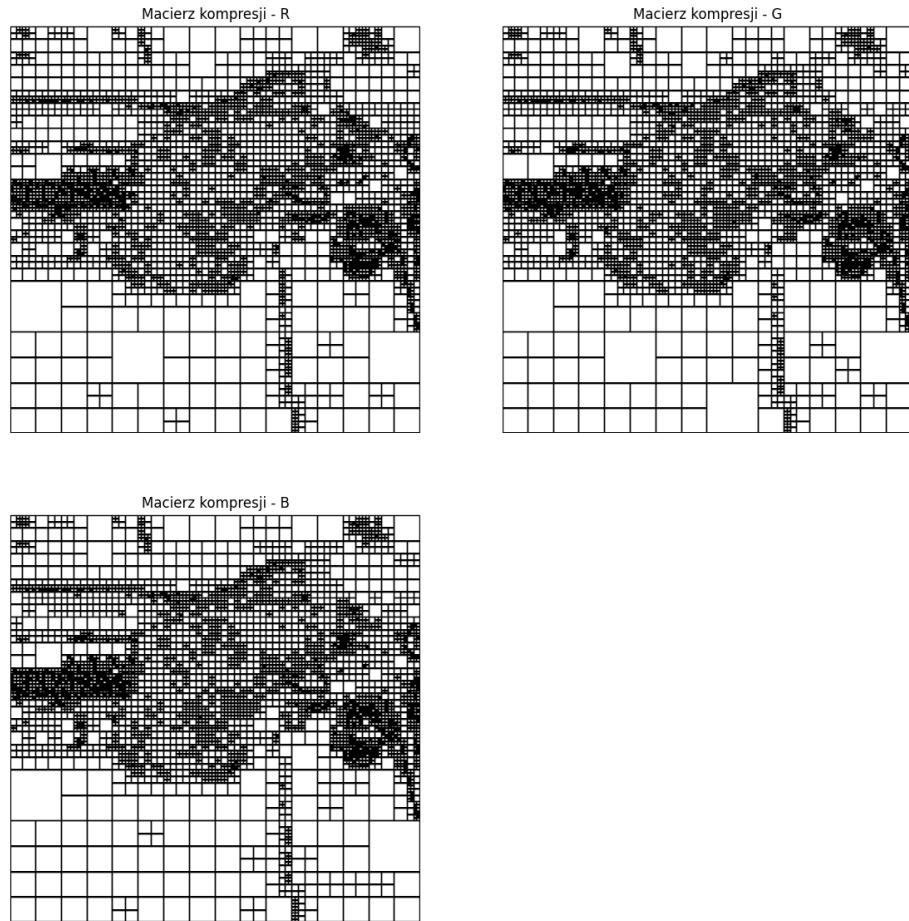


Skompresowana bitmapa - RGB



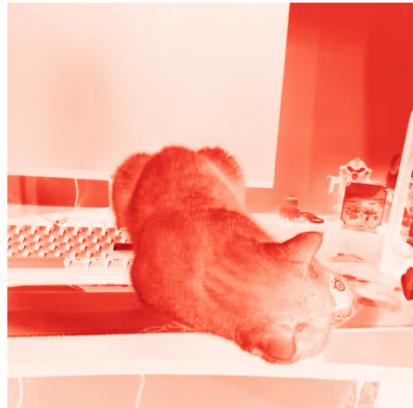
Rysunek 6: Bitmapy dla  $r = 1$  i  $\epsilon = \sigma_{n-1}$

### 5.3 $r = 1, \epsilon = \sigma_{n/2}$

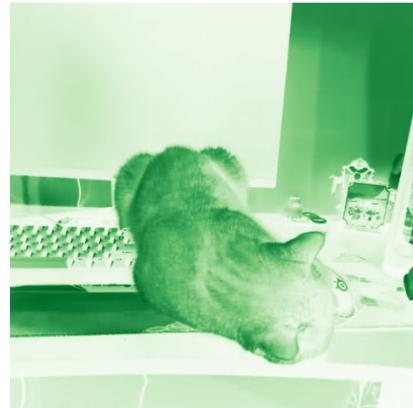


Rysunek 7: Macierze kompresji dla  $r = 1$  i  $\epsilon = \sigma_{n/2}$

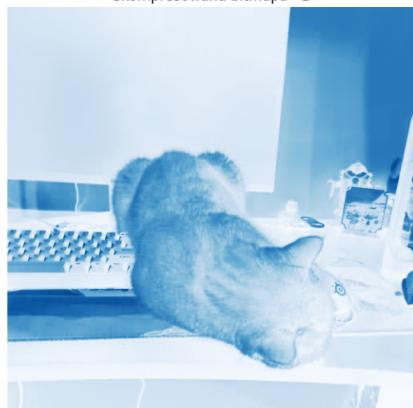
Skompresowana bitmapa - R



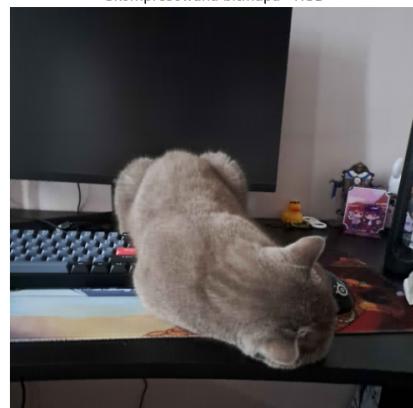
Skompresowana bitmapa - G



Skompresowana bitmapa - B

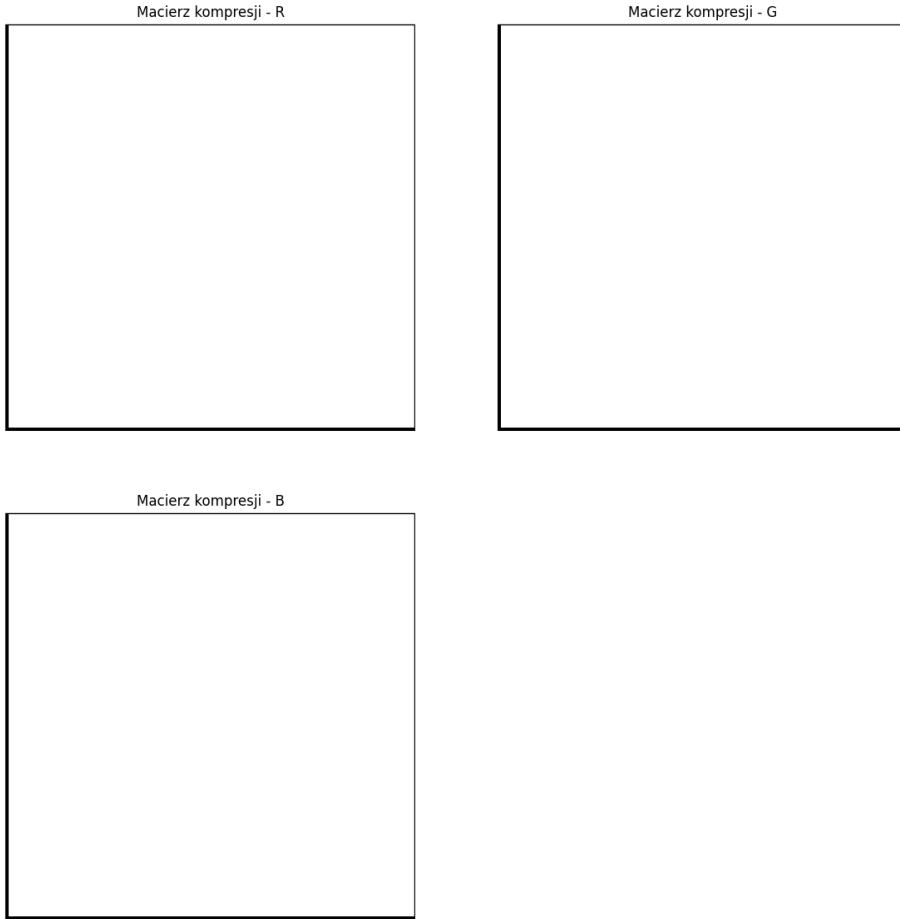


Skompresowana bitmapa - RGB

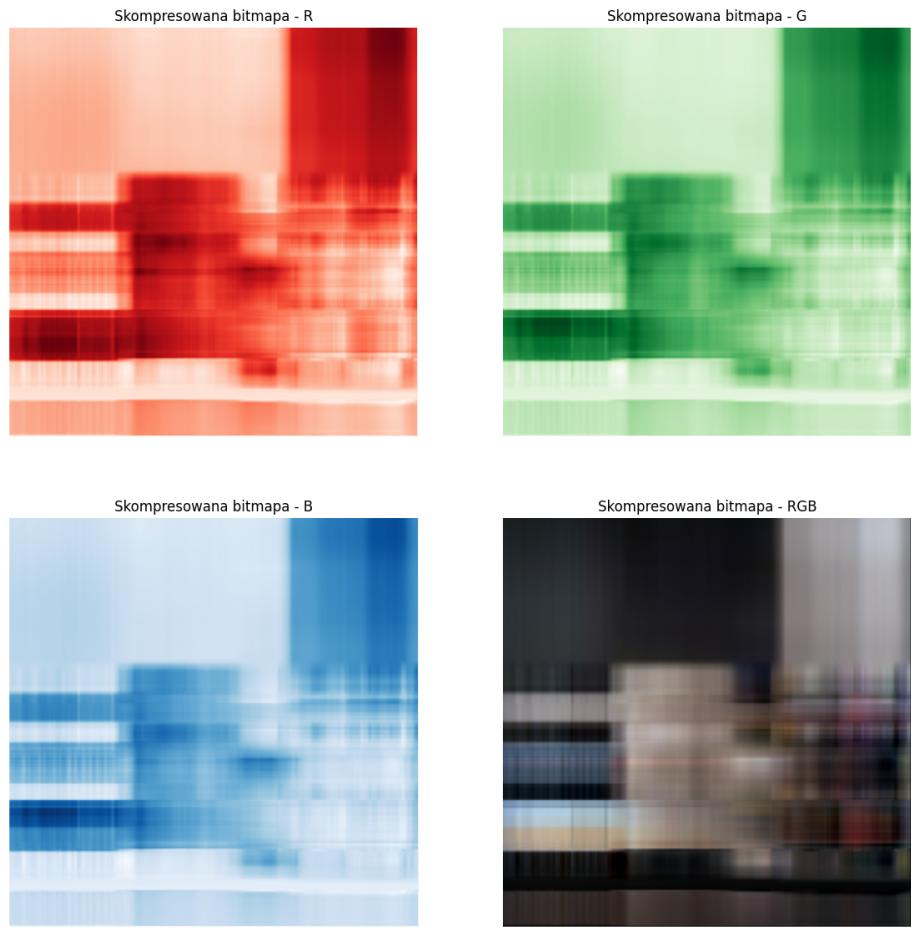


Rysunek 8: Bitmapy dla  $r = 1$  i  $\epsilon = \sigma_{n/2}$

**5.4**    $r = 4, \epsilon = \sigma_0$

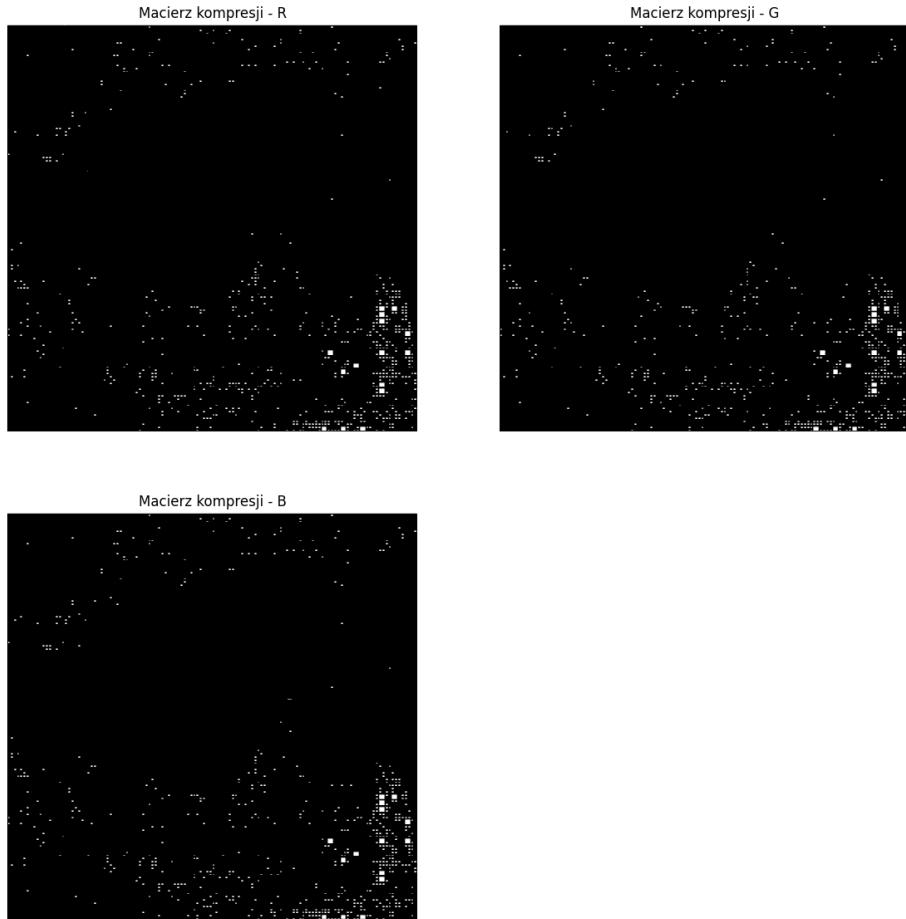


Rysunek 9: Macierze kompresji dla  $r = 4$  i  $\epsilon = \sigma_0$



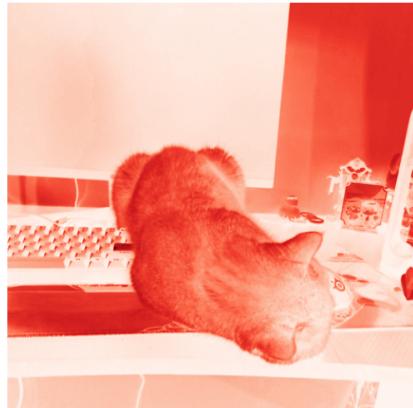
Rysunek 10: Bitmapy dla  $r = 4$  i  $\epsilon = \sigma_0$

## 5.5 $r = 4$ , $\epsilon = \sigma_{n-1}$

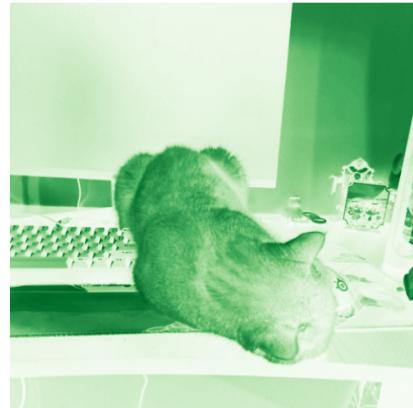


Rysunek 11: Macierze kompresji dla  $r = 4$  i  $\epsilon = \sigma_{n-1}$

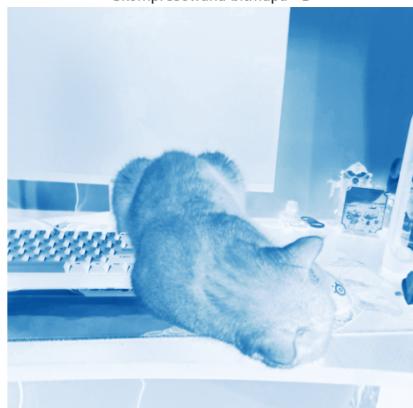
Skompresowana bitmapa - R



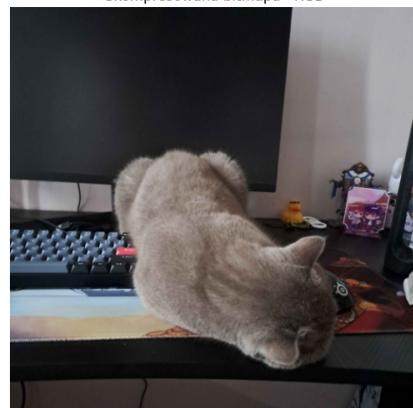
Skompresowana bitmapa - G



Skompresowana bitmapa - B

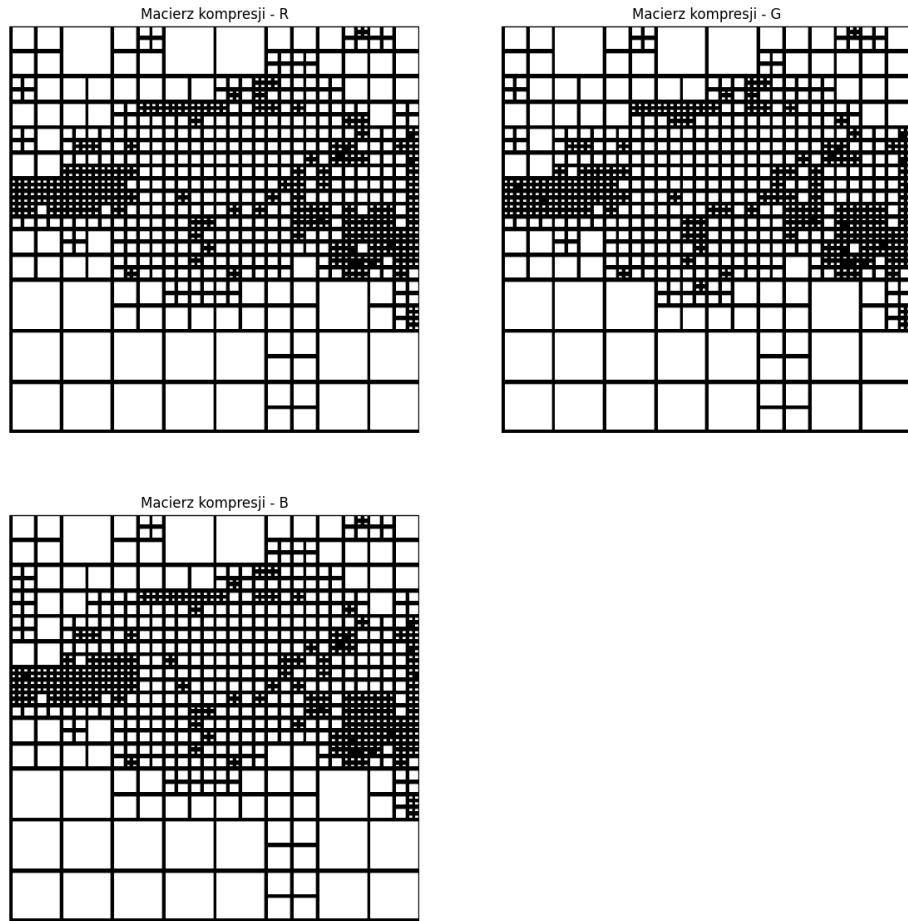


Skompresowana bitmapa - RGB



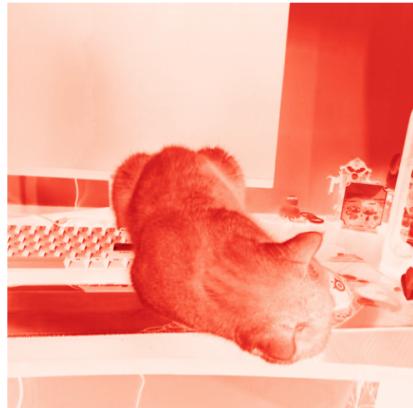
Rysunek 12: Bitmapy dla  $r = 4$  i  $\epsilon = \sigma_{n-1}$

**5.6**    $r = 4$ ,  $\epsilon = \sigma_{n/2}$

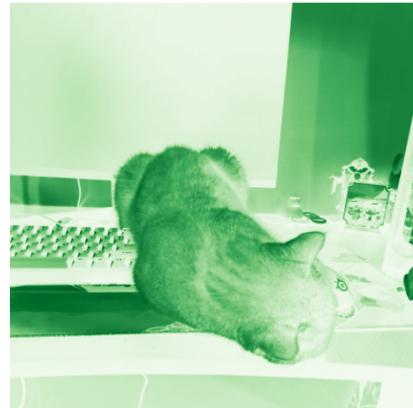


Rysunek 13: Macierze kompresji dla  $r = 4$  i  $\epsilon = \sigma_{n/2}$

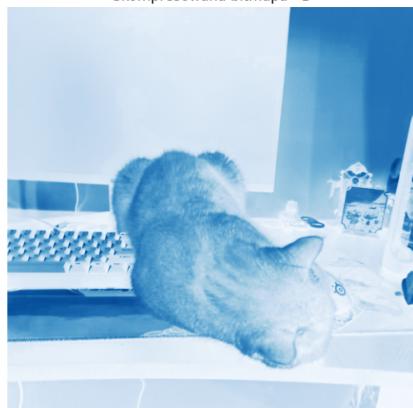
Skompresowana bitmapa - R



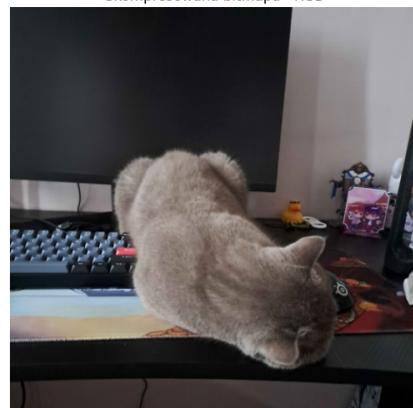
Skompresowana bitmapa - G



Skompresowana bitmapa - B



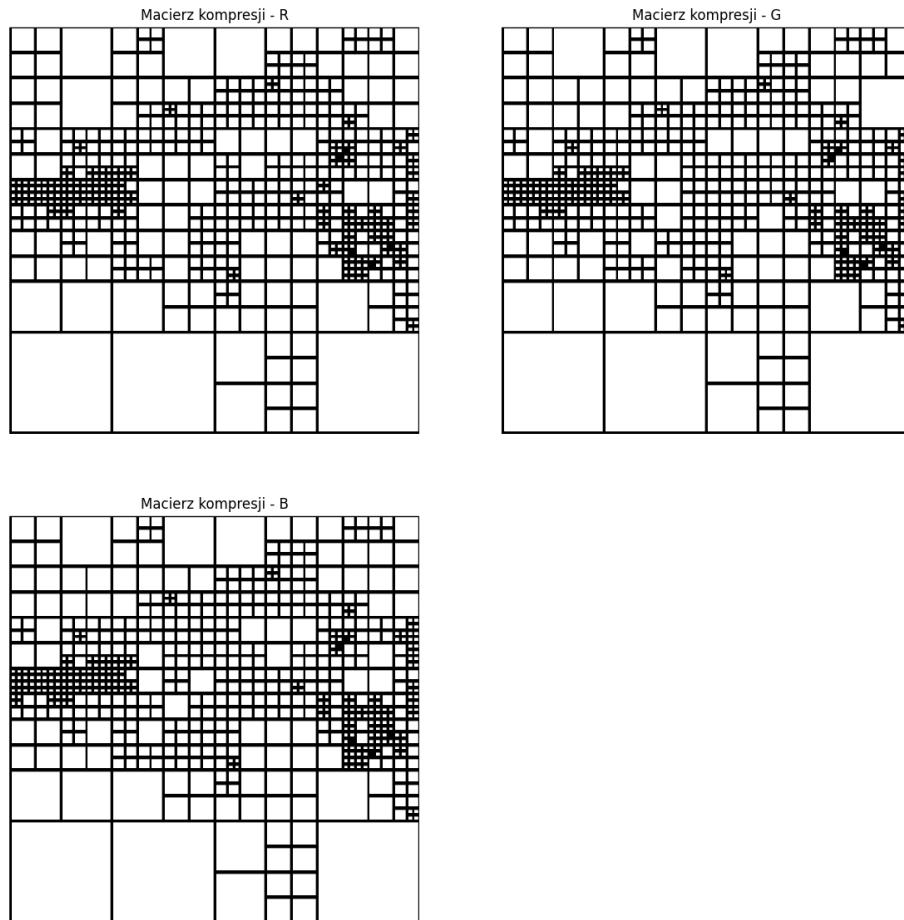
Skompresowana bitmapa - RGB



Rysunek 14: Bitmapy dla  $r = 4$  i  $\epsilon = \sigma_n/2$

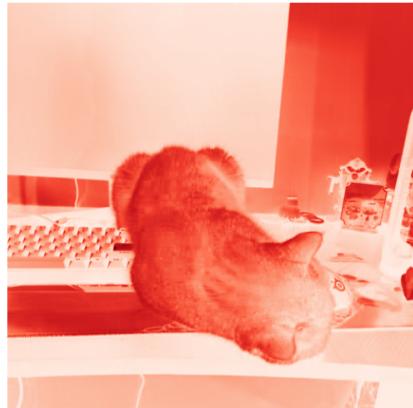
## 5.7 Dopasowane parametry

Jako wybrane parametry wziąłem  $r = 3$  i  $\epsilon = \sigma_{n/3}$ . Dla tych parametrów obraz wynikowy jest nadal w miarę dobrej jakości. Większa kompresja stwarzała bardzo rozmazany obraz.



Rysunek 15: Macierze kompresji dla  $r = 3$  i  $\epsilon = \sigma_{n/3}$

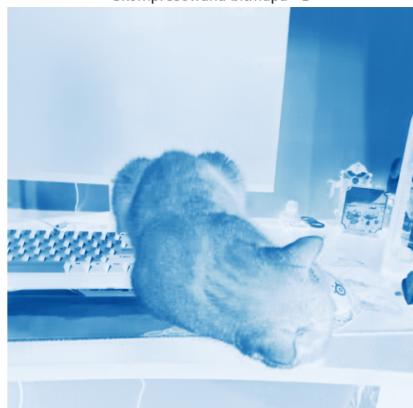
Skompresowana bitmapa - R



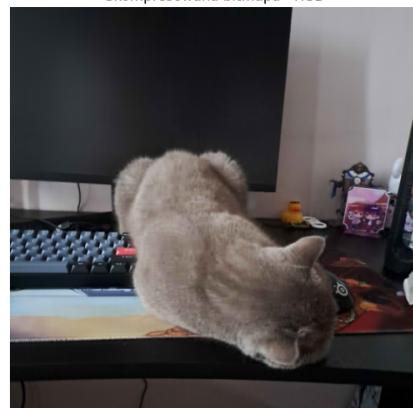
Skompresowana bitmapa - G



Skompresowana bitmapa - B



Skompresowana bitmapa - RGB



Rysunek 16: Bitmapy dla  $r = 3$  i  $\epsilon = \sigma_n/3$