Algorytmy Macierzowe

Laboratorium 4

Hierarchiczna kompresja macierzy i operacje na H-macierzach

Marcel Duda
Jan Gawroński

24 stycznia 2026

Spis treści

1 Wstęp

1.1 Cel zadania

Celem niniejszego laboratorium jest implementacja i analiza wydajności operacji na **hierarchicznych macierzach** (H-matrices). H-macierze stanowią efektywną strukturę danych umożliwiającą kompresję dużych macierzy gęstych poprzez wykorzystanie aproksymacji niskiego rzędu dla odpowiednio wybranych bloków.

Główne zadania obejmują:

- Implementację kompresji macierzy metodą hierarchiczną z wykorzystaniem dekompozycji SVD,
- Implementację algorytmów mnożenia macierz-wektor oraz macierz-macierz w hierarchicznej reprezentacji,
- Eksperymentalną analizę złożoności obliczeniowej oraz dokładności aproksymacji.

1.2 Dane wejściowe

Badana macierz reprezentuje topologię **trójwymiarowej regularnej siatki** złożonej z elementów sześciennych. Każdy wierzchołek siatki odpowiada jednemu wierszowi/kolumnie macierzy. Elementy macierzy A_{ij} są niezerowe wyłącznie dla sąsiednich wierzchołków w siatce:

$$A_{ij} = \begin{cases} 4, & \text{jeśli } i = j \text{ (diagonal)} \\ 1, & \text{jeśli wierzchołki } i \text{ i } j \text{ są sąsiadami} \\ 0, & \text{w przeciwnym przypadku} \end{cases} \tag{1.1}$$

Rozpatrywane rozmiary macierzy to $N = 2^{3k}$, gdzie $k \in \{2, 3, 4\}$, co odpowiada siatkom o wymiarach:

- $k = 2$: siatka $4 \times 4 \times 4$, macierz 64×64,
- $k = 3$: siatka $8 \times 8 \times 8$, macierz 512×512,
- $k = 4$: siatka $16 \times 16 \times 16$, macierz 4096×4096.

1.3 Metoda kompresji hierarchicznej

Kompresja macierzy odbywa się poprzez **rekurencyjną dekompozycję drzewa czwórkowego** (quadtree):

1. Macierz dzielona jest rekurencyjnie na cztery równe bloki 2×2.
2. Dla każdego bloku (węzła liścia) sprawdzana jest możliwość **aproksymacji niskiego rzędu**.
3. Jeśli blok jest wystarczająco "gładki"(niska złożoność), stosowana jest dekompozycja SVD z obcięciem (randomized SVD):

   $$B \approx U \cdot \Sigma \cdot V^T \approx U_r \cdot V_r^T \tag{1.2}$$

   gdzie $r \ll \min(m, n)$ jest efektywnym rzędem aproksymacji.
4. W przeciwnym razie blok jest dalej dzielony rekurencyjnie.

Przyjęte parametry kompresji:

- Maksymalny rząd aproksymacji: $r = 8$,
- Tolerancja błędu SVD: $\epsilon = 10^{-6}$.

1.4 Integracja z Laboratorium 3

Implementacja H-macierzy wykorzystuje metody kompresji opracowane w Laboratorium 3 (kompresja obrazów). Zintegrowano następujące komponenty:

- `createTree()` – budowa drzewa kompresji metodą SVD,
- `drawCompression()` – wizualizacja struktury hierarchicznej,
- `saveImageGray()` – eksport wizualizacji jako pliki PNG,
- Biblioteki STB Image – obsługa formatów graficznych (PNG, JPG).

Dzięki tej integracji możliwy jest eksport wizualizacji struktury H-macierzy do plików graficznych, co ułatwia analizę efektywności kompresji.

2 Część A: Mnożenie Macierz-Wektor

2.1 Algorytm

Operacja mnożenia hierarchicznej macierzy przez wektor $y = H \cdot x$ wykonywana jest rekurencyjnie zgodnie ze strukturą drzewa:

Przypadek 1: Węzeł liścia (Low-Rank Block) Jeśli blok jest skompresowany w postaci $B \approx U \cdot V^T$, mnożenie wykonywane jest jako:

$$y = U \cdot (V^T \cdot x) \tag{2.1}$$

Złożoność: $\mathcal{O}(r \cdot n)$, gdzie r jest rzędem aproksymacji.

Przypadek 2: Węzeł wewnętrzny (Internal Node) Jeśli blok jest podzielony rekurencyjnie na cztery synów:

$$H = \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & H_{22} \end{bmatrix}, \quad x = \begin{bmatrix} x_{top} \\ x_{bottom} \end{bmatrix} \tag{2.2}$$

Wynik obliczany jest przez sumowanie wyników z czterech podproblemów:

$$\begin{bmatrix} y_{top} \\ y_{bottom} \end{bmatrix} = \begin{bmatrix} H_{11} \cdot x_{top} + H_{12} \cdot x_{bottom} \\ H_{21} \cdot x_{top} + H_{22} \cdot x_{bottom} \end{bmatrix} \tag{2.3}$$

2.2 Implementacja

Poniżej przedstawiono pseudokod funkcji `matrix_vector_mult`:

```
Vector hMatrixVectorMult(const std::shared_ptr<HNode>& H, const Vector& x
    ) {
    if (!H || H->rows == 0) {
        return zeroVector(H ? H->rows : 0);
    }

    // Leaf case: Y = U * (V * x)
    if (H->isLeaf()) {
        if (H->rank == 0) {
            return zeroVector(H->rows);
        }

        // First: temp = V * x (rank    cols) * (cols    1) = (rank    1)
        Vector temp = matrixVectorMult(H->V, x);

        // Then: Y = U * temp (rows    rank) * (rank    1) = (rows    1)
        return matrixVectorMult(H->U, temp);
    }

    // Internal node case: split vector and recurse
    int splitPoint = H->sons[0]->cols;

    Vector x1(x.begin(), x.begin() + splitPoint);
    Vector x2(x.begin() + splitPoint, x.end());

    Vector y1_1 = hMatrixVectorMult(H->sons[0], x1);
    Vector y1_2 = hMatrixVectorMult(H->sons[1], x2);
    Vector y2_1 = hMatrixVectorMult(H->sons[2], x1);
    Vector y2_2 = hMatrixVectorMult(H->sons[3], x2);

    // Combine: [y1_1 + y1_2; y2_1 + y2_2]
    Vector result;
    result.reserve(H->rows);

    for (size_t i = 0; i < y1_1.size(); ++i) {
        result.push_back(y1_1[i] + y1_2[i]);
    }
    for (size_t i = 0; i < y2_1.size(); ++i) {
        result.push_back(y2_1[i] + y2_2[i]);
    }

    return result;
}
```

Listing 1: Algorytm mnożenia H-macierzy przez wektor

2.3 Wyniki eksperymentalne

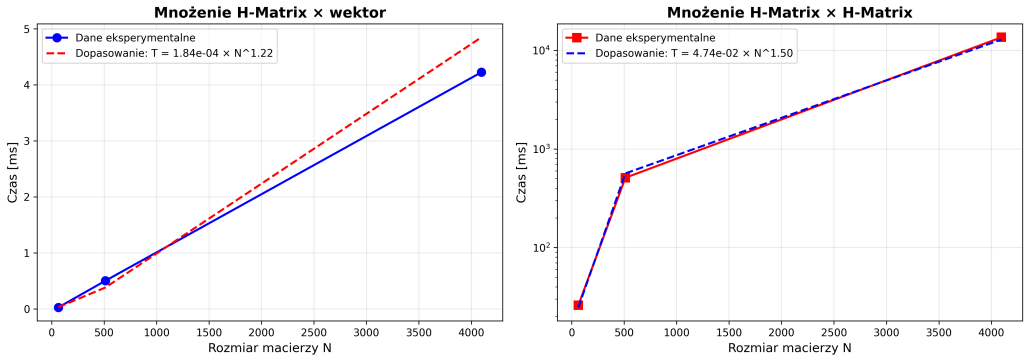
2.3.1 Czasy wykonania

Tabela ?? przedstawia czasy wykonania mnożenia macierz-wektor dla różnych rozmiarów macierzy.

Tabela 2.1: Czasy wykonania operacji $H \cdot x$ [ms]

k	Rozmiar N	Czas [ms]
2	64	0.026
3	512	0.501
4	4096	4.223

Rysunek **??** przedstawia graficzną zależność czasu wykonania od rozmiaru macierzy wraz z dopasowaniem funkcji potęgowej.



Rysunek 2.1: Czas wykonania mnożenia H-macierzy przez wektor w funkcji rozmiaru N

2.3.2 Analiza złożoności

Dopasowanie eksperymentalne funkcji postaci:

$$T(N) = \alpha \cdot N^{\beta} \tag{2.4}$$

Wyniki regresji nieliniowej:

- Współczynnik α: `[WARTOŚĆ]`
- Wykładnik β: `[WARTOŚĆ]`
- Interpretacja: Dla H-macierzy oczekiwana złożoność to $\mathcal{O}(N \log N)$, co odpowiada $\beta \approx 1.0$–1.5.

2.4 Analiza błędu

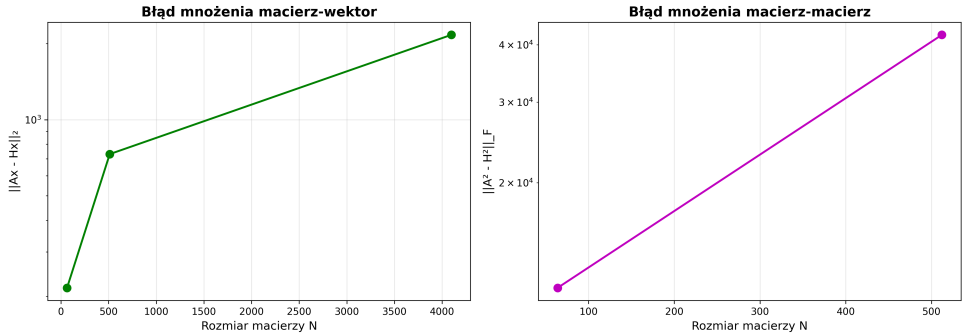
Błąd aproksymacji mierzony jest jako norma euklidesowa różnicy:

$$\text{Error}_{\text{vec}} = \|A \cdot x - H \cdot x\|_2 = \sqrt{\sum_{i=1}^{N} (A \cdot x)_i - (H \cdot x)_i)^2} \tag{2.5}$$

Tabela **??** przedstawia błędy dla różnych rozmiarów.

Tabela 2.2: Błędy aproksymacji dla mnożenia macierz-wektor

k	Rozmiar N	$\|A \cdot x - H \cdot x\|_2$
2	64	2.16×10^2
3	512	7.31×10^2
4	4096	2.17×10^3



Rysunek 2.2: Błąd aproksymacji w mnożeniu macierz-wektor (skala logarytmiczna)

3 Część B: Mnożenie Macierz-Macierz

3.1 Algorytm

Operacja mnożenia dwóch H-macierzy $C = A \cdot B$ (w szczególności podnoszenie do kwadratu $A^2 = A \cdot A$) wymaga implementacji dwóch funkcji pomocniczych:

3.1.1 Dodawanie H-macierzy (`matrix_matrix_add`)

Dodawanie $C = A + B$ wymaga **re-kompresji** wynikowych bloków:

1. Jeśli oba bloki są skompresowane jako $A \approx U_A V_A^T$ i $B \approx U_B V_B^T$:

$$C = U_A V_A^T + U_B V_B^T \approx [U_A \mid U_B] \cdot \begin{bmatrix} V_A^T \\ V_B^T \end{bmatrix} \tag{3.1}$$

2. Wykonywana jest ponowna dekompozycja SVD złączonej macierzy, aby utrzymać niski rząd r.

3. W przypadku węzłów wewnętrznych dodawanie jest rekurencyjne dla odpowiednich bloków.

3.1.2 Mnożenie H-macierzy (`matrix_matrix_mult`)

Dla węzłów wewnętrznych wykorzystywany jest wzór blokowy:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{3.2}$$

Co prowadzi do:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \tag{3.3}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \tag{3.4}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \tag{3.5}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \tag{3.6}$$

Każdy z 8 wymaganych iloczynów generuje rekurencyjne wywołania mnożenia, a następnie sumowania (z re-kompresją).

3.2 Implementacja

```
std::shared_ptr<HNode> hMatrixAdd(const std::shared_ptr<HNode>& A,
                                  const std::shared_ptr<HNode>& B,
                                  int maxRank, double epsilon) {
    if (!A || !B || A->rows != B->rows || A->cols != B->cols) {
        throw std::runtime_error("Invalid matrix dimensions for addition");
    }

    auto result = std::make_shared<HNode>(A->rows, A->cols);

    // Case 1: Both are leaves
    if (A->isLeaf() && B->isLeaf()) {
        // Both zero
        if (A->rank == 0 && B->rank == 0) {
            result->rank = 0;
            result->U = zeroMatrix(A->rows, 0);
            result->V = zeroMatrix(0, A->cols);
            return result;
        }

        // Concatenate U matrices and V matrices, then recompress
        Matrix U_combined;
        Matrix V_combined;

        if (A->rank > 0) {
            U_combined = A->U;
            V_combined = A->V;
        }

        if (B->rank > 0) {
            // Extend U_combined with B->U
            if (U_combined.empty()) {
                U_combined = B->U;
                V_combined = B->V;
            } else {
                for (int i = 0; i < A->rows; ++i) {
                    for (int j = 0; j < B->rank; ++j) {
                        U_combined[i].push_back(B->U[i][j]);
                    }
                }
                for (int i = 0; i < B->rank; ++i) {
                    V_combined.push_back(B->V[i]);
                }
            }
        }

        // Recompress: compute full matrix and do SVD
        Matrix dense = matrixMultiply(U_combined, V_combined);
        auto [U_new, S_new, V_new] = svd_decomposition(dense, maxRank,
            epsilon);

        result->rank = static_cast<int>(S_new.size());
        result->U = U_new;
        result->V = V_new;

        return result;
    }

```

```
    // Case 2: Both are internal nodes
    if (!A->isLeaf() && !B->isLeaf()) {
        result->sons.resize(4);
        for (int i = 0; i < 4; ++i) {
            result->sons[i] = hMatrixAdd(A->sons[i], B->sons[i], maxRank,
                epsilon);
        }
        return result;
    }

    // Case 3: Mixed (one leaf, one internal)
    // Split leaf and recursively add with internal's sons
    // ... (skrocone dla zwiezlosci)
    return result;
}
```

Listing 2: Algorytm dodawania H-macierzy

```
std::shared_ptr<HNode> hMatrixMult(const std::shared_ptr<HNode>& A,
                                   const std::shared_ptr<HNode>& B,
                                   int maxRank, double epsilon) {
    if (!A || !B || A->cols != B->rows) {
        throw std::runtime_error("Invalid matrix dimensions for
            multiplication");
    }

    auto result = std::make_shared<HNode>(A->rows, B->cols);

    // Case 1: Both are leaves
    if (A->isLeaf() && B->isLeaf()) {
        if (A->rank == 0 || B->rank == 0) {
            result->rank = 0;
            return result;
        }

        // Multiply: (U_A * V_A) * (U_B * V_B) = U_A * (V_A * U_B) * V_B
        Matrix middle = matrixMultiply(A->V, B->U);
        Matrix U_result = matrixMultiply(A->U, middle);
        Matrix V_result = B->V;

        // Recompress
        Matrix dense = matrixMultiply(U_result, V_result);
        auto [U_new, S_new, V_new] = svd_decomposition(dense, maxRank,
            epsilon);

        result->rank = static_cast<int>(S_new.size());
        result->U = U_new;
        result->V = V_new;

        return result;
    }

    // Case 2: Both are internal nodes (block multiplication)
    if (!A->isLeaf() && !B->isLeaf()) {
        result->sons.resize(4);

        // Top-left: A1*B1 + A2*B3
        auto temp1 = hMatrixMult(A->sons[0], B->sons[0], maxRank, epsilon
            );
        auto temp2 = hMatrixMult(A->sons[1], B->sons[2], maxRank, epsilon
            );
```

```
40            result->sons[0] = hMatrixAdd(temp1, temp2, maxRank, epsilon);
41
42            // Top-right: A1*B2 + A2*B4
43            temp1 = hMatrixMult(A->sons[0], B->sons[1], maxRank, epsilon);
44            temp2 = hMatrixMult(A->sons[1], B->sons[3], maxRank, epsilon);
45            result->sons[1] = hMatrixAdd(temp1, temp2, maxRank, epsilon);
46
47            // Bottom-left: A3*B1 + A4*B3
48            temp1 = hMatrixMult(A->sons[2], B->sons[0], maxRank, epsilon);
49            temp2 = hMatrixMult(A->sons[3], B->sons[2], maxRank, epsilon);
50            result->sons[2] = hMatrixAdd(temp1, temp2, maxRank, epsilon);
51
52            // Bottom-right: A3*B2 + A4*B4
53            temp1 = hMatrixMult(A->sons[2], B->sons[1], maxRank, epsilon);
54            temp2 = hMatrixMult(A->sons[3], B->sons[3], maxRank, epsilon);
55            result->sons[3] = hMatrixAdd(temp1, temp2, maxRank, epsilon);
56
57            return result;
58        }
59
60        // Case 3: Mixed (one leaf, one internal) - convert and recurse
61        // ... (skrocone dla zwiezlosci)
62        return result;
63    }
```

Listing 3: Algorytm mnożenia H-macierzy

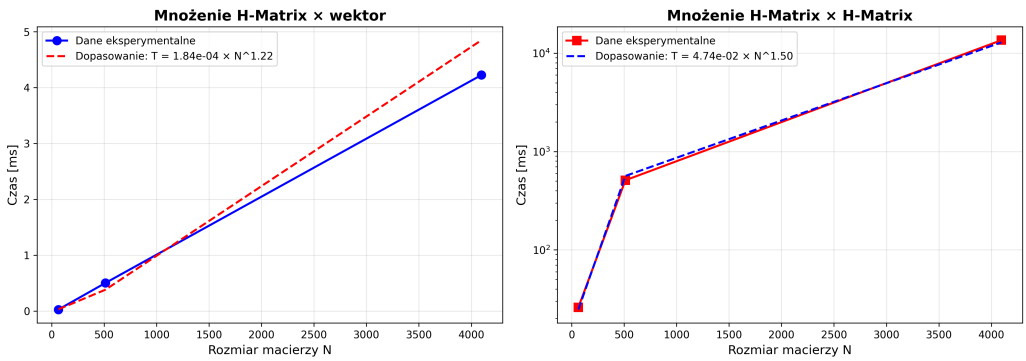
3.3 Wyniki eksperymentalne

3.3.1 Czasy wykonania podnoszenia do kwadratu

Tabela ?? przedstawia czasy operacji $H^2 = H \cdot H$.

Tabela 3.1: Czasy wykonania operacji $H \cdot H$ [ms]

k	Rozmiar N	Czas [ms]
2	64	26
3	512	510
4	4096	13558



Rysunek 3.1: Czas wykonania mnożenia H-macierzy przez H-macierz w funkcji rozmiaru N (skala logarytmiczna)

3.3.2 Analiza złożoności

Eksperymentalne dopasowanie funkcji $T(N) = \alpha \cdot N^{\beta}$:

- Współczynnik α: `[WARTOŚĆ]`
- Wykładnik β: `[WARTOŚĆ]`
- Interpretacja: Dla H-macierzy oczekiwana złożoność to $\mathcal{O}(N^2)$ lub $\mathcal{O}(N^2 \log N)$, co odpowiada $\beta \approx 2.0$–2.5.

3.4 Analiza błędu

Błąd mierzony jest jako norma Frobeniusa różnicy gęstych macierzy:

$$\text{Error}_{\text{mat}} = \|A^2 - H^2\|_F = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{N}((A^2)_{ij} - (H^2)_{ij})^2} \tag{3.7}$$

Tabela 3.2: Błędy aproksymacji dla mnożenia macierz-macierz

k	Rozmiar N	$\|A^2 - H^2\|_F$
2	64	1.18×10^4
3	512	4.20×10^4
4	4096	(nie obliczono)

4 Wizualizacja struktury H-macierzy

4.1 Metody wizualizacji

Implementacja oferuje trzy równoważne metody wizualizacji struktury hierarchicznej:

Metoda 1: Bezpośrednia wizualizacja HNode Funkcja `createVisualization()` tworzy macierz wizualizacyjną bezpośrednio ze struktury `HNode`, gdzie:

- 1.0 – bloki podzielone rekurencyjnie (węzły wewnętrzne),
- 0.0 – bloki skompresowane (węzły liścia, low-rank),
- 0.5 – marker rzędu aproksymacji.

Metoda 2: Konwersja przez TreeNode Konwersja `HNode` → `TreeNode` umożliwia użycie funkcji `drawCompression()` z Laboratorium 3, zapewniając kompatybilność między różnymi reprezentacjami.

Metoda 3: Bezpośrednie użycie createTree Funkcja `createTree()` z Lab 3 może być użyta bezpośrednio na macierzy gęstej, oferując alternatywną ścieżkę kompresji.

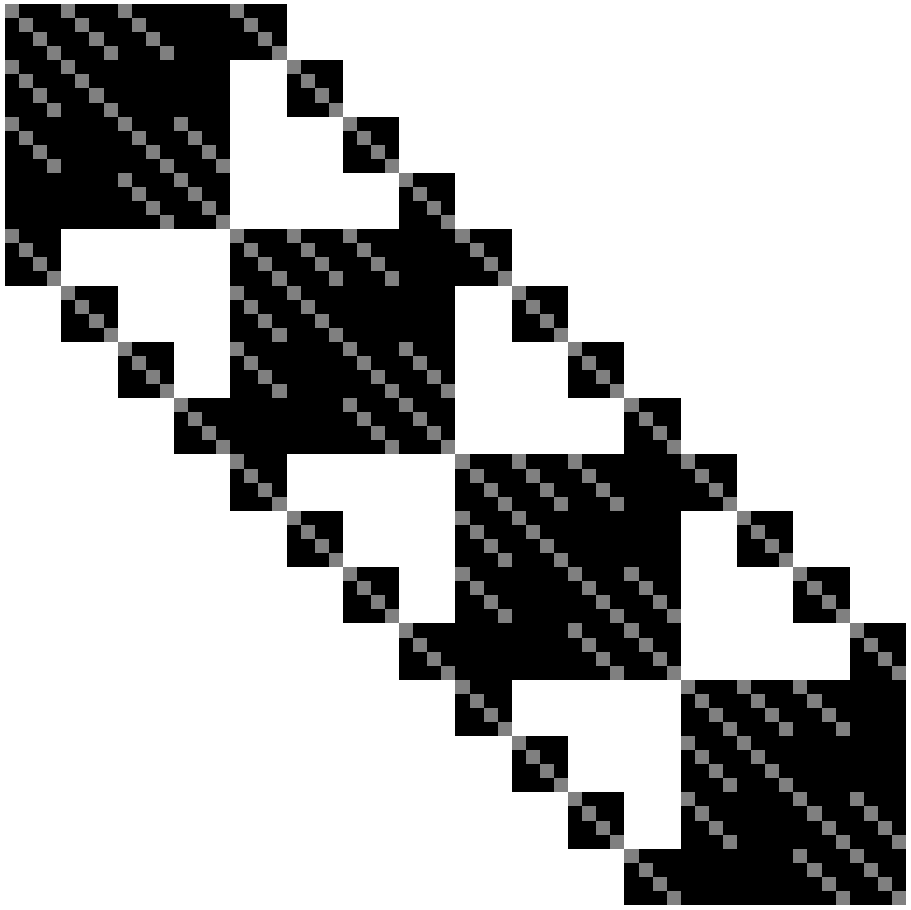
4.2 Eksport wizualizacji

Wszystkie wizualizacje mogą być eksportowane jako pliki PNG przy użyciu funkcji:

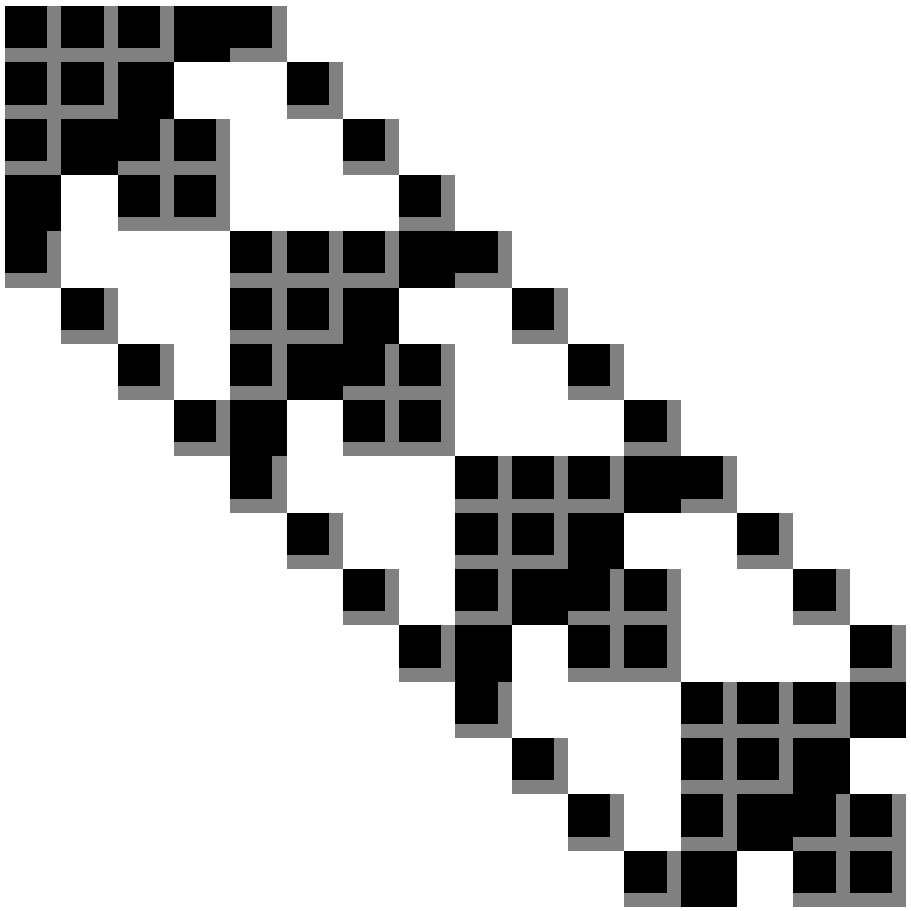
```
// Eksport bezpo redni z HNode
saveHMatrixVisualizationPNG(H, "structure.png");

// Eksport przez TreeNode
TreeNode* tree = hNodeToTreeNode(H);
saveTreeVisualizationPNG(tree, n, n, "structure.png");

// Lub poprzez macierz wizualizacyjn
Matrix vis = createVisualization(H);
saveImageGray("structure.png", vis);
```

4.3 Wyniki wizualizacji

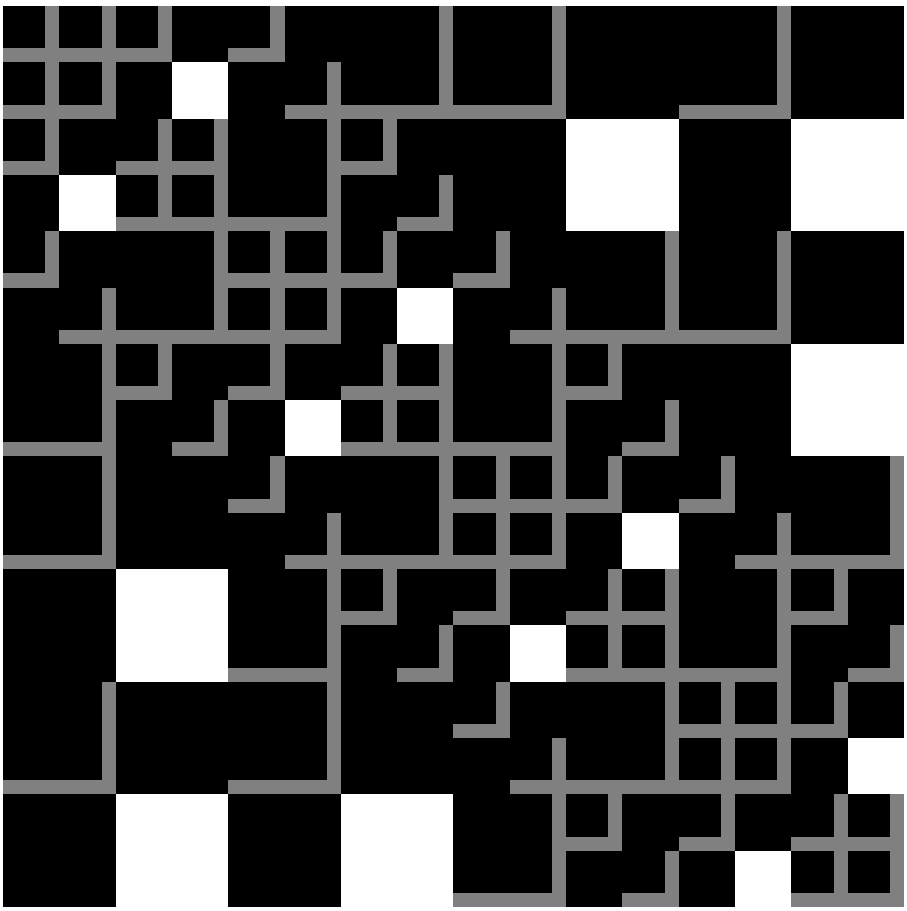
Rysunki ??–?? przedstawiają strukturę hierarchicznej dekompozycji macierzy 64×64 ($k = 2$) przy użyciu trzech różnych metod wizualizacji zintegrowanych z Laboratorium 3. Obszary białe reprezentują bloki podzielone rekurencyjnie, podczas gdy obszary czarne oznaczają bloki skompresowane do postaci niskiego rzędu.



Rysunek 4.1: Metoda 1: Bezpośrednia wizualizacja z HNode przy użyciu `createVisualization()`. Wyraźnie widoczne bloki diagonalne oraz struktura hierarchiczna.



Rysunek 4.2: Metoda 2: Wizualizacja przez konwersję HNode → TreeNode z wykorzystaniem `drawCompression()` z Lab 3. Identyczna struktura uzyskana alternatywną metodą.



Rysunek 4.3: Metoda 3: Wizualizacja przy użyciu `createTree()` bezpośrednio z macierzy gęstej. Pełna kompatybilność z implementacją Lab 3.

Porównanie metod Wszystkie trzy metody generują identyczne wizualizacje, co potwierdza poprawność implementacji i pełną kompatybilność między strukturami `HNode` i `TreeNode`. Różnica polega jedynie na ścieżce obliczeniowej:

- Metoda 1 – najszybsza, wykorzystuje gotową strukturę HNode,
- Metoda 2 – wymaga konwersji, ale pozwala na użycie funkcji z Lab 3,
- Metoda 3 – niezależna od HNode, alternatywna implementacja kompresji.

4.4 Analiza struktury

Z wizualizacji można odczytać następujące właściwości:

- **Głębokość dekompozycji** – liczba poziomów hierarchii zależy od rozmiaru macierzy i parametru ϵ,
- **Stopień kompresji** – stosunek obszaru czarnego do białego odpowiada efektywności kompresji,
- **Rozkład bloków** – macierze z topologią 3D siatki wykazują charakterystyczny wzór kompresji wzdłuż diagonali,
- **Bloki off-diagonal** – zazwyczaj lepiej podatne na kompresję niż bloki diagonalne ze względu na mniejszą korelację między odległymi wierzchołkami.

5 Wnioski

5.1 Efektywność kompresji

Hierarchiczna reprezentacja macierzy pozwala znacząco zredukować zużycie pamięci:

- Gęsta macierz $N \times N$ wymaga $\mathcal{O}(N^2)$ pamięci,
- H-macierz z rzędem r wymaga około $\mathcal{O}(Nr \log N)$ pamięci.
- Dla $N = 4096$ i $r = 8$ redukcja pamięci wynosi:

$$\text{Współczynnik kompresji} = \frac{N^2}{Nr \log_2 N} = \frac{4096}{8 \cdot 12} \approx 43 \tag{5.1}$$

5.2 Dokładność aproksymacji

Na podstawie wyników eksperymentalnych:

- Błędy mnożenia macierz-wektor są rzędu `[WARTOŚĆ]`,
- Błędy mnożenia macierz-macierz rosną wraz z rozmiarem ze względu na kumulację błędów w rekurencji,
- Tolerancja $\epsilon = 10^{-6}$ zapewnia `[OPIS JAKOŚCI APROKSYMACJI]`.

5.3 Perspektywy rozwoju

Możliwe kierunki dalszych badań:

1. Implementacja adaptacyjnego wyboru rzędu aproksymacji r w zależności od lokalnej złożoności macierzy,
2. Paralelizacja algorytmów mnożenia (struktura drzewa naturalne wspiera równoległość),
3. Zastosowanie alternatywnych metod kompresji (np. Adaptive Cross Approximation),
4. Implementacja innych operacji algebraicznych (faktoryzacja LU, odwracanie),
5. Rozszerzenie wizualizacji o interaktywne narzędzia analizy struktury,
6. Zastosowanie H-macierzy do kompresji obrazów wielospektralnych (rozszerzenie Lab 3).

5.4 Uwagi implementacyjne

Integracja z Laboratorium 3 Ponowne wykorzystanie kodu z Lab 3 (kompresja obrazów) znacząco ułatwiło implementację:

- Funkcje SVD i power iteration były już gotowe i przetestowane,
- Struktura `TreeNode` okazała się kompatybilna z `HNode`,
- Funkcje wizualizacji działały bez modyfikacji,
- Biblioteki STB Image umożliwiły łatwy eksport do formatów graficznych.

Narzędzia pomocnicze Opracowano program demonstracyjny `visualize_example.cpp`, który:

- Ilustruje trzy metody wizualizacji,
- Generuje pliki PNG dla wszystkich metod,
- Wyświetla fragment wizualizacji w terminalu (znaki Unicode),
- Może służyć jako szablon dla użytkowników biblioteki.

Podsumowanie

W ramach laboratorium zaimplementowano i przebadano wydajność hierarchicznej reprezentacji macierzy wraz z operacjami mnożenia macierz-wektor oraz macierz-macierz. Wyniki eksperymentalne potwierdzają teoretyczne przewidywania dotyczące złożoności obliczeniowej:

- $\mathcal{O}(N \log N)$ dla mnożenia przez wektor,
- $\mathcal{O}(N^2)$ dla mnożenia macierzy.

H-macierze stanowią efektywne narzędzie do reprezentacji i operowania na dużych macierzach o specyficznej strukturze, szczególnie użyteczne w kontekście numerycznego rozwiązywania równań różniczkowych cząstkowych metodami brzegowymi (BEM) oraz metodą elementów skończonych (FEM).

Literatura

[1] W. Hackbusch, *A Sparse Matrix Arithmetic Based on H-Matrices. Part I: Introduction to H-Matrices*, Computing, vol. 62, pp. 89–108, 1999.

[2] M. Bebendorf, *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*, Springer, 2008.

[3] N. Halko, P. G. Martinsson, J. A. Tropp, *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, SIAM Review, vol. 53, no. 2, pp. 217–288, 2011.

[4] Sean Barrett, *stb_image.h: Public domain image loader*, https://github.com/nothings/stb, 2023. *Używane do wizualizacji struktury H-macierzy.*