ALGORYTMY PERMUTACJI MACIERZY

JAN SMÓŁKA

Abstract. Niniejszy dokument zawiera sprawozdanie z wykonania ćwiczenia 4. w ramach przedmiotu Algorytmy Macierzowe w semestrze zimowym roku akademickiego 2023/24.

1. Algorytmy

W ramach zadania zaimplementowano trzy algorytmy optymalizacji permutacji macierzy w różnych metrykach - algorytm Minimal Degree, Cuthill-McKee oraz Reversed Cuthill-McKee. Wszystkie metody bazują na podejściu grafowym - macierze rzeczywiste są traktowane jako macierze sąsiedztwa grafów nieskierowanych. Taka interpretacja narzuca pewne ograniczenia na postaci macierzy - przede wszystkim wymusza symetryczność.

1.1. **Algorytm Minimal Degree.** Algorytm polega na zachłannym szeregowaniu wierzchołków w kolejności zadanej stopniem wierzchołka - od najmniejszej liczby są siadów do największej. Istotnym krokiem algorytmu jest usuwanie z grafu wierzchołka wybranego w bieżącej iteracji. W konsekwencji, porządkowanie jest związane z topologią grafu i różni się od statycznego posortowania bez usuwania.

Algorithm 1 Permutacja Minimal Degree

Require: $A \in M_{n\times n}(\mathbb{R}) \wedge n \in \mathbb{N}_+ \wedge A = A^T$
  function ORDER($A, rows, cols, rank, tolerance$)
    $p \leftarrow [1, ..., n]$
    $V \leftarrow graph(A)$

    for $i \leftarrow 1, ..., n$ **do**
      $next \leftarrow argmin\{degree(v) : v \in V\}$
      $p[i] \leftarrow next$
      delete(next, V)
    end for
    return p
  end function

Dla prostoty konstrukcję reprezentacji grafu z macierzy wejściowej zapisano jako wywołanie funkcji *graph*. Jej przykładowa implementacja jest przedstawiona na ilustracji 3.

1.2. **Algorytm Cuthill-McKee.** Metoda polega na optymalizacji *rozpiętości* (ang. bandwidth) grafu poprzez sukcesywne przeszukiwanie BFS z sortowaniem sąsiedztwa po rosnących stopniach.

Algorithm 2 Permutacja Cuthill-McKee

Require: $A \in M_{n\times n}(\mathbb{R}) \wedge n \in \mathbb{N}_+ \wedge A = A^T$
```
function ORDER(A, rows, cols, rank, tolerance)
    p ← []
    V ← graph(A)
    Q ← Deque()
    visited ← [false, ..., false] // n elements

    for i ← sort(V.nodes) do
        if visited[i] then
            continue
        end if

        Q.enqueue(i)

        while not Q.empty do
            current ← Q.dequeue()

            if visited[current] then
                continue
            end if

            visited[current] = true
            p.append(current)

            for neighbour ← sort(V.neighbours(current)) do
                if not visited[neighbour] then
                    Q.enqueue(neighbour)
                end if
            end for
        end while
    end for

    return p
end function
```

W algorytmie wykorzystano strukturę danych *talii* (ang. *deque*) - dwustronnej kolejki umożliwiającej dodawanie i usuwanie elementów na obu końcach w zamortyzowanym czasie stałym. Funkcja *sort* sortuje podane wierzchołki po ich stopniach w kolejności rosnącej.

```julia
const Permutation = Vector{UInt}

abstract type Method end

struct MinimalDegree <: Method end
struct CuthillMcKee <: Method end
struct ReversedCuthillMcKee <: Method end
```

FIGURE 1. Pomocnicze typy używane w implementacji

```julia
function permute!(mat::Matrix{T}, method::Method)::Matrix{T} where {T <: Number}
    permute!(
        mat,
        permutation(mat, method)
    )
end

function permute!(mat::Matrix{T}, p::Permutation)::Matrix{T} where {T <: Number}
    permute!.(eachcol(mat), [p])
    permute!.(eachrow(mat), [p])

    mat
end
```

FIGURE 2. Główne do obliczania i wykonywania permutacji

1.3. **Odwrócony algorytm Cuthill-McKee.** Nie różni się znacząco od bazowej wersji - polega na zapisaniu uzyskanej w algorytmie Cuthill-McKee permutacji od końca.

2. Implementacja

Przedstawione w pseudokodzie algorytmy wraz z narzędziami służącymi do prezentacji wyników zaimplementowano w języku Julia.

3. Wyniki działania

Algorytmy przetestowano na macierzach reprezentujących współczynniki w układzie równań liniowych metody elementów skończonych na siatce trójwymiarowej. Przyjęto maksymalny rząd aproksymacji SVD $k = 3$ oraz tolerancję dla wartości osobliwej $\epsilon = 0.1$.

3.1. **Interesująca obserwacja.** W czasie implementacji algorytmów permutacji, odkryłem ciekawy wzorzec - gdy w macierzy współczynników FEM wyznaczy się optymalną permutację Cuthill-McKee i zamieni kolejność wierszy, nie zmieiając kolumn, powstaje piękny kształt przedstawiony na ilustracji 9.

3.2. **Generowanie macierzy testowych.** Macierz współczynników FEM na siatce trójwymiarowej można otrzymać stosując funkcję przedstawioną na ilustracji 8.

```julia
const Neighbourhood = Dict{UInt, Set{UInt}}

function neighbourhood(mat::Matrix{<:Number})::Neighbourhood
    n = size(mat, 1)

    indicator = mat .!= zero(eltype(mat))

    nonzero = indicator |>
        eachrow .|>
        findall .|>
        Set

    Dict(1:n .=> nonzero)
end

function min_degree(nb::Neighbourhood)::UInt
    argmin(
        length ∘ last,
        nb
    ).first
end

function drop!(nb::Neighbourhood, i::UInt)
    pop!(nb, i)

    for set in values(nb)
        i in set && pop!(set, i)
    end
end
```

FIGURE 3. Funkcje pomocnicze dla algorytmu Minimal Degree, zarządzające reprezentacją grafu

3.3. **Rozmiary H-macierzy.** Dla każdego wariantu kompresji wykonany został pomiar rozmiaru struktury H-macierzy w pamięci. Wyniki obrazuje ilustracja 18.

3.4. **Analiza wyników.**

(1) Permutacje Minimal Degree z reguły pogarszają jakość kompresji
(2) Skuteczność prostej i odwróconej metody Cuthill-McKee również jest dyskusyjna. Jakość kompresji ulega poprawie jedynie w narożnikach macierzy - niezerowe komponenty ulegają kondensacji
(3) Macierz współczynników FEM ze względu na swoją topologię jest słabo podatna na kompresję hierarchiczną

```julia
function permutation(mat::Matrix{<:Number}, ::MinimalDegree)::Permutation
    n = size(mat, 1)
    permutation = Vector{UInt}(undef, n)

    nb = neighbourhood(mat)

    for i in 1:n
        next = min_degree(nb)
        permutation[i] = next
        drop!(nb, next)
    end

    permutation
end
```

FIGURE 4. Funkcja realizująca algorytm Minimal Degree

```julia
function indicator(mat::Matrix{<:Number})::BitMatrix
    ind = mat .≠ zero(eltype(mat))

    for i in 1:size(mat, 1)
        ind[i, i] = false
    end

    ind
end

function adjacency(mat::Matrix{<:Number})::Vector{Vector{UInt}}
    mat ▷
    indicator ▷
    eachrow .▷
    findall
end
```

FIGURE 5. Funkcje pomocnicze dla algorytmu Cuthill-McKee, służące do wyznaczania reprezentacji grafu

```julia
function permutation(mat::Matrix{<:Number}, ::CuthillMcKee)::Permutation
    n = size(mat, 1)

    visited = falses(n)
    adj = adjacency(mat)
    degrees = length.(adj)

    perm = Permutation()
    deque = Deque{UInt}()

    descending(vertices::Vector{UInt})::Vector{UInt} =
        sort(vertices, by = i -> degrees[i])

    bfs!(start::UInt)::Nothing = begin
        push!(deque, start)

        while length(deque) > 0
            curr = popfirst!(deque)

            visited[curr] && continue
            visited[curr] = true

            push!(perm, curr)

            for neigh in descending(adj[curr])
                !visited[neigh] && push!(deque, neigh)
            end
        end

        nothing
    end

    for vertex in descending(UInt.(1:n))
        !visited[vertex] && bfs!(vertex)
    end

    perm
end
```

FIGURE 6. Implementacja algorytmu Cuthill-McKee

```julia
function permutation(mat::Matrix{<:Number}, ::ReversedCuthillMcKee)::Permutation
    permutation(mat, CuthillMcKee()) |> reverse
end
```

FIGURE 7. Implementacja odwróconego algorytmu Cuthill-McKee

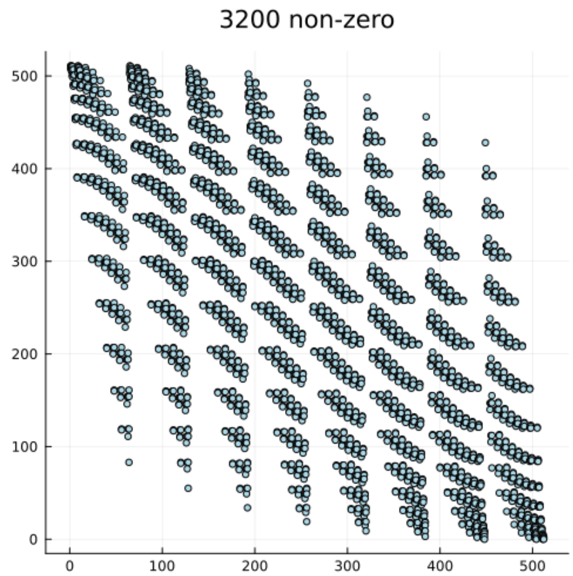
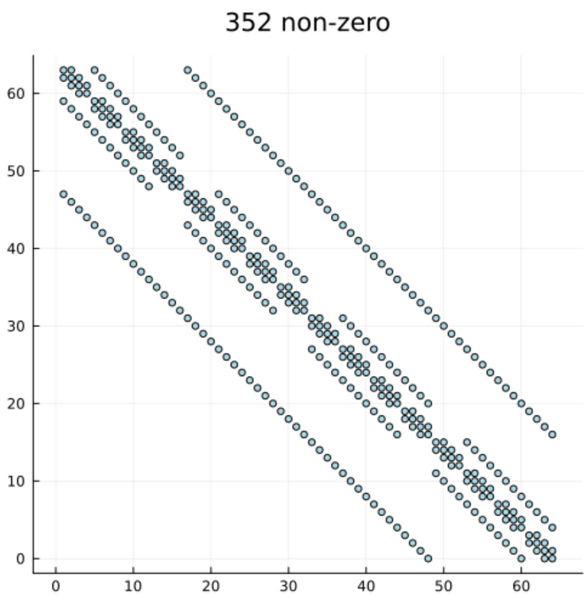


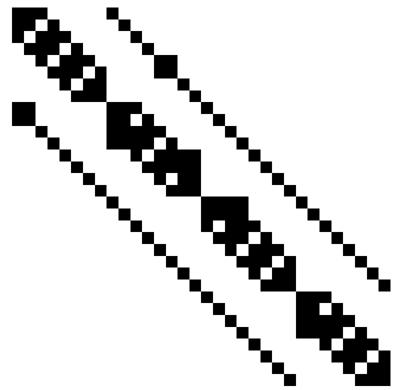
FIGURE 8. Ciekawy wzorzec rzadkości w niepoprawnie spermutowanej macierzy 512x512

```
function fem_3d(k::Int)::Matrix{Float64}
    n = 2 ^ k
    n_2 = n ^ 2
    n_3 = n ^ 3

    matrix = zeros(Float64, n_3, n_3)

    assign!(row::Int, col::Int) = (matrix[row, col] = rand())

    for i in 1:n_3
        level = (i - 1) ÷ n_2
        remainder = (i - 1) % n_2
        row = remainder ÷ n
        col = remainder % n

        assign!(i, i)

        level > 0 && assign!(i, i - n_2)
        level < n - 1 && assign!(i, i + n_2)

        row > 0 && assign!(i, i - n)
        row < n - 1 && assign!(i, i + n)

        col > 0 && assign!(i, i - 1)
        col < n - 1 && assign!(i, i + 1)
    end

    matrix
end
```

FIGURE 9. Funkcja generująca macierze

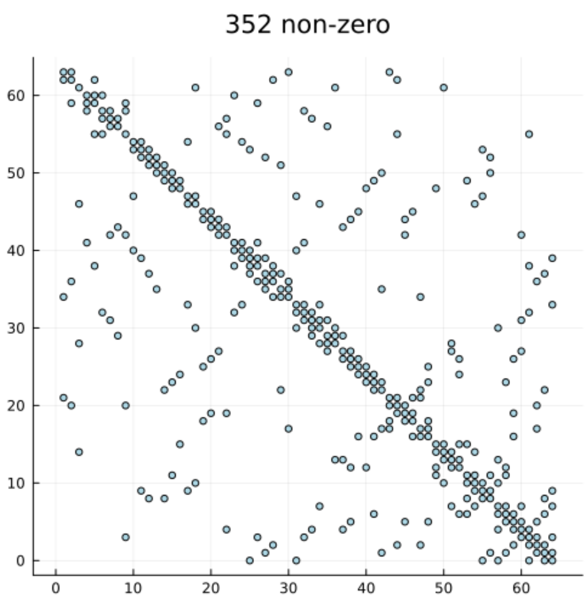


(A) Wzorzec rzadkości

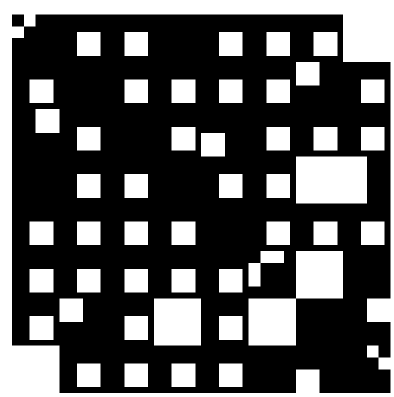


(B) H-macierz

FIGURE 10. Macierz FEM o rozmiarze 64x64

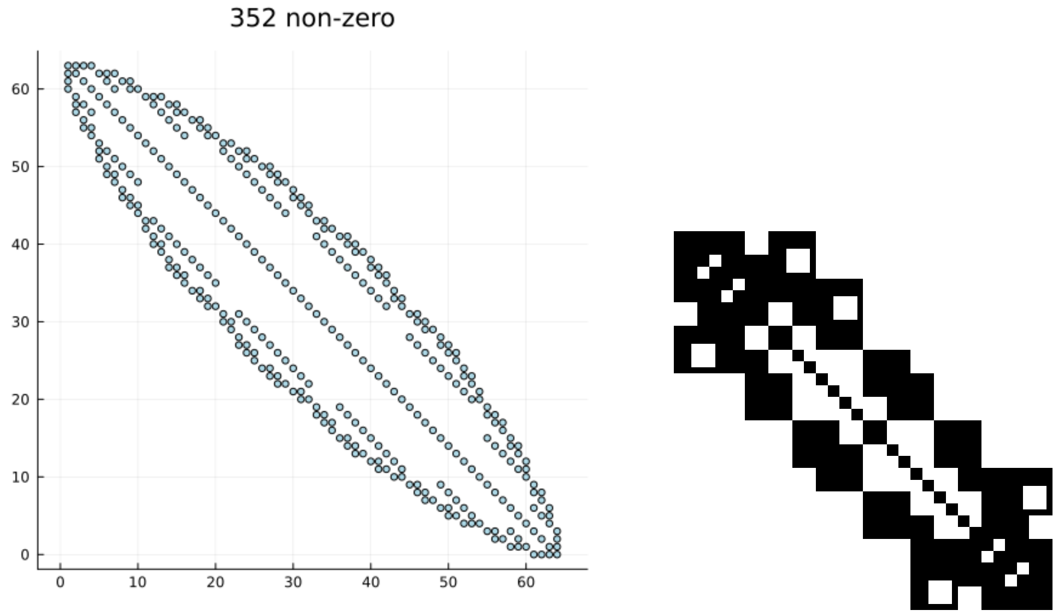


(A) Wzorzec rzadkości



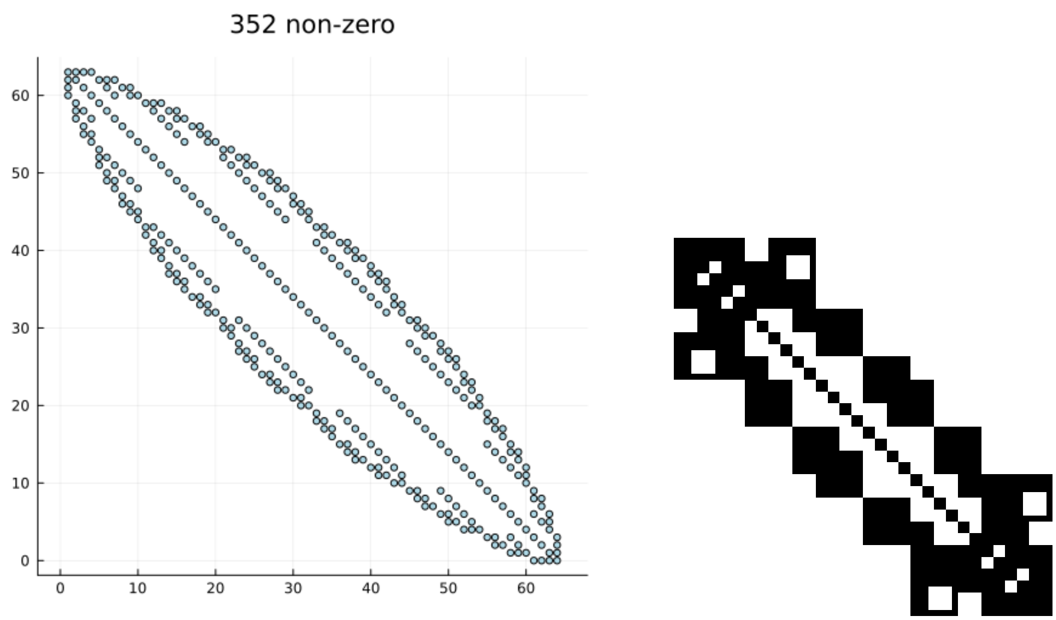
(B) H-macierz

FIGURE 11. FEM 64x64 - Minimal Degree



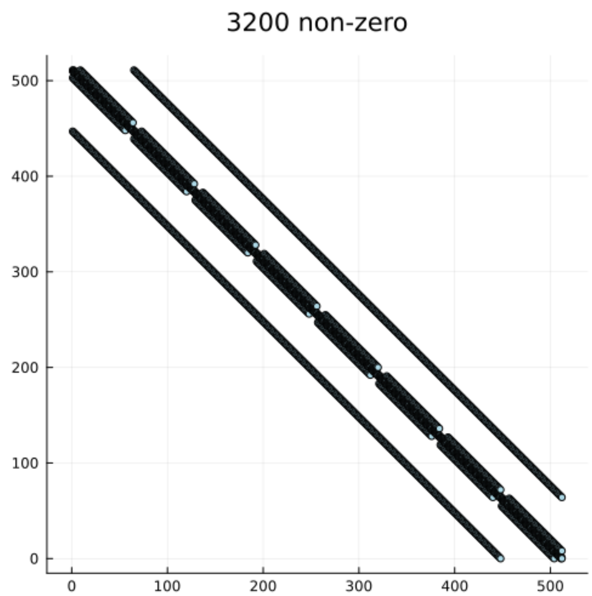
(A) Wzorzec rzadkości (B) H-macierz

Figure 12. FEM 64x64 - Cuthhill-McKee

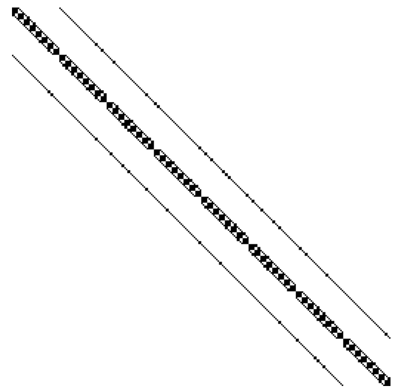


(A) Wzorzec rzadkości (B) H-macierz

Figure 13. FEM 64x64 - Reversed Cuthhill-McKee

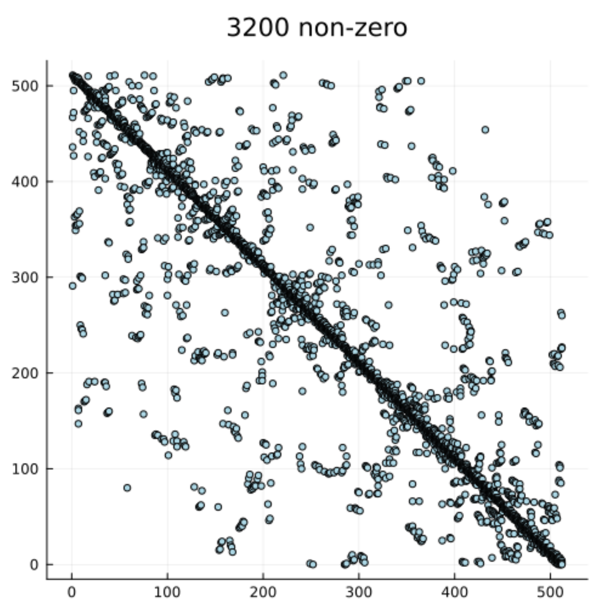


(A) Wzorzec rzadkości

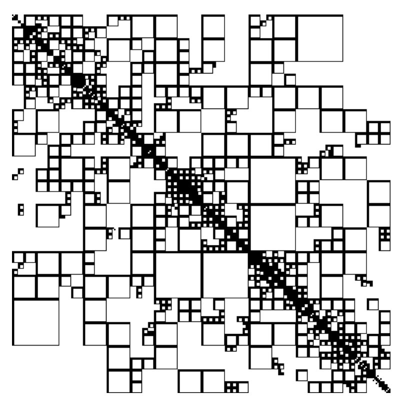


(B) H-macierz

FIGURE 14. Macierz FEM o rozmiarze 512x512

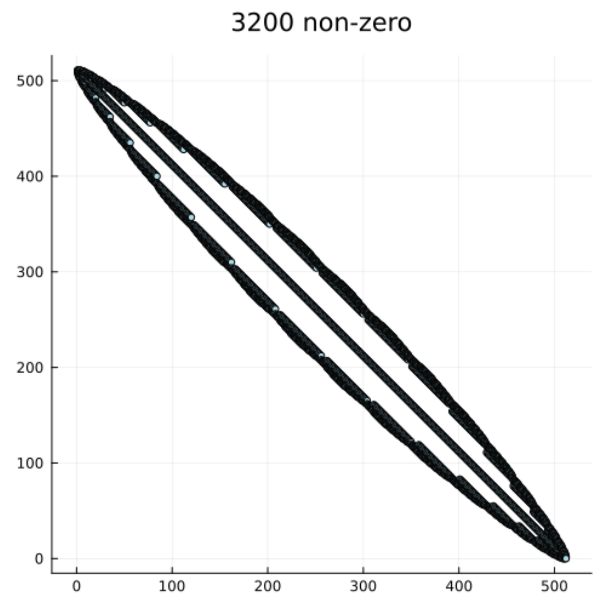


(A) Wzorzec rzadkości

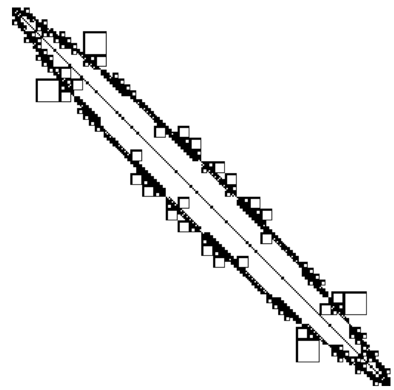


(B) H-macierz

FIGURE 15. FEM 512x512 - Minimal Degree

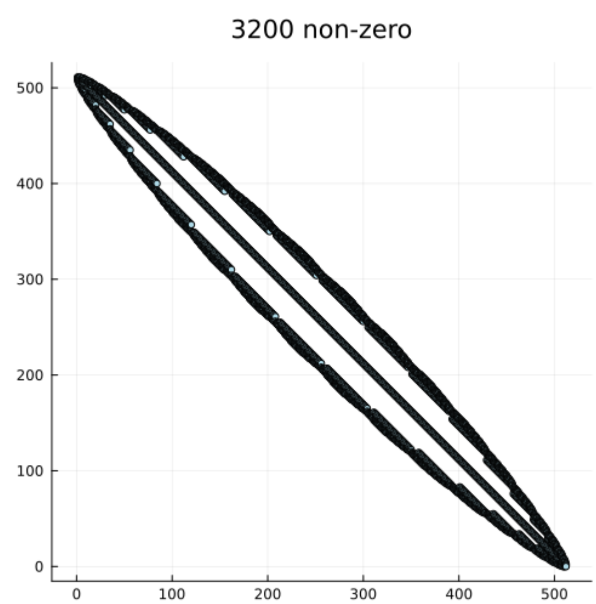


(A) Wzorzec rzadkości

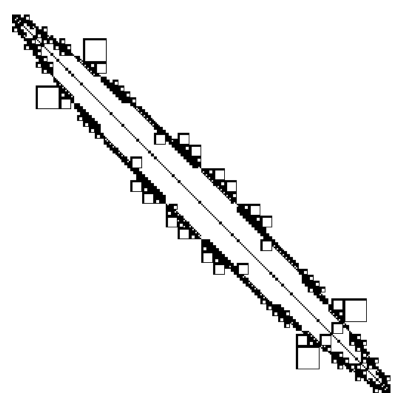


(B) H-macierz

Figure 16. FEM 512x512 - Cuthhill-McKee



(A) Wzorzec rzadkości



(B) H-macierz

Figure 17. FEM 512x512 - Reversed Cuthhill-McKee

Row	matrix_size Int64	base Int64	minimal_degree Int64	cuthill_mckee Int64	reversed_cuthill_mckee Int64
1	64	39968	60800	46760	46760
2	512	439328	780728	512648	512648
3	4096	4064168	9666728	4835240	4835240

Figure 18. Rozmiary H-macierzy w bajtach; *base* - rozmiar bez permutacji, *matrix_ size* - liczba wierszy/kolumn macierzy