ALGORYTMY ODWRACANIA, LU-FAKTORYZACJI, OBLICZANIA WYZNACZNIKA MACIERZY

JAN SMÓŁKA

Abstract. Niniejszy dokument zawiera sprawozdanie z wykonania ćwiczenia 2. w ramach przedmiotu Algorytmy Macierzowe w semestrze zimowym roku akademickiego 2023/24.

1. Algorytmy

W ramach zadania zaimplementowano trzy algorytmy: LU-faktoryzacji (w wariancie LUP), obliczania wyznacznika oraz odwracania macierzy. Każda metoda sformułowana jest rekurencyjnie, jednak z ponieważ pierwsze dwie z nich są wysoce podatne na derekursywację, zostały zaimplementowane iteracyjnie. Również z przyczyn implementacyjnych, procedura odwracania macierzy działa poprawnie przy założeniu, że na wejściu podano macierz kwadratową o rozmiarze będącym potęgą dwójki.

1.1. **Faktoryzacja LUP.** Algorytm rekurencyjny operuje na macierzach kwadratowych o dowolnym rozmiarze. Komponenty L i U są przechowywane w nadpisanej macierzy A, zaś permutacja wierszy skonstruowana w procesie pivotingu - w osobnym komponencie P. *Pivot* może być wybierany na wiele sposobów. W implementacji przyjęto, że jest to element o największej wartości bezwzględnej w danej kolumnie.

Algorithm 1 Faktoryzacja LUP macierzy kwadratowej

Require: $A \in M_{n\times n} \land n \in \mathbb{N}_+$
Ensure: $det(A) \neq 0 \land P = [1, 2, ..., n]$
  for $i \leftarrow 1...n-1$ **do**
    $pivot \leftarrow find\_pivot(A, i)$

    $swap(P, i, pivot)$
    $swap\_rows(A, i, pivot)$

    $divide\_by\_pivot(A[i+1...n, i])$
    $subtract\_schur\_complement(A[i...n, i...n])$
  end for

1.2. **Obliczanie wyznacznika.** Przytoczony algorytm obliczania wyznacznika polega na przeprowadzeniu eliminacji Gaussa i wzięciu iloczynu elementów diagonalnych, z dokładnością do znaku, ustalonego na podstawie liczby zamian wierszy dokonanych w procesie *pivotingu*.

Algorithm 2 Obliczanie wyznacznika macierzy

Require: $A \in M_{n \times n} \wedge n \in \mathbb{N}$
Ensure: $swaps = 0 \wedge result = 1$
  for $i \leftarrow 1...n$ **do**
    $pivot, pivot\_row \leftarrow find\_pivot(A, i)$
    $result \leftarrow result \cdot pivot$

    if $pivot\_row \neq i$ **then**
      $swap\_rows(A, i, pivot\_row)$
      $swaps \leftarrow swaps + 1$
    end if

    $divide\_row(A, i)$
    $reduce\_column(A, i)$
  end forreturn $result \cdot (-1)^{swaps}$

Dla prostoty zapisu algorytmu, w pseudokodzie występują procedury *divide_row* i *reduce_column*. Reprezentują one standardowe kroki w eliminacji Gaussa - podzielenie bieżącego wiersza przez *pivot* oraz wyzerowanie elementów poniżej niego.

1.3. **Wyznaczanie macierzy odwrotnej.** Wybrany algorytm odwracania macierzy jest identyczny z zaprezentowanym na wykładzie, dlatego przedstawienie sprowadza się do przytoczenia wzorów:

$$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}^{-1} \tag{1.1}$$

$$S_{22} = A_{22} \cdot A_{11}^{-1} \cdot A_{12} \tag{1.2}$$

$$B_{11} = A_{11}^{-1} \cdot (I + A_{12} \cdot S_{22}^{-1} * A_{21} * A_{11}^{-1}) \tag{1.3}$$

$$B_{12} = -A_{11}^{-1} \cdot A_{12} \cdot S_{22}^{-1} \tag{1.4}$$

$$B_{21} = -S_{22}^{-1} \cdot A_{21} \cdot A_{11}^{-1} \tag{1.5}$$

$$B_{22} = S_{22}^{-1} \tag{1.6}$$

```julia
@assume_effects :total !:nothrow function lup(matrix::Matrix{Float64}; decompose::Bool = false)::LUP
    result = decompose ?
        DecomposedLUP(matrix) :
        InplaceLUP(matrix)

    n = result.size

    for col ∈ 1:n-1
        pivot_row, pivot = get_pivot(result.factorized, col)

        pivot ≈ 0.0 && throw(SingularException(col))

        if pivot_row ≠ col
            swap!(result.p, col, pivot_row)
            swap_rows!(result.factorized, col, pivot_row)
        end

        scale_column!(result.factorized, col)
        subtract_schur_complement!(result.factorized, col)
    end

    decompose && fill_triangles!(result)

    return result
end
```

FIGURE 1. Funkcja realizująca faktoryzację

2. Implementacja

W implementacjach dużą uwagę poświęcono optymalizacji. Procedury działają sekwencyjnie, jednak szeroko wykorzystują wektoryzację sprzętową SIMD oraz optymalizacje polihedralne dla zagnieżdżonych pętli. Wszystkie działania zmiennoprzecinkowe wykonywane na potrzeby algorytmu odbywają się w standardzie *fastmath*, w celu wykorzystania potencjalnych sprzętowych optymalizacji zależnych od platformy. Język Julia posiada domyślnie kontrolę indeksowania - odwołanie do tablicy poza jej granicami skutkuje wyjątkiem. W celu przyspieszenia działania procedur, kontrola została wyłączona.

2.1. **LU-Faktoryzacja.** Implementacja bezpośrednio odwzorowuje pseudokod. Wywołując główną funkcję można wybrać, czy wynik ma być przechowywany w postaci skompresowanej w nadpisanej macierzy wyjściowej, czy w formie dwóch osobnych macierzy trójkątnych. Dodatkowo zawsze zwracany jest wektor z zapisaną permutacją wierszy.

2.2. **Obliczanie wyznacznika.** W celu dodatkowej optymalizacji czasu działania, po wykryciu osobliwości macierzy (*pivot* równy 0), funkcja zwraca 0. Procedura wykonuje jawną kopię wyjściowej macierzy, w celu nienadpisywania nielokalnej pamięci i zachowania funkcyjnej czystości, co umożliwia dodatkowe optymalizacje przez kompilator.

2.3. **Odwracanie macierzy.** Implementacja ma postać funkcji rekurencyjnej. Działa na kwadratowych macierzach klatkowych o rozmiarach będących potęgami

```julia
function get_pivot(matrix::Matrix{Float64}, col::Int)::Tuple{Int, Float64}
    n = size(matrix, 1)
    @inbounds pivot = abs(matrix[col, col])
    pivot_row = col

    @simd for row in col+1:n
        @inbounds element_abs = abs(matrix[row, col])

        if element_abs > pivot
            pivot = element_abs
            pivot_row = row
        end
    end

    return (pivot_row, pivot)
end
```

FIGURE 2. Funkcja odpowiedzialna za wybór *pivota* w algorytmie LU-faktoryzacji

```julia
function scale_column!(matrix::Matrix{Float64}, col::Int)
    inv_element = inv(matrix[col, col])
    n = size(matrix, 1)

    @simd for i in col+1:n
        @inbounds @fastmath matrix[i, col] *= inv_element
    end
end

@polly function subtract_schur_complement!(matrix::Matrix{Float64}, col::Int)
    n = size(matrix, 1)

    for j in col+1:n
        @simd for i in col+1:n
            @inbounds @fastmath matrix[i, j] -= matrix[i, col] * matrix[col, j]
        end
    end
end
```

FIGURE 3. Funkcje pomocnicze w algorytmie LU-faktoryzacji

dwójki. Zamiera optymalizacje dotyczące wykorzystania pamięci - poszczególne klatki macierzy odwrotnej obliczane są w kolejności wymagającej stworzenia kopii tylko jednej klatki macierzy wyjściowej.

```julia
@assume_effects :total function det(matrix::Matrix{Float64})::Float64
    matrix = copy(matrix)
    n = size(matrix, 1)
    n_rows_permuted = 0
    result = 1.0

    for row_col in 1:n
        pivot_row, pivot = get_pivot(matrix, row_col)

        pivot ≈ 0.0 && return 0.0

        @fastmath result *= pivot

        if pivot_row ≠ row_col
            n_rows_permuted += 1
        end

        swap_rows!(matrix, row_col, pivot_row)

        divide_row!(matrix, row_col)
        reduce_column!(matrix, row_col)
    end

    sgn = isodd(n_rows_permuted) ? -1.0 : 1.0

    return @fastmath result * sgn
end
```

FIGURE 4. Funkcja realizująca algorytm obliczania wyznacznika

```julia
function get_pivot(matrix::Matrix{Float64}, col::Int)::Tuple{Int, Float64}
    n = size(matrix, 1)
    @inbounds pivot = abs(matrix[col, col])
    pivot_row = col

    @simd for row in col+1:n
        @inbounds element_abs = abs(matrix[row, col])

        if element_abs > pivot
            pivot = element_abs
            pivot_row = row
        end
    end

    return (pivot_row, pivot)
end
```

FIGURE 5. Funkcja odpowiedzialna za wybór *pivota* w algorytmie obliczania wyznacznika

```julia
function divide_row!(matrix::Matrix{Float64}, row::Int)::Nothing
    @inbounds inv_pivot = inv(matrix[row, row])
    n = size(matrix, 1)

    @simd for i in row+1:n
        @inbounds @fastmath matrix[row, i] *= inv_pivot
    end

    nothing
end

@polly function reduce_column!(matrix::Matrix{Float64}, col::Int)::Nothing
    n = size(matrix, 1)

    @inbounds for i in col+1:n
        element = matrix[i, col]
        @simd for j in 1:n
            @fastmath matrix[i, j] -= element * matrix[col, i]
        end
    end

    nothing
end
```

FIGURE 6. Funkcje pomocnicze w algorytmie obliczania wyznacznika

3. Koszt obliczeniowy

3.1. **Eksperyment.** By przekonać się jaka jest zależność czasu obliczeń oraz liczby poszczególnych rodzajów operacji zmiennoprzecinkowych od rozmiaru macierzy, przeprowadzono testy dla następujących danych wejściowych: $A \in M_{2^k \times 2^k}(\mathbb{R})$, $k \in \{2, 3, ..., 14\}$. Z powodu ograniczonych możliwości sprzętowych nie było możliwe rozszerzenie zakresu potęg do 16. Uzyskane wyniki obrazują wykresy na rysunku 9-11.

3.2. **Pomiar czasu.** Czas wykonania mierzono za pomocą standardowego makra dostępnego w języku Julia - `@elapsed`. Mimo, że testowanie przebiegało współbieżnie, dokonane pomiary dotyczą fizycznych odczytów zegara procesora, z pominięciem kolejkowania, oczekiwania i przerwań systemowych.

3.3. **Zliczanie operacji arytmetycznych.** Dla każdej funkcji przeprowadzono obliczenie łącznej liczby operacji addytywnych (dodawania i odejmowania) i multiplikatywnych (mnożenia i dzielenia) zmiennoprzecinkowego we wszystkich precyzjach. Pominięto działania całkowitoliczbowe i hybrydowe, mogące wystąpić na pewnych procesorach, np. `muladd`. Użyto makra `@count_ops` z biblioteki *GFlops*.

3.4. **Oszacowanie złożoności asymptotycznej.** Do zebranych danych zastosowano sześcienną aproksymację wielomianową.

```julia
function inv!(matrix::MatrixOrView)::MatrixOrView
    n = size(matrix, 1)
    n_half = n ÷ 2

    if n ≤ 2
        return trivial_inv!(matrix)
    end

    upper = 1:n_half
    lower = n_half+1:n

    @inbounds @views begin
        m11 = matrix[upper, upper]
        m12 = matrix[upper, lower]
        m21 = matrix[lower, upper]
        m22 = matrix[lower, lower]
    end

    m11_inv = inv!(m11)

    @turbo @fastmath m22 .-= m21 * m11_inv * m12
    m22_inv = inv!(m22)

    let m11_inv_copy = copy(m11_inv)
        @turbo @fastmath m11_inv .= m11_inv_copy + m11_inv_copy * m12 * m22_inv * m21 * m11_inv_copy
        @turbo @fastmath m12 .= -m11_inv_copy * m12 * m22_inv
        @turbo @fastmath m21 .= -m22_inv * m21 * m11_inv_copy
    end

    return matrix
end
```

FIGURE 7. Funkcja realizująca algorytm odwracania macierzy

```julia
function trivial_inv!(matrix::MatrixOrView)::MatrixOrView
    @inbounds a, c, b, d = matrix
    @fastmath inv_det = Base.inv(a * d - b * c)

    @inbounds @fastmath begin
        matrix[1, 1] = d * inv_det
        matrix[1, 2] = -b * inv_det
        matrix[2, 1] = -c * inv_det
        matrix[2, 2] = a * inv_det
    end

    return matrix
end
```

FIGURE 8. Funkcja pomocnicza w algorytmie odwracania, obliczająca odwrotność macierzy 2×2

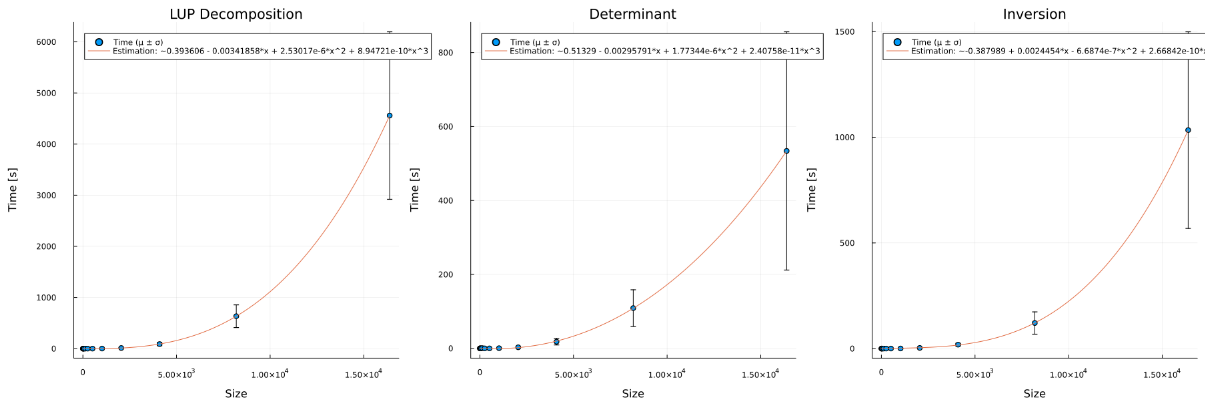


FIGURE 9. Czas działania w zależności od rozmiaru danych wejściowych, dla poszczególnych algorytmów wraz z aproksymacją wielomianową

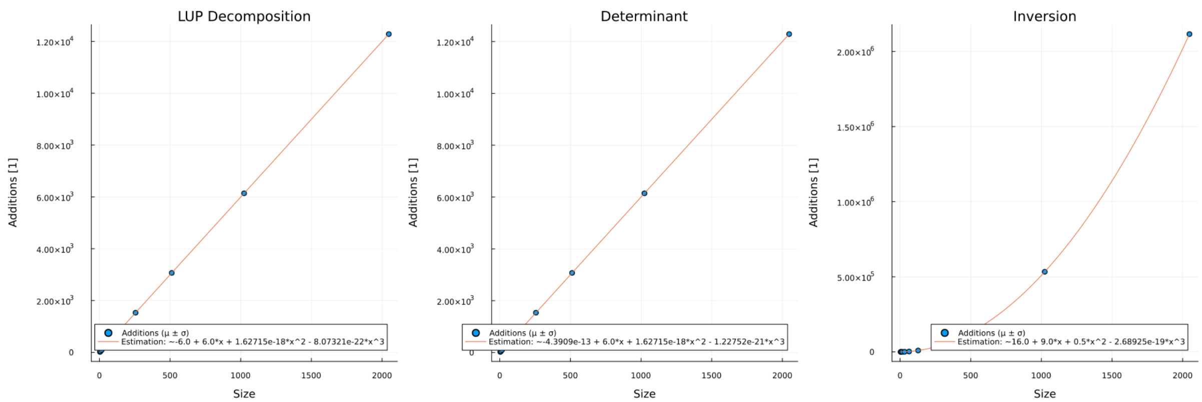


FIGURE 10. Liczba operacji addytywnych w zależności od rozmiaru danych wejściowych, dla poszczególnych algorytmów wraz z aproksymacją wielomianową

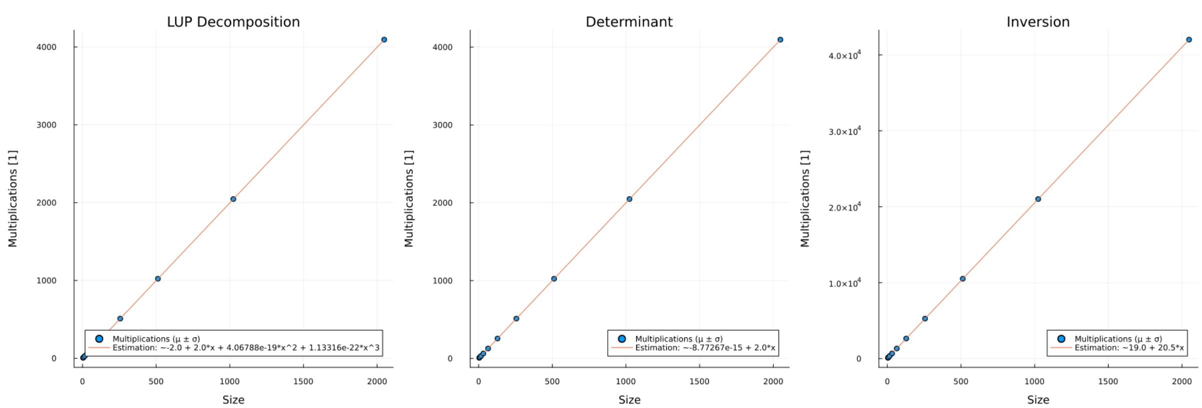
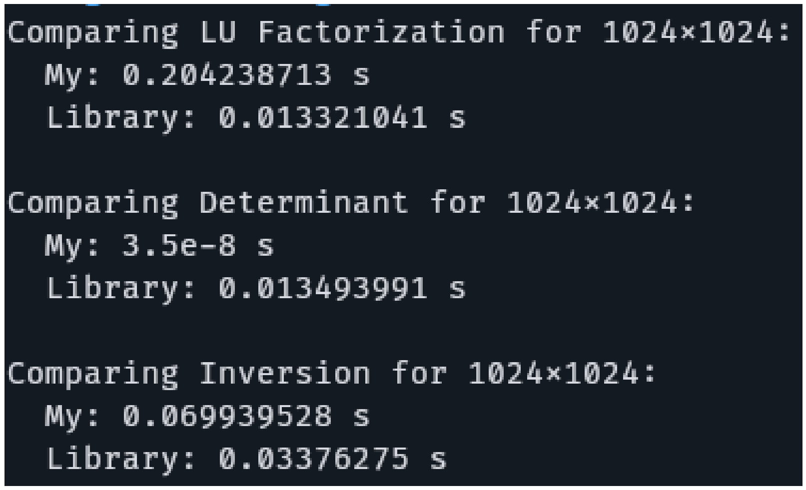


FIGURE 11. Liczba operacji multiplikatywnych w zależności od rozmiaru danych wejściowych, dla poszczególnych algorytmów wraz z aproksymacją wielomianową



```
Comparing LU Factorization for 1024×1024:
  My: 0.204238713 s
  Library: 0.013321041 s

Comparing Determinant for 1024×1024:
  My: 3.5e-8 s
  Library: 0.013493991 s

Comparing Inversion for 1024×1024:
  My: 0.069939528 s
  Library: 0.03376275 s
```

FIGURE 12. Porównanie czasu działania funkcji

Wnioski na temat wyniku oszacowania:

(1) **Jakość aproksymacji:** Ze względu na niewielką liczbę punktów, dopasowanie jest nie najwyższej jakości, jednak obrazuje ogólny trend.
(2) **Współczynniki wielomianów:** Jako etykiety na wykresie przedstawiono dopasowanie wielomiany. Pierwszy (w sensie malejącej kolejności potęg) niezerowy współczynnik można traktować jako estymatę stałej w *asymtpotycznej* złożoności.
(3) **Wiarygodność:** Oszacowania nie zgadzają się z teoretycznymi przewidywaniami na temat złożoności algorytmów. Należy oczekiwać, że dla każdego algorytmu co najmniej jedno z działań arytmetycznych wystąpi w liczbie zaleznej od danych wejściowych sześciennie. Potencjalne przyczyny rozbieżnych oszacowań to: **a)** zbyt mała liczba punktów interpolacji, **b)** pominięcie przez procedurę zliczającą operacje tych pętli w kodzie, które zostały w procesie kompilacji przekształcone przez SIMD lub optymalizator polihedralny.

3.5. **Porównanie z implementacją biblioteczną.** Obraz 12. przedstawia porównanie czasu działania implementacji poszczególnych algorytmów z odpowiadającymi im funkcjami z biblioteki *LinearAlgebra* w Julii.