# Sprawozdanie laboratorium lista 5
# Obliczenia naukowe

Zofia Tarchalska, indeks: 279699

## 1 Wstęp

Celem tej listy było zaimplementowanie trzech poniższych algorytmów o złożoności $O(n)$, gdzie $n$ to rozmiar macierzy $n \times n$.

- funkcja rozwiązująca układ $Ax = b$ metodą eliminacji Gaussa
- funkcja wyznaczająca rozkład $LU$ maciezry $A$ metodą eliminacji Gaussa
- funkcja rozwiązująca układ $Ax = b$ jeśli już wcześniej został wyznaczony rozkład $LU$

Wszystkie te algorytmy mają działać dla macierzy $A$ o specyficznej postaci, która została dokładnie opisana w poleceniu do zadania. Dodatkowo należało zaprogramować wersję podstawową oraz wersję z wybieraniem elementu głównego.

## 2 Metoda eliminacji Gaussa

Główną ideą metody eliminacji Gaussa jest doprowadzenie macierzy do postaci, w której pod przekątną znajdują się same zera. Uzyskujemy to poprzez mnożenie kolejnych wierszy macierzy przez odpowiednie czynniki i odejmowanie ich od wierszy następujących po nich. Poprawny schemat:
Eliminujemy zmienną $x_k$ z wierszy od $k+1$ do $n$. Mnożymy $k-te$ równanie przez

$$l_{ik} = \frac{a_{ik}}{a_{kk}} \quad \text{dla } i \in \{k+1, ..., n\}$$

Kiedy jednak na przekątnej w miejscu $a_{kk}$ będzie 0 metoda może powodować błąd numeryczny, ponieważ nastąpi dzielenie przez 0. W przeciwnym wypadku, po wykonaniu odpowiednich kroków kolejno dla wszystkich wierszy macierzy, otrzymujemy macierz górnotrójkątną.

### 2.1 Wariant pierwszy - bez wyboru elementu głównego

Ten wariant nie zabezpiecza nas przed opisanym powyżej błędem numerycznym.

### 2.1.1 Złożoność czasowa

W niezoptymalizowanej wersji algorytmu złożoność wynosiłaby $O(n^3)$, ponieważ mamy 3 pętle `for` przechodzące po elementach macierzy. Dla każdego wiersza w macierzy wyznaczamy czynnik przez który przemnażamy ten wiersz po czym odejmujemy go od kolejnych wierszy (musimy przejść po wszytskich elementach w danym wierszu, a macierz ma wymiary $n \times n$). Jednak wykorzystując blokową strukturę macierzy A da się sprowadzić ten algorytm do złożoności $O(n)$. Wiemy, że rozmiar pojedynczego bloku ma jakiś określony rozmiar. Oznaczmy go przez l. Zatem pod przekątną macierzy jest maksymalnie l elementów, które są niezerowe. Dodatkowo każdy wiersz, o indeksie większym niż 1 ma od lewej same zera, następnie l leżących po sobie niezerowych wartości i znów zera (chyba, że np. $k + l$ daje już indeks ostatniej kolumny, wtedy nie ma zar od końca). Możemy zatem wykonywać ten algorytm tylko dla wierszy i elementów, dla których ma to sens (oszczędzimy sobie przetwarzania ogromnej liczby samych zer). Możemy zatem odejmować tylko l wierszy i aktualizować w nich l wartości. Ponieważ l jest znacząco mniejsze niż n, złożoność spada do $O(n)$.

### 2.1.2 Złożoność pamięciowa

Zaimplementowana przeze mnie struktura `BlockMatrix` przechowuje naszą macierz blokową A. Korzysta przy tym ze `SparseArrays` dostępnych w języku Julia. `SparseArrays` nie przechowują niezerowych elementów, wobec tego, ze względu na budowę A, mamy tylko elementy zlokalizowane w otoczeniu przekątnej. Ponieważ jest to co najwyżej l elementów przypadających na każdy wiersz/kolumnę, których jest n, to ze względu, że l jest jakąś stałą, złożoność pamięciowa to $O(n)$.

### 2.1.3 Pseudokod

Każde wykonanaie pętli `for` jest ograniczone do pewnych indeksów, dla których występują niezerowe elementy. Dzięki temu dokonujemy optymalizowania opisanego we wcześniejszym paragrafie i ograniczamy nasze obliczenia tylko do tych elementów, które mają rzeczywisty wpływ na wynik. Dodatkowo korzystamy z własności SparseArrays.

- `nzval` - wektor wartości wszystkich niezerowych elementów.
- (colptr) - wektor wskaźników kolumn o długości n+1 (dla macierzy o n wierszach i n kolumnach). Dla kolumny $j$ niezerowe wpisy znajdują się na pozycjach $k = colptr[j] : colptr[j + 1] - 1$,
- `rowval` - wektor indeksów wierszy odpowiadających kolejnym elementom w `nzval`,

```
1: procedure GaussElimination(A, b)
2:     M ← A.matrix, n ← A.size[1]
3:     Zbuduj reprezentację wierszową: rows[1..n] jako mapy kolumna->wartość
4:     for col = 1 to M.n do
5:         for każdy niezerowy wpis (r, v) w kolumnie col do
6:             rows[r][col] = v
7:         end for
8:     end for
9:     ▷ Forward elimination (bez pivotingu), ograniczona do okna blokowego
10:    for k = 1 to n − 1 do
11:        if brak elementu diagonalnego rows[k][k] then
12:            error("Zero pivot")
13:        end if
14:        pivot ← rows[k][k]
15:        for i = k + 1 to min(n, k + A.block_size + 1) do
16:            a_ik ← get(rows[i], k, 0)
17:            if a_ik = 0 then continue
18:            end if
19:            m ← a_ik/pivot
20:            ▷ Zaktualizuj wiersz i: dla kolumn j > k wykonaj a_ij −= m · a_kj
21:            for każde (j, a_kj) w rows[k] z j > k do
22:                val ← get(rows[i], j, 0) − m · a_kj
23:                if |val| < tol then delete rows[i][j]
24:                else rows[i][j]=val
25:                end if
26:            end for
27:            delete rows[i][k]      ▷ eliminowany element poniżej przekątnej
28:            b[i] −= m · b[k]              ▷ aktualizacja wektora prawej strony
29:        end for
30:    end for
31:                                          ▷ Back substitution: rozwiąż Ux = b
32:    x ← wektor zer długości n
33:    if brak rows[n][n] then error("Zero diagonal")
34:    end if
35:    x[n] ← b[n]/rows[n][n]
36:    for i = n − 1 downto 1 do
37:        s ← b[i]
38:        for each (j, a_ij) w rows[i] z j > i do
39:            s −= a_ij · x[j]
40:        end for
41:        if brak rows[i][i] then error("Zero diagonal")
42:        end if
43:        x[i] ← s/rows[i][i]
44:    end for
45:    return x
46: end procedure
```

## 2.2 Wariant drugi - z częściowym wyborem elementu głównego

Ten wariant ma zabezpieczać nas przed potencjalnym dzieleniem przez 0 gdy na przekątnej macierzy taka warotść się znajduje. Biorąc pod uwagę, że obliczenia wykonujemy na komputerze, liczby bardzo zbliżone do 0 również są tymi, które mogą nam zwrócić błąd.
Element główny to nic innego jak wybrana wartość, którą będziemy używać aby wyzerować pozostałe w kolumnie. Wybieramy ją poprzez znaleznienie największej wartości w kolumnie (patrząc na jej moduł). Następnie wyznaczamy czynnik, przez który będziemy mnożyć kolejne wiersze, korzystając już z tej największej wartości. Zapamiętujemy, które wiersze zostały zamienione za pomocą tablicy permutacji. Dzięki temu nie modyfikujemy macierzy A, a odwołując się do konkretnego wiersza zamiast używać jego indeksu wprost, używamy `p[indeks]`.

### 2.2.1 Złożoność czasowa

Algorytm z wyborem elementu głównego jest bardzo podobny do zwykłej eliminacji Gaussa. Jedyna różnica polega na tym, że w pierwszej pętli `for` gdy przechodzimy po kolumnach wybieramy największą wartość w danej kolumnie. Następnie wykonujemy, tak samo, resztę algorytmu na zmienionej kolejności wierszy. Ta jedna dodatkowa pętla nawet w niezoptymalizowanej wersji nie zwiększyłaby złożoności algorytmu. Wobec tego pozostaje, identycznie jak poprzednio, złożność czasowa równa $O(n)$.

### 2.2.2 Złożoność pamięciowa

W tym algorytmie mamy dodatkową tablicę permutacji. Jest ona rozmiaru $n$, który odpowiada liczbie wierszy w macierzy. Poza nią wszystko jest przechowywane tak samo jak poprzednio. Czyli, wszystkie modyfikacje wykonywane są in-place na macierzy A. Jak już wspomniałam wcześniej tablica permutacji ma n elementów, a więc nie zmienia to rzędu pamięci. Wobec tego złożoność pamięciowa to dalej $O(n)$.

### 2.2.3 Pseudokod

Robimy dokładnie to samo co w poprzednim przypadku. Jedynie po pierwszej pętli `for` dla każdej kolumny wybieramy element o największej co do wartości bezwzględnej wartości i wybieramy go jako element główny. Permutacje wierszy przechowujemy w tablicy `p`, którą potem wykorzystujemy. Odwołując się do jakiegokolwiek wiersza `i` używamy po prostu `p[i]`. Poniżej sam schemat częściowego wyboru elementu głównego.

```
pivot ← k
maxval ← | get(rows[p[k]], k, 0) |
for r = k + 1 to bound do
    val ← | get(rows[p[r]], k, 0) |
    if val > maxval then
        maxval ← val
        pivot ← r
    end if
end for
if pivot ≠ k then
    swap p[k] and p[pivot]                ▷ aktualizacja wektora permutacji
end if
return p
```

# 3 Rozkład LU

Rozkład LU to taki podział, w którym A = LU. Czyli zamieniamy macierz A na dwie macierze trójkątne. L jest macierzą dolnotrójkątną, a U górnotrójkątną. L zawiera mnożniki $l_{ik}$ w miejscu zerowanych elementów.

$$L = \begin{pmatrix} 1 & \cdots & & & 1 \\ m_{21} & 1 & \cdots & & \\ m_{31} & m_{32} & 1 & \cdots & \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{pmatrix}$$

$$U = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

Znając LU sprowadzamy nasz problem do rozwiązania

$$Ly = b$$

$$Ux = y$$

Podczas obliczeń, aby odszczędzać miejsce, będziemy modyfikować macierz A, tak aby pod przekątną znalazło się L a pozostałe miejsca były zajęte przez U.

## 3.1 Wariant pierwszy - bez wyboru elementu głównego

### 3.1.1 Złożoność czasowa

Po przyjrzeniu się algorytmowi na wyznaczenie LU zauważymy, że jest on bardzo podobny do procesu eliminacji w eliminacji Gaussa. Występują dokładnie te

same obliczenia z tą różnicą, że macierz A jest modyfikowana i zapisywane są wewnątrz niej wartości L oraz U. Z tego powodu możemy stwierdzić, że złożoność czasowa tego algorytmu to również $O(n)$.

### 3.1.2 Złożoność pamięciowa

Jeśli chodzi o złożoność pamięciową to LU zapisujemy w A (L jest macierzą dolnotrójkątną a U górnotrójkątną). Obydwie macierze mają tą samą strukturę. Więc złożoność pamięciowa wynosi $O(n)$.

### 3.1.3 Uwagi do kodu

Algorytm jest bardzo podobny do zwykłej eliminacji Gaussa z tą różnicą, że zamiast zerować pierwszy niezerowy element poniżej przekątnej (ten, który znajduje się w liczniku podczas obliczania czynnika $m$), zapisujemy w tym miejscu czynnik $m$ - używany do zerowania wierszy poniżej diagonali.

## 3.2 Wariant drugi - z częściowym wyborem elementu głównego

Analogicznie jak w przypadku eliminacji Gaussa z częściowym wyborem, ten wariant zapobiega błędom numerycznym w przypadku bliskim lub równym zera wartościom na przekątnej macierzy A. Lekko modyfikujemy podstawowy algorytm wyznaczania rozkładu LU, zamieniając wiersze miejscami i zapamiętując tablicę permutacji. W tym samym momencie odpowiednio modyfikujemy macierz A tworząc z niej macierz LU.

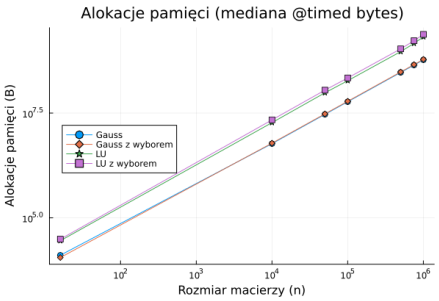
### 3.2.1 Złożoność czasowa

W tym algorytmie występuje taka sama pętla `for`, odpowiedzialna za szukanie największego elementu w kolumnie, jak w wypadku eliminacji Gaussa z częściowym wyborem elementu głównego. Następnie algorytm wyznaczania rozkłau LU wygląda niemal identycznie jak faza eliminacji z tą różnicą, że macierz A jest wtedy odpowiednio modyfikowana. Wobec tego złożoność czasowa algorytmu to $O(n)$.

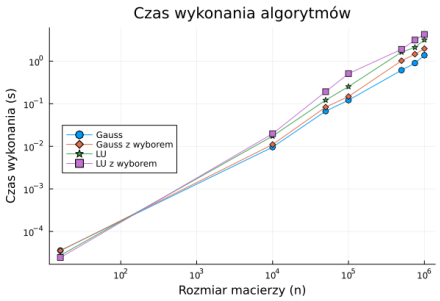
### 3.2.2 Złożoność pamięciowa

Uzasadnienie, że złożoność pamięciowa algorytmu wynosi $O(n)$ nasuwa się samo. Skoro dominującym czynnikiem jest tu macierz A, którą modyfikujemy in-place i w żadnym momencie nie potrzebujemy tworzenia dodatkowej kopii (z wyjątkiem macierzy permutacji, ale zostało już wytłumaczone dlaczego nie jest to problemem) albo nowej struktury to złożoność wynosi $O(n)$.

# 4 Eksperymenty




Rysunek 1: Wykres pamięci od razmiaru macierzy




Rysunek 2: Wykres czasu od rozmiaru macierzy

## 4.1 Wnioski

Powyżej znajduje się porównanie wszystkich czterech zaimplementowanych algorytmów. Widzimy, że obydwa wykresy: pamięci i czasu są liniowe względem rozmiaru macierzy, czyli zaimplementowane przeze mnie rozwiązanie spełnia warunki zadania. W przypadku wykresu pamięci algorytmy, które generują nam macierz LU a następnie rozwiązują układ używając jej, mają trochę większą złożoność niż zwykły algorytm Gaussa. Jest to zgodne z intuicją, ponieważ generując LU automatycznie przechowujemy więcej elementów w samej macierzy niż oryginalnie. Również w przypadku czasu wykonania są one trochę wolniejsze niż metody eliminacji Gaussa z częściowym wyborem elementu głównego oraz bez. Jeśli natomiast chodzi o rozróżnienie algorytmów pod kątem częściowego wyboru elementu głównego to pamięciowo zarówno algorytm z LU jak i zwykły Gauss zajmują pamięciowo porównywalnie tyle samo miejsca, jednak wersje z wyborem elementu głównego są wolniejsze. Jest to zgodne z tym, że pivotowanie wymaga dodatkowych operacji (wyszukiwanie maksimum, permutacje). Koszt ten jednak jest niewielki, a takie algorytmy są bardziej stabilne.