# 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

Zawarte w pseudokodzie metody `get_bottom_row` oraz `get_last_column` zwracają odpowiednio ostatni wiersz i ostatnią kolumnę, w której pojawia się niezerowy element. Dzięki temu dokonujemy optymalizowania opisanego we wcześniejszym paragrafie i ograniczamy nasze obliczenia jedynie do niezerowych elementów.

```
1: procedure GaussElimination(A, b)
2:     n ← size(A)
3:     for k ← 1 to n − 1 do
4:         for i ← k + 1 to get_bottom_row(A,k) do
5:             if A[k, k] = 0 then
6:                 error "Zero on diagonal at (k, k)"
7:             end if
8:             m ← A[i, k]/A[k, k]
9:             A[i, k] ← 0
10:            for j ← k + 1 to get_last_column(A,k) do
11:                A[i, j] ← A[i, j] − m · A[k, j]
12:            end for
13:            b[i] ← b[i] − m · b[k]
14:        end for
15:    end for
16:    x ← 0 ∈ R^n
17:    x[n] ← b[n]/A[n, n]
18:    for i ← n − 1 downto 1 do
19:        s ← 0
20:        for j ← i + 1 to get_last_column(A,i) do
21:            s ← s + A[i, j] · x[j]
22:        end for
23:        x[i] ← (b[i] − s)/A[i, i]
24:    end for
25:    return x
26: 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

**Require:** $A$ – `BlockMatrix` $n \times n$; $b$ – wektor prawych stron dł. $n$
**Ensure:** $x$ – rozwiązanie układu $Ax = b$

1: $n \leftarrow A.size$
2: $p \leftarrow [1, 2, \ldots, n]$ ▷ wektor permutacji (mapowanie wierszy)
3: **for** $k \leftarrow 1$ **to** $n - 1$ **do**
4: $bound \leftarrow \text{get\_bottom\_row}(A, k)$
5: $j \leftarrow k$ ▷ indeks wiersza z największym elementem w kolumnie
6: **for** $r \leftarrow k$ **to** $bound$ **do**
7: **if** $|A[p[r], k]| > |A[p[j], k]|$ **then**
8: $j \leftarrow r$
9: **end if**
10: **end for**
11: $\text{swap}(p[k], p[j])$ ▷ zamiana w wektorze permutacji
12: **for** $i \leftarrow k + 1$ **to** $bound$ **do**
13: $z \leftarrow A[p[i], k] / A[p[k], k]$
14: $A[p[i], k] \leftarrow 0$
15: $last \leftarrow \text{get\_last\_column}(A,\ k + A.block\_size)$
16: **for** $j \leftarrow k + 1$ **to** $last$ **do**
17: $A[p[i], j] \leftarrow A[p[i], j] - z \cdot A[p[k], j]$
18: **end for**
19: $b[p[i]] \leftarrow b[p[i]] - z \cdot b[p[k]]$
20: **end for**
21: **end for**
22: $x \leftarrow \mathbf{0} \in \mathbb{R}^n$
23: $x[n] \leftarrow b[p[n]] / A[p[n], n]$
24: **for** $i \leftarrow n - 1$ **downto** $1$ **do**
25: $s \leftarrow b[p[i]]$
26: $last \leftarrow \text{get\_last\_column}(A,\ i + A.block\_size)$
27: **for** $j \leftarrow i + 1$ **to** $last$ **do**
28: $s \leftarrow s - A[p[i], j] \cdot x[j]$
29: **end for**
30: $x[i] \leftarrow s / A[p[i], i]$
31: **end for**
32: **return** $x$