Lista nr 5

1. Problem rozwiązywania układu równań liniowych

1.1. Definicja

W tym problemie szukamy wektora rozwiązań $x \in \mathbb{R}^n$, który rozwiązuje zadany układ równań:

$$a_{11}x_1 + a_{12}x_2 + ... + a_1nx_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + ... + a_2nx_n = b_2$$

$$...$$

$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n$$

Możemy to też zapisać macierzowo dla zadanej macierzy $A \in \mathbb{R}^{n \times n}$ i wektora $b \in \mathbb{R}^n$:

$$Ax = b$$

1.2. Metoda eliminacja Gaussa

Metoda składa się z serii kroków. W każdym wykorzystujemy jedno z równań, żeby wyeliminować jedną niewiadomą z pozostałych równań. Natomiast na końcu korzystamy z prostego podstawiania. Dla przykładu:

$$1 : a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$2 : a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = b_2$$

$$3 : a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = b_3$$

Możemy usunąć z równań 2 i 3 niewiadomą x_1 poprzez odjęcie od nich przeskalowanego równania 1 (odpowiednio przez $\frac{a_{21}}{a_{11}}$ i $\frac{a_{31}}{a_{11}}$):

$$1 : a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$2 : a'_{22}x_2 + a'_{23}x_3 = b'_2$$

$$3 : a'_{32}x_2 + a'_{33}x_3 = b'_3$$

Ten krok możemy powtórzyć, tym razem wykorzystując równanie 2 do usunięcia niewiadomej x_2 z równania 3.

$$1 : a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$2 : a'_{22}x_2 + a'_{23}x_3 = b'_2$$

$$3 : a''_{33}x_3 = b''_3$$

Poprzez wykorzystywanie jedynie operacji elementarnych uzyskany układ jest równoważny z układem początkowym i do tego trywialny do rozwiązania. Z równania 3 otrzymujemy wartość niewiadomej x_3, którą następnie podstawiamy do równania 2 i wyliczamy niewiadomą

x_2. Obydwie poznane niewiadome podstawiamy do równania 1 i otrzymujemy x_1, tym samym rozwiązując układ równań.

1.3. Rozkład LU

Cały proces eliminacji Gaussa możemy zapisać w postaci działań na macierzach. Realizacja kroku k algorytmu możemy zapisać jako:

$$A^{(k+1)} = L^{(k)}A^{(k)} \quad b^{(k+1)} = L^{(k)}b^{(k)}$$

$$L^{(k)} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -l_{k+1,k} & 1 & & \\ & & & -l_{k+2,k} & & 1 & \\ & & & \vdots & & & \ddots \\ & & & -l_{n,k} & & & 1 \end{bmatrix}$$

$$l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}$$

Macierze $L^{(k)}$ składają się z jedynek na przekątnej i współczynników skalujących l_{ik} w kolumnie k. Przemnożenie przez tą macierz powoduje wyzerowanie czynników przy niewiadomej x_k od równania $k+1$ do n.

W wyniku tych nakładania tych działań uzyskujemy macierz górną trójkątną U:

$$U = L^{(n)}...L^{(2)}L^{(1)}A$$

Inaczej:

$$A = {L^{(1)}}^{-1}{L^{(2)}}^{-1}...{L^{(n)}}^{-1}U$$

Co składamy do postaci:

$$A = LU$$

Po przeliczeniu dochodzimy, że macierz L jest postaci:

$$L = \begin{bmatrix} 1 & & & & \\ l_{21} & 1 & & & \\ l_{31} & l_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{bmatrix}$$

Teraz rozwiązanie układu równań:

$$LUx = b$$

Sprowadza się do rozwiązania dwóch równań, które ze względu na trójkątną postać macierzy, stają się trywialne:

$$Ly = b \quad Ux = y$$

1.4. Wybór elementu głównego

Algorytm w takiej postaci zawsze używa równania 1 do wyeliminowania niewiadomej x_1, równania 2 do wyeliminowania x_2, itd. Jednak jak wiemy zamiana równań miejscami nie zmieni rozwiązań danego układu. Równie dobrze możemy do wyeliminowania x_1 wykorzystać dowolne równanie.

To pozwala rozwiązać problem, gdy w wyniku działania algorytmu, element główny - $a_{kk}^{(k)}$ - stanie się zerem lub będzie bliski zeru. Dzielenie przez niego może się wiązać z wprowadzeniem dużego błędu numerycznego.

Częściowy wybór elementu głównego polega na znalezieniu elementu maksymalnego pod względem modułu w kolumnie k wśród pozostałych równań:

$$|a_{pk}^{(k)}| = \max_{k \leq i \leq n} |a_{ik}^{(k)}|$$

Wtedy następuje zamiana miejscami równania p z równaniem k. Ostatecznie skończymy z pewną permutacją P oryginalnej macierzy:

$$LU = PA$$

Stąd w celu rozwiązania układu należy rozwiązać dwa równania:

$$Ly = Pb \quad Ux = y$$

1.5. Złożoność obliczeniowa i pseudokod

W implementacji komputerowej rozkładu LU nie potrzebujemy dodatkowej pamięci, ponieważ obydwie macierze mieszczą się w miejscu oryginalnej macierzy A.

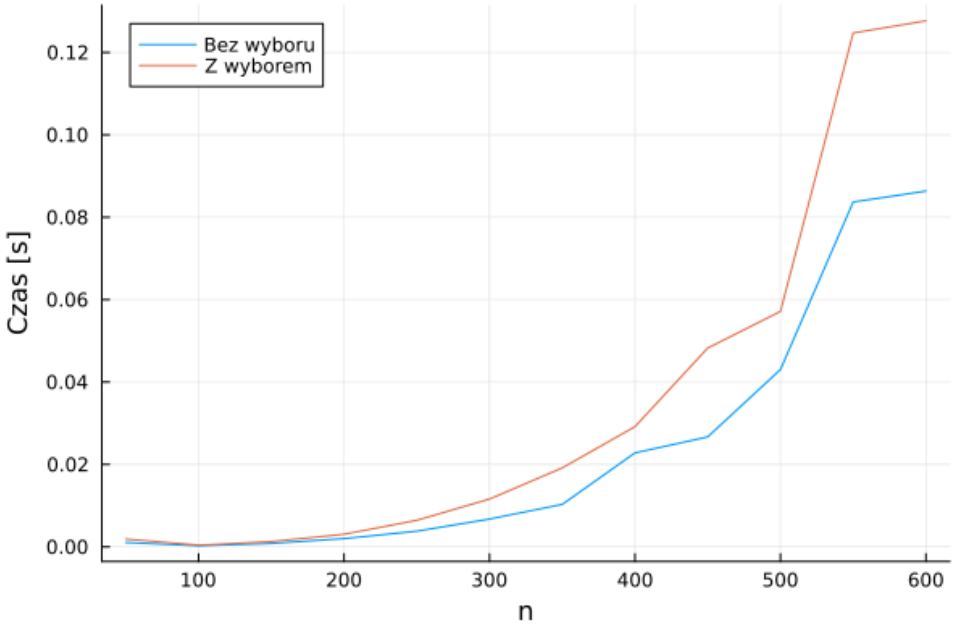
1: **function** ROZKŁAD-LU($a \in \mathbb{R}^{n \times n}$)
2: **for** k from 1 to $n-1$ **do**
3: **for** i from $k+1$ to n **do**
4: $l \leftarrow \frac{a_{ik}}{a_{kk}}$
5: **for** j from $k+1$ to n **do**
6: $a_{ij} \leftarrow a_{ij} - l \cdot a_{kj}$
7: $a_{ik} \leftarrow l$
8: **return** a

Żeby nie robić niepotrzebnej pracy poprzez kopiowanie zawartości wierszy z jednego miejsca w pamięci komputera do drugiego, wykorzystujemy wektor permutacji p. Na początku zaczyna jako permutacja $[1, 2, ..., n]$. Zamiany wierszy realizujemy jako zamiana elementów tej permutacji. p_i mówi nam które z równań (według oryginalnej numeracji) jest teraz na miejscu równania i.

1: **function** ROZKŁAD-LU-Z-WYBOREM($a \in \mathbb{R}^{n \times n}$)
2: $p \leftarrow [1, 2, ..., n]$
3: **for** k from 1 to $n-1$ **do**
4: $m \leftarrow \text{argmax}_{k \leq i \leq n} \ |a_{p_i k}|$

```
5:            p_m, p_k ← p_k, p_m
6:            for i from k + 1 to n do
7:                l ← a_{p_i k} / a_{p_k k}
8:                for j from k + 1 to n do
9:                    a_{p_i j} ← a_{p_i j} − l · a_{p_k j}
10:               a_{p_i k} ← l
11:       return a, p
```

Wnioskując z prostego pseudokodu, oba algorytmy wykonują się w czasie $O(n^3)$.



Rysunek 1: Czas znajdywania rozkładu LU tradycyjnym algorytmem

2. Adaptacja algorytmu dla specjalnych macierzy

2.1. Opis problemu

W niektórych środowiskach ma się do czynienia z macierzami o rozmiarach rzędu setek tysięcy, gdzie rozwiązywanie równania $Ax = b$ przy pomocy tradycyjnego algorytmu eliminacji Gaussa staje się nieosiągalne. Często da się jednak wykorzystać specjalną strukturę tych macierzy do uproszczenia obliczeń.

W tym zadaniu mamy do czynienia z macierzą, której jedyne elementy niezerowej są podzielone na bloki znajdujące się przy przekątnej. Zadany jest rozmiar bloku l i podzielny przez niego rozmiar macierzy n. Liczba bloków to $v = \frac{n}{l}$.

$$A_k = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1l} \\ a_{21} & a_{22} & \dots & a_{2l} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \dots & a_{ll} \end{bmatrix} \quad B_k = \begin{bmatrix} 0 & \dots & 0 & b_1^k \\ 0 & \dots & 0 & b_2^k \\ \vdots & & \vdots & \vdots \\ 0 & \dots & 0 & b_l^k \end{bmatrix} \quad C_k = \begin{bmatrix} c_1^k & 0 & \dots & 0 \\ 0 & c_2^k & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & c_l^k \end{bmatrix}$$

$$A = \begin{bmatrix} A_1 & C_1 & & & & \\ B_2 & A_2 & C_2 & & & \\ & B_3 & A_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & B_{v-1} & A_{v-1} & C_{v-1} & \\ & & & B_v & A_v & C_v \end{bmatrix}$$

2.2. Adaptacja rozkładu LU bez wyboru elementu głównego

2.2.1. Pomysł

Do optymalizacji tego algorytmu wystarczy jedynie to, które elementy danych wierszy są niezerowe. Dla wizualizacji oznaczyłem elementy pochodzące z bloków typu A_k jako a, z B_k jako b, z C_k jako c.

$$A = \begin{bmatrix} a & a & c & & & \\ a & a & & c & & \\ & b & a & a & c & \\ & b & a & a & & c \\ & & & b & a & a \\ & & & b & a & a \end{bmatrix}$$

Na tym przykładzie widać, że elementy niezerowe danego wiersza i zaczynają się od jakiejś kolumny s_i i kończą na kolumnie t_i. Obie z tych wartości zależą tylko od numeru wiersza i oraz wartości l i n, które opisują wygląd macierzy.

Do wyliczenia wartości s_i oraz l_i najpierw potrzeba jest znać, jaki jest numer bloku, którego wartości znajdują się w tym wierszu. Wiersze z $i \in [1, l]$ mają numer bloku 1, z $i \in [l+1, 2l]$ mają 2, itd. Tę liczbę dla danego wiersza i można wyrazić jako $u_i = \lfloor \frac{i-1}{l} \rfloor$.

Liczba s_i to inaczej numer kolumny, gdzie pojawia się w tym wierszu element typu b lub 1, gdy w danym wierszu nie ma takiego elementu. Wystąpienie elementu b zależy natomiast od numeru bloku. Konkretnie:

$$s_i = \max\{u_i \cdot l, 1\} = \max\left\{ \left\lfloor \frac{i-1}{l} \right\rfloor \cdot l, 1 \right\}$$

Podobnie liczba t_i to inaczej kolumna, gdzie w tym wierszu znajduje się element typu c lub n, gdy nie ma takiego elementu. To wyraża się zależnością:

$$t_i = \min\{i + l, n\}$$

Tak samo można spojrzeń na niezerowe elementy danej kolumny j. Kończą się na jakiejś liczbie b_j. Z uwagi na wystającą w lewo kolumnę elementów typu b, ta wartość zależy od tego, gdzie kończą się bloki $j+1$.

$$b_j = \min\big\{ (u_{j+1} + 1) \cdot l, n \big\} = \min\left\{ \left(\left\lfloor \frac{j}{l} \right\rfloor + 1 \right) \cdot l, n \right\}$$

Spójrzmy jak wygląda sytuacja w kroku k

$$\begin{bmatrix}
a_{11} & a_{21} & \dots & & & \dots & & & & \\
0 & a_{22} & \dots & & & \dots & & & & \\
\vdots & & \ddots & & & \dots & & & & \\
0 & \dots & 0 & a_{kk} & a_{kk+1} & \dots & a_{kt_k} & 0 & \dots & 0 \\
0 & \dots & 0 & a_{k+1k} & a_{k+1k+1} & \dots & a_{k+1t_k} & a_{k+1t_{k+1}} & 0 \dots & 0 \\
\vdots & & & \vdots & & & \vdots & \ddots & \ddots & \vdots \\
0 & \dots & 0 & a_{b_kk} & a_{b_kk+1} & \dots & a_{b_kt_k} & \dots & a_{b_kt_{b_k}} & 0 \dots 0 \\
0 & \dots & 0 & 0 & & \dots & & & & \\
\vdots & & \vdots & \vdots & & \dots & & & &
\end{bmatrix}$$

Skoro b_k wyraża to gdzie kończą się niezerowe elementy kolumny k, to wystarczy wyzerować tą kolumnę jedynie w wierszach pomiędzy $k+1$ a b_k. Podobnie skoro t_k wyraża gdzie kończą się niezerowe elementy wiersza k, to tylko elementy kolumn od $k+1$ do t_k będą brały udział w odejmowaniu.

Dla danych wierszy i kolumn k liczby t_k i b_k nie zmieniają się w czasie działania algorytmu. Bierze się to z faktu, że ciągi $(t_k)_k$ i $(b_k)_k$ są niemalejące. Dla danego wiersza i elementy zerowe o indeksach $j > t_k$ nadal pozostaną zerowe. Gdyby tak nie było, to znaczy że odjęliśmy od czynnika w tej kolumnie jakiś niezerowy czynnik z któregoś wiersza wyżej. Jednak każdy wiersz wyżej także ma element zerowy z tej kolumnie.

2.2.2. Pseudokod

function ADAPTOWANY-ROZKŁAD-LU(a, n, l)
    for k from 1 to $n-1$ **do**
        $b \leftarrow \min\Big(\Big(\Big\lfloor \frac{k}{l} \Big\rfloor + 1\Big) \cdot l, n\Big)$
        $t \leftarrow \min(k+l, n)$
        for i from $k+1$ to b **do**
            $l \leftarrow \frac{a_{ik}}{a_{kk}}$
            for j from $k+1$ to t **do**
                $a_{ij} \leftarrow a_{ij} - l \cdot a_{kj}$
            $a_{ik} \leftarrow l$
    return a

Liczbę operacji w danym kroku k możemy ograniczyć z góry przez l^2. Zatem ostateczna złożoność tego algorytmu to jedynie $O(nl^2)$. Gdy l ustalimy jako stałą, to możemy zapisać złożoność jako $O(n)$

Oryginalny algorytm rozwiązania układu przy pomocy rozkładu LU miał złożoność $O(n^2)$. Tutaj też można zastosować wcześniejsze obserwacje do ograniczenia obliczeń.

function ADAPTOWANY-SOLVE-LUXB(a, n, l, b)
    $x \leftarrow b$
    for k from 1 to n **do**
        $s \leftarrow \max\Big(\Big\lfloor \frac{k-1}{l} \Big\rfloor \cdot l, 1\Big)$
        for i from s to $k-1$ **do**
            $x_k \leftarrow x_k - x_i \cdot a_{ki}$

7: **for** k from n to 1 **do**
8: $t \leftarrow \min(k+l, n)$
9: **for** i from $k+1$ to t **do**
10: $x_k \leftarrow x_k - x_i * a_{ki}$
11: $x_k \leftarrow \frac{x_k}{a_{kk}}$
12: **return** x

Znowu w każdym kroku obu pętel można ograniczyć z góry liczbę działań do l. Co daje ponownie złożoność $O(n)$.

2.3. Adaptacja rozkładu LU z wyborem elementu głównego

2.3.1. Problem

W przypadku, kiedy dopuścimy zamianę miejscami wierszy, tracimy część dobrych właściwości, które mieliśmy przy rozkładzie bez wyboru. Tym razem liczby s_i i t_i ulegają zmianom w czasie działania algorytmu i nie tworzą już niemalejących ciągów.

$$A = \begin{bmatrix}
1: & a & a & a & c & & & & \\
2: & a & a & a & & c & & & \\
3: & a & a & a & & & c & & \\
4: & & & b & a & a & a & c & \\
5: & & & b & a & a & a & & c \\
6: & & & b & a & a & a & & \\
7: & & & & & & b & a & a & a \\
8: & & & & & & b & a & a & a \\
9: & & & & & & b & a & a & a
\end{bmatrix}$$

Na tym przykładzie można zauważyć, że zamiana wierszy w obrębie bloków (np. 1 i 2, 4 i 6, itd.) zmienia liczby t_i dla tych wierszy, ale pozostawia taką samą s_i. Mogą się też zdarzyć zamiany wierszy pomiędzy blokami (np. 3 i 4), które zmieniają także liczby s_i. W specyficznym przypadku może dojść do zamiany wiersza 1 z 9 (ciąg zamian $1 \leftrightarrow 3 \leftrightarrow 6 \leftrightarrow 9$), który spowoduje pojawienie się na samym dole wiersza, którego potencjalnie każdy element jest niezerowy.

Okazuje się jednak, że nadal możemy bez problemu wyliczyć liczby s'_i i t'_i na podstawie wiersza. Trzeba jedynie wykorzystać do tego wektor permutacji p.

$$s'_i = s_{p_i} = \max\left\{\left\lfloor \frac{p_i - 1}{l} \right\rfloor \cdot l, 1\right\}$$

$$t'_i = \max_{1 \le j \le i} t_{p_j} \le \min\left\{\left(\left\lfloor \frac{k}{l} \right\rfloor + 2\right) \cdot l, n\right\}$$

Uzasadnienie: Najpierw zauważmy, że w kroku k żadna liczba b_j dla $j \ge k$ nie zmieni się. Wymagałoby to zamiany pomiędzy niesąsiadującymi blokami.

Tym samym w kroku k maksymalnie może nastąpić zamiana z wierszem b_k. Wszystkie wiersze pomiędzy nimi w ramach odejmowania zostają „poszerzone". Stąd ograniczenie $t'_k \le t_{b_k} = \min\left\{\left(\min\left\{\left(\left\lfloor \frac{k}{l} \right\rfloor + 1\right) \cdot l, n\right\}\right) + l, n\right\} = \min\left\{\left(\left\lfloor \frac{k}{l} \right\rfloor + 2\right) \cdot l, n\right\}$.

W przypadku s'_i nie ma problemu. Elementy pod przekątną i na lewo od kolumny k nie biorą udziału w odejmowaniu, bo należą do macierzy L. Stąd zamiana wierszy powoduje jedynie zamiany pozycji elementu startowego w tych wierszach.

2.3.2. Pseudokod

Ostateczny algorytm wykorzystuje wyżej podane ograniczenia i obserwacje.

function ADAPTOWANY-ROZKŁAD-LU-Z-WYBOREM(a, n, l)
    $p \leftarrow [1, 2, ..., n]$
    for k from 1 to $n - 1$ **do**
        $b \leftarrow \min\Big(\Big(\Big\lfloor \frac{k}{l} \Big\rfloor + 1\Big) \cdot l, n\Big)$
        $t \leftarrow \min\Big(\Big(\Big\lfloor \frac{k}{l} \Big\rfloor + 2\Big) \cdot l, n\Big)$
        $m \leftarrow \text{argmax}_{k \le i \le b} \ |a_{p_i k}|$
        $p_m, p_k \leftarrow p_k, p_m$
        for i from $k + 1$ to b **do**
            $l \leftarrow \frac{a_{p_i k}}{a_{p_k k}}$
            for j from $k + 1$ to t **do**
                $a_{p_i j} \leftarrow a_{p_i j} - l \cdot a_{p_k j}$
            $a_{p_i k} \leftarrow l$
    return a, p

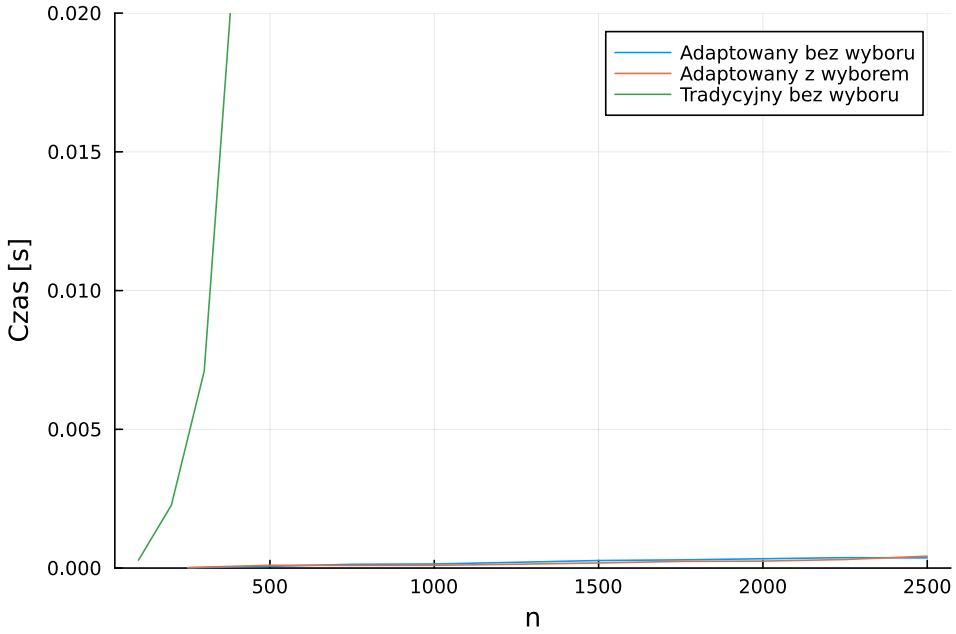
W każdym kroku liczbę operacji możemy ograniczyć z góry przez $l + l \cdot 2l$. Stąd pomimo wykonywania większej liczby operacji, udało się zachować złożoność $O(n)$.

Tak samo sytuacja wygląda dla algorytmu rozwiązywania układu z jedną subtelnością. Kroki pierwszej pętli nie można ograniczyć przez l. Jednak, jak później przy optymalizacji pamięciowej będzie pokazane, całą pętle i tak możemy ograniczyć przez $O(n)$.

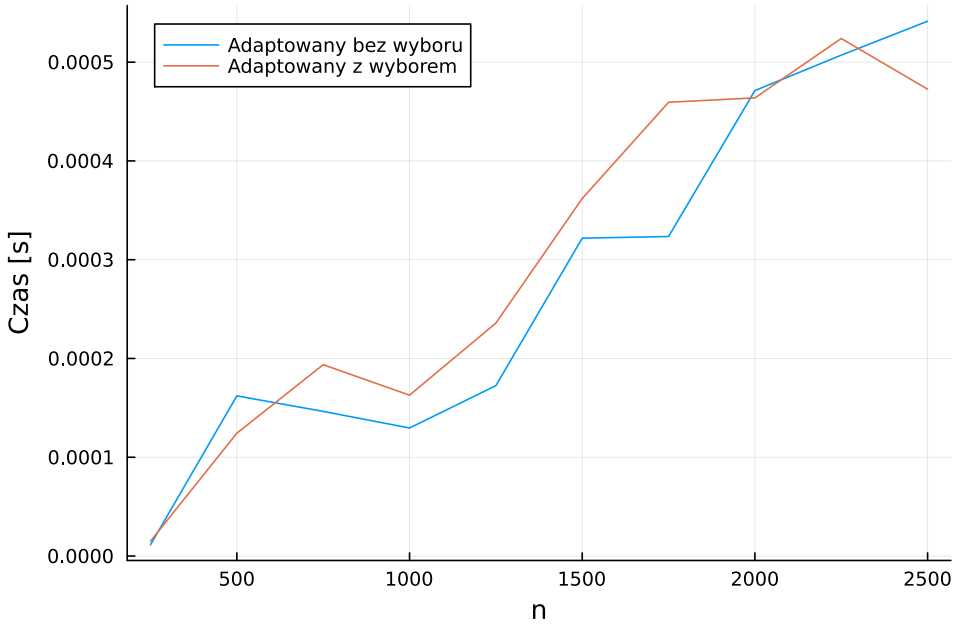
function ADAPTOWANY-SOLVE-LUXB-Z-WYBOREM(a, n, l, p, b)
    for k from 1 to n **do**
        $s \leftarrow \max\Big(\Big\lfloor \frac{p_k - 1}{l} \Big\rfloor \cdot l, 1\Big)$
        $x \leftarrow b_{p_k}$
        for i from s to $k - 1$ **do**
            $x_k \leftarrow x_k - x_i \cdot a_{p_k i}$
    for k from n to 1 **do**
        $t \leftarrow \min\Big(\Big(\Big\lfloor \frac{k}{l} \Big\rfloor + 2\Big) \cdot l, n\Big)$
        for i from $k + 1$ to t **do**
            $x_k \leftarrow x_k - x_i * a_{p_k i}$
        $x_k \leftarrow \frac{x_k}{a_{p_k k}}$
    return x

2.4. Porównanie

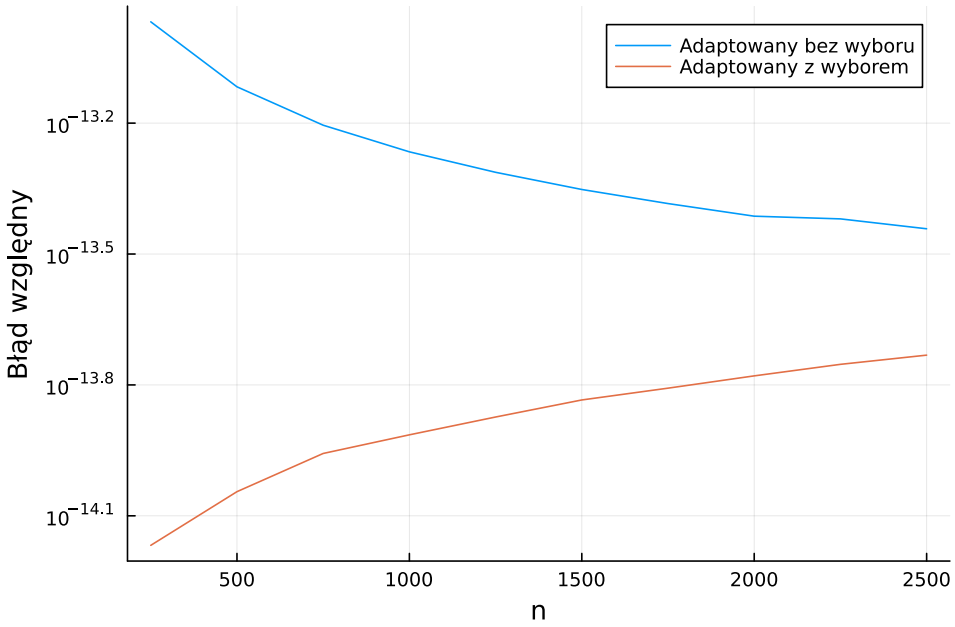
Na tym dość ekstremalnym wykresie widać, jak dużą różnice daje zastosowanie zaadaptowanego algorytmy o liniowej złożoności. Także zgodnie z oczekiwaniami, algorytm z częściowym wyborem zajmuje więcej czasu, ale charakteryzuje się mniejszym błędem.



Rysunek 2: Porównanie zaadaptowanych algorytmów z tradycyjnym rozkładem LU



Rysunek 3: Porównanie zaadaptowanych algorytmów ze sobą



Rysunek 4: Porównanie błędów zaadaptowanych algorytmów ze sobą

3. Optymalizacja pamięciowa

3.1. Lepsza struktura danych

Oprócz samego algorytmu potrzeba też zoptymalizować strukturę danych przechowującą macierz. Jeśli rozmiar macierzy ma sięgać setek tysięcy, to niemożliwym staje się przechowywanie jej w strukturze $O(n^2)$. Proponowana struktura będzie wykorzystywała fakt, że niezerowe elementy danego wiersza i znajdują się między s_i oraz t_i.

$$A = \begin{bmatrix} 1: & a & a & a & c & & & & \\ 2: & a & a & a & & c & & & \\ 3: & a & a & a & & & c & & \\ 4: & & b & a & a & a & c & & \\ 5: & & b & a & a & a & & c & \\ 6: & & b & a & a & a & & & c \\ 7: & & & & & b & a & a & a \\ 8: & & & & & b & a & a & a \\ 9: & & & & & b & a & a & a \end{bmatrix} \rightarrow A' = \begin{bmatrix} 1: & & a & a & a & c & & \\ 2: & & a & a & a & & c & \\ 3: & & a & a & a & & & c \\ 4: & b & a & a & a & c & & \\ 5: & b & a & a & a & & c & \\ 6: & b & a & a & a & & & c \\ 7: & b & a & a & a & & & \\ 8: & b & a & a & a & & & \\ 9: & b & a & a & a & & & \end{bmatrix}$$

Ta struktura zajmuje już jedynie $n \cdot (2l + 1)$ pamięci. $a_{i,j}$ w A odpowiada $a_{i,j-s_i+1}$ w A'. Jednak ta struktura nie zadziała w przypadku algorytmu rozkładu LU z częściowym wyborem elementu głównego. Opiera się na pomyśle, że $t_i - s_i \leq 2l + 1$, co nie jest spełnione w przypadku zamiany wierszy. Tam dochodziło do „poszerzania”, kiedy następowały zamiany pomiędzy różnymi blokami.

Jednak te zamiany następują w bardzo przewidywalny sposób. Jedyny moment, kiedy to się może stać, to na końcu bloku, kiedy $l \mid k$ i $k \neq n$ (zamiany w ramach tego samego bloku są załatwione poprzez już znajdujące się w strukturze puste pola na prawym końcu). Wtedy wiersz k zamienia się z wierszem i o większym t_i, przez co musi się „poszerzyć” do tej długości, co możemy ograniczyć przez l. Sumarycznie więc w wyniku takich „poszerzeń” pojawi

się jedynie $n - l$ nowych potencjalnie niezerowych elementów. Dodatkowo każdy z tych elementów będzie znajdował się w innej kolumnie. W związku z tym można je przechowywać w dodatkowym wektorze $u \in \mathbb{R}^n$ i prosto się do niego odnosić.

Ostatecznie więc definiujemy mapowanie elementów pełnowymiarowej macierzy do elementów tej struktury:

$$a_{i,j} = \begin{cases} a'_{i,j-\lfloor \frac{i-1}{l} \rfloor \cdot l+1} \text{ gdy } j \leq \left(\lfloor \frac{i-1}{l} \rfloor + 2\right) \cdot l \\ u_j \text{ w przeciwnym wypadku} \end{cases}$$

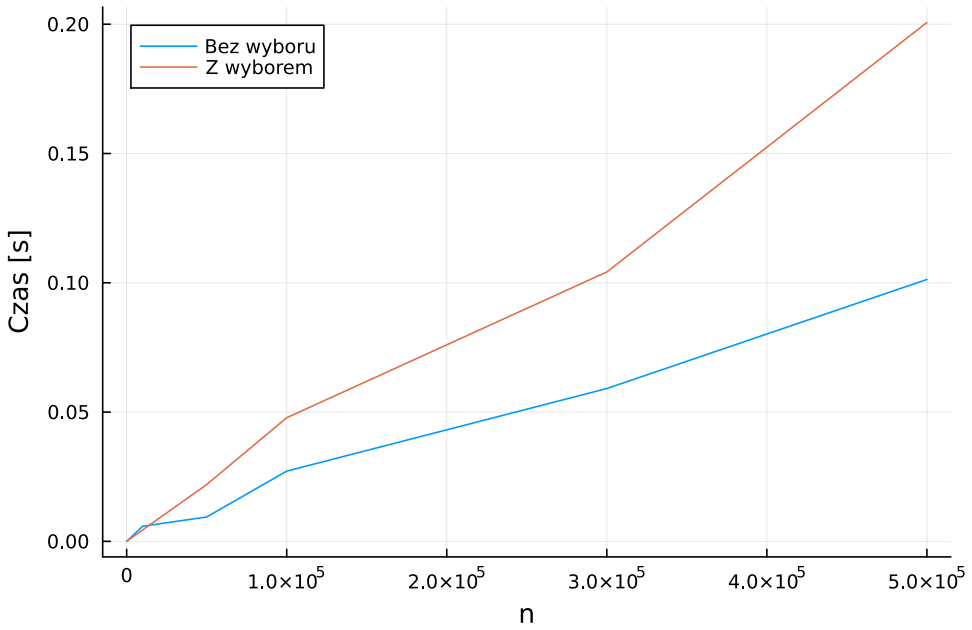
Zakładamy, że zawsze będziemy się odwoływali jedynie do elementów potrzebnych do działania tego algorytmu. Wykorzystujemy fakt, że „poszerzanie" jest jednoznaczne w stostunku do kolumn. W takim razie, jeżeli w trakcie działania algorytm będzie chciał odwołać się do wiersza poza jego oryginalnym t_i (a właściwie to jego górnym ograniczeniem $\left(\lfloor \frac{i-1}{l} \rfloor + 2\right) \cdot l$ oznaczającym najbardziej na prawo wysuniętą kolumnę), to wiemy że doszło do „poszerzenia" i na podstawie kolumny decydujemy, który element wektora u zwrócić.

Tym samym pamięć potrzebna na przechowanie takiej struktury to $n \cdot (2l + 2)$, co znaczy $O(n)$.

3.2. Wyniki

n, l	$16, 4$	$10000, 5$	$50000, 4$	$100000, 5$	$300000, 4$	$500000, 4$
Gauss	$5.201 \cdot 10^{-6}s$	$0.002523s$	$0.009766s$	$0.02724s$	$0.0539s$	$0.09688s$
Gauss z wyb.	$6.304 \cdot 10^{-6}s$	$0.007336s$	$0.02121s$	$0.05067s$	$0.1129s$	$0.1728s$
LU	$3.448 \cdot 10^{-6}s$	$0.00248s$	$0.01802s$	$0.0274s$	$0.06349s$	$0.1035s$
LU z wyb.	$1.067 \cdot 10^{-5}s$	$0.004479s$	$0.04068s$	$0.05085s$	$0.1179s$	$0.1741s$

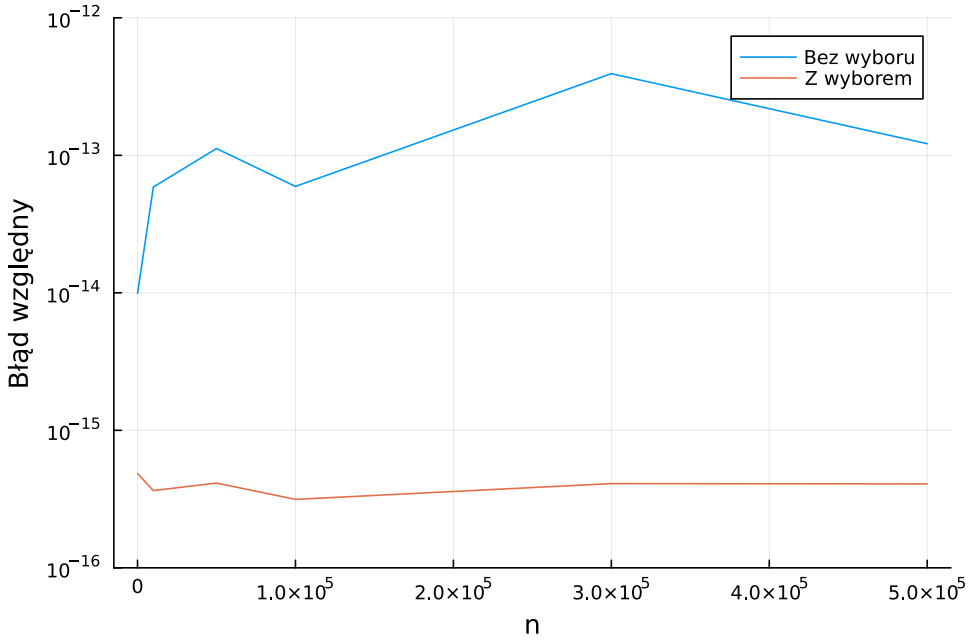
Tabela 1: Porównanie czasu rozwiązywania układu równań przy pomocy rozkładu LU



Rysunek 5: Porównanie czasu dla algorytmu rozkładu LU

n, l	$16, 4$	$10000, 5$	$50000, 4$	$100000, 5$	$300000, 4$	$500000, 4$
Bez wyboru	$9.9 \cdot 10^{-15}$	$5.9 \cdot 10^{-14}$	$1.1 \cdot 10^{-13}$	$5.9 \cdot 10^{-14}$	$3.9 \cdot 10^{-13}$	$1.2 \cdot 10^{-13}$
Z wyborem	$4.9 \cdot 10^{-16}$	$3.6 \cdot 10^{-16}$	$4.1 \cdot 10^{-16}$	$3.1 \cdot 10^{-16}$	$4.1 \cdot 10^{-16}$	$4.1 \cdot 10^{-16}$

Tabela 2: Porównanie błędu względnego rozwiązywania układu równań przy pomocy LU



Rysunek 6: Porównanie błędów względnych dla algorytmu LU

3.3. Wnioski

Wykresy dowodzą, że udało się zaimplementować algorytm eliminacji Gaussa oraz rozkładu LU w czasie liniowym i z liniową pamięcią. Pokazują też, że opłaca się dobry wybór elementu głównego. Poprawia błąd o dwa rzędy wielkości kosztem niecałego podwojenia czasu, który i tak jest wyjątkowo krótki. Sama eliminacja Gaussa wykonuje mniej operacji niż pełny rozkład LU, jednak okazała się jedynie nieznacznie szybsza.