Obliczenia Naukowe - L5

Piotr Maciejończyk

3 stycznia 2024

1 Opis problemu

Otrzymujemy specyficzną blokową macierz rzadką $\mathbf{A} \in \mathbb{R}^{n \times n}$, którą można przedstawić w taki oto sposób:

$$\mathbf{A} = \begin{bmatrix} A_1 & C_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ B_2 & A_2 & C_2 & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & B_3 & A_3 & C_3 & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{0} & B_{v-2} & A_{v-2} & C_{v-2} & \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & B_{v-1} & A_{v-1} & C_{v-1} \\ \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \mathbf{0} & B_v & A_v \end{bmatrix}$$

Zmienna v jest równa wartości $\frac{n}{l} \in \mathbb{N}$, gdzie n to rozmiar macierzy $\mathbf{A}$ oraz l to rozmiar każdego z bloków. Bloki A_k, B_k, C_k oraz $\mathbf{0}$ przyjmują takie oto formy:

1. $$A_k = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1l-1} & a_{1l} \\ a_{21} & a_{22} & \dots & a_{2l-1} & a_{2l} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{l-11} & a_{l-12} & \dots & a_{l-1l-1} & a_{l-1l} \\ a_{l1} & a_{l2} & \dots & a_{ll-1} & a_{ll} \end{bmatrix}$$

   Macierze $A_k \in \mathbb{R}^{l \times l}$ (gdzie $k \in \{1, 2, \dots, v\}$) to macierze gęste, w których znajduje się zero lub stosunkowo niewielka liczba pozycji zerowych.

2. $$B_k = \begin{bmatrix} 0 & 0 & \dots & 0 & b_1^k \\ 0 & 0 & \dots & 0 & b_2^k \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & b_{l-1}^k \\ 0 & 0 & \dots & 0 & b_l^k \end{bmatrix}$$

   Macierze $B_k \in \mathbb{R}^{l \times l}$ (gdzie $k \in \{2, 3, \dots, v\}$) charakteryzują się tym, iż jedynie ich ostatnie kolumny są niezerowe.

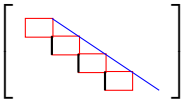
3. $$C_k = \begin{bmatrix} c_1^k & 0 & \dots & 0 & 0 \\ 0 & c_2^k & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & c_{l-1}^k & 0 \\ 0 & 0 & \dots & 0 & c_l^k \end{bmatrix}$$

Macierze $C_k \in \mathbb{R}^{l \times l}$ (gdzie $k \in \{1, 2, \ldots, v-1\}$) to macierze diagonalne, które posiadają niezerowe pozycje jedynie na swoich przekątnych.

4.
$$\mathbf{0} = \begin{bmatrix} 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 0 & 0 \\ 0 & 0 & \ldots & 0 & 0 \end{bmatrix}$$

Macierze $\mathbf{0}$ to macierze zerowe stopnia l, gdzie wszystkie ich współczynniki są równe zeru.

Dla lepszego zobrazowania macierzy $\mathbf{A}$ postanowiłem umieścić poniżej jej reprezentację, w której pod uwagę wziąłem tylko pozycje niezerowe. Czerwone kwadraty to macierze A_k, czarne linie - miejsca niezerowe B_k, a niebieska linia to miejsca niezerowe tworzone przez macierze C_k:

$\mathbf{A} =$ 

Naszym celem jest efektywne znalezienie rozwiązania równania $\mathbf{Ax} = \mathbf{b}$, gdzie $\mathbf{b} \in \mathbb{R}^n$ to wektor prawych stron, $n \geq 4$, a $\mathbf{x}$ to poszukiwane przez nas rozwiązanie układu równań.

2 Metoda eliminacji Gaussa

Metoda eliminacji Gaussa to technika rozwiązywania układów równań liniowych poprzez przekształcenie macierzy współczynników do postaci trójkątnej górnej poprzez eliminację zmiennych. Po otrzymaniu tej postaci można wyznaczyć rozwiązania układu równań za pomocą metody substytucji wstecznej.

Kroki metody eliminacji Gaussa

1. **Tworzenie macierzy rozszerzonej:** Połącz macierz współczynników A z wektorem prawych stron b w tzw. macierz rozszerzoną: $[A|b]$.
2. **Eliminacja zmiennych:** Dążymy do uzyskania macierzy trójkątnej górnej poprzez eliminację zmiennych. Iteracyjnie dla każdej kolumny k od 1 do $n-1$, eliminujemy zmienne w kolumnie k pod przekątną. Dla każdego wiersza i od $k+1$ do n, mnożymy równanie i-te przez pewien współczynnik, aby wyeliminować zmienną x_k.
3. **Metoda substytucji wstecznej:** Po uzyskaniu macierzy trójkątnej górnej, możemy znaleźć rozwiązania układu równań poprzez zastosowanie metody substytucji wstecznej. Polega ona na obliczaniu kolejnych rozwiązań układu równań za pomocą już obliczonych.

Rozważmy układ n równań liniowych z n zmiennymi i przeanalizujmy pierwsze kilka kroków algorytmu:

$$A^{(1)} = \begin{array}{ccccccccc} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ a_{21}^{(1)}x_1 & + & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}^{(1)}x_1 & + & a_{n2}^{(1)}x_2 & + & \cdots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)} \end{array}$$

(zapis $a_{ij}^{(k)}$ oraz $b_i^{(k)}$ oznacza wartości tych pozycji po ostatnim $k+1$ kroku, który przyczynił się do ich zmiany; z kolei $A^{(k)}$ oznacza oryginalną macierz A po $k+1$ kroku).

W pierwszym kroku chcemy pozbyć się zmiennej x_1 z równań od 2-go do n-tego. W tym celu, dla i-tego wiersza (gdzie $i \in \{2, 3, \ldots, n\}$), obliczamy wartość $I_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$, mnożymy ją przez równanie pierwsze i odejmujemy ją od tego wiersza. Po tym pierwszym kroku, otrzymujemy poniższy układ równań:

$$A^{(2)} = \begin{array}{rcrcccrcl} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \ldots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{22}^{(2)}x_2 & + & \ldots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ & & a_{32}^{(2)}x_2 & + & \ldots & + & a_{3n}^{(2)}x_n & = & b_3^{(2)} \\ & & \vdots & & & & \vdots & = & \vdots \\ & & a_{n2}^{(2)}x_2 & + & \ldots & + & a_{nn}^{(2)}x_n & = & b_n^{(2)} \end{array}$$

Zatem po $n-1$ krokach otrzymujemy następująco macierz górnotrójkątną:

$$A^{(n)} = \begin{array}{rcrcccrcl} a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \ldots & + & a_{1n}^{(1)}x_n & = & b_1^{(1)} \\ & & a_{22}^{(2)}x_2 & + & \ldots & + & a_{2n}^{(2)}x_n & = & b_2^{(2)} \\ & & & & \ddots & & \vdots & & \vdots \\ & & & & & & a_{nn}^{(n)}x_n & = & b_n^{(n)} \end{array}$$

Z tej postaci łatwo możemy otrzymać $x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}}$. Kolejne wartości otrzymujemy ze wzoru:

$$x_k = \frac{b_k - \sum_{j=k+1}^{n} u_{kj}x_j}{u_{kk}}, \text{ dla } k \in \{n-1, \ldots, 1\}$$

Poniżej zamieściłem pseudokod metody eliminacji Gaussa:

Algorithm 1 Metoda eliminacji Gaussa

```
1:  procedure EliminacjaGaussa(A, b)
2:      for k ← 1 to n − 1 do                          ▷ Iteracja po kolumnach
3:          for i ← k + 1 to n do                      ▷ Iteracja po wierszach
4:              I ← A[i][k] / A[k][k]                  ▷ Współczynnik eliminacji
5:              A[i][k] ← 0                            ▷ Zerowanie elementu
6:              for j ← k + 1 to n do                  ▷ Aktualizacja macierzy
7:                  A[i][j] ← A[i][j] − I · A[k][j]
8:              end for
9:              b[i] ← b[i] − I · b[k]                 ▷ Aktualizacja wektora prawych stron
10:         end for
11:     end for
12: end procedure
```

Rozważmy złożoność obliczeniową metody eliminacji Gaussa dla układu n równań z n zmiennymi:

1. Iterujemy po kolumnach macierzy A, co daje $n-1$ iteracji.
2. W każdej iteracji wykonujemy eliminację zmiennych w odpowiednich wierszach, co wymaga przeglądu $n-k$ wierszy dla danej kolumny k.
3. Dla każdego wiersza wykonujemy operacje elementarne na zmiennych, co wymaga aktualizacji $n+1$ współczynników.

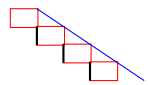
Stąd złożoność obliczeniowa metody eliminacji Gaussa wynosi $O(n^3)$.

3 Optymalizacja dla naszego problemu

W oryginalnej wersji metody eliminacji Gaussa mamy do czynienia z trzema pętlami **for**. Pętla pierwsza nie jest możliwa do zoptymalizowania, gdyż musimy się przemieścić przez $n-1$ kolumn. Natomiast, udało mi się znaleźć optymalizacje dla drugiej oraz trzeciej pętli, zatem ta część dotyczyć będzie pętli drugiej.

Przypomnijmy, że tak wygląda nasza macierz $\mathbf{A}$:

$$\mathbf{A} = \left[\quad \right]$$



Rozważmy przykładową macierz $\mathbf{M}$ o parametrach $n = 9$ oraz $l = 3$:

$$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ 0 & 0 & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & 0 & 0 & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Dla każdej kolumny k, obliczmy teraz największy indeks wiersza i, dla którego macierz przyjmuje wartości niezerowe. Robimy tak, ponieważ, jeśli pod znalezionym największym indeksem znajdują się już same miejsca zerowe, to współczynniki $I_{ik} = 0$, przez co nie zmieniamy w żaden sposób pozycji w macierzy:

Indeks kolumny k	Znaleziony indeks wiersza i
1	3
2	3
3	6
4	6
5	6
6	9
7	9
8	9
9	9

Wartość dla ostatniej kolumny możemy zignorować, ponieważ iterujemy jedynie do $n-1$ kolumny. Można zauważyć, że dla kolumny k, jeśli $l|k$, to znaleziony indeks powiększa się o l. Z tego powodu, zamiast iterować po wierszach od $(k+1)$-go do n-tego, za koniec zakresu przyjąłem $(k \div l + 1) \cdot l$ (gdzie $\div$ oznacza dzielenie całkowite).

Przyjrzyjmy się teraz trzeciej pętli w naszym algorytmie. Służy ona do aktualizowania współczynników w podanym wierszu. Wzór na tę aktualizację wygląda następująco:

$$a_{ij} = a_{ij} - I_{ik} \cdot a_{kj}$$

Pozbyliśmy się już przypadku, w którym to $I_{ik} = 0$, zatem skupmy się na zakresie, gdzie $a_{kj} \neq 0$ (inaczej: znajdźmy pierwszy od prawej indeks kolumny j, że dla k-tego wiersza współczynnik a_{kj} jest niezerowy). Na podstawie macierzy $\mathbf{M}$ otrzymujemy:

Indeks wiersza k	Znaleziony indeks kolumny j
1	4
2	5
3	6
4	7
5	8
6	9
7	9
8	9
9	9

Po tej obserwacji, nietrudno jest stwierdzić, że jako koniec zakresu dla trzeciej pętli możemy przyjąć wartość $\min(n, l + k)$. To samo tyczy się obliczania wektora rozwiązań $\mathbf{x}$, gdzie też możemy tak samo to zoptymalizować. Zatem tak prezentuje się kod zoptymalizowanego algorytmu metody eliminacji Gaussa dla macierzy opisanych w problemie:

Algorithm 2 Zoptymalizowana metoda eliminacji Gaussa

```
1:  procedure ZoptymalizowanaEliminacjaGaussa(A, b)
2:      n ← A.size
3:      l ← A.block_size
4:      for k ← 1 to n − 1 do                          ▷ Iteracja po kolumnach
5:          bottom_index ← (k ÷ l + 1) · l
6:          for i ← k + 1 to bottom_index do           ▷ Iteracja po wierszach
7:              I ← A[i][k] / A[k][k]                  ▷ Współczynnik eliminacji
8:              A[i[k] ← 0
9:              column_index ← min(n, l + k)
10:             for j ← k + 1 to column_index do       ▷ Aktualizacja macierzy
11:                 A[i][j] ← A[i][j] − I · A[k][j]
12:             end for
13:             b[i] ← b[i] − I · b[k]                 ▷ Aktualizacja wektora prawych stron
14:         end for
15:     end for
16: end procedure
```

Złożoność obliczeniowa tego algorytmu wynosi $O(l^2 \cdot (n-1)) = O(n)$ (l jest stałą). Jest to prawdą, ponieważ pętla k wykonuje się $n - 1$ razy, pętla i wykona się maksymalnie l razy, a pętla j - również maksymalnie l razy. Zredukowaliśmy zatem złożoność tego problemu z $O(n^3)$ do $O(n)$.

4 Częściowy wybór elementu głównego dla metody eliminacji Gaussa

Ważnym problemem numerycznym podczas korzystania z metody eliminacji Gaussa są zerowe lub małe wartości bezwzględne występujące na przekątnej. Spowodowane jest to koniecznością obliczania współczynników I_{ik}, gdzie otrzymując mały co do wartości (lub zerowy) współczynnik a_{kk} dochodzi do błędów obliczeniowych (lub błędu dzielenia przez zero).

W celu uniknięcia tego problemu, korzystamy z częściowego wyboru elementu głównego i szukamy w każdej kolumnie pod diagonalą elementu maksymalnego, który umieścimy na przekątnej. W skrócie, szukamy takiego elementu, że:

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

a następnie przestawiamy w macierzy $A^{(k)}$ wiersz p-ty z k-tym oraz element p-ty z k-tym w wektorze $b^{(k)}$.

W mojej implementacji tej metody, wszelkie permutacje wierszy zapamiętywałem w tablicy o rozmiarze n. Samo szukanie elementu maksymalnego w pojedynczej kolumnie to (korzystając ze zoptymalizowanego algorytmu) maksymalnie l iteracji, zatem złożoność obliczeniowa nie ulega zmianie.

Ważną natomiast zmianą, jest sposób liczenia górnego ograniczenia, jeżeli chodzi o zakres najbardziej zagnieżdżonej pętli. Przytoczmy tu ponownie przykład macierzy $\mathbf{M}$:

$$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ 0 & 0 & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & 0 & 0 & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Zauważmy, że niezerowy współczynnik jest oddalony od współczynnika na przekątnej o maksymalnie l pozycji (np. a_{33} oraz b_{63}). Zamieńmy zatem wiersz trzeci z szóstym:

$$\mathbf{M} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Wcześniej, pierwszy od prawej indeks kolumny j-tej, taki że dla k-tego wiersza współczynnik a_{kj} jest niezerowy, wynosił $\min(n, l+k)$. W tym przypadku natomiast, przy zamianie ze sobą wierszy, musimy zwiększyć ten zakres. Po powyższej przykładowej zamianie, znaleziony indeks w trzecim wierszu równa się 9, zamiast 6. Zatem zauważmy, że największa taka zmiana może wynieść dokładnie l pozycji.

Wniosek: nowy wzór na *column_index* jest równy $\min(n, 2 \cdot l + k)$.

Tak prezentuje się pseudokod tego algorytmu:

Algorithm 3 Zoptymalizowana metoda eliminacji Gaussa z częściowym wyborem

```
1: procedure ZoptymalizowanyCzęściowyWybórGauss(A, b)
2:     n ← A.size
3:     l ← A.block_size
4:     P ← [1, 2, ..., n]                                  ▷ Tablica permutacji wierszy
5:     for k ← 1 to n − 1 do
6:         bottom_index ← (k ÷ l + 1) · l
7:         index_found = k
8:         for i ← k + 1 to bottom_index do                ▷ Szukanie elementów głównych
9:             if |A[P[i]][k]| > |A[P[index_found]][k]| then
10:                index_found = i
11:            end if
12:        end for
13:        if index_found ≠ k then                         ▷ Zamiana wierszy
14:            swap(P[k], P[index_found])
15:        end if
16:        for i ← k + 1 to bottom_index do
17:            I ← A[P[i]][k] / A[P[k]][k]
18:            A[P[i]][k] ← 0
19:            column_index ← min(n, 2 · l + k)            ▷ Zmodyfikowany sposób
20:            for j ← k + 1 to column_index do
21:                A[P[i]][j] ← A[P[i]][j] − I · A[P[k]][j]
22:            end for
23:            b[P[i]] ← b[P[i]] − I · b[P[k]]
24:        end for
25:    end for
26: end procedure
```

Złożoność powyższej procedury to również $O(n)$ z uwagi na fakt, że algorytm rozbudowaliśmy o pętlę szukającą elementów głównych, co nie zmienia złożoności w rozumieniu notacji dużego O. Algorytm został również zmodyfikowany, jeżeli chodzi o zakres najbardziej zagnieżdżonej pętli, lecz modyfikacja ta zwiększyła liczbę wykonywanych operacji o maksymalnie l, co również pozostawia nas z wynikiem $O(n)$.

5 Rozkład LU

Warto zauważyć, że wykonując $(k+1)$-szy krok algorytmu Gaussa, czyli przechodząc z macierzy $A^{(k)}$ do $A^{(k+1)}$ i od wektora $b^{(k)}$ do $b^{(k+1)}$, możemy ten krok zapisać jako:

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

gdzie:

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

Można zatem spostrzec, że macierz górnotrójkątna $A^{(n)}$ powstaje w $n-1$ krokach i można ją opisać wzorem $A^{(n)} = L^{(n-1)} \dots L^{(2)} L^{(1)} A^{(1)}$. Oznaczmy $A^{(n)}$ jako macierz U oraz zauważmy, że $A^{(1)} = A$. Wynika z tego, że: $A = {L^{(1)}}^{-1} {L^{(2)}}^{-1} \dots {L^{(n-1)}}^{-1} U$. Ostatnim spostrzeżeniem, jest fakt, iż:

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

Zatem rozkład LU macierzy A można przedstawić jako iloczyn macierzy dolnotrójkątnej L i górnotrójkątnej U:

$A = LU$, gdzie:

$$L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ I_{21} & 1 & 0 & \dots & 0 \\ I_{31} & I_{32} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I_{n1} & I_{n2} & I_{n3} & \dots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Chcąc efektywnie posługiwać się rozkładem LU macierzy A w obliczeniach komputerowych, najlepiej jest przechowywać obie te macierze jako macierz o postaci:

$$LU = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ I_{21} & u_{22} & u_{23} & \dots & u_{2n} \\ I_{31} & I_{32} & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ I_{n1} & I_{n2} & I_{n3} & \dots & u_{nn} \end{bmatrix}$$

Jest to możliwe, jeżeli pamiętamy, że na przekątnej macierzy L znajdują się same jedynki!

Mając na uwadze specyfikę macierzy opisanej w naszym problemie, możemy odpowiednio dostosować algorytm do tworzenia takiego rozkładu LU, który przedstawiłem poniżej:

Algorithm 4 Tworzenie rozkładu LU dla macierzy A

```
1:  procedure RozkładLU(A)
2:      n ← A.size
3:      l ← A.block_size
4:      for k ← 1 to n − 1 do
5:          bottom_index ← (k ÷ l + 1) · l
6:          for i ← k + 1 to bottom_index do
7:              I ← A[i][k] / A[k][k]
8:              A[i][k] ← I                ▷ Zamiast zerować pozycję, ustawiamy na jej miejsce I_ik
9:              column_index ← min(n, l + k)
10:             for j ← k + 1 to column_index do
11:                 A[i][j] ← A[i][j] − I · A[k][j]
12:             end for
13:         end for
14:     end for
15: end procedure
```

Sama procedura jest niemal identyczna co do procedury podstawowej wersji eliminacji Gaussa i ma złożoność obliczeniową $O(n)$, lecz ważną różnicą jest fakt, że:

1. Wykonujemy $A[i][k] \leftarrow I_{ik}$, zamiast $A[i][k] \leftarrow 0$
2. Nie modyfikujemy w żaden sposób wektora prawych stron

Znając rozkład $A = LU$, możemy sprowadzić zadanie rozwiązania $Ax = b$ do dwóch układów trójkątnych:

$$Ly = b \text{ i } Ux = y$$

Zacznijmy od rozwiązania układu równań $Ly = b$. W przeciwieństwie do metody eliminacji Gaussa, w tym przypadku mamy do czynienia z macierzą dolnotrójkątną, zatem musimy skorzystać z metody substytucji wstecznej, ale zaczynając od 1-szego wiersza. Rozwiązywany układ równań Y można przedstawić w takiej formie:

$$Y = \begin{array}{cccccc} u_{11}y_1 & 0 & 0 & \dots & 0 & = b_1 \\ I_{21}y_1 & u_{22}y_2 & 0 & \dots & 0 & = b_2 \\ I_{31}y_1 & I_{32}y_2 & u_{33}y_3 & \dots & 0 & = b_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ I_{n1}y_1 & I_{n2}y_2 & I_{n3}y_3 & \dots & u_{nn}y_n & = b_n \end{array}$$

Łatwo jest zauważyć, że $y_1 = \frac{b_1}{u_{11}}$ Chcąc obliczyć y_k, możemy zastosować wzór:

$$y_k = \frac{b_k - \sum_{j=1}^{k-1} I_{kj}y_j}{u_{kk}}, \text{ dla } k \in \{2, \dots, n\}$$

Należy jednak pamiętać, że na diagonali mamy tak naprawdę do czynienia z jedynkami, zatem $y_1 = b_1$, a powyższy wzór możemy zmodyfikować do:

$$y_k = b_k - \sum_{j=1}^{k-1} I_{kj}y_j, \text{ dla } k \in \{2, \dots, n\}$$

Nie każde jednak I w naszym układzie równań Y jest różne od zera. W danej kolumnie k, te współczynniki są niezerowe na przedziale wierszy od $(k+1)$-go do $bottom\_index$ $(= (k \div l + 1) \cdot l)$. Natomiast jak to wygląda w kontekście jednego wiersza?

Zauważmy, że dla pierwszych l wierszy, wszystkie współczynniki I są niezerowe. Dla wierszy od $l+1$ do $2 \cdot l$ pierwszy niezerowy współczynnik znajduje się w kolumnie l-tej. Stąd wniosek, żeby poszukiwany od lewej, pierwszy, niezerowy współczynnik I był w kolumnie $left\_index = max(1, ((k-1) \div l) \cdot l)$.

Zatem algorytm do rozwiązywania układu równań $Ly = b$ wygląda w ten sposób:

Algorithm 5 Rozwiązywanie układu równań $Ly = b$

```
1:  function SolveLy(L, b)
2:      n ← A.size
3:      l ← A.block_size
4:      y[n] ← initialize array
5:      y[1] ← b[1]
6:      for k ← 2 to n do
7:          difference ← 0.0
8:          left_index ← max(1, ((k − 1) ÷ l) · l)
9:          for j ←left_index to k − 1 do
10:             difference + = L[k][j] · y[j]
11:         end for
12:         y[k] = b[k] − difference
13:     end for
14:     return y
15: end function
```

Pętla obliczająca zmienną *difference* wykonuje się maksymalnie l razy, zatem cały algorytm ma złożoność $O(n)$.

Obliczyliśmy układ równań $Ly = b$, zatem jesteśmy teraz w stanie przejść do kolejnego układu równań, jakim jest $Ux = y$. W tym przypadku sposób obliczania wektora x jest identyczny co do tego stosowanego przy metodzie eliminacji Gaussa, ponieważ znowu mamy do czynienia z macierzą górnotrójkątną U. Zatem, ponownie stosujemy technikę substytucji wstecznej, obliczając rozwiązania od x_n do x_1. Oto pseudokod algorytmu do rozwiązywania układów równań $Ux = y$ (z którego korzystałem przy poprzedniej metodzie Gaussa):

Algorithm 6 Rozwiązywanie układu równań $Ux = y$

```
1:  function SolveUx(U, y)
2:      n ← A.size
3:      l ← A.block_size
4:      x[n] ← initialize array
5:      x[n] ← y[n]/U[n][n]
6:      for k ← n − 1 down to 1 do
7:          difference ← 0.0
8:          column_index ← min(n, l + k)
9:          for j ← k + 1 to column_index do
10:             difference += U[k][j] · x[j]
11:         end for
12:         x[k] = (y[k] − difference) / U[k][k]
13:     end for
14:     return x
15: end function
```

Złożoność obliczeniowa tego algorytmu to $O(n)$ (jako, że wewnętrzna pętla wykonuje się maksymalnie l razy). **Zatem obliczenie obu układów równań jest złożoności $O(2n) = O(n)$.**

6 Częściowy wybór elementu głównego dla rozkładu LU

Tak samo jak dla metody eliminacji Gaussa z częściowym wyborem, chcemy w podobny sposób zmodyfikować algorytm tworzący rozkład LU dla wybranej macierzy. Ponownie wszelkie permutacje wierszy zapamiętujemy w tablicy n-elementowej.

Pierwszą różnicą między algorytmem dla rozkładu LU a dla metody eliminacji Gaussa, jest fakt, iż nie zerujemy pozycji pod diagonalą, a zastępujemy je współczynnikami I. Po drugie, nie modyfikujemy w żaden sposób wektora prawych stron i zwracamy tablicę permutacji. Oto pseudokod:

Algorithm 7 Zoptymalizowany rozkład LU z częściowym wyborem

```
1:  function ZoptymalizowanyCzęściowyWybórLU(A)
2:      n ← A.size
3:      l ← A.block_size
4:      P ← [1, 2, ..., n]                                  ▷ Tablica permutacji wierszy
5:      for k ← 1 to n − 1 do
6:          bottom_index ← (k ÷ l + 1) · l
7:          index_found = k
8:          for i ← k + 1 to bottom_index do                ▷ Szukanie elementów głównych
9:              if |A[P[i]][k]| > |A[P[index_found]][k]| then
10:                 index_found = i
11:             end if
12:         end for
13:         if index_found ≠ k then                         ▷ Zamiana wierszy
14:             swap(P[k], P[index_found])
15:         end if
16:         for i ← k + 1 to bottom_index do
17:             I ← A[P[i]][k] / A[P[k]][k]
18:             A[P[i]][k] ← I                              ▷ Tworzenie tablicy L
19:             column_index ← min(n, 2 · l + k)
20:             for j ← k + 1 to column_index do
21:                 A[P[i]][j] ← A[P[i]][j] − I · A[P[k]][j]
22:             end for
23:         end for
24:     end for
25:     return P
26: end function
```

Złożoność tego algorytmu wynosi $O(n \cdot (l + l^2)) = O(n)$, gdyż l jest stałą.

Tak samo jak wcześniej, mamy do rozwiązania dwa układy równań, a mianowicie:

$$Ly = Pb \text{ oraz } Ux = y$$

Najpierw przedstawię poniżej pseudokod algorytmu dla $Ly = Pb$:

Algorithm 8 Rozwiązywanie układu równań $Ly = Pb$

```
1:  function SolveLyPb(L, b, P)
2:      n ← A.size
3:      l ← A.block_size
4:      y[n] ← initialize array
5:      y[1] ← b[P[1]]
6:      for k ← 2 to n do
7:          difference ← 0.0
8:          left_index ← max(1, ((k − 1) ÷ l) · l − l)
9:          for j ←left_index to k − 1 do
10:             difference + = L[P[k]][j] · y[j]
11:         end for
12:         y[k] = b[P[k]] − difference
13:     end for
14:     return y
15: end function
```

Porównując ten algorytm do algorytmu służącego do rozwiązywania układu równań $Ly = b$, odwołując się do wektora prawych stron, korzystamy tutaj z tablicy permutacji P i za jej pomocą odwołujemy się do odpowiedniej wartości. Tak samo przy liczeniu zmiennej *difference*, szukane wartości znajdują się w $P[k]$ wierszu (nie k-tym).

Zmodyfikowałem również sposób liczenia zmiennej *left_ index* i postanowiłem zmniejszyć jej wartość do $max(1, ((k-1) \div l) \cdot l - l)$, gdyż przy zmianie wierszy, liczba niezerowych współczynników I w danym wierszu może wzrosnąć o rozmiar bloku l.

Układ równań $Ux = y$ można rozwiązać w taki sposób:

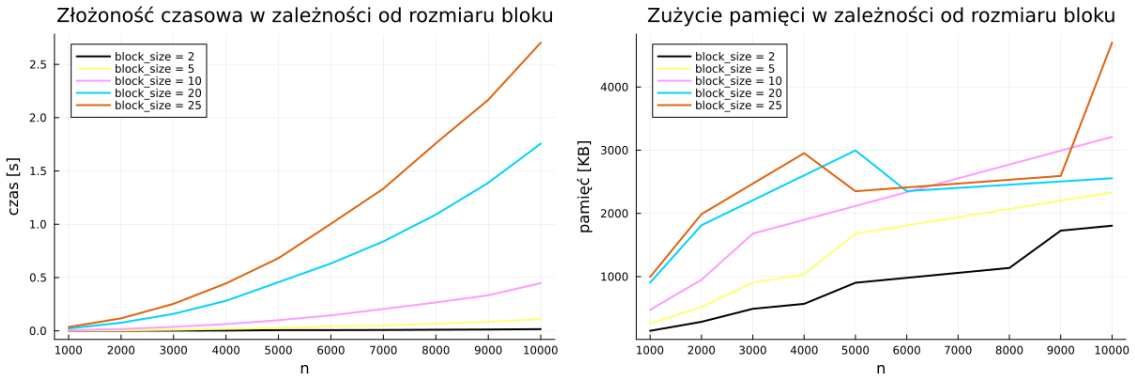
Algorithm 9 Rozwiązywanie układu równań $Ux = y$

```
1:  function PSolveUx(U, y, P)
2:      n ← A.size
3:      l ← A.block_size
4:      x[n] ← initialize array
5:      x[n] ← y[n]/U[P[n]][n]
6:      for k ← n − 1 down to 1 do
7:          difference ← 0.0
8:          column_index ← min(n, 2 · l + k)
9:          for j ← k + 1 to column_index do
10:             difference + = U[P[k]][j] · x[j]
11:         end for
12:         x[k] = (y[k] − difference) / U[P[k]][k]
13:     end for
14:     return x
15: end function
```

W porównaniu do rozkładu LU bez wyboru elementu głównego, otrzymujemy większą liczbę operacji, ale jedynie o stałą, zatem złożoność nadal jest wielkości $O(n)$.

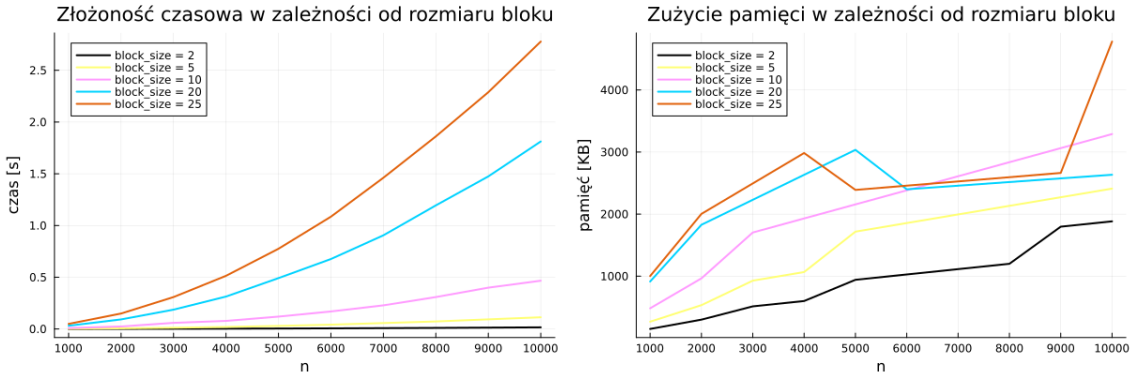
7 Eksperymenty i wyniki

Swoje rozważania rozpocząłem od wpływu rozmiaru bloku l na złożoność czasową oraz pamięciową działania algorytmów. Poniżej zamieściłem wykresy dla każdego poszczególnego algorytmu:



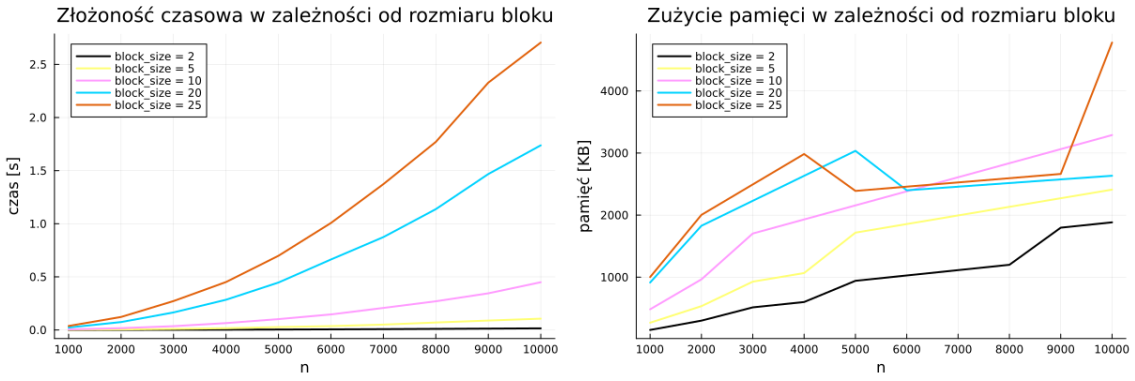
(a) (b)

Rysunek 1: Eliminacja Gaussa bez wyboru



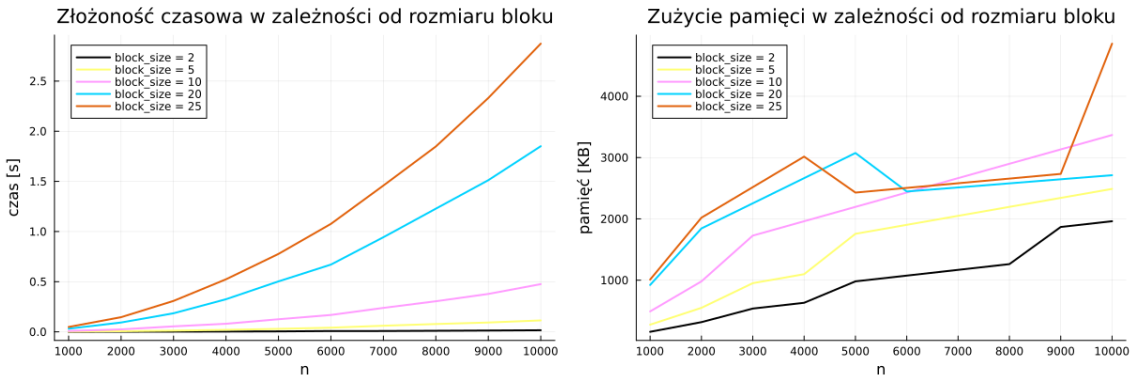
(a) (b)

Rysunek 2: Eliminacja Gaussa z częściowym wyborem



(a) (b)

Rysunek 3: Rozkład LU bez wyboru

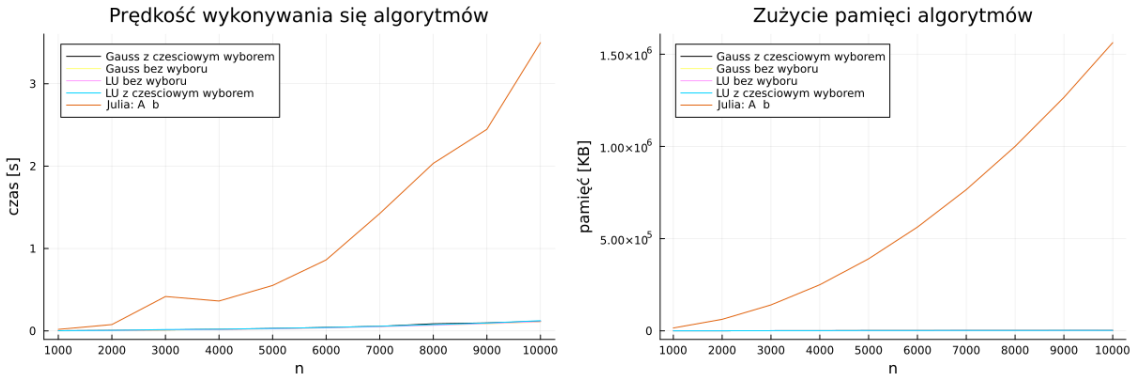


(a) (b)

Rysunek 4: Rozkład LU z częściowym wyborem

Wnioski: Jak można zauważyć, im większy rozmiar bloku l, tym więcej czasu wykonuje się algorytm oraz więcej pamięci potrzebuje. Nie jest to zaskoczeniem z racji, że zoptymalizowane przeze mnie programy zależne są od stałej l i im ta liczba jest większa, tym więcej iteracji jest potrzebnych do zakończenia przez program działania. Zapotrzebowanie pamięciowe wzrasta, gdyż zwiększa się liczba niezerowych pozycji w tworzonej macierzy.

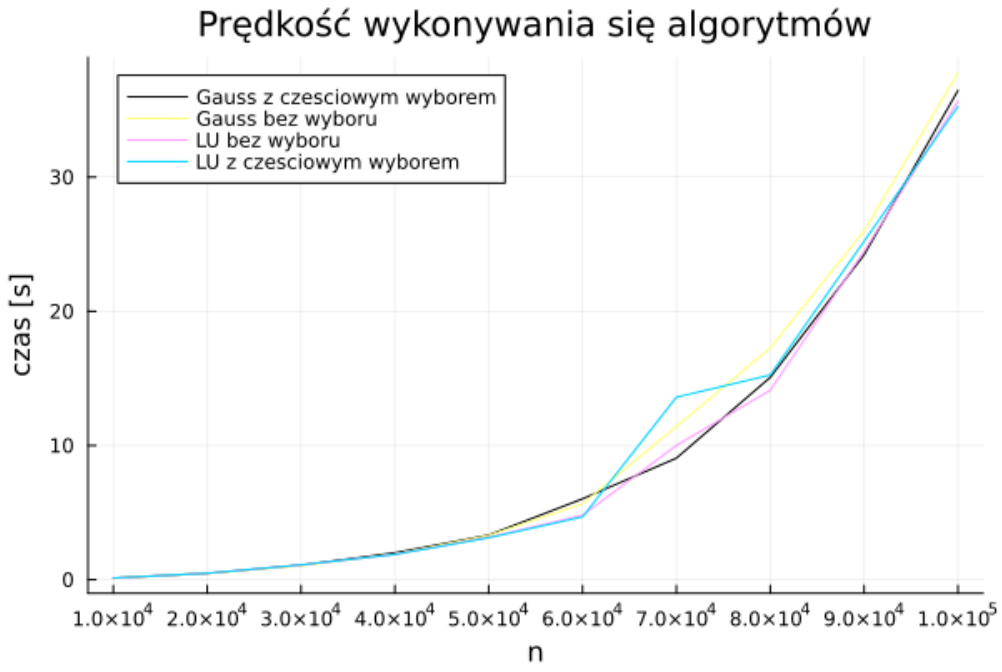
Do dalszych testów za rozmiar bloku przyjąłem $l = 5$, a za rozmiar macierzy $n \in \{10000, 20000, \ldots, 100000\}$. Na początku porównałem złożoność czasową i pamięciową wszystkich algorytmów z uwzględnieniem niezoptymalizowanego sposobu $A \setminus b$:



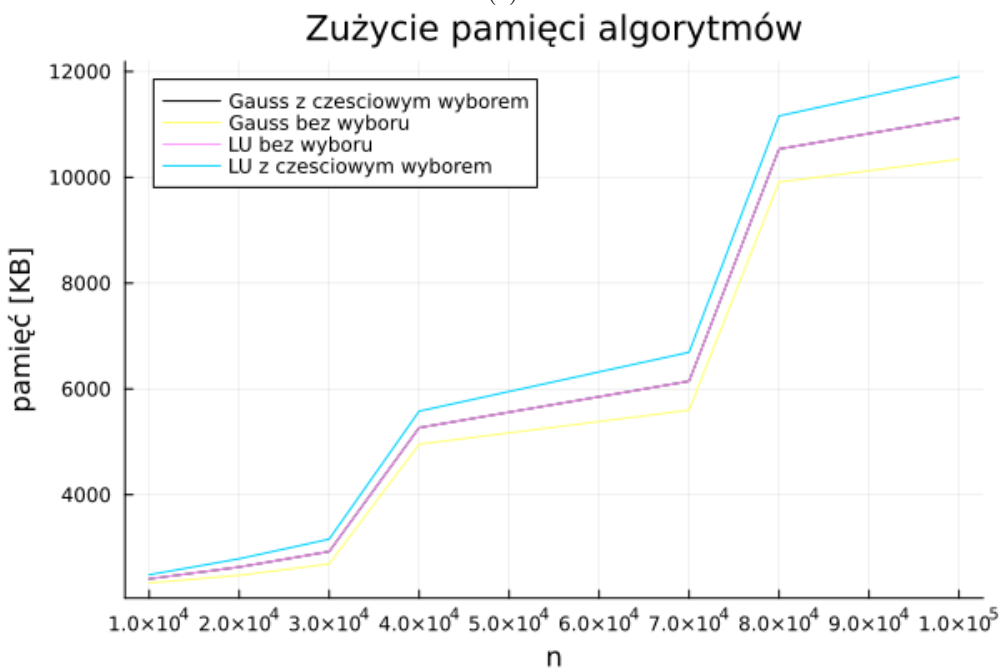
(a) (b)

Rysunek 5: Porównanie działania algorytmów z niezoptymalizowanym $A \setminus b$

Od razu można zauważyć ogromną różnicę. Zaadaptowanie algorytmów do specyficznej budowy macierzy znacząco zminimalizowało czas, jak i pamięć potrzebną do wykonania programów. Tak natomiast prezentują się wyniki dla czterech zoptymalizowanych algorytmów:



(a)



(b)

Rysunek 6: Porównanie działania zoptymalizowanych algorytmów

Wszystkie cztery metody do rozwiązania układu równań są bardzo podobne do siebie pod względem złożoności czasowej i pamięciowej. Zakładaliśmy, że wykonują się one w czasie $O(n)$ pod warunkiem, iż wszystkie operacje na macierzy wykonują się w czasie $O(1)$. Wiadome jest jednak, że operacje na strukturach *SparseArrays* kosztują więcej czasu, przez co nie otrzymujemy wykresu funkcji liniowej.

Można natomiast zauważyć tę liniowość w odniesieniu do zużytej pamięci podczas korzystania z każdego algorytmu. Algorytmy wykorzystujące częściowy wybór elementu głównego korzystają z większej ilości pamięci ze względu na potrzebę przechowywania tablicy permutacji. Algorytmy z rozkładem LU potrzebują więcej pamięci, lecz raz obliczony rozkład może zostać użyty do różnych wektorów prawych stron.