Obliczenia naukowe lista nr 5

Sebastian Woźniak 268491

January 2024

1 Opis problemu

Potrzebny jest moduł posiadający funkcje do efektywnego rozwiązywania układu równań liniowych $Ax = b$, dla danej macierzy współczynników $A \in \mathbb{R}^{n \times n}$ i wektora prawych stron $b \in \mathbb{R}^n$, $n \geqslant 4$. Z powodu dużej ilości zer w macierzy A oraz jej specyficznej strukturze, tj.:

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

gdzie $v = \frac{n}{l}$ i l jest rozmiarem wszystkich kwadratowych macierzy wewnętrznych, będzie potrzebna specjalna struktura optymalizująca wykorzystanie pamięci.

2 Struktura MyMatrix

Struktura służy do efektywnego pamieciowo przechowywania macierzy A oraz związanych z nią parametrów:

```
struct MyMatrix:
    n - rozmiar macierzy A
    l - rozmiar wewnętrznych macierzy A_k, B_k, C_k
    v - liczba wewnętrznych macierzy
    A - v-elementowa tablica bloków wewnętrznych A_k
    B - (v-1)-elementowa tablica bloków wewnętrznych B_k
    C - (v-1)-elementowa tablica bloków wewnętrznych C_k
```

Tablica B oraz C są w rzeczywistości v-elementowe jednak istotne bloki znajdują się odpowiednio pod indexami 2:v oraz 1:v-1. Dzięki takiemu zapisowi jesteśmy w stanie uzyskać złożoność pamięciową $O(v \cdot l^2)$ co sprowadza się do $O(n)$, gdy l jest stałą.

2.1 Indeksowanie elementów macierzy

Aby ułatwić operacje na strukturze powstała funkcja obliczająca index wartości w jednej z tablic należących do struktury $MyMatrix$ oraz jej rząd oraz kolumne wewnątrz danego bloku wewnętrznego, przyjmując rzeczywisty numer wiersza oraz kolumny w macierzy o rozmiarze $n \times n$.

Algorithm 1: priv_mymatrix_get_idx

Data: struct MyMatrix - m, column number - i, row number - j
Result: block column - iv, block row - jv,
inside block column number - il, inside block row number - jl

1 iv = $\lfloor \frac{i-1}{m.l} \rfloor + 1$
2 jv = $\lfloor \frac{j-1}{m.l} \rfloor + 1$
3 il = $(i-1) \mod m.l + 1$
4 jl = $(j-1) \mod m.l + 1$
5 return iv, jv, il, jl

Z obliczonymi wartościami możemy odczytać lub przypisać wartość do macierzy MyMatrix wiedząc, że gdy $iv = jv$ to $A[i, j]$ dla kwadratowej macierzy A odpowiada $MyMatrix.A[iv][il, jl]$. Gdy $iv = jv + 1$ to $MyMatrix.B[jv][il, jl]$, a dla $iv + 1 = jv$, $MyMatrix.C[iv][il, jl]$. W każdym innym przypadku wartością pod danym indexem będzie 0 (nie jest zapisana w strukturze MyMatrix). Pozwala to na prostą implementacji funkcji mymatrix_set oraz mymatrix_get.

2.2 Przestawianie rzędów

Dla wariantów algorytmów, w których odbywa się częściowy wybór elementu głównego potrzebna jest funkcja umożliwiająca przestawienie wybranych rzędów. Znając specyfikacje macierzy oraz zakres numerów rzędów wystarczającą implementacją do API struktury MyMatrix będzie:

Algorithm 2: mymatrix_swap_rows

```
Data: struct MyMatrix - m, row number - x, row number - y
Result: MyMatrix with swapped row x and y
1  xv, yv, xl, yl = priv_mymatrix_get_idx(m, x, y)
2  for j ← 1 to m.l do
3      if xv == yv then
4          swap(m.A[xv][xl, j], m.A[yv][yl, j])
5          if xv > 1 then
6              swap(m.B[xv - 1][xl, j], m.B[yv - 1][yl, j])
7          if yv < m.v then
8              swap(m.C[xv][xl, j], m.C[yv][yl, j])
9      else if yv == xv + 1 then
10         swap(m.A[xv][xl, j], m.B[yv - 1][yl, j])
11         if xv > 1 then
12             m.B[xv - 1][xl, j] = 0
13         if yv < m.v then
14             swap(m.C[xv][xl, j], m.A[yv][yl, j])
```

Iterując po elementach należących do rzędu w wewnętrznych strukturach z rzędu x podmieniamy je z elementami ze struktur wewnętrznych z rzędu y. Funkcja na wejściu zapewnia, że $x < y$.

3 Rozwiązywanie układu Ax = b metodą eliminacji Gaussa

Pierwszą częścią modułu jest funkcja, która przyjmuje macierz A zapisaną w strukturze MyMatrix oraz wektor prawych stron b i rozwiązuje równanie $Ax = b$ metodą eliminacji Gaussa. Do wyboru jest wariant z częściowym wyborem elementu głownego (b) oraz bez (a). Algorytm wyprowadza macierz górną trójkątną i jego standardowa wersja ma złożoność obliczeniową $O(n^3)$, jednak wykorzystując specjalną strukturę macierzy jesteśmy w stanie uzyskać złożoność $O(n)$.

3.1 Zmodyfikowany algorytm eliminacji Gaussa (a)

3.1.1 Opis i złożność obliczeniowa

Algorytm oblicza macierz górną trójkątną i następnie wykorzystuje ją do obliczenia wartości z wektora x.

Algorithm 3: gauss_elimination (a)

```
1  //Wyznaczenie macierzy górnej trójkątnej
2  for k ← 1 to A.n − 1 do
3      for i ← k + 1 to min(n, k + A.l) do
4          mult = mymatrix_get(A, i, k)/mymatrix_get(A, k, k) for
            j ← k + 1 to min(n, k + A.l) do
5              mymatrix_set(A, i, j,
6              mymatrix_get(A, i, j) - mult · mymatrix_get(A, k, j) )
7          b[i] -= mult · b[k]
8  //Obliczenie x
9  x[A.n] = b[n] / mymatrix_get(A, A.n, A.n)
10 for i ← A.n − 1 to 0 do
11     s = 0
12     for j ← i + 1 to min(n, i + 2 · A.l) do
13         s += mymatrix_get(A, i, j)*x[j]
14     x[i] = (b[i] - s)/mymatrix_get(A, i, i)
```

Na macierzy wykonywane są operacje mające na celu wyzerowanie elementów znajdujących się pod przekątną. W k-tej iteracji odejmujemy od rzędów $i > k$ wielokrotność *mult* rzędu k. Wiedząc, że wszystkie wartości dla rzędów $h > k + l$ można skrócić wewnętrzne pętle do max $O(l)$ iteracji. W ten sposób otrzymujemy macierz górną trójkątną w czasie $O(n \cdot l^2)$, czyli dla stałego l jest to $O(n)$. Następnie ze złożonością obliczeniową $O(n * l)$ odbywa się obliczenie wyniku x korzystając ze wzoru:

$$x_n = \frac{b_n}{A_{n,n}}$$
$$x_i = (b_i - \Sigma_{j=i+1}^{n} x_j)/a_{i,i}$$

Dzięki specyficznej budowie macierzy sprowadzonego do:

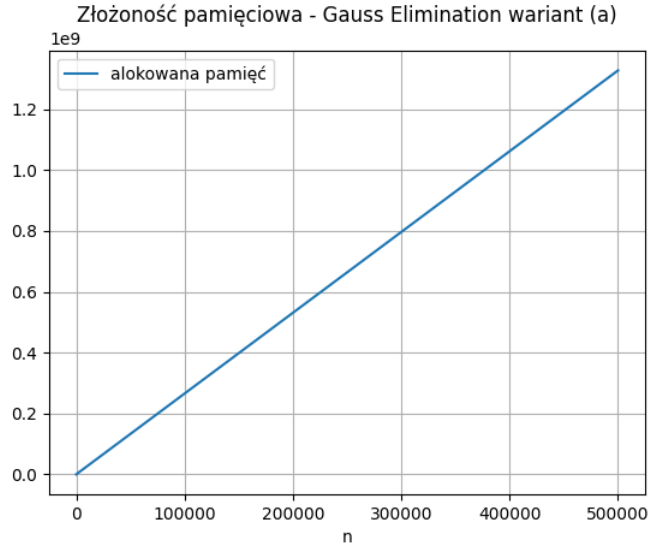
$$x_n = \frac{b_n}{A_{n,n}}$$
$$x_i = (b_i - \Sigma_{j=i+1}^{min(n,j+l)} x_j)/a_{i,i}$$

3.1.2 Testy

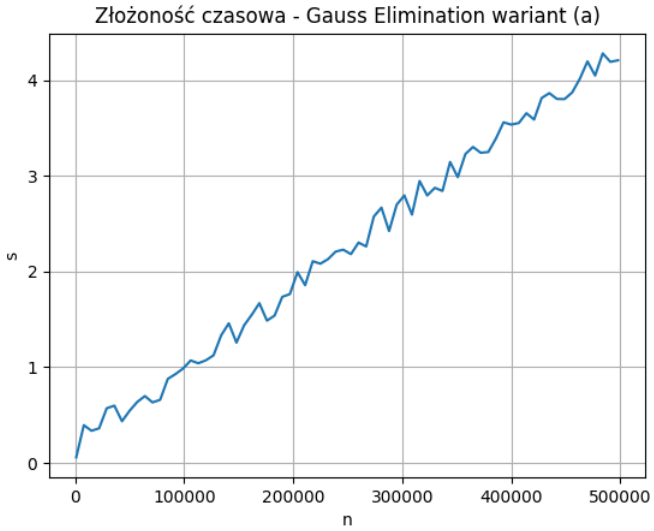
Rozmiar n macierzy A	Błąd względny $\hat{x}$
16	6.906622391776103e-15
50000	1.5393951981921677e-13
100000	4.681482376769146e-14
300000	5.066539517131064e-13
500000	1.2844098793303182e-13

Powyższa tabela przedstawia błąd względny uzyskany przy obliczaniu wektora prawych stron b dla różnych rozmiarów macierzy A, ze wzoru $b = Ax$, gdzie

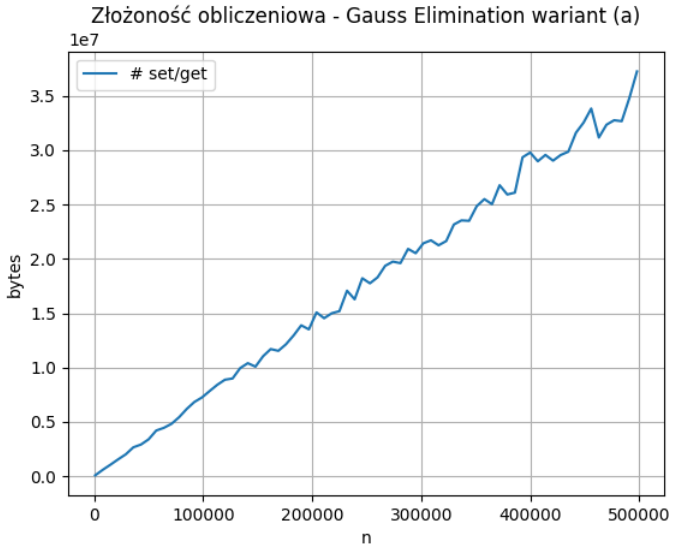
$x = (1, ..., 1)^T$.
Złożoność pamięciowa została obliczona makrem @timed w języku Julia, które zwraca informacje o alokowanej liczbie bajtów. Testy przeprowadzono dla macierzy A oraz wektora prawych stron b o rozmiarze od 1000 do 500000.



Widać, że zgodnie z przeprowadzonymi obliczeniami złożoność pamięciowa to $O(n)$.
Do obliczenia złożoności czasowej również zostało wykorzystane makro @timed.



Do obliczenia złożoności obliczenowej została uzyta liczba odwołań do elementów macierzy A:



Wszystkie wykresy potwierdzają liniową złożonośc obliczeniową oraz pamięciową algorytmu.

3.2 Zmodyfikowany algorytm eliminacji Gaussa (b)

3.2.1 Opis i złożoność obliczeniowa

Ten wariant algorytmu eliminacji Gaussa jest rozwinięty o częściowy wybór głównego elementu co pomaga rozwiązać problem wykonywania działań na bardzo małych wartościach na przekątnej macierzy co mogło powodować niedokładność w obliczeniach. Celem jest wyznacznie wiersza p, dla którego:

$$|A_{p,k}| = max_{k \leqslant i \leqslant n} |A_{i,k}|$$

W przypadku specyficznej budowy macierzy A wystarczy:

$$|A_{p,k}| = max_{k \leqslant i \leqslant min(n,k+l)} |A_{i,k}|$$

Co zmniejsza nam obszar poszukiwań maksymalnej wartości do nie więcej niż l wartości. Po znalezionu p następuje zamiana wierszy p oraz k w macierzy A oraz w wektorze prawych stron b. Implementacja algorytmu eliminacji Gaussa z częściowym wyborem elementu głównego:

Algorithm 4: gauss_elimination (b)

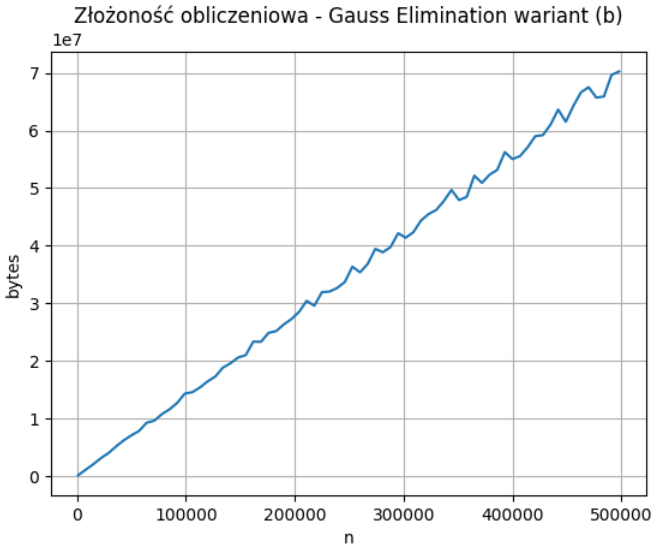
```
1  for k ← 1 to A.n − 1 do
2      //Wyznaczenie elementu głównego
3      main_elem_row = k
4      for i ← k + 1 to min(n, k + A.l) do
5          if —mymatrix_get(A, i, k)— ¿ —mymatrix_get(A,
            main_elem_row, k)— then
6              main_elem_row = i
7      mymatrix_swap_rows(A, k, main_elem_row)
8      swap(b[k], b[main_elem_row])
9      for i ← k + 1 to min(n, k + A.l) do
10         mult = mymatrix_get(A, i, k)/mymatrix_get(A, k, k) for
            j ← k + 1 to min(n, k + 2 · A.l) do
11             mymatrix_set(A, i, j,
12             mymatrix_get(A, i, j) - mult · mymatrix_get(A, k, j) )
13         b[i] -= mult · b[k]
```

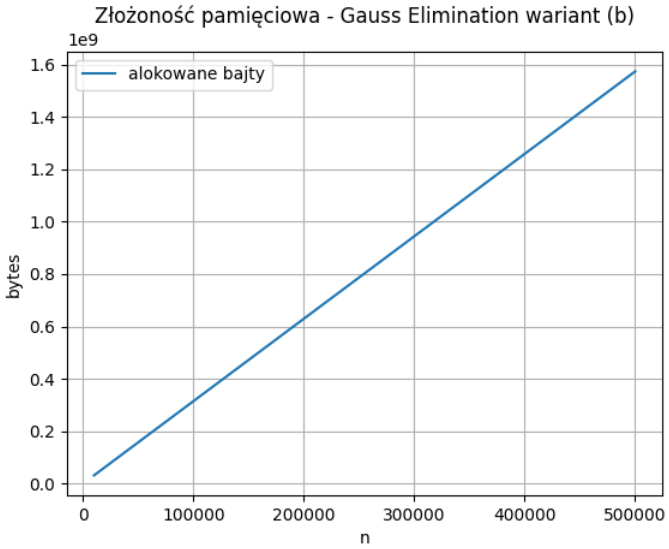
Dzięki skróceniu dodanej pętli uzyskano złożoność obliczeniową $O(n \cdot l^2)$ co przy stałej wartości l daje liniową złożoność $O(n)$. Wyznaczenie wartości wektora x przebiega w taki sam sposób jak w poprzedniej wersji algorytmu.

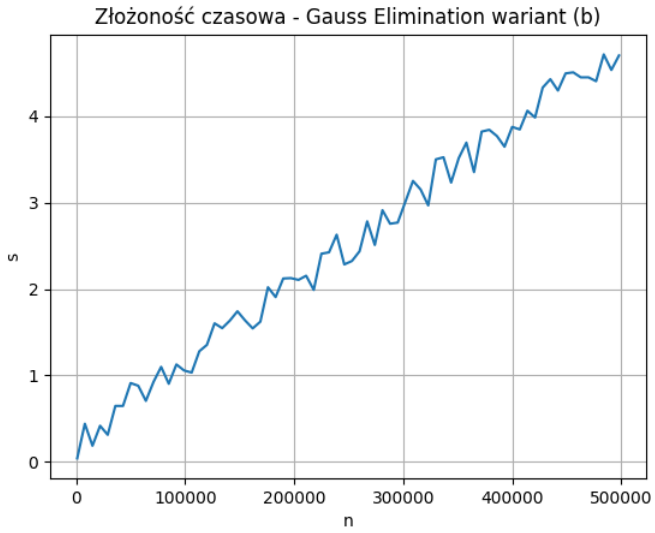
3.2.2 Testy

Poniższa tabela przedstawia błąd względny uzyskany przy obliczaniu wektora prawych stron b dla różnych rozmiarów macierzy A, ze wzoru $b = Ax$, gdzie $x = (1, ..., 1)^T$, porównane z rezultatmi dla poprzedniego wariantu. Widoczna jest poprawa pod względem błędu względnego, tzn. jest on mniejszy dla wariantu z częściowym wyborem elementu głównego.

Rozmiar n macierzy A	Błąd względny $\hat{x}$ wariant (a)	Błąd względny $\hat{x}$ wariant (b)
16	6.906622391776103e-15	3.744431865468923e-16
50000	1.5393951981921677e-13	4.057095823578191e-16
100000	4.681482376769146e-14	2.986658830105658e-16
300000	5.066539517131064e-13	4.006581448861896e-16
500000	1.2844098793303182e-13	3.983755283200343e-16


Złożoność obliczeniowa - Gauss Elimination wariant (b)
1e7
bytes
n
0
1
2
3
4
5
6
7
0
100000
200000
300000
400000
500000


Złożoność pamięciowa - Gauss Elimination wariant (b)
1e9
alokowane bajty
bytes
n
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0
100000
200000
300000
400000
500000



4 Rozkład LU

Podejście to polega na wyznacznie podziału macierzy $A = LU$ na macierz dolnotrójkątną L oraz górnotrójkątną U. Istnieje wtedy możliwośc wyznacznie rozwiązania poprzez rozwiązanie dwóch równań:

$$Lz = b$$
$$Ux = z$$

Wyznaczony rozkład prezentuje się następująco:

$$LU = \begin{bmatrix} A'_{1,1} & A'_{1,2} & A'_{1,3} & \dots & A'_{1,n} \\ l_{2,1} & A'_{2,2} & A'_{2,3} & \dots & A'_{2,n} \\ l_{3,1} & l_{3,2} & A'_{2,3} & \dots & A'_{2,n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & A'_{n,n} \end{bmatrix}, \tag{2}$$

W strukturze MyMatrix zapisywane są wszytskie niezerowe bloki $A'_{i,j}$ oraz mnożniki wykorzystane do ich uzyskania, a $l_{i,j}$ jest obliczane ze wzoru $l_{i,j} = \frac{A_{i,j}}{A_{j,j}}$. Zachowana jest więc złożoność pamięciowa $O(n)$ do przechowywania rozkładu LU.

4.1 Algorytm wyznaczania rozkładu - wariant (a)

4.1.1 Opis i złożoność obliczeniowa

Można zauważyc, że podczas poprzedniego algorytmu eliminacji Gaussa zostaje wyznaczona macierz górna trójkątna U. Możemy, więc z niego skorzystać wprowadzając jedynie zapisywanie mnożników w wektorze B struktury MyMatrix.

Algorithm 5: gauss_LU (a)

```
1 for k ← 1 to A.n − 1 do
2     for i ← k + 1 to min(n, k + A.l) do
3         mult = mymatrix_get(A, i, k)/mymatrix_get(A, k, k)
4         mymatrix_set(A, i, k, mult)
5         for j ← k + 1 to min(n, k + A.l) do
6             mymatrix_set(A, i, j,
7             mymatrix_get(A, i, j) - mult · mymatrix_get(A, k, j) )
```

Algorytm rózni się jedynie dodaniem jednego wywołanie mymatrix_set oraz usunięciem modyfikacji wektora prawych stron b w środkowej pętli. Złożnośc obliczniowa to $O(n \cdot l^2)$, czyli $O(n)$ dla stałego l.
Następnymi krokami do rozwiązania równania z użyciem rozkładu LU jest rozwiązanie równań $Lz = b$ oraz $Ux = z$. Wykorzystując zapisane wcześniej mnożniki wyznaczane są kolejne wartości wektora z poprzez obliczenie złożenia wszystkich przekształceń L.

$$L^{(n-1)^{-1}} \dots L^{(2)^{-1}} L^{(1)^{-1}} = L$$

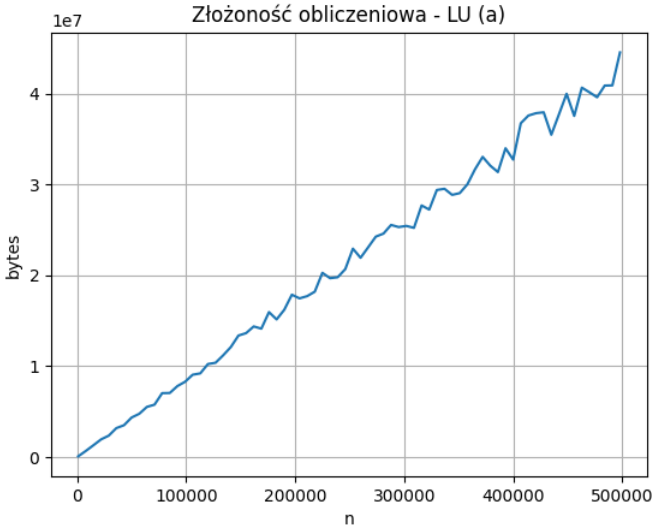
Algorithm 6: Rozwiązanie Lz = b

```
1 z = copy(b)
2 for k ← 1 to A.n − 1 do
3     for i ← k + 1 to min(n, k + A.l) do
4         z[i] = z[i] - z[k]*mymatrix_get(A, i, k)
```

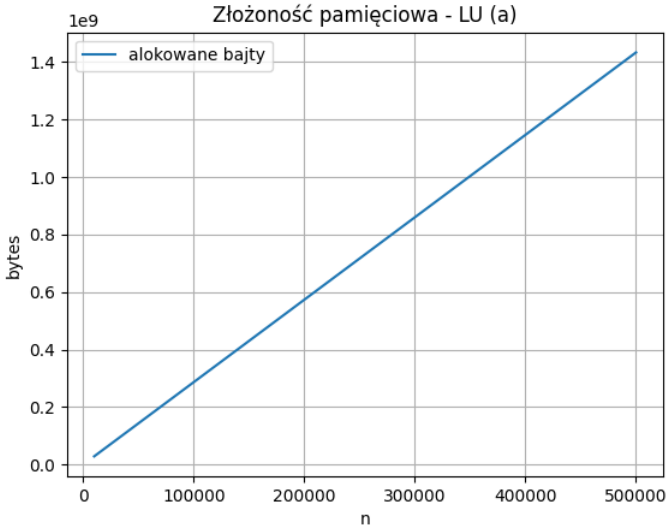
Z obliczonym z możemy użyc identycznego algorytmu jak w poprzedniej sekcji do rozwiązania $Ux = z$ zamiast $Ax = b$. Ten krok, więc również ma złożoność obliczeniową $O(n)$.

4.1.2 Testy

Rozmiar n macierzy A	Błąd względny $\hat{x}$
16	6.906622391776103e-15
50000	1.5393951981921677e-13
100000	4.681482376769146e-14
300000	5.066539517131064e-13
500000	1.2844098793303182e-13



/be



4.2 Algorytm wyznaczania rozkładu - wariant (b)

4.2.1 Opis i złożoność obliczeniowa

Tak jak w wariancie (b) dla algorytmu eliminacji Gaussa również przy wyznaczaniu rozkładu LU można zastosować częściowy wybór elementu głównego co pozwoli uniknąć działań na małych wartościach z przekątnej macierzy. Wybór odbywa się w identyczny sposób:

Algorithm 7: gauss_LU (b)

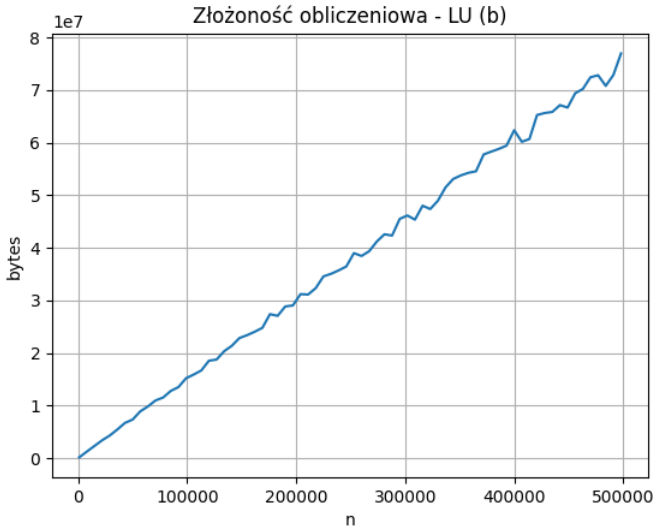
```
1  for k ← 1 to A.n − 1 do
2      main_elem_row = k
3      for i ← k + 1 to min(n, k + A.l) do
4          if —mymatrix_get(A, i, k)— ¿ —mymatrix_get(A,
            main_elem_row, k)— then
5              main_elem_row = i
6      mymatrix_swap_rows(A, k, main_elem_row)
7      swap(b[k], b[main_elem_row])
8      for i ← k + 1 to min(n, k + A.l) do
9          ...
10         for j ← k + 1 to min(n, k + 2 · A.l) do
11             ...
```

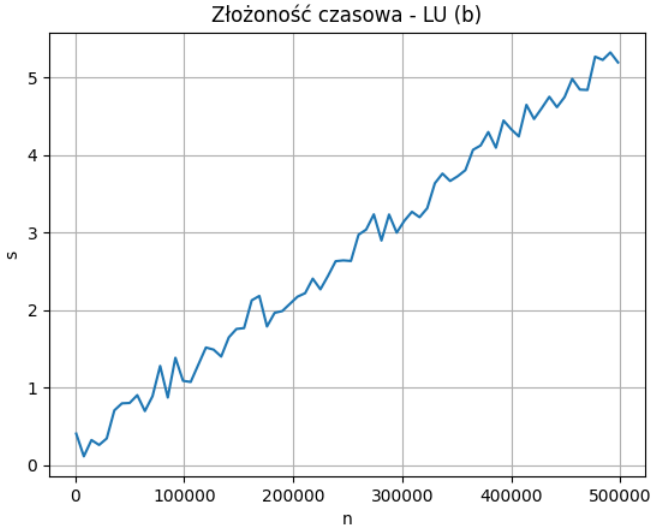
Dodanie wewnętrznej pętli o złożoności $O(l)$ nadal daje linową złożoność obliczeniową całego algorytmu $O(n)$ dla stałego l. Dalsze kroki pozostają bez zmian.

4.2.2 Testy

Tabela zawiera porównanie wariantu (a) z wariantem (b):

Rozmiar n macierzy A	Błąd względny $\hat{x}$ wariant (a)	Błąd względny $\hat{x}$ wariant (b)
16	6.906622391776103e-15	3.744431865468923e-16
50000	1.5393951981921677e-13	4.057095823578191e-16
100000	4.681482376769146e-14	2.986658830105658e-16
300000	5.066539517131064e-13	4.006581448861896e-16
500000	1.2844098793303182e-13	3.983755283200343e-16





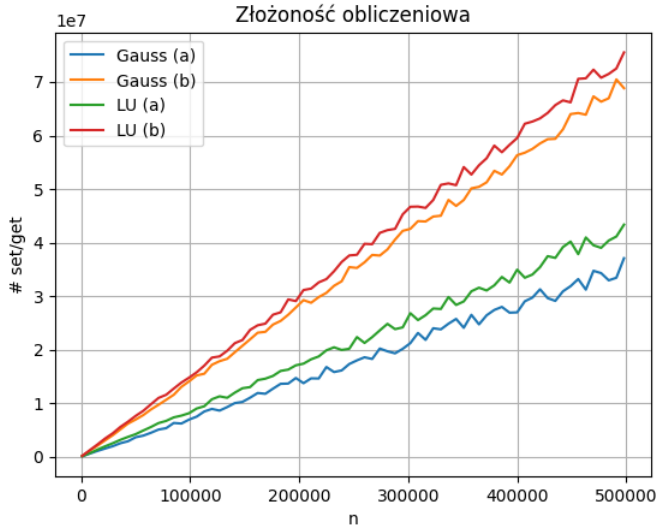
Złożoność pamięciowa jest niezmienna względem wariantu (a).

5 Podsuwowanie testów

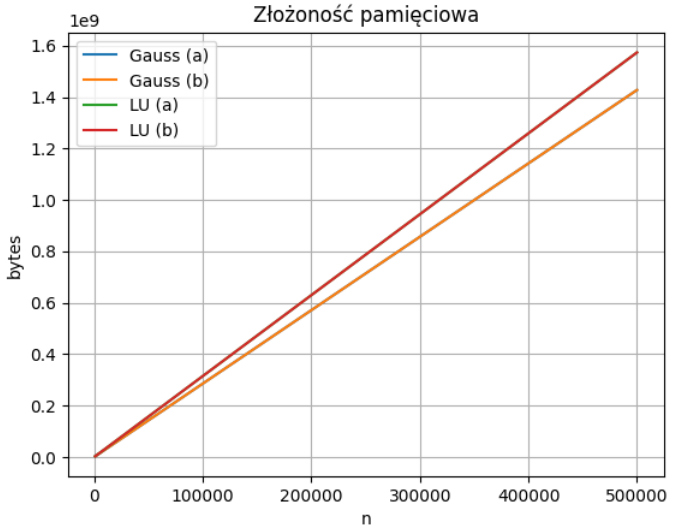
Do zdobycia danych zostały wykorzystane zawarte w module funkcje:

- **solve_and_save**
  Zapisuje w pliku tekstowym obliczone rozwiązanie x oraz błąd względny obliczonego b, gdzie macierz A była czytana z pliku, a wektor prawych stron b był obliczany $b = Ax$, gdzie $x = (1, ..., 1)^T$

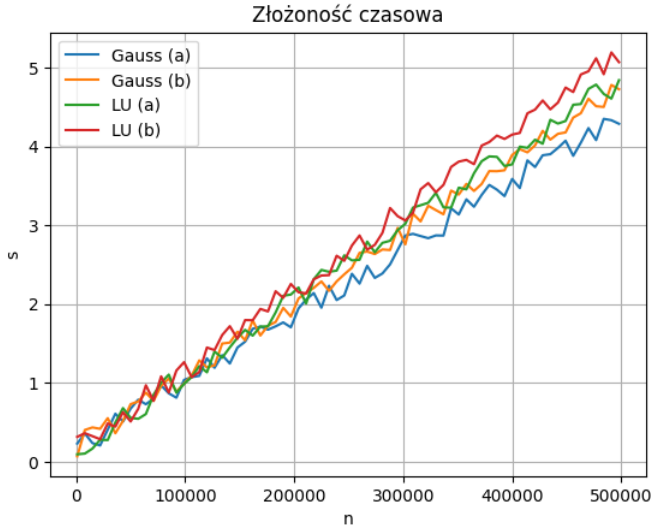
- **solve_and_save_2**
  Odczytuje z pliku A oraz b i zapisuje w pliku tekstowym rozwiązanie $Ax = b$ oraz błąd względny.



Jak widać oba warianty algorytmu z rozkładem LU wypadają lekko gorzej od odpowiadających wariantów rozwiązania poprzez eliminacje Gaussa. Pamiętając jednak rozkład LU dla macierzy A możliwe jest dużo szybsze obliczenie x dla różnych wektorów b. Warianty z wyborem elementu głównego wymagały ponad 2 razy więcej operacji set/get ze względu na zamiane rzędów macierzy.



Dzięki wykorzystaniu struktury MyMarix we wszystkich algorytmach osiągnięto złożoność pamięciową O(n). Algorytmy z rozkładem LU alokowały trochę wiecej bajtów na wektor y przy rozwiązywaniu $Ly = b$.



Również czas wykonania algorytmów rósł liniowo. Rozwiązywanie poprzez obliczenie rozkładu LU trwało dłużej jednak posiadając rozkład jesteśmy w stanie szybko obliczyć x z równania $Ax = b$, dla różnych wektorów prawych stron b bez ponownego wykonywania algorytmu eliminacji Gaussa. Oczywiście warianty z częściowym wyborem ze względu na dodatkowe operacje potrzebowały więcej czasu niż wariant podstawowy.

6 Wnioski

Odpowiednia analiza problemu i danych wejściowych może pozwolić na modyfikacje znanego algorytmu w sposób znacznie zwiększający wydajność. W tym przypadku udało się wykorzystać specyficzną budowę macierzy A do zejścia ze złożoności obliczeniowej $O(n^3)$ do $O(n \cdot l^2)$ co przy stałej wartości l zapewniało liniową złożoność. Również złożoność pamięciowa okazała się liniowa zamiast kwadratowej dzięki wykorzystaniu specjalnej struktury przechowującej tylko znaczące informacje o macierzy.