Optymalizacja algorytmów rozwiązywania układów równań liniowych

z wykorzystaniem efektywnych struktur danych

Ewa Kasprzak

6 stycznia 2025

1 Wstęp

W dzisiejszych badaniach z zakresu chemii kwantowej czy astronomii, szczególnie w dużych jednostkach badawczych, rozwiązywanie układów równań liniowych o macierzach o dużych rozmiarach stanowi istotne wyzwanie obliczeniowe.

Celem tej pracy jest rozważenie zagadnienia rozwiązywania układów równań postaci $Ax = b$, gdzie A jest macierzą rzadką o blokowej strukturze, z dominującą liczbą elementów zerowych. Ze względu na rozmiar macierzy, przechowywanie jej w standardowej postaci pełnej macierzy $n \times n$ oraz stosowanie tradycyjnych algorytmów staje się niewystarczające pod względem czasu obliczeń i zużycia pamięci. Zaprezentowane zostaną metody efektywnych obliczeń z uwzględnieniem rzadkości i blokowej struktury macierzy A. Struktura ta pozwala na oszczędności pamięciowe i redukcję złożoności obliczeniowej. Omówimy techniki reprezentacji macierzy, które przechowują tylko elementy niezerowe oraz adaptację algorytmów numerycznych, umożliwiających efektywne rozwiązanie problemu przy minimalnym zużyciu zasobów obliczeniowych.

2 Struktura macierzy

Macierz A jest macierzą blokową o wymiarze $n \times n$, gdzie n jest podzielne przez parametr ℓ, rozmiar poszczególnych bloków. Strukturę macierzy A można zapisać w postaci:

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

Gdzie:

1. **Bloki** A_k: Są to macierze gęste o wymiarach $\ell \times \ell$.

2. **Bloki** B_k: Macierze B_k mają szczególną strukturę, w której jedynie dwie ostatnie kolumny są niezerowe.

$$B_k = \begin{pmatrix} 0 & \cdots & 0 & b_{1\ell-1} & b_{1\ell} \\ 0 & \cdots & 0 & b_{2\ell-1} & b_{2\ell} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b_{\ell\ell-1} & b_{\ell\ell} \end{pmatrix}$$

3. **Bloki** C_k: Macierze C_k są diagonalne, co oznacza, że jedynie elementy na głównej przekątnej są różne od zera.

$$C_k = \begin{pmatrix} c_1 & 0 & 0 & \cdots & 0 \\ 0 & c_2 & 0 & \cdots & 0 \\ 0 & 0 & c_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & c_\ell \end{pmatrix}$$

Wszystkie te bloki są rozmieszczone w sposób regularny w macierzy A. Daje to blokową strukturę, a jednocześnie rzadką, ponieważ poza tymi elementami macierz zawiera dużą liczbę zer.

2.0.1 Możliwa implementacja

Taka struktura umożliwia przechowywanie macierzy w sposób bardziej efektywny pamięciowo niż tradycyjna tablica $n \times n$. Wystarczy zapamiętać jedynie poszczególne bloki, ponieważ podczas rozwiązywania układów równań nie będziemy korzystać z elementów, które nie należą do żadnego z bloków.

Jedną z możliwych implementacji jest przechowywanie bloków każdego typu w oddzielnych wektorach. Przykładowo:

- **Wektor bloków** A**:** składa się z obiektów, które, ze względu na swoją gęstość, muszą być przechowywane jako macierze o wymiarach $l \times l$. Jeśli zaś parametr l jest stały, rozmiar tych macierzy będzie również stały.
- **Wektor bloków** B**:** to wektor słowników, w których klucze odpowiadają współrzędnym elementów w oryginalnej macierzy, a wartości to elementy przechowywane w macierzy głównej, w komórkach wyznaczonych przez te współrzędne. Optymalniejszym rozwiązaniem byłoby przechowywanie tej struktury w dwuwymiarowej tablicy $2 \times l$, jednak mogłoby to utrudnić dostęp do elementów z bloku B, które będą potrzebne podczas rozwiązywania układu równań reprezentowanego przez tę macierz.
- **Wektor bloków** C**:** mógłby być wektorem jednowymiarowych tablic o długości l. Niemniej jednak, z tych samych powodów co przy blokach B, bloki C są przechowywane w pamięci jako słowniki.

Oprócz wektorów bloków, w strukturze macierzy przechowywane są również informacje o wymiarze wejściowej macierzy n oraz wymiarach bloków l.

2.0.2 Efektywność przechowywanej struktury

Implementacja tej struktury danych umożliwia przechowywanie macierzy wejściowej w sposób efektywny pamięciowo.

- Liczba elementów w każdym z typów bloków wynosi co najwyżej n/l.
- Bloki A wymagają pamięci rzędu $O(l^2)$.
- Bloki B również wymagają pamięci rzędu $O(l^2)$, choć w rzeczywistości ta złożoność dotyczy jedynie przypadków, w których l jest małe, np. $l = 2$. W średnim przypadku złożoność ta zbliża się raczej do $O(l)$.
- Bloki C także wymagają pamięci rzędu $O(l^2)$, a w średnim przypadku złożoność ta jest również bliższa $O(l)$.

Pesymistyczne przypadki dla bloków B i C dotyczą jednak macierzy, które mogą wystąpić po wykonaniu operacji na tych blokach. Początkowo bloki A zajmują $O(l^2)$ pamięci, a bloki B i C zajmują $O(l)$.

W sumie złożoność pamięciowa wynosi:

$$3 \cdot \left(\frac{n}{l}\right) \cdot O(l^2) = 3 \cdot n \cdot O(l^2),$$

gdzie, jeśli l jest stałe, otrzymujemy złożoność rzędu $O(n)$.

Zmniejszenie złożoności pamięciowej z $O(n^3)$ (wymaganej przy tradycyjnym przechowywaniu macierzy) do $O(n)$ stanowi znaczną poprawę, której efekty będą widoczne w poprawie wydajności zaimplementowanych algorytmów.

3 Wybrane zagadnienia algebry liniowej

3.1 Funkcja rozwiązująca układ równań $Ax = b$ metodą eliminacji Gaussa

1. **Bez wyboru elementu głównego:** Eliminacja Gaussa to klasyczna metoda rozwiązywania układów równań liniowych, polegająca na przekształceniu macierzy układu A do postaci górnotrójkątnej przy użyciu operacji elementarnych. W wariancie bez wyboru elementu głównego podczas eliminacji po prostu przekształcamy kolejne wiersze macierzy, eliminując zmienne za pomocą odpowiednich operacji na wierszach. W tej wersji założenie jest takie, że nie będziemy poszukiwać największych elementów w kolumnach, co może prowadzić do problemów numerycznych, zwłaszcza w przypadku macierzy źle uwarunkowanych.

```
1. Dla każdej kolumny k (od 1 do n-1):
    a. Dla każdego wiersza i (od k+1 do n):
        - Oblicz współczynnik m, który jest równy A[i,k] / A[k,k].
        - Zaktualizuj wiersz i, odejmując m razy wiersz k od wiersza i.
        - Zaktualizuj wektor b, odejmując m razy b[k] od b[i].

2. Po uzyskaniu macierzy trójkątnej górnej, rozwiązuj układ Ux = b:
```

```
   a. Rozpocznij od ostatniego wiersza (i = n):
       - Oblicz x[i] = b[i] / U[i,i].
   b. Następnie iteruj wstecz dla każdego wiersza i (od n-1 do 1):
       - Oblicz x[i] = (b[i] - suma(U[i,i+1:n] * x[i+1:n])) / U[i,i].
   c. Powtarzaj aż do i = 1, uzyskując rozwiązanie x.
```

2. **Z częściowym wyborem elementu głównego:** W tej wersji eliminacji Gaussa przed każdą operacją eliminacyjną wybieramy największy element (w sensie wartości bezwzględnej) w kolumnie jako element główny. Dzięki temu zmniejszamy ryzyko błędów numerycznych wynikających z niewielkich elementów, które mogą prowadzić do niestabilności obliczeniowych. Częściowy wybór elementu głównego polega na przestawianiu wierszy, aby element główny znajdował się na przekątnej. Ta wersja jest stabilniejsza numerycznie i pozwala uzyskać dokładniejsze wyniki.

```
1. Dla każdej kolumny k (od 1 do n-1):
   a. Znajdź wiersz p, gdzie element A[p,k] jest
       największy w wartościach bezwzględnych (p >= k).
   b. Zamień wiersze k i p w macierzy A oraz w wektorze b.
   c. Dla każdego wiersza i (od k+1 do n):
       - Oblicz współczynnik m = A[i,k] / A[k,k].
       - Zaktualizuj wiersz i: A[i, :] = A[i, :] - m * A[k, :].
       - Zaktualizuj wektor b: b[i] = b[i] - m * b[k].
2. Po uzyskaniu macierzy trójkątnej górnej, rozwiązuj układ
   Ux = b w taki sam sposób jak w podstawowej wersji algorytmu.
```

3.2 Funkcja wyznaczająca rozkład LU macierzy A jako drogi do rozwiązywania układów równań $Ax = b$ metodą eliminacji Gaussa

1. **Bez wyboru elementu głównego:** Rozkład LU polega na dekompozycji macierzy kwadratowej A na iloczyn dwóch macierzy: macierzy dolnej (L) i macierzy górnej (U), gdzie $A = LU$. Wariant bez wyboru elementu głównego będzie polegał na tym, że podczas eliminacji Gaussa nie będziemy przeprowadzać żadnego wyboru elementu głównego. Proces będzie polegał na sekwencyjnej eliminacji wierszy bez optymalizacji przez przestawianie wierszy.

   Po wyznaczeniu rozkładu LU macierzy $A = LU$, rozwiązanie układu równań $Ax = b$ staje się znacznie prostsze i szybsze. Aby znaleźć x, rozwiążemy dwa układy równań:

   (a) $Ly = b$, gdzie L jest macierzą dolną z rozkładu LU (rozwiązanie tego układu jest prostsze, ponieważ L jest macierzą trójkątną dolną).

   (b) $Ux = y$, gdzie U jest macierzą górną (rozwiązanie tego układu jest również szybkie, ponieważ U jest macierzą trójkątną górną).

```
1. Dla każdej kolumny k (od 1 do n-1):
    a. Dla każdego wiersza i (od k+1 do n):
        - Oblicz współczynnik m = A[i,k] / A[k,k].
        - Ustaw L[i,k] = m.
        - Zaktualizuj wiersz i: A[i, :] = A[i, :] - m * A[k, :].
2. Po zakończeniu:
```

```
    - Macierz A zawiera macierz górną U.
    - Macierz L jest generowana z wartości m
        (L[i,j] dla j < i i 1 na przekątnej).
3. Rozwiąż układy równań
    2. Rozwiąż układ Ly = b (metoda podstawiania w przód):
        a. Rozpocznij od pierwszego wiersza (i = 1):
            - Oblicz wartość y[i] jako
            (b'[i] - suma(L[i,1:i-1] * y[1:i-1])) / L[i,i].
        b. Iteruj dla każdego wiersza i (od 2 do n).
    3. Rozwiąż układ Ux = y (metoda podstawiania wstecz):
        a. Rozpocznij od ostatniego wiersza (i = n):
            - Oblicz wartość x[i] jako y[i] / U[i,i].
        b. Iteruj wstecz dla każdego wiersza i (od n-1 do 1):
            - Oblicz wartość x[i] jako
            (y[i] - suma(U[i,i+1:n] * x[i+1:n])) / U[i,i].
    4. Zwróć rozwiązanie x.
```

2. **Z częściowym wyborem elementu głównego:** W tym wariancie podczas procesu eliminacji Gaussa będziemy wybierać elementy główne (tak jak w przypadku rozwiązania układu $Ax = b$ z wyborem elementu głównego). Oznacza to, że przed każdą operacją eliminacyjną przeprowadzimy zamianę wierszy, aby na przekątnej znajdował się element o największej wartości bezwzględnej w danej kolumnie. Dzięki temu rozkład LU będzie bardziej stabilny numerycznie.

   W wyniku częściowego wyboru elementu głównego, macierz A zostaje przekształcona w postać $PA = LU$, gdzie P jest macierzą permutacyjną reprezentującą kolejność wierszy po zamianach. Tak więc rozkład LU nie jest już wyłącznie dekompozycją $A = LU$, ale macierzy permutowanej A.

   W związku z tym, aby rozwiązać układ $Ax = b$, musimy najpierw rozwiązać układ $Ly = Pb$, gdzie L jest macierzą dolną, a P jest macierzą permutacyjną, która uwzględnia dokonane zamiany wierszy. Następnie rozwiązujemy układ $Ux = y$, gdzie U jest macierzą górną z rozkładu LU. Dzięki temu podejściu zachowujemy stabilność numeryczną i dokładność obliczeń, a rozwiązanie jest uzyskiwane w dwóch etapach.

```
1. Dla każdej kolumny k (od 1 do n-1):
    a. Znajdź wiersz p, gdzie element A[p,k] jest największy
        w wartościach bezwzględnych (p >= k).
    b. Zamień wiersze k i p w macierzy A oraz w macierzy L
        (jeśli istnieje).
    c. Dla każdego wiersza i (od k+1 do n):
        - Oblicz współczynnik m = A[i,k] / A[k,k].
        - Ustaw L[i,k] = m.
        - Zaktualizuj wiersz i: A[i, :] = A[i, :] - m * A[k, :].
2. Po zakończeniu:
    - Macierz A zawiera macierz górną U.
    - Macierz L jest generowana z wartości m
        (L[i,j] dla j < i i 1 na przekątnej).
    - Tworzymy macierz permutacyjną \(P\), która
        odzwierciedla zamiany wierszy.
3. Rozwiąż układy równań
    1. Przekształć wektor b za pomocą macierzy permutacyjnej P,
```

```
    tak aby b' = Pb.
2. Rozwiąż układ Ly = b' (metoda podstawiania w przód):
    a. Rozpocznij od pierwszego wiersza (i = 1):
        - Oblicz wartość y[i] jako
        (b'[i] - suma(L[i,1:i-1] * y[1:i-1])) / L[i,i].
    b. Iteruj dla każdego wiersza i (od 2 do n).
3. Rozwiąż układ Ux = y (metoda podstawiania wstecz):
    a. Rozpocznij od ostatniego wiersza (i = n):
        - Oblicz wartość x[i] jako y[i] / U[i,i].
    b. Iteruj wstecz dla każdego wiersza i (od n-1 do 1):
        - Oblicz wartość x[i] jako
        (y[i] - suma(U[i,i+1:n] * x[i+1:n])) / U[i,i].
4. Zwróć rozwiązanie x.
```

Złożoność wszystkich powyższych algorytmów wynosi $O(n^3)$, jeśli macierz pamiętana jest jako tablica $n \times n$.

4 Dostosowanie algorytmów do specyficznej struktury danych

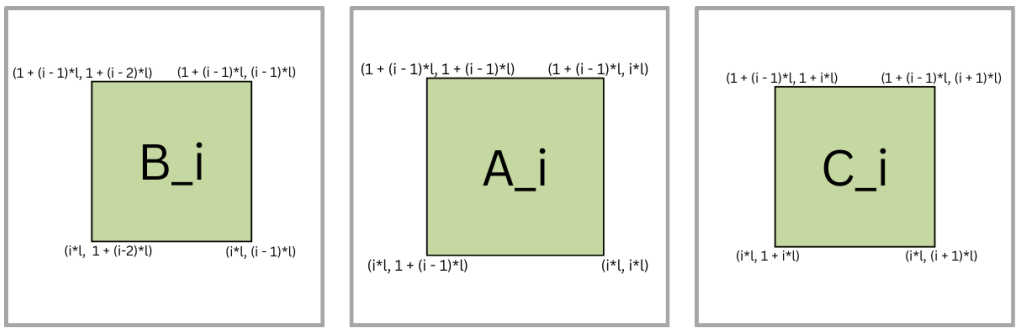
Aby zoptymalizować algorytmy w kontekście struktury danych, konieczne było zaimplementowanie kilku kluczowych funkcji.

Pierwszą z nich jest możliwość mapowania na odpowiedni element w bloku na podstawie współrzędnych. W tym przypadku, po podaniu współrzędnych (x, y), algorytm identyfikuje odpowiedni blok oraz współrzędne wewnątrz niego, a następnie zwraca właściwą wartość. Ważne jest, aby ta operacja była wykonywana w czasie stałym, ponieważ wpływa to bezpośrednio na efektywność zaproponowanej struktury.

Zajmijmy się szczegółowo tym problemem.

4.1 Poszukiwanie wartości komórki na podstawie współrzędnych w głównej macierzy

Ze względu na regularność struktury, dla każdego elementu macierzy możemy wyliczyć jego współrzędne w odpowiednim bloku, bazując na jego pozycji w wektorze bloków:



Rysunek 1: Indeksowanie rogów w i-tych blokach

Dla elementu o współrzędnych (x, y), jego blok można zidentyfikować jako:

$$\left\lfloor \frac{x-1}{l} \right\rfloor + 1.$$

Dzięki tej informacji możemy szybko ustalić, do którego bloku należy dany element.

Ponieważ bloki B i C przechowywane są w postaci słowników, dostęp do elementów w obrębie tych bloków również odbywa się w czasie stałym, gdyż kluczem jest para współrzędnych, podanych na początku poszukiwań.

Jeśli natomiast element należy do bloku A, jego pozycja w bloku jest obliczana jako:

$$(x - (i-1) \cdot l, y - (i-1) \cdot l),$$

przy założeniu, że (x, y) to współrzędne w głównej macierzy, a i to numer bloku A w wektorze bloków.

Jeżeli element nie należy do żadnego bloku, zwracamy wartość 0.

W ten sposób mamy metodę, która w sposób efektywny mapuje współrzędne z głównej macierzy na odpowiedni blok oraz pozycje w tym bloku.

4.2 Wyznaczanie liczby elementów należących do bloków w zadanym kierunku od elementu

Drugą funkcją, która znacząco może przyspieszyć obliczenia, jest funkcja, która w czasie stałym zwraca liczbę elementów znajdujących się wewnątrz jakiegoś bloku w określonym kierunku.

Dzięki temu, przekształcając macierz do postaci górnotrójkątnej, LU, bądź wykonując obliczenia metodą podstawiania wstecznego, nie będziemy musieli przeszukiwać całej macierzy. Wystarczy, że przeanalizujemy jedynie te elementy, które mają wpływ na ostateczny wynik, tj. te, które należą do któregoś z bloków.

1. **Przeszukiwanie w prawo:** Obliczanie liczby elementów w blokach po prawej stronie od danego elementu jest konieczne jedynie dla elementów z bloków A i B. Aby to zrobić, niezbędna jest znajomość numeru kolumny, w której znajduje się dany element.

Jeśli element należy do bloku A, należy policzyć pozostałe elementy w bloku A po prawej stronie oraz, w przypadku, gdy za blokiem A znajduje się blok C (tzn. nie jest to ostatni wiersz bloków), także elementy w wierszu C.

Liczba elementów po prawej stronie w bloku A wynosi:

$$l - ((col - 1) \mod l) + 1.$$

Liczba elementów w bloku C wynosi maksymalnie l.

Łącznie, do przeszukania będzie:

$$2l - ((col - 1) \mod l) + 1.$$

Jeśli element należy do bloku B, liczba elementów do przeszukania wynosi również:

$$l - ((col - 1) \mod l) + 1.$$

Należy jednak uwzględnić również elementy z bloku A (w liczbie l) oraz elementy z bloku C, o ile nie jesteśmy w ostatnim wierszu bloków.

W sumie, do przejrzenia będzie:

$$3l - ((col - 1) \mod l) + 1.$$

2. **Przeszukiwanie w lewo:** Liczba elementów, które musimy przejrzeć po lewej stronie od danego elementu, jest powiązana z liczbą elementów po prawej stronie.

   Zauważmy, że liczba elementów po lewej stronie plus liczba elementów po prawej stronie wynosi zawsze $l - 1$. Dlatego liczba elementów po lewej stronie w tym samym bloku wynosi:

   $$l - (l - (col - 1) \mod l + 1) - 1 = (col - 1) \mod l.$$

   Analogicznie, dla bloków innych niż B, liczbę elementów po lewej stronie można obliczyć uwzględniając elementy z pozostałych bloków w wierszu.

   Dla bloku B liczba elementów po lewej stronie wynosi:

   $$(col - 1) \mod l.$$

   Dla bloku A, jeżeli element znajduje się w wierszu bloków większym niż pierwszy, liczba elementów po lewej stronie to:

   $$((col - 1) \mod l) + l,$$

   natomiast w przypadku pierwszego wiersza, liczba elementów to:

   $$(col - 1) \mod l.$$

   Dla bloku C, jeżeli element znajduje się w wierszu bloków większym niż pierwszy, liczba elementów po lewej stronie wynosi:

   $$((col - 1) \mod l) + 2l,$$

   natomiast w pierwszym wierszu liczba elementów to:

   $$((col - 1) \mod l) + l.$$

3. **Przeszukiwanie w dół:** Liczenie liczby elementów w kierunku dolnym jest istotne jedynie dla elementów na przekątnej, gdyż to pod nimi będą usuwane w procesie przekształcania macierzy do postaci górnotrójkątnej.

   Ze względu na strukturę macierzy, elementy z przekątnej zawsze należą do jednego z bloków A. Aby określić liczbę elementów w kierunku dolnym, musimy znać położenie elementu w bloku A.

   Jeśli element znajduje się w bloku A w wierszu row, liczba elementów w kierunku dolnym w tym bloku wynosi:

   $$l - (((row - 1) \mod l) + 1),$$

   co jest analogiczne do obliczania liczby elementów po prawej stronie.

   Dodatkowo, jeżeli element znajduje się w jednej z dwóch ostatnich kolumn bloku A (tj. $l - (((row - 1) \mod l) + 1) == 0$ lub $l - (((row - 1) \mod l) + 1) == 1$), to poniżej bloku A znajdują się również elementy z bloku B, których liczba wynosi l.

   Łącznie, do przeszukania będzie:

   $$2l - (((row - 1) \mod l) + 1).$$

4.3 Zmiana wartości komórki na podstawie współrzędnych z głównej macierzy

Ostatnią funkcjonalnością jest efektywna zmiana wartości w zadanej komórce macierzy. Operacja ta wymaga zlokalizowania elementu w odpowiednim bloku i zmiany jego wartości. Jest to funkcjonalność analogiczna do funkcji wyszukującej wartość na zadanej pozycji, z tą różnicą, że zamiast zwracać wartość, zmieniamy ją na wartość przekazaną w argumentach funkcji.

5 Implementacja efektywnych algorytmów

Mając pewność, że na naszej strukturze danych jesteśmy w stanie wykonać niezbędne do jej efektywności funkcje w czasie stałym możemy przejść do implementacji optymalnych algorytmów.

Funkcja rozwiązująca układ równań $Ax = b$ metodą eliminacji Gaussa

1. **Bez wyboru elementu głównego:**

   Krok 1: (eliminacja do macierzy górnotrójkątnej)

```
for k in 1:(n-1)
 for i in (k+1):(k+left_down)
  l_ik = find_cell_at(i, k) / find_cell_at(k, k)
  set_cell_at(i, k, 0)

  for j in (k+1):(k+left_right)
   value = find_cell_at(i, j) - l_ik * find_cell_at(k, j)
   set_cell_at(i, j, value)
```

```
  endfor

  b[i] = b[i] - l_ik * b[k]
 endfor
endfor
```

Krok 2: (rozwiązanie układu równań)

```
for k in n:1
 suma = 0
 for j in (k+1):(k+left_right)
  suma += find_cell_at(k, j)*x[j]
 endfor
 x[k] = (b[k] - suma)/fin_cell_at(k, k)
endfor
```

2. **Z częściowym wyborem elementu głównego:**
W wersji z częściowym wyborem konieczne jest wprowadzenie funkcji do zamiany wierszy `swap rows` oraz znajdującej maksymalny element co do wartości bezwzględnej wierszu `find max`. Ze względu na zaimplementowane funkcje do znajdowania, gdzie kończą się elementy w bloku w zadanym kierunku są one wykonywane w czasie zależnym od l, $O(l)$.

Krok 1: (eliminacja do macierzy górnotrójkątnej)

```
for k in 1:(n-1)
    index = find_max(k)

    if index != k
        swap_rows(k, index)
    endif

    for i in (k+1):(k+left_down)
        l_ik = find_cell_at(i, k) / find_cell_at(k, k)
        set_cell_at(i, k, 0)

        for j in (k+1):(k+left_right)
            value = find_cell_at(i, j) - l_ik * find_cell_at(k, j)
            set_cell_at(i, j, value)
        endfor

        b[i] = b[i] - l_ik * b[k]
    endfor
endfor
```

Krok 2: (rozwiązanie układu równań)

```
for k in n:1
    suma = 0
    for j in (k+1):(k+left_right)
        suma += find_cell_at(k, j)*x[j]
    endfor
    x[k] = (b[k] - suma)/fin_cell_at(k, k)
endfor
```

Funkcja wyznaczająca rozkład LU macierzy A jako drogi do rozwiązywania układów równań $Ax = b$ metodą eliminacji Gaussa

Efektywność rozkładu LU dla zadanej struktury wynika z możliwości pamietania macierzy L oraz U w ramach jednej macierzy A. Elementy nad przekątną oraz na przekątnej to elementy macierzy U, natomiast elementy pod przekątną są elementami macierzy L. Niejawnie pamiętamy również jedynki występujące na przekątnej macierzy L.

Takie rozwiązanie jest możliwe dzięki temu, że macierz LU ma taką samą strukturę blokową jak macierz A

1. **Bez wyboru elementu głównego:**

   Modyfikacją w stosunku do eliminacji do macierzy górnotrójkątnej bez wyboru głównego jest tutaj przypisywanie wartości l_{ik} zamiast 0 elementom pod przekątną. Nie przypisujemy również zmodyfikowanych wartości w wektorze b, a rozwiązywanie układu równań $Ax = b$ jest dwustopniowe.

   Krok 1: (eliminacja do macierzy LU)

   ```
   for k in 1:(n-1)
       for i in (k+1):(k+left_down)
           l_ik = find_cell_at(i, k) / find_cell_at(k, k)
           set_cell_at(i, k, l_ik)

           for j in (k+1):(k+left_right)
               value = find_cell_at(i, j) - l_ik * find_cell_at(k, j)
               set_cell_at(i, j, value)
           endfor
       endfor
   endfor
   ```

   Krok 2: (rozwiązanie układu równań Ly = b)

   ```
   for k in 1:L.n
       y[k] = b[k]
       for i in (k - left_left):k-1
               y[k] = y[k] - find_cell_at(k, i) * y[i]
       endfor
   endfor
   ```

   Krok 3: (rozwiązanie układu równań Ux = y)

   ```
   for k in n:1
       suma = 0
       for j in (k+1):(k+left_right)
           suma += find_cell_at(k, j)*x[j]
       endfor
       x[k] = (y[k] - suma)/fin_cell_at(k, k)
   endfor
   ```

2. **Z częściowym wyborem elementu głównego:**

   Modyfikacją w stosunku do eliminacji do macierzy górnotrójkątnej z częściowym wyborem elementu głównego jest tutaj przypisywanie wartości l_{ik} zamiast 0 elementom pod przekątną. Nie przypisujemy również zmodyfikowanych wartości w

wektorze b, a rozwiązywanie układu równań $Ax = b$ jest dwustopniowe. Zmieniona kolejność wierszy pamiętana jest w osobnym wektorze `perm`, który wykorzystywany jest później do utworzenia odpowiedniej permutacji elementów wektora b.

Krok 1: (eliminacja do macierzy LU)

```
for k in 1:(n-1)
    index = find_max(k)

    if index != k
        swap_rows(k, index)
        perm[k], perm[index] = perm[index], perm[k]
    endif

    for i in (k+1):(k+left_down)
        l_ik = find_cell_at(i, k) / find_cell_at(k, k)
        set_cell_at(i, k, l_ik)

        for j in (k+1):(k+left_right)
            value = find_cell_at(i, j) - l_ik * find_cell_at(k, j)
            set_cell_at(i, j, value)
        endfor
    endfor
endfor
```

Krok 2: (rozwiązanie układu równań Ly = b)

```
b_prim = b[perm]
for k in 1:L.n
    y[k] = b_prim[k]
    for i in (k - left_left):k-1
            y[k] = y[k] - find_cell_at(k, i) * y[i]
    endfor
endfor
```

Krok 3: (rozwiązanie układu równań Ux = y)

```
for k in n:1
    suma = 0
    for j in (k+1):(k+left_right)
        suma += find_cell_at(k, j)*x[j]
    endfor
    x[k] = (y[k] - suma)/fin_cell_at(k, k)
endfor
```

6 Złożoność zaimplementowanych algorytmów

Funkcja rozwiązująca układ równań $Ax = b$ metodą eliminacji Gaussa

1. **Bez wyboru elementu głównego:**

   Funkcję tę można szacować z góry przez:

$$\sum_{k=1}^{n-1}\left(\sum_{i=k+1}^{k+2l}\left(\sum_{j=k+1}^{k+3l} O(1)\right)\right)+\sum_{k=n}^{1}\left(\sum_{j=k+1}^{k+3l} O(1)\right)$$

Zatem złożoność algorytmu:

$$O(n)\cdot O(l)\cdot O(l)=O(n\cdot l^2)$$

Zakładając, że l jest stałe dostajemy, że algorytm ma złożoność

$$O(n)$$

2. **Z częściowym wyborem elementu głównego:**

$$\sum_{k=1}^{n-1}\left(2\cdot O(l)+\sum_{i=k+1}^{k+2l}\left(\sum_{j=k+1}^{k+3l} O(1)\right)\right)+\sum_{k=n}^{1}\left(\sum_{j=k+1}^{k+3l} O(1)\right)$$

Złożoność algorytmu również wynosi więc $O(n\cdot l^2)$, co w efekcie daje $O(n)$.

Funkcja wyznaczająca rozkład LU macierzy A jako drogi do rozwiązywania układów równań $Ax=b$ metodą eliminacji Gaussa

1. **Bez wyboru elementu głównego:**

$$\sum_{k=1}^{n-1}\left(\sum_{i=k+1}^{k+2l}\left(\sum_{j=k+1}^{k+3l} O(1)\right)\right)+\sum_{k=1}^{n}\left(\sum_{j=k-3l}^{k-1} O(1)\right)+\sum_{k=n}^{1}\left(\sum_{j=k+1}^{k+3l} O(1)\right)$$

Złożoność wynosi $O(n\cdot l^2)$, dla l stałego mamy więc $O(n)$.

2. **Z częściowym wyborem elementu głównego:**

$$\sum_{k=1}^{n-1}\left(2\cdot O(l)+\sum_{i=k+1}^{k+2l}\left(\sum_{j=k+1}^{k+3l} O(1)\right)\right)+\sum_{k=1}^{n}\left(\sum_{j=k-3l}^{k-1} O(1)\right)+\sum_{k=n}^{1}\left(\sum_{j=k+1}^{k+3l} O(1)\right)$$

Złożoność wynosi $O(n\cdot l^2)$, dla l stałego mamy więc $O(n)$.

W wyniku optymalizacji struktury danych, udało się zmniejszyć złożoność algorytmów z $O(n^3)$ do $O(n)$, co umożliwia na pracę ze zdecydowanie większymi danymi.

7 Eksperymentalne wyznaczenie własności algorytmów

W celu potwierdzenia teoretycznych rozważań zostały napisane programy testujące zaimplementowane algorytmy. W celu ocenienia ich poprawności zostały one porównane z wbudowanymi w język Julia funkcjami rozwiązującymi te algebraiczne problemy.

Testy zostałe przeprowadzone na danych wygenerowanych za pomocą funkcji `blockmat` dostarczonej przez prof. Pawła Zielińskiego.

7.1 Dane testowe

Testy zostały podzielone na dwie części:

1. **Testy uwzględniające wbudowane w język Julia funkcje:**

   Zostały one przeprowadzone na danych o rozmiarze z zakresu $[1000, 20000]$ z krokiem 1000. Dla każdego rozmiaru testy przeprowadzane były 10 razy.

   W trakcie testu zbierane były wielkości takie jak:

   - czas potrzebny na przeprowadzenie obliczeń wykorzystując funkcje wbudowane (w sekundach)
   - czasy potrzebne na przeprowadzenie obliczeń dla każdego z zaimplementowanych algorytmów (w sekundach)
   - wielkość pamięci potrzebna do zaalokowania podstawowej struktury danych (w bajtach)
   - wielkość pamięci potrzebna do zaalokowania zoptymalizowanej struktury danych (w bajtach)
   - błąd względny znalezionego rozwiązania w stosunku do rozwiązania wyliczonego przy pomocy wbudowanych funkcji

2. **Testy uwzględniające jedynie zoptymalizowane algorytmy**

   Ze względu na małą moc obliczeniową testy dla większych danych nie mogły zostać przeprowadzone z wykorzystaniem funkcji i struktur wbudowanych w język.

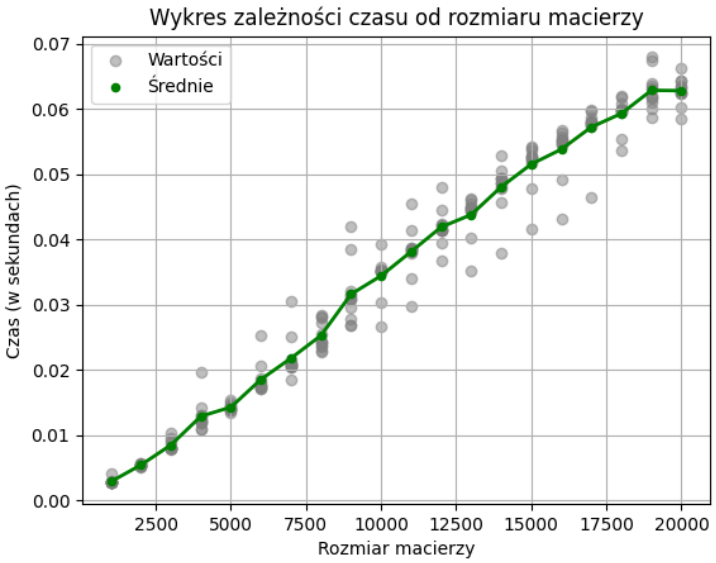
   Zostały jednak przeprowadzone na danych o rozmiarze z zakresu $[50000, 1000000]$ z krokiem 50000. Dla każdego rozmiaru testy przeprowadzane były 10 razy.

   W trakcie testu zbierane były wielkości takie jak:

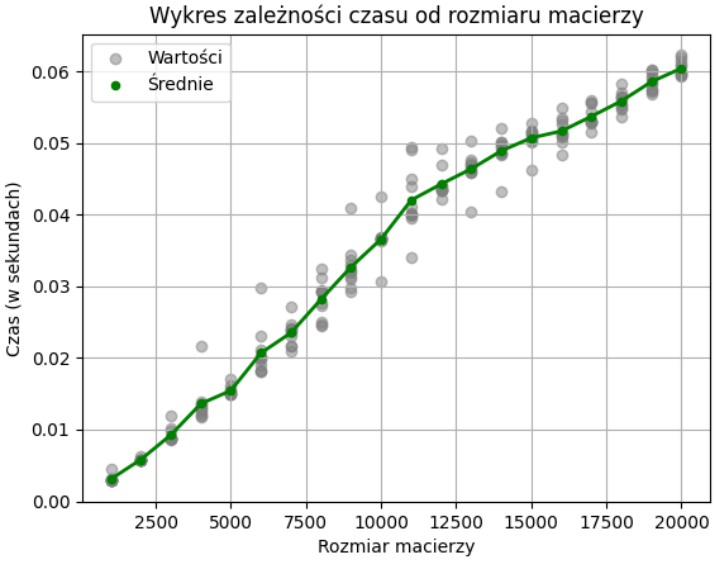
   - czasy potrzebne na przeprowadzenie obliczeń dla każdego z zaimplementowanych algorytmów (w sekundach)
   - wielkość pamięci potrzebna do zaalokowania zoptymalizowanej struktury danych (w bajtach)

7.2 Wyniki

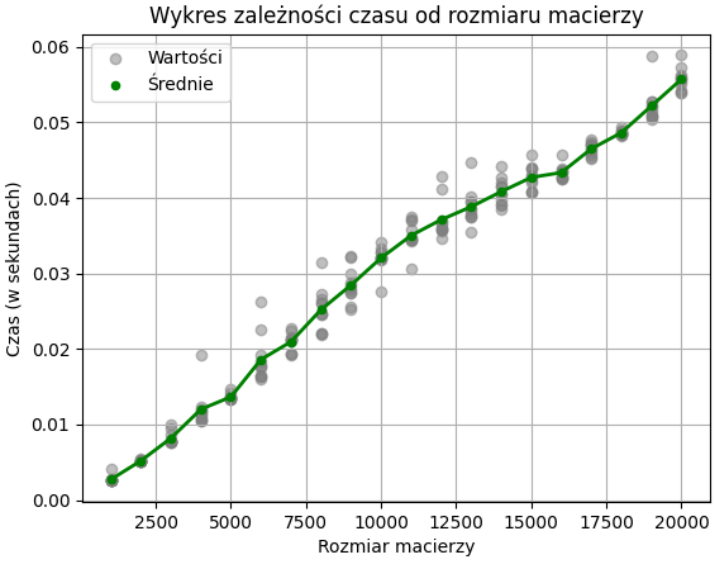
1. **Testy uwzględniające wbudowane w język Julia funkcje:**



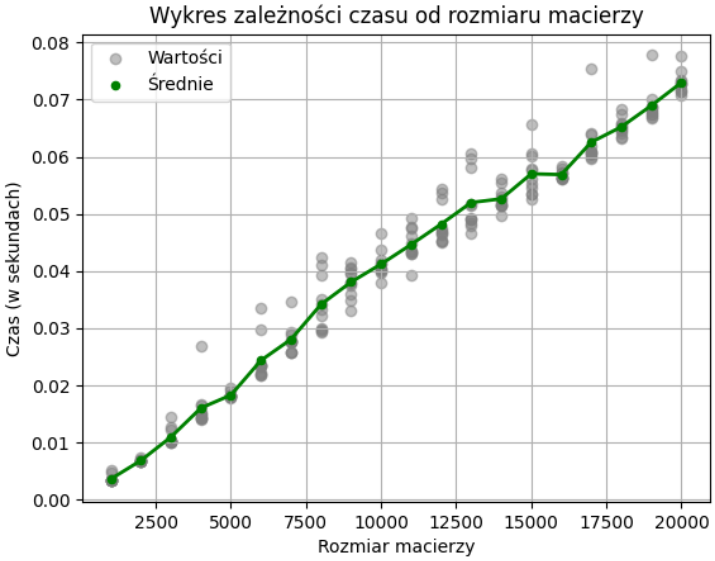
Rysunek 2: Gauss



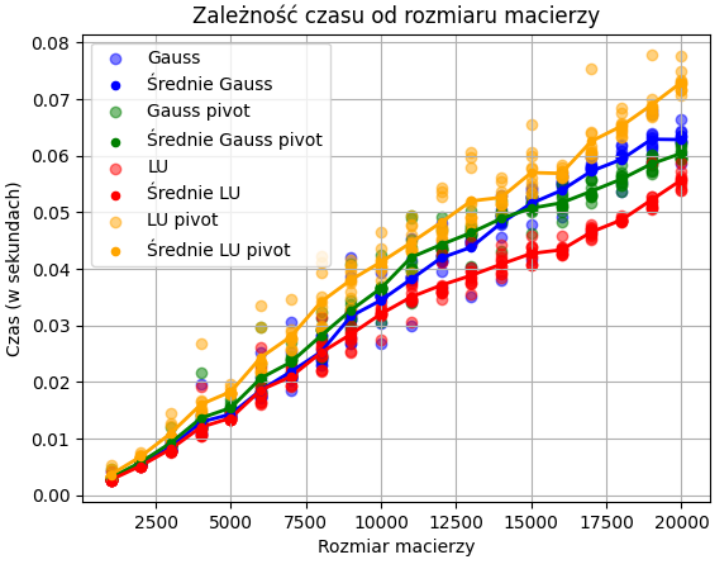
Rysunek 3: Gauss pivot



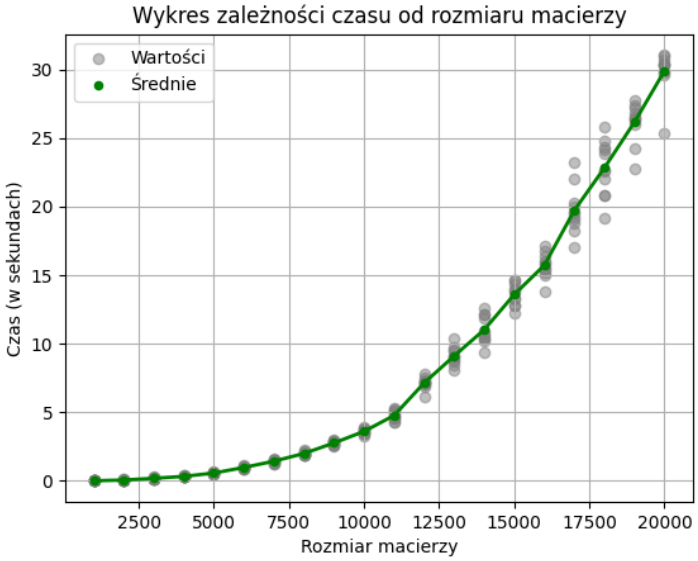
Rysunek 4: LU



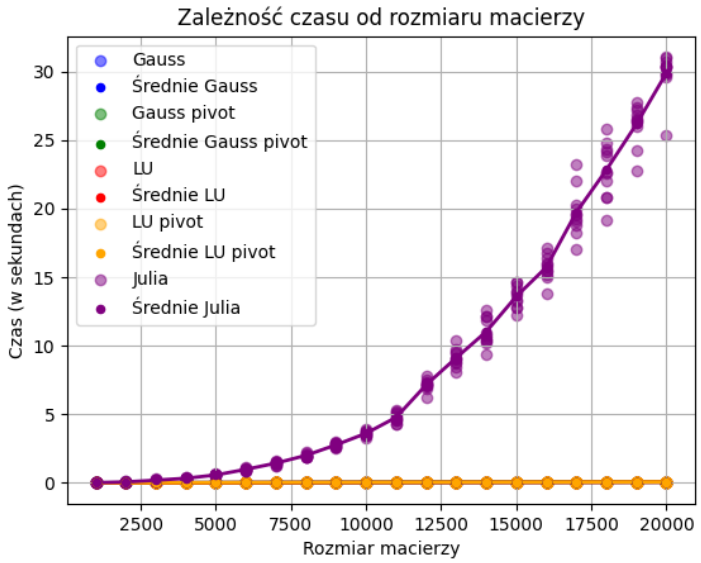
Rysunek 5: LU pivot



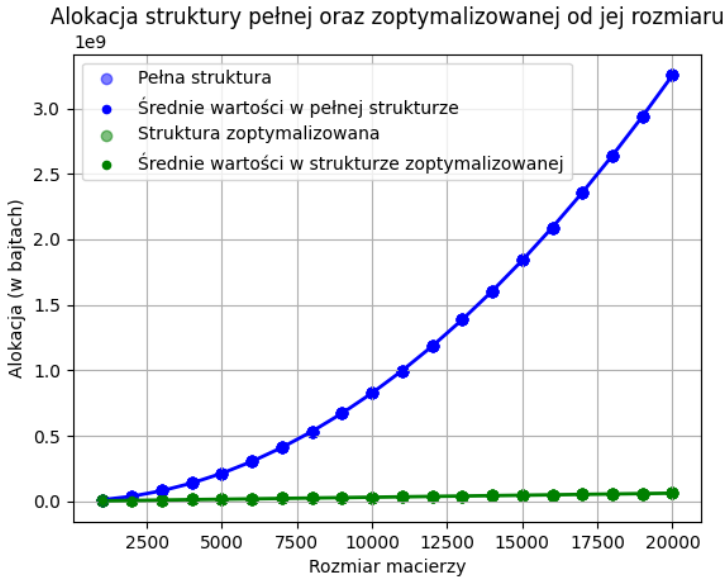
Rysunek 6: All custom



Rysunek 7: Julia matrix



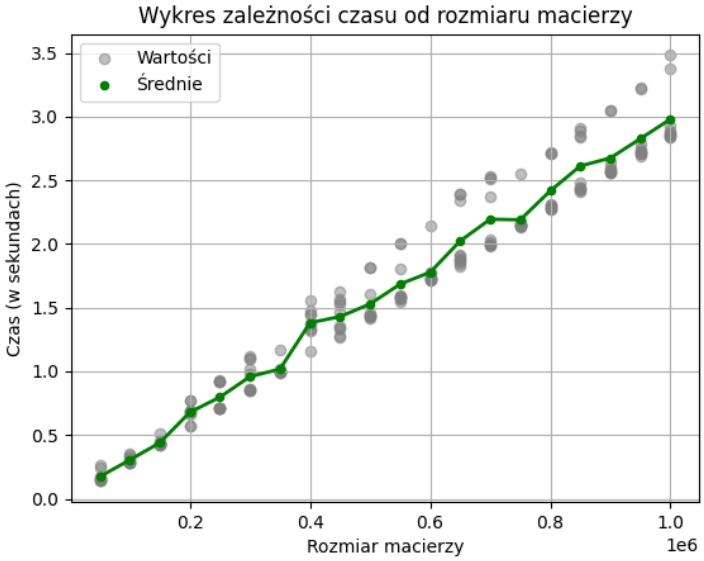
Rysunek 8: All



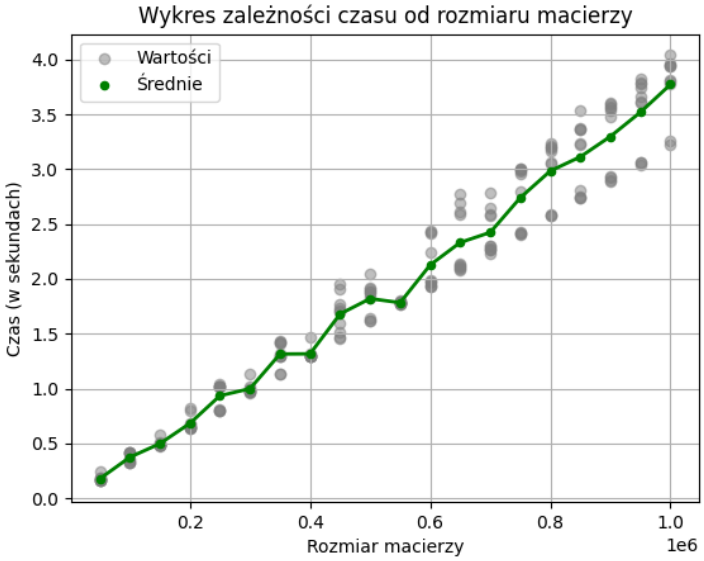
Rysunek 9: Memory alloc

Przeprowadzone testy wykazały, że zaimplementowane funkcje zostały skutecznie zoptymalizowane do złożoności $O(n)$ i wypadły zdecydowanie lepiej od funkcji wbudowanych, zarówno pod względem złożoności czasowej, jak i pamięciowej. Ponadto, charakteryzują się one dużą dokładnością. Dla funkcji z częściowym wyborem elementu głównego, błędy względne wynosiły rzędu 10^{-16}, natomiast dla funkcji bez tego wyboru, które są bardziej narażone na błędy numeryczne, błędy te wyniosły rzędu 10^{-14}.

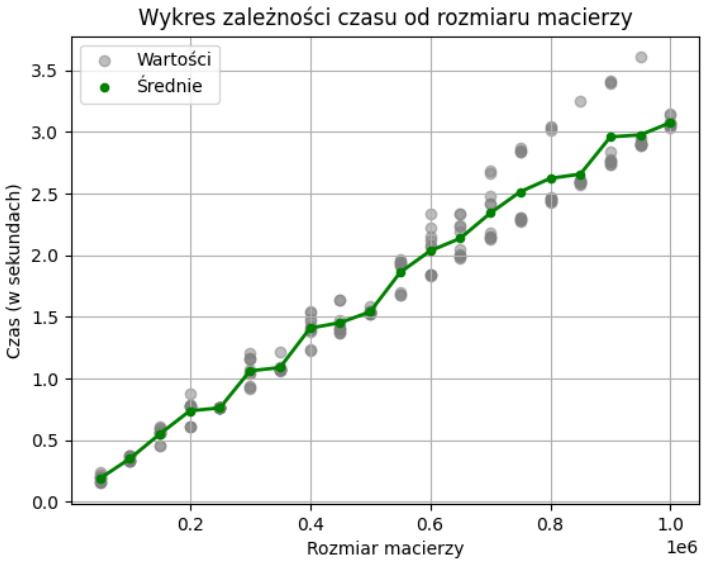
2. **Testy uwzględniające jedynie zoptymalizowane algorytmy**



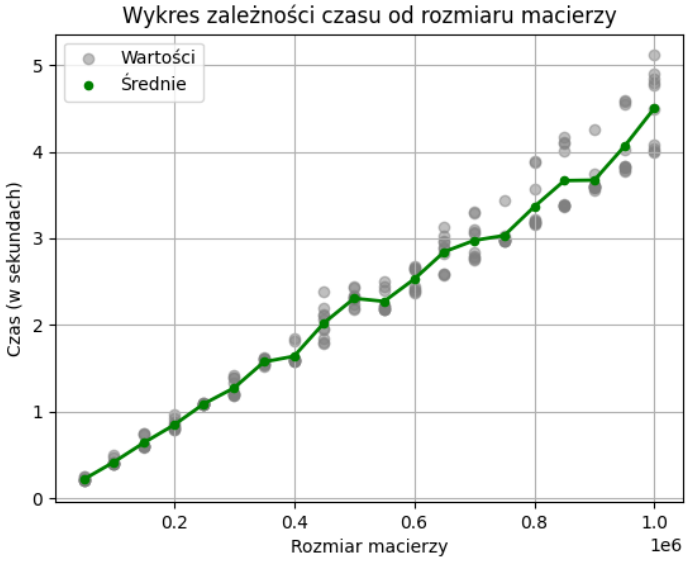
Rysunek 10: Gauss



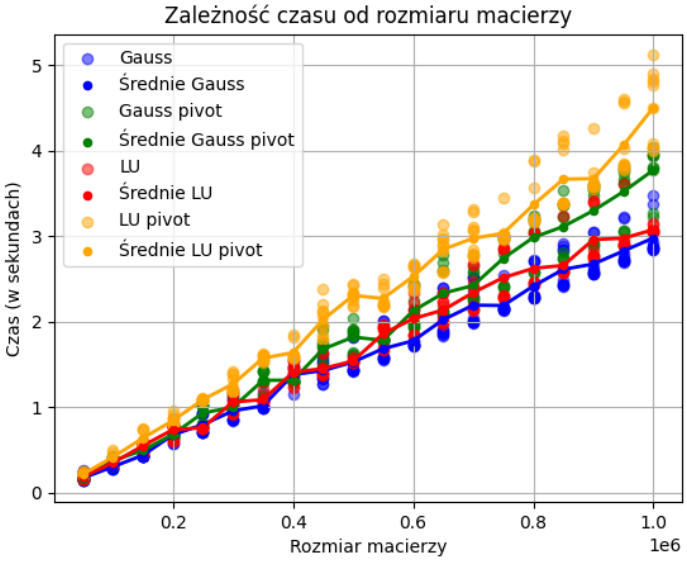
Rysunek 11: Gauss pivot



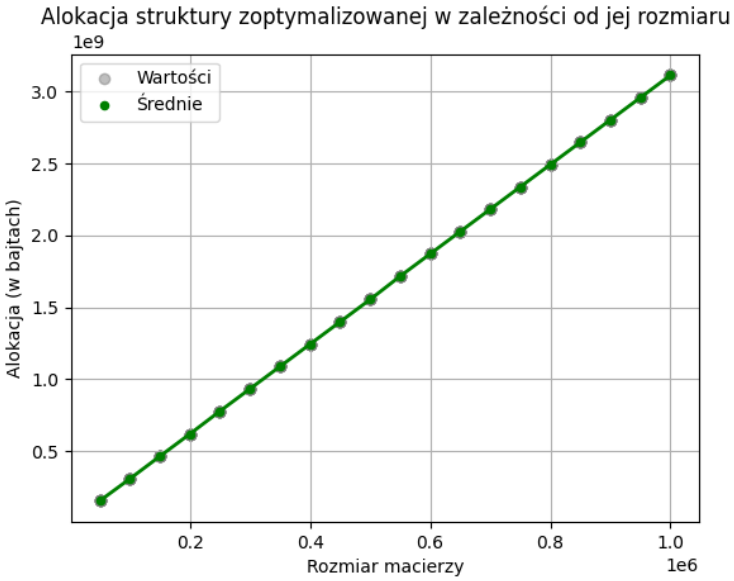
Rysunek 12: LU



Rysunek 13: LU pivot



Rysunek 14: All custom



Rysunek 15: All custom

Testy przeprowadzone na większych zbiorach danych wykazały, że dla nich również złożoność pamięciowa i czasowa algorytmów utrzymuje się w granicach $O(n)$. Co istotne, dla macierzy o rozmiarze milion na milion, algorytmy potrzebowały zaledwie kilku sekund na znalezienie rozwiązania. Tak imponujący czas wykonania wynika z efektywnego dopasowania struktury danych do rzadkiej i blokowej natury macierzy.

8 Podsumowanie

W niniejszej pracy przedstawiono implementację i optymalizację algorytmów rozwiązywania układów równań liniowych $Ax = b$ metodą eliminacji Gaussa oraz rozkładu LU. Celem było zaprojektowanie struktur danych i algorytmów, które umożliwiłyby znaczną redukcję złożoności czasowej do $O(n)$ w przypadku dużych macierzy. Osiągnięto to dzięki zastosowaniu efektywnych struktur danych, które umożliwiły wykonanie kluczowych operacji w czasie stałym.

Przeprowadzone testy eksperymentalne potwierdziły teoretyczne założenia dotyczące złożoności algorytmów. Testy porównawcze z funkcjami wbudowanymi w język Julia wykazały, że zaimplementowane algorytmy są znacząco szybsze, a także mniej pamięciochłonne, zwłaszcza w przypadku dużych danych. Dodatkowo, algorytmy wykazały wysoką dokładność, z błędami względnymi wynoszącymi rzędu 10^{-16} dla wersji z częściowym wyborem elementu głównego.

Wyniki eksperymentalne potwierdzają, że zoptymalizowane algorytmy działają efektywnie nawet na macierzach o rozmiarze miliona na milion, osiągając czas wykonania rzędu kilku sekund. Zoptymalizowana struktura danych pozwala na wykorzystanie tej metodologii do rozwiązywania układów równań liniowych w zastosowaniach wymagających dużej skali danych.

9 Wnioski

Odpowiednie dopasowanie struktury danych do konkretnego problemu jest kluczowe dla osiągnięcia optymalnej wydajności algorytmów. Zależność między strukturą danych a efektywnością algorytmu jest szczególnie widoczna w przypadku rozwiązywania układów równań liniowych, gdzie różne typy macierzy, takie jak macierze rzadkie czy gęste, wymagają zastosowania różnych metod przechowywania danych. Wybór niewłaściwej struktury może prowadzić do nadmiernego zużycia pamięci i wydłużenia czasu obliczeń, nawet przy zastosowaniu najwydajniejszych algorytmów. Dlatego tak istotne jest, aby przy każdym zadaniu dokładnie analizować charakterystykę danych wejściowych i dobrać do nich odpowiednią strukturę.