Obliczenia naukowe Lista 5

Radosław Wojtczak

Numer indeksu: 254607

Spis treści

1 Omówienie problematyki **2**

2 Eliminacja Gaussa **4**
2.1 Wprowadzenie 4
2.2 Analiza algorytmu 5

3 Eliminacja Gaussa z częściowym wyborem **8**
3.1 Wprowadzenie 8
3.2 Analiza algorytmu 8

4 Rozkład LU **11**
4.1 Wprowadzenie 11
4.2 Analiza algorytmu 11
4.3 LU() 11
4.4 LUSolver() 12

5 Rozkład LU z częściowym wyborem elementu głównego **14**
5.1 Wprowadzenie 14
5.2 LUPivot() 14
5.3 LUSolver() 16

6 Wyniki i interpretacje **17**

7 Wnioski **19**

Omówienie problematyki

Problematyka zadania opierała się na efektywnym (pod względem złożoności obliczeniowej jak i pamięciowej) rozwiązaniu układu równań liniowych postaci:

$$\mathbf{Ax} = \mathbf{b} \tag{1.1}$$

dla szczególnej postaci macierzy $\mathbf{A}$ ($\mathbf{A} \in \mathbb{R}^{n \times n}$):

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

oraz wektora prawych stron $\mathbf{b}$ ($\mathbf{b} \in \mathbb{R}^n$). Macierze $\mathbf{A_k}, \mathbf{B_k}, \mathbf{C_k}$ są następujących postaci:

1. Macierz $\mathbf{A_k}$, $k = 1, ..., v$,gdzie $v = n/l$, przy założeniu, że n jest podzielne przez l oraz $l \geq 2$, jest macierzą gęstą (oznacza to, że ów macierz ma wszystkie elementy niezerowe)
2. Macierz $\mathbf{B_k}$, $k = 2, ..., v$ (v jak powyżej) jest macierzą zawierającą jedyne niezerowe elementy umieszczone w dwóch ostatnich kolumnach
3. Macierz $\mathbf{C_k}$, $k = 1, ..., v-1$ (v jak powyzej) jest macierzą diagonalną

Dodatkowo macierze $\mathbf{0}$ są macierzami kwadratowymi zerowego stopnia l. W pierwszej części zadania, do rozwiązania układu liniowego z tak specyficzną specyfikacją macierzy $\mathbf{A}$, należało odpowiednio zmodyfikować metodę eliminacji Gaussa dla dwóch wariantów:

- bez wyboru elementu głównego
- z częściowym wyborem elementu głównego

W drugiej zaś należało wyznaczyć rozkład $\mathbf{LU}$ macierzy $\mathbf{A}$ przy pomocy wcześniej zmodyfikowanej metody eliminacji Gaussa (również jak w przypadku Gaussa dla dwóch wariantów- z, jak i z częściowym wyboremelementu głównego) oraz

napisać funkcję rozwiązującą zadany układ równań dla wcześniej wyznaczonego rozkładu **LU** przez powyżej wspomnianą funkcję.
Dalsza część tego dokumentu będzie składała się z omówienia poszczególnych części wchodzących w skład zadania oraz przedstawienia, jak i zinterpetowania otrzymanych wyników.
Wszystkie funkcje zostały zaimplementowane w module o nazwie **blocksys.jl**, zgodnie z poleceniem zadania. Rozwiązanie wykorzystuje gotową strukturę danych o nazwie *SparseArray*, która w efektywny sposób pozwala nam na zapamiętanie niezerowych elementów macierzy.
Pliki

- gaussTest.jl
- gaussPivotTest.jl
- LuTest.jl
- LuPivotTest.hl

implementują testy odpowiednio dla: metody eliminacji gausa bez wyboru elementu głównego, z częściowym wyborem elementu głównego, rozkładu **LU** bez elementu głównego, z częściowym wyborem elementu głównego. Testy wykorzystują dostarczony z poleceniem zadania moduł **matrixgen.jl)** zawierający funkcję **blockmat()** odpowiedzialną za generowanie odpowiednich macierzy. Testy wykonywane są dla $n \in \{1000, 1010, 1020, ..., 50000\}$ oraz stałego $l = 4$. Sprawdzany jest czas, jak i wykorzystana pamięć, przez każdą z funkcji w trakcie rozwiązywania układu równań.

Eliminacja Gaussa

2.1 Wprowadzenie

Metoda eliminacja Gaussa jest algorytmem stosowanym do rozwiązywania układów równań liniowych jak i wyznaczania rozkładu LU, wykorzystując operacje elementarne. Operacje elementarne wykonywane na macierzach to:

- Dodawanie do jednego wiersza macierzy innego wiersza pomnożonego przez liczbę
- zamienienie dwóch wierszy miejscami
- pomnożenie wierszy przez liczbę różną od zera

W ów zadaniu podstawowy algorytm eliminacji Gaussa należało zmodyfikować tak, aby wykorzystywał specyficzną strukturę macierzy w celu szybszej realizacji postawionego problemu. Analizę zaczniemy od wizualizacji macierzy dla następujących danych wejściowych: $n = 16$, $l = 4$, z czego automatycznie otrzymujemy $v = 4$.

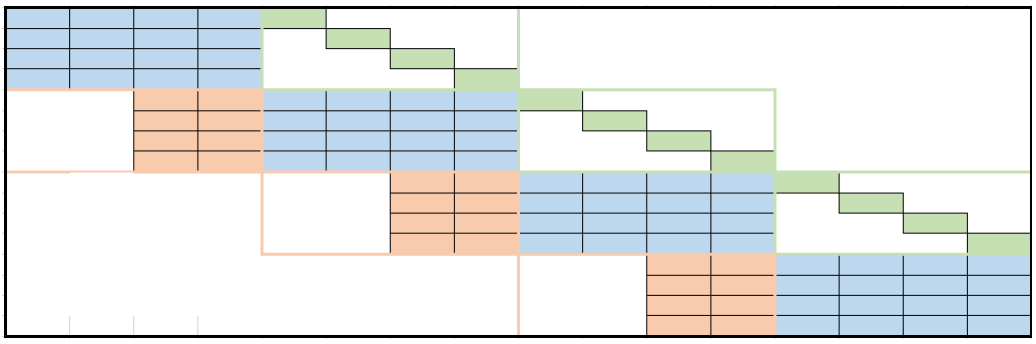

Figure 2.1: Struktura macierzy dla $n = 16$ oraz $l = 4$

Białym wypełnieniem oznaczone są komórki, w których znajdują się wartości zerowe. Obramowaniem jasnoniebieskim oznaczone są macierze $\mathbf{A_k}$, jasnopomarańczowym- $\mathbf{B_k}$, a $\mathbf{C_k}$- jasnozielonym. Dodatkowo komórki, które w tych macierzach są niezerowane zostały wypełnione kolorem odpowiadającym kolorowi obramowania. Pierwszym spostrzeżeniem jest fakt, iż liczba elementów niezerowych zależy od liczby l. Dodatkowo zauważamy, że

- Każda z macierzy $\mathbf{A_k}$ składa się z l^2 elementów niezerowych
- Każda z macierzy $\mathbf{B_k}$ składa się z $2 * l$ elementów niezerowych
- Każda z macierzy $\mathbf{C_k}$ składa się z l elementów niezerowych

Wiedząc, że v oznacza liczbę takich macierzy możemy obliczyć sumę niezerowych komórek dla danej macierzy $\mathbf{A}$: $v * l^2 + (v-1) * 2 * l + (v-1) * l = n/l * l^2 + (n/l-1) * 2 * l + (n/l-1) * l = n * l + 2 * n - 2 * l + n - l = n * l + 3 * n - 3 * l = n * l + 3(n-l)$ co pomoże nam w dokładnym alokowaniu wektorów w dalszym działaniu programu.

2.2 Analiza algorytmu

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **gauss()**:

Algorithm 1 Metoda eliminacji Gaussa

INPUT:

- matrix- struktura typu *SparseArray*
- b- wektor prawych stron
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

OUTPUT:

- result- wektor o rozmiarze n zawierający pierwiastki równania

```
for k ← 1 to n − 1 do
  for i ← k + 1 to min(n, k + l + 1) do
    if eps(Float64)>abs(matrix[k,k]) then
      print:Zero w mianowniku
      break
    end if
    I ← matrix[i, k]/matrix[k, k]
    matrix[i, k] ← 0
    for j ← k + 1 to min(n, k + l) do
      matrix[i, j] ← matrix[i, j] − I ∗ matrix[k, j]
    end for
    b[i] ← b[i] − I ∗ b[k]
  end for
  result ← Array[n] = {0}
  for a ← n downto 1 do
    sum ← 0
    for z ← a + 1 to min(n, a + l) do
      sum ← sum + matrix[a, z] ∗ result[z]
    end for
    result[a] ← (b[a] − sum)/matrix[a, a]
  end for
end for
return result
```

Algorytm rozpoczynamy pętlą przechodzącą po wszystkich kolumnach macierzy. Kolejna pętla- wewnętrzna, odpowiada za iterowanie po wierszach. Tutaj korzystamy z własności macierzy i zauważamy, iż liczba elementów do sprawdzenia w danym wierszu jest nie większa niż l+1. Następnie pod zmienną I podstawiamy wynik dzielenia aktualnie rozpatrywanego wiersza w danej kolumnie z wartością diagonalną po czym zerujemy aktualnie rozpatrywaną komórkę. Notacja matrix[a,b] jest równoważna elementowi w rzędzie **a** (numerowanym od 1) i kolumnie **b**. Po wyelimowaniu zmiennej z równania przechodzimy do odejmowania I od pozostałych wyrazów w danym wierszu.
Po zakończeniu działania pierwszej pętli **FOR** otrzymujemy macierz górnotrójkątną. Następnie deklarujemy tablicę o nazwie **result**, w której będziemy przechowywać pierwiastki równania. W drugiej pętli **FOR** przechodzimy wstecz, wyliczając pierwiastki od ostatniego do pierwszego. Taką metodę postępowania nazywamy

podstawianiem wstecz. Po wykonaniu drugiej pętli **FOR** w tablicy **result** znajdują się wszystkie pierwistki równania, której zwracamy użytkownikowi. W tym momencie funkcja kończy swoje działanie pomyślnie.

Złożoność: W algorytmie Gaussa przechodzimy po przekątnej macierzy, zerując elementy, które się pod nią znajdują. Nie posiadając żadnych dodatkowych informacji na temat obsługiwanych macierzy zauważamy, że każda z wewnętrznych pętli **for** musiałaby zostać wykonana aż do końca macierzy, czyli do n. To dałoby nam złożonośc na poziomie $\mathbf{O(n^3)}$. W tym miejscu korzystamy z specyfikacji zadania. Szybko zauważamy, że w takim przypadku maksymalna liczba elementów do rozpatrzenia w każdym wierszu, jak i w każdej kolumnie jest równa $l+1$. Oczywistym jest, ów liczba nie jest stała- przykładem będzie chociażby pierwsza kolumna dla $l=4$. W tej sytuacji macierz **B** ma niezerowe elementy jedynie w kolumnie 3 i 4 macierzy **A**, więc wystarczyłoby rozpatrzeć jedynie l elementów, jednakże implementacja korzysta z uproszczenia i wszystkie pętle rozpatrują $l+1$ elementów (z wyjątkiem sytuacji, w których n jest liczbą mniejszą niż $l+1$- wtedy wychodzilibyśmy poza ramy tablicy, co kończyłoby się błędem kompilacji). Oznacza to, że przyjmując l jako liczbę stałą schodzimy ze złożoności $\mathbf{O(n^3)}$ na $\mathbf{O(n*(l^2+l))}$, co uznajemy jako $\mathbf{O(n)}$.
Ze względu na to, że macierz składa się z $n*l+3(n-l)$ elementów niezerowych (a tylko takie zapamiętujemy) łatwo zauważamy, iż złożonośc pamięciowa wynosi $\mathbf{O(n)}$.

Eliminacja Gaussa z częściowym wyborem

3.1 Wprowadzenie

W tym przypadku dokonujemy drobnej modyfikacji standardowego algorytmu rozwiązywania układu równań metodą eliminacji Gaussa. Wprowadzamy nowe pojęcie częściowego wyboru, który polega na znalezieniu elementu, które jest największy co do wartości bezwzględnej w danym wierszu. Następnie takowy znaleziony element determinuje nam wiersz, który zamieniamy z wierszem aktualnie rozpatrywanym. Przestawienie dotyczy się również macierzy prawych stron.
W ten sposób zmniejszamy błędy wynikające z niedokładnej reprezentacji liczb zmiennoprzecinkowych w komputerach. Dzięki wyborowi elementu o największej co do wartości bezwzględnej wartości zmniejszamy szansę na wykonywanie działań na bardzo małych liczbach (co, jak pokazywaliśmy wielokrotnie w trakcie wykonywania zadań na ów kursie, może drastycznie wpływać na otrzymane wyniki)

3.2 Analiza algorytmu

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **gaussPivot()**:

Algorithm 2 Metoda eliminacji Gaussa z wykorzystaniem częściowego wyboru elementu głównego

INPUT:

- matrix- struktura typu *SparseArray*
- b- wektor prawych stron
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

OUTPUT:

- result- wektor o rozmiarze n zawierający pierwiastki równania

$pivot \leftarrow \{1, ..., n\}$
for $k \leftarrow 1$ **to** $n-1$ **do**
  $row \leftarrow k$
  $value = abs(matrix[pivot[k], k])$
  for $i \leftarrow k+1$ **to** $min(n, k+l+1)$ **do**
    if $abs(matrix[pivot[i], k]) > value$ **then**
      $value \leftarrow matrix[pivot[i], k]$
      $row \leftarrow i$
    end if
  end for
  $pivot[row], pivot[k] \leftarrow pivot[k], pivot[row]$
  for $i \leftarrow k+1$ **to** $min(n, k+l+l+1)$ **do**
    if eps(Float64)>abs(matrix[pivot[k],k]) **then**
      print:Zero w mianowniku
      break
    end if
    $I \leftarrow matrix[pivot[i], k]/matrix[pivot[k], k]$
    $matrix[pivot[i], k] \leftarrow 0$
    for $j \leftarrow k+1$ **to** $min(n, k+l+l)$ **do**
      $matrix[pivot[i], j] \leftarrow matrix[pivot[i], j] - I * matrix[pivot[k], j]$
    end for
    $b[pivot[i]] \leftarrow b[pivot[i]] - I * b[pivot[k]]$
  end for
  $result \leftarrow Array[n] = \{0\}$
  for $a \leftarrow n$ **downto** 1 **do**
    $sum \leftarrow 0$
    for $z \leftarrow a+1$ **to** $min(n, a+l+l)$ **do**
      $sum \leftarrow sum + matrix[pivot[a], z] * result[z]$
    end for
    $result[a] \leftarrow (b[pivot[a]] - sum)/matrix[pivot[a], a]$
  end for
end for
return result

Na pierwszy rzut oka spostrzegamy duże podpobieństwo względem poprzedniego algorytmu. Różnice jednak występują w dodaniu specjalnej tablicy, której

początkowymi elementami są liczby naturalne od 1 do n o nazwie *pivot*, która ma za zadanie przechowywać rotację w indeksach po ustaleniu częściowego elementu głównego. Konsekwencją tego jest dodanie dodatkowej pętli **FOR** w linii numer 5, oraz dwóch zmiennych lokalnych: *row*, która przechowuje wiersz, w którym znajduje się maksymalna co do wartości bezwzględnej wartość, oraz *value*, która ów wartość przechowuje. Ustalenie elementu maksymalnego zachodzi w trywialny sposób- liniowo porównujemy wszystkie elementy zapamiętując największy. W reszcie algorytmu w kolumnie wierszy wszystkie napisy z **x** zostały zamienione na **pivot[x]**, gdyż to właśnie tablica *pivot* przechowuje indeksy po zmianie. **Złożoność:** W algorytmie Gaussa z częściowym wyborem elementu głównego dochodzi nam jeden **FOR** na drugim poziomie zagnieżdżenia, która wykonuje się zawsze $l+1$ razy. Dodatkowo we wszystkich późniejszych pętlach **FOR** na poziomie drugim liczba sprawdzanych indeksów wzrosła o l (z $l+1$ do $l+l+1$). Wynika to z faktu, iż mogło dojść do zamiany wiersza k-tego z wierszem k+l-tym. W tej sytuacji dochodzi do zamian w całych wierszach, co ma znaczenie ze względu na strukturę macierzy $\mathbf{C_k}$, która jest macierzą diagonalną. Weźmy pod uwagę sytuację, gdy k=1, a maksymalna wartość znajduje się k=l. Wtedy należy zamienić ze sobą wiersze 1 i l, co prowadzi również do zamiany elementów znajdujących się w macierzy $\mathbf{C_k}$- w rozpatrywanym wierszu ostatni element będzie znajdował się na pozycji k+l-tej.
Z teoretycznego punktu widzenia ów zmiana nie wpływa na złożoność obliczeniową, która dalej wynosi **O(n)**, przy założeniu, że l jest liczbą stałą. Zwiększy się jedynie współczynnik stojący przy wyrazie n, jednakże nie wpływa to na asymptotykę ów funkcji. Ze względu na to, iż macierz składa się z $n*l+3(n-l)$ elementów niezerowych (a tylko takie zapamiętujemy) łatwo zauważamy, iż złożonosc pamięciowa wynosi **O(n)**.

Rozkład LU

4.1 Wprowadzenie

Rozkład **LU** (ang. lower-upper) polega na rozłożeniu macierzy kwadratowej na iloczyn dwóch macierzy trójkątnych- macierzy dolnotrójątknej **L** (której elementy diagonalne są równe 1) oraz macierzy górnotrójkątnej **U**. Z wykładu wiemy, iż Eliminację Gaussa możemy traktować jako algorytm rozkładu trójkątnego.
$\mathbf{A}^{(n)} = \mathbf{L}^{(n-1)}...\mathbf{L}^{(2)} * \mathbf{L}^{(1)} * \mathbf{A}^{(1)}$ gdzie zapis $\mathbf{A}^{(n)}$ oznacza stan macierzy **A** po $n-1$ krokach.
Następnie oznaczamy przez **U** - $\mathbf{A}^{(n)}$. Wychodząc z $\mathbf{A} = \mathbf{A}^{(1)}$ otrzymujemy:
$\mathbf{U} = \mathbf{L}^{(n-1)}...\mathbf{L}^{(2)}\mathbf{L}^{(1)}\mathbf{A}^{(1)}$, $\mathbf{A} = \mathbf{L}^{(1)^{-1}}\mathbf{L}^{(2)^{-1}}...\mathbf{L}^{(n-1)^{-1}}\mathbf{U}$
gdzie $\mathbf{L} = \mathbf{L}^{(1)^{-1}}\mathbf{L}^{(2)^{-1}}...\mathbf{L}^{(n-1)^{-1}}$.
Z tego od razu wnioskujemy, iż eliminacja Gaussa jest równoważna rozkładowi macierzy na postać **LU**

4.2 Analiza algorytmu

Ze względu na polecenie zadania funkcja **LU()** wyznacza rozkład **LU** macierzy **A** metodą eliminacji Gaussa, natomiast funkcja **LUSolver()** rozwiązuje układ równań $\mathbf{Ax} = \mathbf{b}$ dla wcześniej wyznaczonego rozkładu **LU**

4.3 LU()

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **LU()**:

Algorithm 3 rozkład LU

INPUT:

- matrix- struktura typu *SparseArray*
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

WYNIK:

- Macierz w postaci **LU**

for $k \leftarrow 1$ **to** $n-1$ **do**
  for $i \leftarrow k+1$ **to** $min(n, k+l+1)$ **do**
    if eps(Float64)>abs(matrix[k,k]) **then**
      print:Zero w mianowniku
      break
    end if
    $I \leftarrow matrix[i,k]/matrix[k,k]$
    $matrix[i,k] \leftarrow I$
    for $j \leftarrow k+1$ **to** $min(n, k+l)$ **do**
      $matrix[i,j] \leftarrow matrix[i,j] - I * matrix[k,j]$
    end for
    $b[i] \leftarrow b[i] - I * b[k]$
  end for
end for

Zauważamy, iż jedyną różnicą między ów algorytmem, a algorytmem eliminacji Gaussa jest podstawienie za element $matrix[i,k]$ zmiennej **I**. Wynika to z faktu, iż macierz **U** na przekątnej posiada obliczone wartości, które przy łączeniu z macierzą **I** tworzą spójną całość. Należy zauważyć, iż wszystko trzymane jest w jednej macierzy, podział na dwie jest zbędny, co zostało pokazane we wcześniejszej sekcji, co prowadzi do efektywniejszego zapamiętywania rozkładu jeśli chodzi o wykorzystaną pamięć. **Złożoność:** Występujące pętle są tożsame z pętlami występującymi występującymi w funkcji **gauss()**, stąd wnioskujemy, iż złożoność pamięciowa jak i czasowa jest na poziomie **O(n)**.

4.4 LUSolver()

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **LUSolver()**:

Algorithm 4 Metoda eliminacji Gaussa

INPUT:

- matrix- struktura typu *SparseArray*
- b- wektor prawych stron
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

OUTPUT:

- result- wektor o rozmiarze n zawierający pierwiastki równania

for $k \leftarrow 1$ **to** $n-1$ **do**
  for $i \leftarrow k+1$ **to** $min(n, k+l+1)$ **do**
    $b[i] \leftarrow b[i] - matrix[i,k] * b[k]$
  end for
  $result \leftarrow Array[n] = \{0\}$
  for $a \leftarrow n$ **downto** 1 **do**
    $sum \leftarrow 0$
    for $z \leftarrow a+1$ **to** $min(n, a+l)$ **do**
      $sum \leftarrow sum + matrix[a,z] * result[z]$
    end for
    $result[a] \leftarrow (b[a] - sum)/matrix[a,a]$
  end for
end for
return result

Ów algorytm jest tożsamy z drugą pętlą **FOR** na drugim poziomie zagnieżdżenia w algorytmie eliminacji gaussa. Również złożonośc pamięciowa jak i obliczeniowa ów algorytmu wynosi **O(n)**

Rozkład LU z częściowym wyborem elementu głównego

5.1 Wprowadzenie

Z podobnych powódek jak w metodzie eliminacji Gaussa, przy wykonywaniu rozkładu **LU** można zastosować wyznaczenie częściowego elementu głównego, w celu redukcji błędów związanych z zaokrągleniami oraz wykonywania obliczeń na liczbach bliskich zeru. Podobnie jak w przypadku Rozkładu LU bez częściowego wyboru postały dwie funkcje, **LUPivot()** wyznacza rozkład **LU** macierzy **A** metodą eliminacji Gaussa przy użyciu częściowego wyboru elementu głównego oraz funkcja **LUPivotSolver()** rozwiązująca układ równań $\mathbf{Ax} = \mathbf{b}$ dla wcześniej wyznaczonego rozkładu **LU** z częściowym wyborem elementu głównego

5.2 LUPivot()

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **LUPivot()**:

Algorithm 5 rozkład LU

INPUT:

- matrix- struktura typu *SparseArray*
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

OUTPUT:

- pivot- Macierz indeksów

$pivot \leftarrow \{1, ..., n\}$
for $k \leftarrow 1$ **to** $n-1$ **do**
  $row \leftarrow k$
  $value = abs(matrix[pivot[k], k])$
  for $i \leftarrow k+1$ **to** $min(n, k+l+1)$ **do**
    if $abs(matrix[pivot[i], k]) > value$ **then**
      $value \leftarrow matrix[pivot[i], k]$
      $row \leftarrow i$
    end if
  end for
  $pivot[row], pivot[k] \leftarrow pivot[k], pivot[row]$
  for $i \leftarrow k+1$ **to** $min(n, k+l+l+1)$ **do**
    if eps(Float64)>abs(matrix[pivot[k],k]) **then**
      print:Zero w mianowniku
      break
    end if
    $I \leftarrow matrix[pivot[i], k]/matrix[pivot[k], k]$
    $matrix[pivot[i], k] \leftarrow I$
    for $j \leftarrow k+1$ **to** $min(n, k+l+l)$ **do**
      $matrix[pivot[i], j] \leftarrow matrix[pivot[i], j] - I * matrix[pivot[k], j]$
    end for
    $b[pivot[i]] \leftarrow b[pivot[i]] - I * b[pivot[k]]$
  end for
  $result \leftarrow Array[n] = \{0\}$
  for $a \leftarrow n$ **downto** 1 **do**
    $sum \leftarrow 0$
    for $z \leftarrow a+1$ **to** $min(n, a+l+l)$ **do**
      $sum \leftarrow sum + matrix[pivot[a], z] * result[z]$
    end for
    $result[a] \leftarrow (b[pivot[a]] - sum)/matrix[pivot[a], a]$
  end for
end for
return result

5.3 LUSolver()

Pseudokod algorytmu zaimplementowanego w module **blocksys.jl** w funkcji **LUSolver()**:

Algorithm 6 Metoda eliminacji Gaussa

INPUT:

- matrix- struktura typu *SparseArray*
- b- wektor prawych stron
- n- rozmiar macierzy
- l- rozmiar bloku wewnętrznego

OUTPUT:

- result- wektor o rozmiarze n zawierający pierwiastki równania

for $k \leftarrow 1$ **to** $n-1$ **do**
  for $i \leftarrow k+1$ **to** $min(n, k+l+1)$ **do**
    $b[i] \leftarrow b[i] - matrix[i,k] * b[k]$
  end for
  $result \leftarrow Array[n] = \{0\}$
  for $a \leftarrow n$ **downto** 1 **do**
    $sum \leftarrow 0$
    for $z \leftarrow a+1$ **to** $min(n, a+l)$ **do**
      $sum \leftarrow sum + matrix[a,z] * result[z]$
    end for
    $result[a] \leftarrow (b[a] - sum)/matrix[a,a]$
  end for
end for
return result

Analogicznie do sytuacji z metodą eliminacji Gaussa- **O(n)** jest złożonością zarówno pamięciową jak i czasową.

Wyniki i interpretacje

Poniższe wykresy przedstawiają otrzymane wyniki w ramach testów zaimplementowanych funkcji. Wszystkie testy zostały wykonane dla $l = 4$. Dodatkowo wyniki, na podstawie których ów wykresy zostały wykonane znajdują się w plikach:

- gaussPivotTestResult.txt
- gaussTestResult.txt
- LUTestResult.txt
- LUPivotTestResult.txt

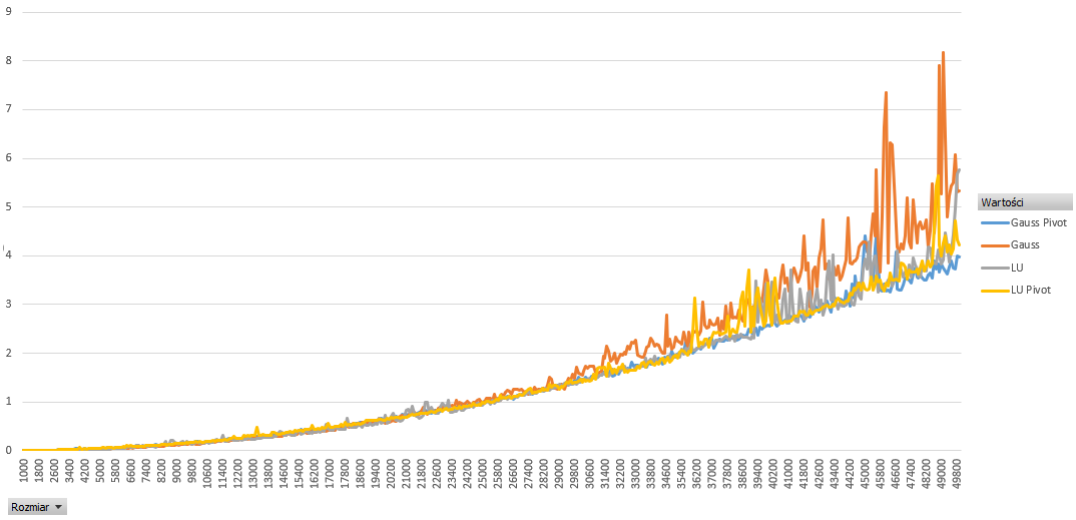

Figure 6.1: Osiągniecia czasowe zaimplementowanych algorytmów

UWAGA: Otrzymane wyniki mogą być narażone na błąd spowodowany innymi procesami działającymi w tle w trakcie wykonywania testów!

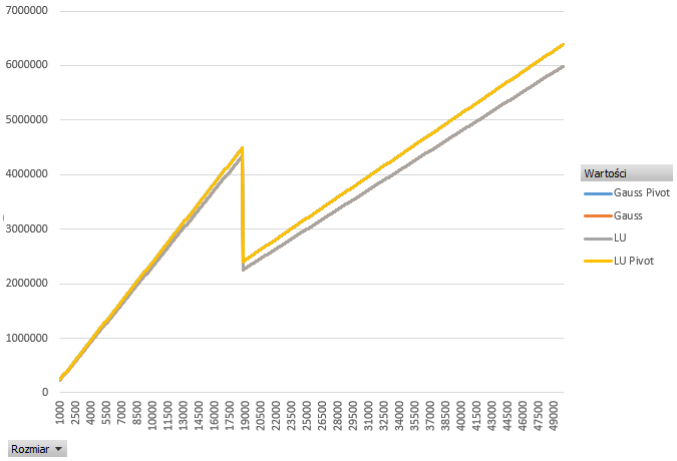

Figure 6.2: Użyta pamięć w zaimplementowanych algorytmów

/	$n = 16$	$n = 10000$	$n = 50000$
Gauss	1.3189010295585306e-15	5.166229763032209e-12	4.920922321731882e-14
Gauss Pivot	2.8576114088871287e-16	4.443944163031559e-16	4.556461336521988e-16
LU	1.3189010295585306e-15	5.166229763032209e-12	4.920922321731882e-14
LU Pivot	2.8576114088871287e-16	4.443944163031559e-16	4.556461336521988e-16

Table 6.1: Wartości błędów względnych dla danych podanych wraz z treścią zadania

Interpretacja: Patrząc na wykres odpowiadający za złożoność pamięciową zaimplementowanych algorytmów zauważamy, iż faktycznie ów złożoność jest liniowa, natomiast niepokój budzą wyniki odnośnie czasu. Zauważamy tendencję paraboliczną dla ów implementacji, co wskazuje na to, że w praktyce ta złożoność liniowa nie jest. Spowodowane jest to faktem, iż dyksretnie założyliśmy stałość wszystkich operacji wykonywanych wewnątrz pętel **FOR**, jednakże okazuje się, iż dostęp do elementów znajdujacych się w *SparseArray* stały nie jest, co odciska swoje piętno na złożności czasowej całego algorytmu. W celu uzyskania złożoności liniowej należałoby skorzystać z innej struktury, lub utworzyć własną, która bierze pod uwagę specyfikację badanych macierzy.

Wnioski

Porównując otrzymane wyniki zauważamy, iż sposób z wykorzystaniem częściowego wyboru elementu głównego zwraca poprawniejsze wyniki, z mniejszym błędem względnym, kosztem nieznacznie większego zużycia pamięci. Czasowo funkcje realizowane są w bardzo podobnym tempie. Dodatkowo zauważamy, iż algorytmy z częściowym wyborem radzą sobie w sytuacji, gdy na przekątnej znajdują się elementy zerowe, czego niestety nie można powiedzieć o ich odpowiednich bez wcześniejszych rotacji.