Obliczenia naukowe

Lista 5

Jakub Brodziński
229781

1 Zadanie 1

1.1 Opis problemu

Celem zadania było napisanie funkcji rozwiązującej układ **Ax=b** metodą eliminacji Gaussa, która uwzględni specyficzną postać macierzy **A**. Funkcja powinna móc rozwiązywać podany układ bez wyboru elementu głównego lub też z częściowym wyborem elementu głównego. Macierz $\mathbf{A} \in R^{n\times n}$ (dla $n \geq 4$)jest macierzą blokową, rzadką o następującej strukturze:

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

gdzie $v = n/l$, zakładając, że n jest podzielne przez l oraz $l \geq 2$ jest rozmiarem wszystkich macierzy wewnętrznych (bloków): $\mathbf{A}_k$, $\mathbf{B}_k$, $\mathbf{C}_k$. $\mathbf{A}_k \in R^{l\times l}$, $k = 1, .., v$ jest macierzą gestą, $\mathbf{0}$ jest kwadratową macierzą zerową stopnia l, macierz $\mathbf{B}_k \in R^{l\times l}$, $k = 2, .., v$ jest następującej postaci:

$$B_k = \begin{pmatrix} 0 & \cdots & 0 & b^k_{1,l-1} & b^k_{1,l} \\ 0 & \cdots & 0 & b^k_{2,l-1} & b^k_{2,l} \\ \vdots & & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & b^k_{l,l-1} & b^k_{l,l} \end{pmatrix}$$

tj. tylko twie ostatnie kolumny nie są kolumnami zerowymi. Macierz $\mathbf{C}_k \in R^{l\times l}$, $k = 1, .., v-1$ jest macierzą diagonalną:

$$C_k = \begin{pmatrix} c^k_1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & c^k_2 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & c^k_{l-1} & 0 \\ 0 & \cdots & 0 & 0 & c^k_l \end{pmatrix}$$

natomiast $b \in R^n$, $n \geq 4$ jest wektorem prawych stron.

1.2 Opis rozwiązania

Metoda eliminacji Gaussa polega na sprowadzeniu układu równań **Ax=b** do postaci $A'x = b'$, gdzie macierz A' jest macierzą trójkątną górną, co czyni układ $A'x = b'$ trywialnym do rozwiązania. Oba równania są sobie równoważne, ze względu na fakt, że przekształcając macierz A korzystamy z elementarnych operacji takich jak zamiana miejscami dwóch równań w układzie, pomnożenie obu stron równania przez liczbę różną od zera oraz dodanie stronami do równania wielokrotności innego równania. W każdym kolejnym kroku eliminacji Gaussa kolejna niewiadoma jest eliminowana z rzędów j takich ,że $i \leq j \leq n$, gdzie $n \times n$ to rozmiar macierzy, a i to numer iteracji algorytmu. W przypadku zwykłego algorytmu wykorzystującego eliminacje Gaussa, macierz gęsta Z po i iteracji wygląda:

$$Z = \begin{pmatrix} z^i_{1,1} & z^i_{1,2} & z^i_{1,3} & \cdots & z^i_{1,i} & z^i_{1,i+1} & \cdots & z^i_{1,n} \\ 0 & z^i_{2,2} & z^i_{2,3} & \cdots & z^i_{2,i} & z^i_{2,i+1} & \cdots & z^i_{2,n} \\ 0 & 0 & z^i_{3,3} & \cdots & z^i_{3,i} & z^i_{3,i+1} & \cdots & z^i_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & z^i_{i,i} & z^i_{3,i+1} & \cdots & z^i_{3,n} \\ 0 & 0 & 0 & \cdots & 0 & z^i_{i,i+1} & \cdots & z^i_{i,n} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & z^i_{n-1,i+1} & \cdots & z^i_{n-1,n} \\ 0 & 0 & 0 & \cdots & 0 & z^i_{n,i+1} & \cdots & z^i_{n,n} \end{pmatrix}$$

Dla macierzy gęstej w każdej iteracji musimy przejść po każdym rzędzie oraz dla każdego rzędu po każdym jego elemencie, co skutkuje złożonością $O(n^3)$. W przypadku macierzy o postaci przedstawionej na początku tej sekcji możemy zmodyfikować standardowy algorytm eliminacji Gaussa, zmniejszając również jego złożność. Macierz wejściowa $\mathbf{A}$ jest takiej postaci, że zerując pierwszą kolumne algorytm musi przejść jedynie po pierwszych l (z wyłączeniem pierwszego) rzędach zerując $a_{i,1}$, gdzie $1 < i \leq l$, jako że od kolejnych rzędów odejmowany jest pierwszy rząd algorytm musi zaktualizować jedynie pierwszych $l+1$ kolumn, ponieważ rząd pierwszy jest postaci:

$$a_1 = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,l} & c_1 & 0 & 0 & 0 & \cdots \end{pmatrix}$$

Ze względu na taką postać rzędu pierwszego, przy eliminowaniu pierwszej niewiadomej, zmianie może ulec pierwse $l+1$ kolumn. W kolejnym kroku eliminowana jest druga niewiadoma, tylko rzędy a_i dla $2 < i \leq l$ nie mają zer w drugiej kolumnie, druga kolumny jest postaci:

$$a_2 = \begin{pmatrix} 0 & a^1_{2,2} & \cdots & a^1_{2,l} & x & c_1 & 0 & 0 & \cdots \end{pmatrix}$$

Korzystając z faktu, żę wiersz a_2 przemnożony przez stały czynnik jest odejmowany od rzędów, w których $a_{i,2} \neq 0$ możemy wywnioskować, że w tym kroku rownież nie musimy aktualizować więcej niż $l+1$ kolumn (ponieważ wiersz a_2 posiada nie więcej niż $l+1$ niezerowych komórek). Przez x została oznaczona wartość otrzymana przez odjęcie od rzędu drugiego rząd pierwszy pomnożony przez wcześniej wyznaczoną stała. Podobnie jak we wcześniejszych iteracjach w $l-1$ iteracji musimy martwić się o zaktualizowanie jedynie $l+1$ kolumn, ponieważ dla rzędu a_{l-1} pierwsze $l-2$ komórek jest równe 0, gdyż te niewiadome zostały wyeliminowane we wcześniejszych iteracjach. Z postaci macierzy wejściowej A wynika natomiast, że po $l-2$ iteracjach rząd a_{l-1} może mieć maksymalnie $l+1$ pól różnych od 0. Poniżej została przedstawiona część macierzy poglądowej, ilustrującej postać przed $l-1$ iteracją. W celu zachowania przejrzystości przedstawiona macierz jest rozmiaru 16×16, dla $v = 4$, gdzie symbolem x zostały oznaczone wartości policzone we wcześniejszych iteracjach, natomiast $a^i_{j,k}$ oznacza komórke $a_{j,k}$ po i iteracjach.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & c_1 & 0 & 0 & 0 & \cdots \\ 0 & a^1_{2,2} & a^1_{2,3} & a^1_{2,4} & x & c_2 & 0 & 0 & \cdots \\ 0 & 0 & a^2_{3,3} & a^2_{3,4} & x & x & c_3 & 0 & \cdots \\ 0 & 0 & a^2_{4,3} & a^2_{4,4} & x & x & x & c_4 & \cdots \\ 0 & 0 & b_{1,3} & b_{1,4} & c_1 & 0 & 0 & 0 & \cdots \\ 0 & 0 & b_{2,3} & b_{2,4} & 0 & c_2 & 0 & 0 & \cdots \\ 0 & 0 & b_{3,3} & b_{3,4} & 0 & 0 & c_3 & 0 & \cdots \\ 0 & 0 & b_{4,3} & b_{4,4} & 0 & 0 & 0 & c_4 & \cdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \end{pmatrix}$$

W l-tej iteracji algorytmu musimy wyeliminować kolejną niewiadomą jedynie w l równaniach, analogicznie martwiąc się jedynie o aktualizacje $l+1$ kolumn. Po l iteracjach algorytmu "wracamy do punktu wyjścia", ponieważ algorytm napotyka analogiczną sytuacje jak w pierwszej iteracji. Algorytm w $l-n$ pierwszych iteracjach napotyka tę samą sytuacje co l iteracji. W przypadku l ostatnich iteracji sytuacja wygląda nieco inaczej. Ponieważ w ostatnich l rzędach w macierzy A nie występują elementy macierzy C_{v-1}, ze względu na to w i-tej iteracji, gdzie $i \in \{n-l+1, n-l+2, .., n-1, n\}$ musimy się martwić o akutalizacje jedynie kolejno $\{l, l-1, ..., 2, 1\}$ kolumn.
Należy pamiętać, że wektor prawych stron b również powininen byc aktualizowany podczas iteracji algorytmu. Wynikiem algorytmu jest macierz trojkątna górna A' oraz zmodyfikowany wektor prawych storn b'. Taki układ możemy rozwiązac w sposób trywialny korzystając ze wzoru:

$$x_i = (b_i - \textstyle\sum_{j=i+1}^{n} a_{i,j} x_j)/a_{i,i}$$

Należy pamietąć, że korzystając z tego wzoru zaczynamy wyznaczanie od x_n "idąc w góre" macierzy A'. W tym konkretnym przypadku korzystając z faktu, że w $n-l$ pierwszych $n-l$ rzędów macierz A' ma jedynie po $l+1$ nie zerowych komórek oraz, że w l ostatnich rzędach liczba niezerowych elementów wynosi $1 \leq i \leq l$ możemy korzystać z takiego wzoru:

$$x_i = \begin{cases} (b_i - \sum_{j=i+1}^{i+l+1} a_{i,j} x_j)/a_{i,i} & x \leq l-n \\ (b_i - \sum_{j=i+1}^{n} a_{i,j} x_j)/a_{i,i} & x > l-n \end{cases}$$

W przypadku wariantu z częściowym wyborem elementu głównego wystarczy, że szukamy elementu głównego jedynie w zasięgu tych rzędów, w których eliminujemy daną niewiadomą, nie w całej macierzy. Wybór elementu głównego sprowadza się do znalezienia permutacji rzędów macierzy, na której później pracujemy, zaczynając od wyznaczenia skali rzędu czyli tablicy s określonej wzorem:

$$s_i = \max_{1 \leq j \leq n} |a_{i,j}| \text{ dla } i = 1, .., n$$

Sposób obliczania tablicy s nie został zmodyfikowany pod względem postaci macierzy wejściowej, ponieważ sposób implementacji struktury $SparseMatrixCSC$ w $Julii$ (struktury w której macierz wejsciowa rzadka jest przechowywana) pozwala na szybkie iterowanie po elementach macierzy kolumnami zarazem posiadając wiedze o rzędzie do którego dana wartość należy.
Przy wyznaczeniu pierwszego wiersza głównego, czyli wiersza dla którego iloraz $|a_{i,1}|/s_i$ jest największy. Niech będzie to j-ty wiersz. Wtedy w permutacji (która początkowo była identycznością) zmieniamy wartości p_1 z p_j. Pierwszego wiersza głównego szukamy w pierwszych l wierszach, ponieważ tylko te wiersze nie maja 0 w pierwszej komórce. Po znalezieniu elementu głównego oraz zmianie jego wskaźnika w permutacji wykonujemy eliminacje pierwszej niewiadomej korzystając z zmodyfikowanej permutacji. Drugiego wiersza głównego szukamy w wierszach $2, .., l$. Jego iloraz $\left|a'_{i,2}\right|/s_i$ musi być największy, modyfikujemy permutacje a następnie eliminujemy kolejną niewiadomą zgodnie z zmodyfikowaną dla tego przykładu eliminacją Gaussa wykorzystując spermutowaną macierz. Przed każdą eliminacją niewiadomej należy wyznaczyć kolejny wiersz główny.
W przypadku wariantu z częściowym wyborem elementu głównego nie możemy korzystać z faktu, że po i-tej iteracji eliminacji Gaussa pierwsze i rzędów ma co najwyżej $l + 1$ obok siebie wartości niezerowych, ponieważ ze względu na permutacje rzędów tablicy wejściowej A przez co możemy spotkać się z sytuacją, gdzie rząd a_k dla $k = \{l, 2l, .., n\}$ może być wierszem głównym, co skutkuje tym, że zostanie zmodyfikowanych do $2l$ kolumn podczas eliminacji. Poniżej znajduje się pseudokod funkcji rozwiązujących układ $Ax = b$ metodą eliminacji Gaussa z częściowym wyborem elementu głównego, które jest rozbudowaniem funkcji rozwiązujacej ten sam układ bez wyboru elementu głównego. Składaja się na to trzy funckje.

```
1:  function gaussDomKdownMright(A,b,row,k,m,p,s)
2:      swp ← x takie, że p_x = max_{row≤j≤row+k} |A[p_j, row]|/s_r
3:      swap(p_row, p_swp)
4:      swap(s_row, s_swp)
5:      for i = row + 1 to row + k do
6:          max ← row + m − 1
7:          if max > size(A) then
8:              max ← size(A)
9:          end if
10:         factor ← A[p_i, row]/A[p_row, row]
11:         for g = row to max do
12:             A[p_i, g] ← A[p_i, g] − factor ∗ A[p_row, g]
13:         end for
14:         b[p_i] ← b[p_i] − factor ∗ b[p_row]
15:     end for
16: end function
```

Funkcja przedstawiona powyżej eliminuje niewiadomą z kolumny row, która znajduje się w wierszach $\{row, row + 1, .., row + k\}$, po tym jak zostanie wybrany wiersz główny (linijki $2 - 4$), na podstawie skali rzędu w tablicy s oraz permutacji w tablicy p, funkcja przy eliminacji Gaussa zaktualizuje kolumny $\{row + 1, .., row + m\}$ (linijki $5 - 15$).

```
1: function gaussDom(A,b,n,l)
2:     for i = 1 to n do
3:         s[i] ← s_i = max_{1≤j≤n} |a_{i,j}|
4:     end for
5:     p ← id
6:     it ← l
7:     for i = 1 to n − l do
8:         if it > 2 then
9:             gaussDomKdownMright(A, b, i, it − 1, 2 ∗ l, p, s)
10:        else
11:            gaussDomKdownMright(A, b, i, it + l − 1, 2 ∗ l, p, s)
12:        end if
13:        it ← (it − 1)_{mod l}
14:    end for
15:    it ← l
16:    for i = n − 1 + 1 to n do
17:        gaussDomKdownMright(A, b, i, it − 1, it, p, s)
18:        it ← (it − 1)_{mod l}
19:    end for
20:    it ← 1
21:    for i = n downto n − l do
22:        findXiDom(A, b, i, it, p, s)
23:        it ← (it + 1)
24:    end for
25:    for i = n − l − 1 downto 1 do
26:        findXiDom(A, b, i, 2 ∗ l, p, s)
27:    end for
28:    return p ∗ b
29: end function
```

Funkcja przedstawiona powyżej pobiera macierz A oraz wektor prawych stron b, jak rownież rozmiar macierzy A wraz z rozmiarem macierzy, której składa się macierz blokowa A. Zwraca wektór $(x_1, ..., x_n)$, który zawiera rozwiązanie układu $Ax = b$. Bierze pod uwage wszystkie przypadki, o których była mowa wcześniej. Po obliczeniu wartości tablicy s (linijki $2 - 4$) eliminuje $n - l$ kolejnych niewiadomych (linijki $6 - 13$) w $n - l$ pierwszych rzędach, natrafiając cyklicznie co l na identyczną sytuacje. Następnie zostaje wykonują się ostatnie l iteracji eliminacji Gaussa (linijki $15 - 18$), a reszta funkcji to rozwiązanie trywialnego równania $A'x = b'$

```
1: function findXiDom(A,b,row,depth,p,s)
2:     max ← row + depth − 1
3:     if max > size(A) then
4:         max ← size(A)
5:     end if
6:     for i = row + 1 to max do
7:         b[p_row] ← b[p_row] − A[p_row, i] ∗ b[p_i]
8:         A[p_row, i] ← 0
9:     end for
10:    b[p_row] ← b[p_row]/A[p_row, row]
11: end function
```

Ostatnia funkcja jest implementacją wzoru poniżej, który wcześniej został wytłumaczony, pozwalał on rozwiązac układ $A'x = b'$, gdzie A' to macierz trójkątna górna. Funkcja zaimplementowana w Julii pracuje na tablicy b zamiast dodatkowej tablicy x, co skutkuje mniejszym zużyciem pamięci podręcznej.Przyjmuje ona na wejsciu macierz trójkątną górną, wektor prawych stron, rząd w którym ma pracować (przy założeniu, że każdy rząd z $\{row + 1, .., n - 1, n\}$ został już wcześniej policzon), jak głęboko ma szukać wartości w danych rzędach oraz wektor permutacji.

$$x[row] = \begin{cases} (b[p_{row}] - \sum_{i=row+1}^{row+depth-1} a[p_{row}, i]x_i)/A[p_{row}, row] & row + depth - 1 \leq n \\ (b[p_{row}] - \sum_{i=row+1}^{n} a[p_{row}, i]x_i)/A[p_{row}, row] & row + depth - 1 > n \end{cases}$$

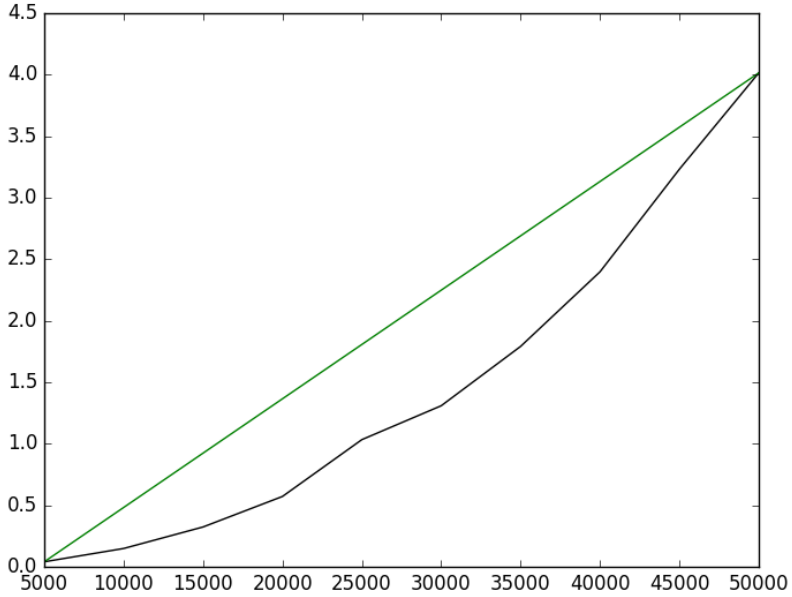
1.3 Wyniki

W tabeli poniżej zostały przedstawione błędy względne w zależności od rozmiaru macierzy blokowej, jak rownież macierzy wewnętrznych dla funkcji z wariantem częściowego wyboru elementu głównego jak rownież wariantu bez wyboru elementu głównego dla macierzy o wskaźniku uwarunkowania 10.0.

n	*wariant bez elementu gwnego*		*wariant z elementem gwnym*	
	$l = 4$	$l = 5$	$l = 4$	$l = 5$
5000	$1.3345928314674054e-14$	$1.0432921655595746e-14$	$1.3428450346798256e-14$	$1.0527117624716868e-14$
10000	$4.309438733486324e-14$	$2.8152619811308132e-14$	$4.3082991718942607e-14$	$2.971327360914021e-14$
15000	$2.990073308165054e-14$	$1.2765196102161818e-13$	$5.527191782833452e-14$	$1.2779439819433568e-13$
20000	$1.1320922312971717e-12$	$1.8845055663747355e-13$	$1.1337506822274095e-12$	$1.8851978472780802e-13$
25000	$2.6594183739953338e-14$	$7.124237970699825e-14$	$2.6804866769433998e-14$	$7.313274825470416e-14$
30000	$1.0660395978279995e-13$	$4.2395150469991125e-14$	$1.1491069358960369e-13$	$4.2817054121022945e-14$
35000	$2.125770206973623e-13$	$9.05067310931645e-14$	$2.9239648608116325e-13$	$9.134079663128835e-14$
40000	$3.25149690662533e-14$	$7.318542167173057e-14$	$3.534428144571841e-14$	$7.76707369096008e-14$
45000	$5.487895274086159e-14$	$4.97906807376808e-14$	$5.508212359397843e-14$	$5.1066986353220836e-14$
50000	$1.4516793222710255e-12$	$2.581504179274979e-12$	$1.4517059370135028e-12$	$2.5847476918927036e-12$

W drugiej tabeli zostały zapisane czasy, w trakcie których algorytm policzył rozkład Gaussa oraz wyznaczył wektor $x = (x_1, ..., x_n)$ dla $l \in \{4, 5\}$ oraz $n \in \{1000i : i \in \{5, 10, ..., 50\}\}$. Pod tabelą z czasami znajduje się wykres zależności rozmiaru macierzy A od czasu dla $l = 4$.

n	*wariant bez elementu gwnego*		*wariant z elementem gwnym*	
	$l = 4$	$l = 5$	$l = 4$	$l = 5$
5000	$0.02707234s$	$0.054348988s$	$0.040612958s$	$0.070535455s$
10000	$0.157828962s$	$0.226459116s$	$0.14885509s$	$0.276497047s$
15000	$0.319136371s$	$0.593793937s$	$0.323671504s$	$0.545769766s$
20000	$0.584740938s$	$0.955125879s$	$0.571654866s$	$0.885067485s$
25000	$0.87263218s$	$1.691298331s$	$1.033474966s$	$1.61737976s$
30000	$1.267299485s$	$2.069301047s$	$1.309104571s$	$3.008853431s$
35000	$1.738218298s$	$3.013875774s$	$1.791567045s$	$3.502696003s$
40000	$2.266891683s$	$3.613615407s$	$2.397898831s$	$4.285924386s$
45000	$2.913301529s$	$4.561855497s$	$3.231910063s$	$6.016201141s$
50000	$3.59854977s$	$7.233716675s$	$4.01503439s$	$5.70034519s$



Rysunek 1: Zależnośc czasu od n dla $l = 4$ z wyborem elementu głównego

1.4 Wnioski

1.4.1 Złożoność obliczeniowa

Pętla w funkcji $findXiDom$ wykona się $max - row - 1$ razy, co w najgorszym przypadku wynosi $depth$ razy, z czego wnioskujemy, że złożoność obliczeniowa tego algorytmu to $O(depth)$, a pamięciowa jest równa $O(1)$. Analogicznie wewnętrzna pętla for w funkcji $gaussDomKdownMright$ wykona się m razy, przy każdym z k wykonań pętli zewnętrznej. Aby wyznaczyć wartość zmiennej swp potrzeba k iteracji pętli, z czego wynika, że złożonośc obliczeniowa funkcji $gaussDomKdownMright$ to $O(k*m)$, natomiast pamięciowa jest równa $O(1)$. Dzięki wykorzystaniu struktury danych stworzonej przez Julie złożoność obliczeniowa i pamięciowa obliczenia tablicy s to kolejno $O(n*(l+3)$ oraz $O(n)$.Zastępując wywołanie funkcji jej złożonością czasową symbolicznie możemy zapisać (ograniczając zmienną it zmienną l od góry):

1: **function** $gaussDom(A,b,n,l)$
2: $O(n*l)$
3: $it \leftarrow l$
4: **for** $i = 1$ **to** $n - l$ **do**
5: $O(l^2)$
6: $it \leftarrow (it-1)_{mod\ l}$
7: **end for**
8: $it \leftarrow l$
9: **for** $i = n - 1 + 1$ **to** n **do**
10: $O(l^2)$
11: $it \leftarrow (it-1)_{mod\ l}$
12: **end for**
13: $it \leftarrow 1$
14: **for** $i = n$ **downto** $n - l$ **do**
15: $O(l)$
16: $it \leftarrow (it+1)$
17: **end for**
18: **for** $i = n - l - 1$ **downto** 1 **do**
19: $O(l)$
20: **end for**
21: **return** $p * b$
22: **end function**

Z czego wynika, że złożonośc obliczeniowa algorytmu jest równa $O(n*l^3)$, co przy założeniu, że l jest stałą jest równe $O(n)$. W przypadku złożoności pamięciowej możemy algorytm symbolicznie zapisać w sposób jak poniżej, zastępując wywołanie funkcji jej złożonością pamięciową.

1: **function** $gaussDom(A,b,n,l)$
2: $O(n)$
3: $it \leftarrow l$
4: **for** $i = 1$ **to** $n - l$ **do**
5: $O(1)$
6: $it \leftarrow (it-1)_{mod\ l}$
7: **end for**
8: $it \leftarrow l$
9: **for** $i = n - 1 + 1$ **to** n **do**
10: $O(1)$
11: $it \leftarrow (it-1)_{mod\ l}$
12: **end for**
13: $it \leftarrow 1$
14: **for** $i = n$ **downto** $n - l$ **do**
15: $O(1)$
16: $it \leftarrow (it+1)$
17: **end for**

```
18:     for i = n − l − 1 downto 1 do
19:         O(1)
20:     end for
21:     return p ∗ b
22: end function
```

wynika z tego jasno, że złożonośc pamięciowa algorytmu jest równa złożoności czasowej i wynosi $O(n)$.

1.4.2 Wybór wierszy głównych

Dobry algorytm wykorzystujący metode eliminacji Gaussa powinien uwzględniać przestawianie rownań układu, gdy wymagają tego okoliczności, ponieważ tak naprawde nie wykorzystując jedynie tablice permutacji (a nie modyfikując macierzy) nie przedłuża to obliczeń,a chroni nas przed sytuacja, gdzie element główny jest zerem, lub też jest bardzo mały w porównaniu do reszty wiersza, co może skutkować zwiększeniem błędów występujących w obliczeniach.

2 Zadanie 2

2.1 Opis problemu

Celem zadania było napisanie funckji wyznaczającej rozkład LU macierzy A metodą eliminacji Gaussa uwzględniająca specyficzną postać macierzy wejściowej w dwóch wariantach: bez wyboru elementu głównego oraz z częściowym wyborem elementu głównego.

2.2 Opis rozwiązania

Wyznaczenie rozkłądu LU mocno bazuje na zadaniu numer 1, gdzie macierz końcowa A' była tak naprawdę macierzą U, z taką różnicą, że podczas wyznaczania rozkładu LU nie uzyskujemy rozwiązania układu. Algorytm który został zaimplementowany przeze mnie w ten sam sposób jak algorytm z zadania 1 eliminuje kolejne zmienne z kolejnych rzędów lecz przy każdej eliminacji zapisuje czynnik przez który przemnaża wiersz, który odejmuje od innych wierszy w macierzy L. Algorytm ten jest mocno związany z sposobem inicjalizacji struktury danych w języku programowania $Julia$ w którym możemy przechowywać efektywnie macierze rzadkie. W dalej części opisu algorytmu będe odniosił się tylko i wyłącznie do algorytmu z wariantem wyboru elementu głównego, gdyż jest on bardziej rozbudowaną wersją algorytmu bez tego wariantu, a w przypadku, gdy wektor $p = id$ otrzymamy takie same wyniki w obu algorytmach. W przypadku obu macierzy L i U pracując na nich wykorzystujemy permutacje, następnie na końcu algorytmu permutujemy macierze i zwracamy trójkę (L, U, b). Mimo, że nie zmieniamy poszczególnych wartości w wektorze prawych stron, to również i jego musimy spermutować. Zachodzi równość $LU = PA$, co zostanie pokazane później. Poniżej został przedstawiony pseudokod wraz z krótkim komentarzem.

1: **function** $gaussDomKdownMrightLU(A,row,k,m,p,s)$
2: $sparse_{row} \leftarrow [\,]$
3: $sparse_{column} \leftarrow [\,]$
4: $sparse_{value} \leftarrow [\,]$
5: $swp \leftarrow x$ takie, że $p_x = \max_{row \le j \le row+k} |A[p_j, row]|/s_r$
6: $swap(p_{row}, p_{swp})$
7: $swap(s_{row}, s_{swp})$

8: $sparse_{row}.push(p_{row})$
9: $sparse_{column}.push(row)$
10: $sparse_{value}.push(1)$
11: $sparse_{row}.push(p_{row})$
12: $sparse_{column}.push(row)$
13: $sparse_{value}.push(1.0)$
14: **for** $i = row + 1$ **to** $row + k$ **do**
15: $max \leftarrow row + m - 1$
16: **if** $max > size(A)$ **then**
17: $max \leftarrow size(A)$
18: **end if**
19: $factor \leftarrow A[p_i, row]/A[p_{row}, row]$
20: **for** $g = row$ **to** max **do**
21: $A[p_i, g] \leftarrow A[p_i, g] - factor * A[p_{row}, g]$
22: **end for**
23: $sparse_{row}.push(p_i)$
24: $sparse_{column}.push(row)$
25: $sparse_{value}.push(factor)$
26: **end for**
27: **return** $(sparse_{row}, sparse_{column}, sparse_{value})$
28: **end function**

Funkcja przedstawiona powyżej eliminuje niewiadomą z kolumny row, która znajduje się w wierszach $\{row, row + 1, .., row + k\}$, po tym jak zostanie wybrany wiersz główny (linijki $2 - 4$), na podstawie skali rzędu w tablicy s oraz permutacji w tablicy p, funkcja przy eliminacji Gaussa zaktualizuje kolumny

$\{row+1, .., row+m\}$ (linijki $5-15$). Aby później zbudować macierz rzadką L musimy posiadać trzy tablice, posiadające w odpowiadających sobie indeksach wiersz,kolumne oraz wartość komórki, dlatego też w tabliach $sparse_{row}$,$sparse_{column}$,$sparse_{value}$ są przechowywane czynniki przez które przemnażamy wiersze, w tych eliminujemy zmienne podczas metody eliminacji Gaussa. W odróżnieniu do poprzedniego zadania wektor b nie jest aktualizowany.

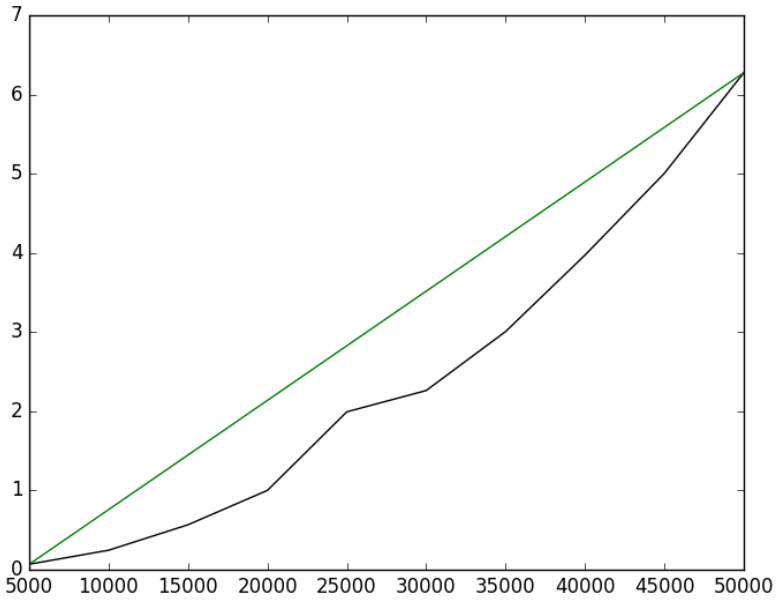
1: **function** $LUmatrixesDom(A,b,n,l)$
2: $sparse_{row} \leftarrow [\,]$
3: $sparse_{column} \leftarrow [\,]$
4: $sparse_{value} \leftarrow [\,]$
5: **for** $i = 1$ **to** n **do**
6: $s[i] \leftarrow s_i = \max_{1 \le j \le n} |a_{i,j}|$
7: **end for**
8: $p \leftarrow id$
9: $it \leftarrow l$
10: **for** $i = 1$ **to** $n - l$ **do**
11: **if** $it > 2$ **then**
12: $(r, c, v) \leftarrow gaussDomKdownMrightLU(A, i, it-1, 2*l, p, s)$
13: **else**
14: $(r, c, v) \leftarrow gaussDomKdownMrightLU(A, b, i, it+l-1, 2*l, p, s)$
15: **end if**
16: $sparse_{row}.append(r)$
17: $sparse_{column}.append(c)$
18: $sparse_{value}.append(v)$
19: $it \leftarrow (it-1)_{mod\ l}$
20: **end for**
21: $it \leftarrow l$
22: **for** $i = n - 1 + 1$ **to** n **do**
23: $(r, c, v) \leftarrow gaussDomKdownMrightLU(A, i, it-1, it, p, s)$
24: $it \leftarrow (it-1)_{mod\ l}$
25: $sparse_{row}.append(r)$
26: $sparse_{column}.append(c)$
27: $sparse_{value}.append(v)$
28: **end for**
29: $L \leftarrow sparse(sparse_{row}, sparse_{column}, sparse_{value})$
30: **return** $(p \circ L, p \circ A, p \circ b)$
31: **end function**

Algorytm w analogiczny sposób do algorytmu z pierwszego zadania wywołuje funkcje $gaussDomKdownMrightLU$, która zwraca współrzędne oraz wartości części macierzy L wyznaczonej przez tą funkcje, pracuje ona na referencji macierzy A dlatego też owa macierz staje się macierzą U. Aby funkcja $LUmatrixesDom$ zwracała macierz trójkątną dolna oraz trójkątna górna przez zakończeniem funkcji wiersze macierzy L i U (nasza wejściowa macierz A) są permutowane zgodnie z permutacją p co zostało zaznaczone symbolicznie w pseudokodzie jako np $p \circ L$.

2.3 Wyniki

Tabela poniżej przedstawia czas wywołań zaimplementowanego przez mnie algorytmu dla $n \in= \{5000, 10000, ..., 50000\}$ oraz dla $l \in \{4, 5\}$. Wykres natomiast prezentuje zależnośc $(f(n))$ czasu od rozmiaru macierzy wejściowej dla funkcji z wyborem elementu głównego dla $l = 4$ oraz linią prostą przechodząca przez pierwszy i ostatni punkt $f(n)$

n	*wariant bez elementu glownego*		*wariant z elementem glownym*	
	$l = 4$	$l = 5$	$l = 4$	$l = 5$
5000	$0.034873598s$	$0.057140037s$	$0.044189743s$	$0.069281358s$
10000	$0.149162337s$	$0.22606255s$	$0.163973052s$	$0.24585715s$
15000	$0.378810569s$	$0.515256262s$	$0.344433418s$	$0.567440066s$
20000	$0.652203852s$	$0.966692766s$	$0.607166089s$	$1.001632374s$
25000	$0.998385502s$	$1.382362857s$	$0.950785276s$	$1.993403493s$
30000	$1.477835342s$	$2.430161414s$	$1.410933703s$	$2.262357833s$
35000	$1.866421816s$	$2.978352526s$	$1.933857517s$	$3.006949317s$
40000	$2.473494649s$	$3.809623088s$	$2.421870202s$	$3.972160413s$
45000	$3.395498598s$	$5.075152178s$	$4.213304299s$	$5.005106047s$
50000	$4.423652015s$	$7.149150898s$	$6.778759442s$	$6.275006387s$



Rysunek 2: Zależnośc czasu od n dla $l = 4$ z wyborem elementu głównego

2.4 Wnioski

Obie funkje bardzo mocno bazują na funkcjach z zadania pierwszego, których złożność wynosiła $O(n)$ przy uznaniu, że l była stałą. Przy założeniu, że funkcja $Array.push(Integer)$ jest $O(1)$ nie zmienia ona nam asymptotycznej złożności algorytmu, który asympottycznie zbiega $O(l^2)$. Przy założeniu, że l jest stałą funkcja $LUmatrixesDom$ analogicznie do funkcji $gaussDom$ asympotycznie zbiega do $O(n)$. Co do złożoności pamięciowej funkcji $LUmatrixesDom$ zostaje stoworzona tablica s $(O(n))$, permutacja p ($O(n)$) oraz trzy struktury $sparse_{row}$, $sparse_{column}$, $sparse_{value}$, których długość możemy odgórnie ograniczyć względem ilości elementów w macierzy wejściowych, których jest maksymalnie $n * (l + 3)$, ponieważ ze struktury macierzy wejściowej wynika, że na początku w danym rzędzie jest co najwyżej

$l+3$ elementów. Wynika z tego, że złożnośc pamięciowa funkcji $LUmatrixesDom$ asympottycznie zbiega do $n * l$, co przy założeniu, że l jest stałe możemy ograniczyć jakos $O(n)$.

3 Zadanie 3

3.1 Opis problemu

Celem zadania było zaimplementowanie funckji rozwiązującej układ równań $Ax = b$ przy uwzględnieniu specyficznej postaci macierzy A posiadając jej rozkład LU taki, że $LU = PA$, który został wyznaczony przez funkcje z zadania drugiego.

3.2 Opis rozwiązania

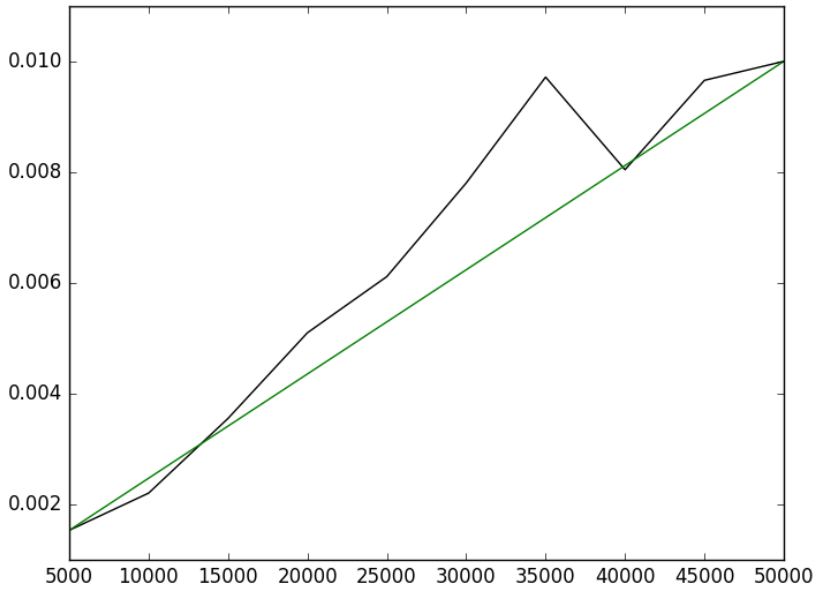
Zadanie sprowadza się do rozwiązania dwóch równań $Lz = b$ względem z, a następnie $Ux = z$ względem x. L to macierz trójkątna dolna, natomiast U to macierz trójkątna górna. Z algorytmu wyznaczania macierzy L wynika, że w i-tej kolumnie poniżej przekątnej posiada ona tyle samo niezerowych komórek co liczba wierszów w ktorej eliminowaliśmy niewiadomą x_i w macierzy wejściowej A. Opierając się na wnioskach wyciągniętych w zadaniu 1 dla macierzy A takiej, że $n = 16$ oraz $l = 4$ będzie to kolejno $3, 2, 5, 4, 3, 2, 5, 4..., 5, 4, 3, 2, 1, 0$. Zauwazając pewną regularność możemy zoptymalizować algorytm rozwiązywania równania z macierzą trojkątna dolna, aby zawsze "zaglądał" do komórek tylko niepustych. Poniżej znajduje się pseudokod zaimplementowanego algorytmu.

```
1:  function findXWithLU(L,U,b,n,l)
2:      off ← l − 1
3:      for i = 1 to l − 2 do
4:          for j = 1 to off do
5:              b[i + j] ← b[i + j] − L[i + j, i] ∗ b[i]
6:          end for
7:          off ← off − 1
8:      end for
9:      off ← l + 1
10:     for i = 1 to l − 2 do
11:         for j = 1 to off do
12:             b[i + j] ← b[i + j] − L[i + j, i] ∗ b[i]
13:         end for
14:         off ← off − 1
15:         if off < 2 then
16:             off ← l + 1
17:         end if
18:     end for
19:     b[n] ← b[n] − b[n − 1] ∗ L[n, n − 1]
20:     for i = n downto 1 do
21:         sum ← 0.0
22:         j ← 1
23:         while j < 2 ∗ l && (i + j) ≤ n do
24:             sum ← sum + U[i, i + j] ∗ x[i + j]
25:             j ← j + 1
26:         end while
27:         x[i] ← (b[i] − sum)/U[i, i]
28:     end for
29:     return x
30: end function
```

Po rozwiązaniu pierwszego równania w linijkach $3 - 19$ rozwiązujemy drugi układ w ten sam sposób jak w zadaniue 1 w linijkach $20 - 28$. Funkcja zwraca wektor zawierający rozwiązanie układu $Ax = b$

3.3 Wyniki

Wykres przedstawia zależność czasu wykonywania funckji $findXWithLU$ od rozmiaru macierzy dla $l = 4$ oraz $n = 5000, 10000, ..., 50000$.



Rysunek 3: Zależnośc czasu od n dla $l = 4$

3.4 Wnioski

Zmienna off jest zawsze mniejsza bądz równa $l+1$ tak więc pierwszą część algorytmu możemy ograniczyć jako $O(n * l)$. W drugiej części pętla $while$ wykonuje się zawsze co najwyżej $2l$ razy tak więć i drugą część zadania możemy ograniczyć $O(n*l)$ z czego wynika, że cały algorytm przy założeniu, że l jest stałą asympottycznie zbiega do $O(n)$. Jako, że zwracamy nowy wektor x zadeklarowan w ciele funkcji, to jej złożonośc pamięciowa zbiega do $O(n)$.