Obliczenia naukowe. Sprawozdanie z listy 5.

Maciej Bazela 261743

29 grudnia 2022

Spis treści

1 Opis problemu

Zadaniem na tej liście było zaimplementowanie algorytmów rozwiązywania układu równań liniowych:

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

gdzie $\mathbf{A}$ to macierz współczynników $A \in \mathbb{R}^{n \times n}$, a $\mathbf{b}$ to wetkor prawych stron $b \in \mathbb{R}^n$, $n \geqslant 4$.

Ponadto macierz $\mathbf{A}$ jest macierzą rzadką o poniższej blokowej strukturze:

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

gdzie $v = n/l$ (n jest podzielne przez l), $l \geqslant 2$ jest rozmiarem kwadratowych ($\mathbb{R}^{l \times l}$) bloków $\mathbf{A_i}$, $\mathbf{B_i}$, $\mathbf{C_i}$, a $i = 1, \ldots, v$.

Każdy z bloków ma swoją własną strukturę:

- Bloki $\mathbf{A_i}$ są macierzami gęstymi o rozmiarze $l \times l$.
- $\mathbf{0}$ jest macierzą zerową stopnia l.
- Bloki $\mathbf{B_i}$ są macierzami o następującej postaci:

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

- Bloki $\mathbf{C_i}$ są macierzami diagonalnymi:

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

Wartości współczynników macierzy są liczbami zmiennoprzecinkowymi w arytmetyce *Float64*.

Problemem tej listy jest efektywne przechowywanie macierzy (z pominięciem współczynników zerowych) oraz adaptacja standardowych algorytmów rozwiązywania układów równań liniowych, tak, aby zredukować czas ich wykonywania do $O(n)$.

2 Sposób przechowywania macierzy

W celu przechowywania macierzy w postaci blokowej, stworzyłem strukturę *BlockSparseMatrix*, która korzystając z własności zadanej macierzy, przechowuje jedynie niezerowe elementy w obrębie 2-3 bloków w wierszu macierzy.

Przed wczytywaniem wartości macierzy wyliczane są zakresy niezerowych elementów dla każdego wiersza/kolumny w naszej macierzy. W ten sposób, każdy wiersz możemy zapisać w postaci tablicy, która

zawiera jedynie niezerowe elementy, a odwoływanie się do nich polega na przesunięciu szukanego indeksu o początek niezerowego zakresu dla danego wiersza.

Do wyznaczania zakresów niezerowych elementów stworzyłem 4 funkcje:

- **first_nonzero_row** - zwraca indeks pierwszego niezerowego wiersza dla danej kolumny
- **first_nonzero_column** - zwraca indeks pierwszego niezerowego wiersza dla danego wiersza
- **last_nonzero_row** - zwraca indeks ostatniego niezerowego wiersza dla danej kolumny
- **last_nonzero_column** - zwraca indeks ostatniego niezerowego wiersza dla danego wiersza

2.1 Funkcja first_nonzero_row

Rozważmy poniższą macierz ($n = 9$, $l = 3$):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Możemy zauważyć, że pierwszy niezerowy wiersz zależny jest od wielkości bloku i numeru kolumny, którą rozpatrujemy.

Aby uzyskać numer pierwszego niezerowego wiersza wystarczy policzyć $col - l$. W przypadku gdy $col - l < 1$, pierwszy niezerowy wiersz jest równy 1, więc możemy to zapisać jako:

$$\text{first\_nonzero\_row}(l,\ col) = \max(1,\ col - l)$$

2.2 Funkcja first_nonzero_column

Rozważmy poniższą macierz ($n = 9$, $l = 3$):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Znalezienie wzoru na pierwszą niezerową kolumnę jest nieco bardziej skomplikowane.

Zapiszmy, które kolumny są niezerowe dla danego wiersza w ciągu:

$$\begin{array}{ll} \text{wiersz:} & 1, 2, 3, 4, 5, 6, 7, 8, 9 \\ \text{niezerowa kolumna:} & 1, 1, 1, 1, 3, 3, 4, 6, 6 \end{array}$$

Oznaczę kolorami dwa typy wartości w ciągu:

wiersz:	1, 2, 3, 4, 5, 6, 7, 8, 9
niezerowa kolumna:	1, 1, 1, 1, 3, 3, 4, 6, 6

Szukanie pierwszej niezerowej kolumny możemy rozbić na dwa przypadki:

- Jeżeli $row - 1 \mod l \equiv 0$, to pierwsza niezerowa kolumna znajduje się w kolumnie $row - l$.
- W przeciwnym wypadku, pierwsza niezerowa kolumna znajduje się w kolumnie $\lfloor \frac{row-1}{l} \rfloor \cdot l$.

Sprawdzając powyższe wzory dla macierzy z poprzedniego przykładu, otrzymujemy:

$$
\begin{aligned}
&l = 3, \\
&row = 1,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{1-1}{3} \rfloor \cdot 3 = 0 \\
&row = 2,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{2-1}{3} \rfloor \cdot 3 = 0 \\
&row = 3,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{3-1}{3} \rfloor \cdot 3 = 0 \\
&row = 4,\ \text{first\_nonzero\_column}(l,\ row) = 4 - 3 = 1 \\
&row = 5,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{5-1}{3} \rfloor \cdot 3 = 3 \\
&row = 6,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{6-1}{3} \rfloor \cdot 3 = 3 \\
&row = 7,\ \text{first\_nonzero\_column}(l,\ row) = 7 - 3 = 4 \\
&row = 8,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{8-1}{3} \rfloor \cdot 3 = 6 \\
&row = 9,\ \text{first\_nonzero\_column}(l,\ row) = \lfloor \frac{9-1}{3} \rfloor \cdot 3 = 6
\end{aligned}
$$

Aby pozbyć się zerowych wartości dla pierwszych l argumentów row wystarczy zastosować funkcję max:

$$
\text{first\_nonzero\_column}(l,\ row) = \begin{cases} row - l : (row - 1) \mod l \equiv 0 \\ \max(1, \lfloor \frac{row-1}{l} \rfloor \cdot l) : \text{otherwise} \end{cases}
$$

2.3 Funkcja last_nonzero_row

Rozważmy poniższą macierz (z poprzedniego przykładu):

$$
\begin{bmatrix}
a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\
a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\
a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\
b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\
0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\
0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\
0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\
0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\
0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99}
\end{bmatrix}
$$

W tym wypadku, możemy zauważyć, że jeśli $col \mod l \equiv 0$, to ostatni niezerowy wiersz znajduje się w wierszu $col + l$. W przeciwnym wypadku musimy dodawać do wielokrotności l jedynkę.

Dla $col < l$ mamy ostatnie niezerowe wiersze: $1 \cdot l + 1$.

Dla $l < col < 2l$ mamy ostatnie niezerowe wiersze: $2 \cdot l + 1$.

etc.

W takim razie, możemy zwinąć to do wzoru:

$$\text{last\_nonzero\_row}(n, l,\ col) = \begin{cases} col + l : col \bmod l \equiv 0 \\ \min(n, \lceil \frac{col}{l} \rceil \cdot l + 1) : \text{otherwise} \end{cases}$$

$min(n, \dots)$ jest potrzebne, aby uniknąć wyjścia poza zakres macierzy.

2.4 Funkcja last_nonzero_column

Rozważmy poniższą macierz (z poprzedniego przykładu):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Jak widzimy wartości *last_nonzero_column* są zależne od niezerowych wartości c_{ik}.

Analogicznie do funkcji *first_nonzero_row* musimy po prostu do numeru wiersza dodać l oraz brać $min(n, \dots)$, aby nie wyjść poza zakres macierzy:

$$\text{last\_nonzero\_column}(l,\ row) = \min(n, row + l)$$

2.5 Przechowywanie wartości współczynników

Mając wszystkie funkcje, możemy w czasie budowania nowej macierzy wyliczać dla każego wiersza i kolumny zakresy niezerowych elementów. Wykorzystujemy te zakresy, aby alokować pamięć na macierz.

Wartości współczynników przechowujemy w tablicy tablic, gdzie każda tablica to wiersz z niezerowymi elementami macierzy.

W takim razie, skoro mamy n wierszy, to dla każdego wiersza będziemy alokować tablice długości *last_nonzero_column* - *first_nonzero_column* + 1.

W taki sposób nie tracimy pamięci na niezerowe elementy, które i tak są nam niepotrzebne podczas obliczeń:

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

W każdym wierszu przechowujemy co najwyżej $2l+1$ niezerowych wartości, więc złożoność pamięciowa macierzy wynosi $O(n \cdot (2l+1)) = O(n)$.

2.6 Odwoływanie się do wartości współczynników

Aby zapewnić odpowiednie zwracanie i modyfikowanie wartości macierzy musimy pamiętać, że każdy wiersz ma jakieś przesunięcie.

Dlatego przeciążając bazowe funkcje z języka *Julia*: *getindex* i *setindex*! możemy uwzględnić w nich offset, który zawsze będzie wynosił *first_nonzero_column*:

```
function Base.getindex(
  matrix::BlockSparseMatrix,
  row::Int64,
  column::Int64
)::Float64
  matrix.observability.no_getindex += 1

  if !(column in matrix.row_ranges[row])
    return 0.0
  end

  return matrix.values[row][column - matrix.row_ranges[row].start + 1]
end
```

gdzie *row_ranges* to mapa zakresów niezerowych elementów w każdym wierszu (tworzona podczas konstruowania obiektu *BlockSparseMatrix*).

Co więcej, jako że interesuje nas liczba odwołań do wartości współczynników macierzy, to możemy podczas wywołania tej funkcji zinkrementować zmienną, która odnotowuje tę liczbę.

Jeżeli próbujemy odwołać się do wartości spoza zakresu, to zwracamy 0.0.

```
function Base.setindex!(
  matrix::BlockSparseMatrix,
  value::Float64,
  row::Int64,
  column::Int64
)
  matrix.observability.no_setindex += 1

  if !(column in matrix.row_ranges[row])
    push!(matrix.values[row], 0.)
    matrix.row_ranges[row] = \
      matrix.row_ranges[row].start:matrix.row_ranges[row].stop+1
  end

  matrix.values[row][column - matrix.row_ranges[row].start + 1] = value
end
```

Modyfikowanie wartości w macierzy jest podobne do odwoływania się do niej. Tutaj także musimy uwzględnić przesunięcie na podstawie *first_nonzero_column*.

Jeżeli jednak próbujemy modyfikować wartość spoza zakresu, to musimy zwiększyć zakres o jeden element i dodać go do odpowiedniego wiersza.

Formalnie powinniśmy sprawdzać, czy dodawana kolumna nie odbiega znacząco od naszego pierwotnego zakresu, ale w trakcie implementowania algorytmów z listy zauważyłem, że będą one zawsze odwoływać się do indeksu oddalonego o 1 od obecnego zakresu. Dlatego na spokojnie można pominąć tę część kodu.

3 Algorytmy rozwiązujące układy równań liniowych o zadanej macierzy

Mieliśmy zaimplementować poniższe algorytmy:

1. Rozwiązywanie układu $\mathbf{Ax} = \mathbf{b}$ metodą eliminacji Gaussa bez wyboru elementu główneego,
2. Rozwiązywanie układu $\mathbf{Ax} = \mathbf{b}$ metodą eliminacji Gaussa z częściowym wyborem elementu głównego,
3. Wyznaczanie rozkładu LU na podstawie eliminacji Gaussa bez wyboru elementu głównego,
4. Wyznaczanie rozkładu LU na podstawie eliminacji Gaussa z częściowym wyborem elementu głównego,
5. Rozwiązywanie układu równań $\mathbf{Ax} = \mathbf{b}$ z wykorzystaniem rozkładu LU bez wyboru elementu głównego,
6. Rozwiązywanie układu równań $\mathbf{Ax} = \mathbf{b}$ z wykorzystaniem rozkładu LU z częściowym wyborem elementu głównego.

Wszystkie algorytmy zostały zoptymalizowane tak, aby czas wykonywania każdego z nich był co najwyżej $O(n)$.

3.1 Eliminacja Gaussa bez wyboru elementu głównego

Eliminacja Gaussa bez wybory elementu głównego polega na wyzerowaniu elementów pod główną przekątną macierzy $\mathbf{A}$ przy pomocy elementarnych operacji na wierszach macierzy. Wiadomo, że taka macierz, będzie równoznaczna bazowemu układowi równań.

W k-tym kroku algorytmu:

$$\begin{pmatrix} a_{11} & a_{12} & & \dots & & a_{1n} \\ 0 & a_{22} & & \dots & & a_{2n} \\ & & \ddots & \vdots & & \vdots \\ & & & a_{kk} & \dots & a_{kn} \\ & & & a_{kk+1} & \dots & a_{k+1n} \\ & & & \vdots & & \vdots \\ & & & a_{kn} & \dots & a_{nn} \end{pmatrix}$$

Aby wyzerować współczynnik $a_{km}, m = k+1, \dots, n$, przemnażamy k-ty wiersz przez $-\frac{a_{km}}{a_{kk}}$ i dodajemy do m-tego wiersza.

Po wykonaniu tych operacji, mamy:

$$\begin{pmatrix} a_{11} & a_{12} & & \dots & & a_{1n} \\ 0 & a_{22} & & \dots & & a_{2n} \\ & & \ddots & \vdots & & \vdots \\ & & & a_{kk} & \dots & a_{kn} \\ & & & 0 & \dots & a'_{k+1n} \\ & & & \vdots & & \vdots \\ & & & 0 & \dots & a'_{nn} \end{pmatrix}$$

Musimy także zaktualizować wektor b:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \\ b_{k+1} \\ \vdots \\ b_n \end{pmatrix}$$

Odejmując $-\frac{a_{km}}{a_{kk}} \cdot b_k$ od $b_m, m = k+1, \dots, n$ otrzymujemy:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \\ b'_{k+1} \\ \vdots \\ b'_n \end{pmatrix}$$

W przypadku gdy któraś z wartości na diagonali macierzy jest zerem lub bliska zeru, algorytm zawodzi (przez dzielenie $\frac{a_{km}}{a_{kk}}$). W takim wypadku należy zastosować częściowy wybór elementu głównego.

Aby znaleźć wektor rozwiązania $\mathbf{x}$, należy rozwiązać układ równań $\mathbf{Ux} = \mathbf{b}$, gdzie $\mathbf{U}$ jest macierzą górną trójkątną powstałą w wyniku eliminowania współczynników pod diagonalą macierzy $\mathbf{A}$.

Iterujemy od n do 1 i korzystamy ze wzorów:

$$x_n = \frac{b_n}{a_{nn}}$$
$$x_i = \frac{b_i - \sum_{j=i+1}^{n} x_j}{a_{ii}}$$
$$i = n-1, \dots, 1$$

3.1.1 Złożoność obliczeniowa

Nietrudno zauważyć, że złożoność powyższego algorytmu wynosi $O(n^3)$ (w k-tym kroku aktualizujemy $n-k$ wierszy i $n-k$ wartości, powtarzamy dla $n-1$ wierszy).

Wykorzystując fakt, że macierz $\mathbf{A}$ ma swoją specyficzną postać, można zredukować złożoność do $O(n)$.

Zauważmy, że skoro **A** jest macierzą blokową, to w k-tym kroku będziemy zerować co najwyżej l wierszy. Co więcej, odejmując od siebie wiersze odejmujemy zawsze stałą liczbę niezerowych współczynników (bo po lewej od diagonali mamy same zera, a ostatnia niezerowa kolumna znajduje się l kolumn po prawej od diagonali). W takim razie odejmując l niezerowych wartości od l wierszy $n-1$ razy otrzymujemy złożoność $O(l^2 \cdot (n-1)) = O(n)$.

3.1.2 Pseudokod

Algorytm 1: Zoptymalizowana eliminacja Gaussa bez częściowego wyboru

Dane: A - macierz współczynników , b - wektor prawych stron,
n - wielkość macierzy, l - wielkość bloku
Wynik: x, takie, że $Ax = b$

```
for k = 1 to n − 1 do
    for m = k + 1 to last_nonzero_row(n, l, k) do
        low ← a_mk / a_kk;
        a_mk = 0;
        for j = k + 1 to last_nonzero_column(n, l, k) do
            a_mj ← a_mj − low · a_kj;
        end
        b_m ← b_m − low · b_k;
    end
end
x ← b;
for i = n − 1 to 1 do
    for j = i + 1 to last_nonzero_column(n, l, k) do
        x_i ← x_i − a_ij · x_j;
    end
    x_i ← x_i / a_ii;
end
return x
```

3.2 Eliminacja Gaussa z częściowym wyborem elementu głównego

Eliminacja Gaussa bez częściowego wyboru zawodzi w przypadku gdy któraś z wartości na diagonali macierzy jest zerem lub bliska zeru. Aby zapobiec temu, należy zastosować częściowy wybór elementu głównego.

Częściowy wybór elementu głównego polega na poszukiwaniu w danym kroku algorytmu elementu o największej wartości bezwzględnej w kolumnie k. Jeśli taki element istnieje, to zamieniamy wiersze k i m, dzięki czemu pozbywamy się niepożądanych współczynników na diagonali.

Fizyczne zamienianie wierszy w macierzy byłoby uciążliwe i kosztowne (musielibyśmy zamieniać miejscami wektory w pamięci, przemapować zakresy kolumn i wierszy etc.). Dlatego wykorzystamy wektor permutacji, który będzie przechowywał informację o tym, które wiersze zostały zamienione. Dzięki temu, możemy w czasie stałym odwoływać się do zamienionych wierszy macierzy.

Algorytm przebiega podobnie jak w wersji bez częściowego wyboru, z tą różnicą, że w każdym kroku szukamy elementu o największej wartości bezwzględnej w kolumnie k.

3.2.1 Złożoność obliczeniowa

Podobnie jak w przypadku eliminacji Gaussa bez częściowego wybroru złożoność $O(n^3)$ możemy zredukować do $O(n)$ dla podanego rodzaju macierzy.

Pod diagonalą mamy maksymalnie l niezerowych wartości, dlatego liczba modyfikowanych wierszych nie zmieni się. Jednak przy zamianie wierszy liczba niezerowych kolumn może być większa niż l.

Rozważmy poniższą macierz:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Chcielibyśmy zastosować eliminację dla kolumny zaznaczonej na czerwono. Może się zdarzyć, że największy współczynnik w kolumnie 1 będzie znajdował się w wierszu 4. W takim wypadku musimy zamienić wiersze 1 i 4:

$$\begin{bmatrix} b_{41} & b_{42} & b_{43} & a_{44} & a_{45} & a_{46} & c_{47} & 0 & 0 \\ a_{11} & a_{12} & a_{13} & c_{14} & 0 & 0 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{33} & 0 & c_{25} & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & c_{36} & 0 & 0 & 0 \\ 0 & 0 & b_{53} & a_{54} & a_{55} & a_{56} & 0 & c_{58} & 0 \\ 0 & 0 & b_{63} & a_{64} & a_{65} & a_{66} & 0 & 0 & c_{69} \\ 0 & 0 & 0 & b_{74} & b_{75} & b_{76} & a_{77} & a_{78} & a_{79} \\ 0 & 0 & 0 & 0 & 0 & b_{86} & a_{87} & a_{88} & a_{89} \\ 0 & 0 & 0 & 0 & 0 & b_{96} & a_{97} & a_{98} & a_{99} \end{bmatrix}$$

Ale teraz mamy o l niezerowych kolumn więcej do odjęcia od wierszy poniżej niż mieliśmy przed zamianą. Musimy więc rozszerzyć zakres kolumn, które będziemy modyfikować.

Łatwo zauważyć, że liczba niezerowych kolumn rośnie o 1 wraz z każdym wierszem w dół (przed zamianą). Po zamianie będziemy mieć maksymalnie $2l$ niezerowych kolumn, które będziemy odejmować od l wierszy poniżej.

Możemy więc używając funkcji *last_nonzero_column* wyliczyć końcowy zakres modyfikowanych kolumn o l wierszy w dół, ponieważ w naszym przypadku będzie to po prostu $k + 2l$.

Po zastosowaniu wszystkich powyższych zmian względem algorytmu bez częściowego wyboru złożoność obliczeniowa dalej będzie wynosić $O(n)$ dla podanego rodzaju macierzy (w każdym kroku modyfikujemy l wierszy, maksymalnie $2l$ niezerowych kolumn, mamy $n - 1$ kroków: $O(2l^2 \cdot (n - 1)) = O(n)$).

3.2.2 Pseudokod

Algorytm 2: Zoptymalizowana eliminacja Gaussa z częściowym wyborem

Dane: A - macierz współczynników , b - wektor prawych stron,
n - wielkość macierzy, l - wielkość bloku
Wynik: x, takie, że $Ax = b$

```
permutation ← [1..n] ;
for k = 1 to n − 1 do
    max_row ← k;
    for i = k + 1 to last_nonzero_row(n, l, k) do
        if |a_ik| > |a_max_rowk| then
            max_row ← i;
        end
    end
    if max_row ≠ k then
        temp ← permutation[k];
        permutation[k] ← permutation[max_row];
        permutation[max_row] ← temp;
    end
    for m = k + 1 to last_nonzero_row(n, l, k) do
        low ← a_permutation[m]k / a_permutation[k]k;
        a_permutation[m]k = 0;
        for j = k + 1 to last_nonzero_column(n, l, k + l) do
            a_permutation[m]j ← a_permutation[m]j − low · a_permutation[k]j;
        end
        b_permutation[m] ← b_permutation[m] − low · b_permutation[k];
    end
end
x ← [0, . . . , 0];
for i = n to 1 do
    x_i ← b_permutation[i];
    for j = i + 1 to last_nonzero_column(n, l, k + l) do
        x_i ← x_i − a_permutation[i]j · x_j;
    end
    x_i ← x_i / a_permutation[i]i;
end
return x
```

3.3 Rozkład LU bez częściowego wyboru

Rozkład LU polega na rozłożeniu algorytmu elimincji Gaussa na dwie części. W pierwszej z nich wyznaczamu macierz L i U takie, że $A = LU$. L jest macierzą trójkątną dolną, a U jest macierzą trójkątną górną.

W drugiej części algorytmu szukamy rozwiązania układu $Ax = b$ korzystając z faktu, że $A = LU$:

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

W ten sposób, oddzielamy fazę faktoryzacji macierzy od fazy rozwiązywania układu równań. W normalnym przypadku koszt faktoryzacji do postaci LU wynosi $O(n^3)$, natomiast koszt rozwiązania układu równań wynosi $O(n^2)$. W tym wypadku otrzymujemy dużo lepszą złożoność niż przy wyznaczaniu rowziązania eliminacją Gaussa ($O(n^3)$).

Taki rozkład jest bardzo przydatny w praktyce, ponieważ często potrzebujemy rozwiązać wiele układów równań z taką samą macierzą współczynników.

Wyznaczanie rozkładu LU w praktyce przebiega tak samo jak eliminacja Gaussa, z tą różnicą, że zamiast zerować współczynniki pod diagonalą, możemy wpisywać w miejsce współczynniku mnożnik użyty do wyzerowania go.

Macierz L ma poniższą postać:

$$L = \begin{pmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix}$$

gdzie l_{ij} to mnożnik użyty do wyzerowania współczynnika a_{ij}.

Macierz U ma poniższą postać:

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}$$

gdzie u_{ij} to współczynniki uzyskane w wyniku eliminacji Gaussa.

Aby przechować rozkład $A = LU$ nie musimy przechowywać macierzy L i U osobno. Wystarczy zauważyć, że współczynniki pod diagonalą w macierzy L można wpisać w zera w macierzy U. Trzeba jedynie pamiętać, że w macierzy L na diagonali są jedynki.

Wyznaczenie rozwiązań układu $LUx = b$ musimy przeprowadzić dwuetapowo. Najpierw wyznaczamy $Lz = b$. Korzystamy z faktu, że na diagonali L są jedynki, więc aby wyznnaczyć z musimy po prostu zmodyfikować wektor b w prosty sposób:

$$\begin{aligned} m &= last\_nonzero\_row(k) \\ b_i &= b_i - \sum_{j=k+1}^{m} l_{ij} \cdot b_k \end{aligned}$$

gdzie $k = 1, \dots, n-1$ to obecnie rozważany wiersz.

Następnie wyznaczamy $Ux = z$. W tym przypadku wykonujemy analogiczne operacje jak w eliminacji Gaussa, ale przemnażamy odejmowane kolumny przez odpowiednie współczynniki u:

$$\begin{aligned} x &= b \\ m &= last\_nonzero\_column(i) \\ x_i &= \frac{b_i - \sum_{j=i+1}^{m} u_{ij} \cdot x_j}{u_{ii}} \end{aligned}$$

gdzie $i = n, n-1, \dots, 1$ to obecnie rozważany wiersz.

W ten sposób otrzymujemy rozwiązanie układu równań.

3.3.1 Złożoność obliczeniowa

Faktoryzacja macierzy A do postaci LU ma złożoność obliczeniową $O(n^3)$. Znajdywanie rozwiązania układu równań $LUx = b$ ma złożoność $O(n^2)$. Korzystając z rozważań jak w przypadku eliminacji Gaussa, możemy zredukować koszt do $O(n)$.

Wyznaczanie rozkładu LU różni się po prostu od eliminacji Gaussa tym, że zamiast zerować współczynniki pod diagonalą, wpisujemy w miejsce współczynnika mnożnik użyty do wyzerowania go. W takim razie koszt faktoryzacji jest taki sam jak w przypadku eliminacji Gaussa - $O(n)$.

Rozwiązywanie układu równań $LUx = b$ polega na dwuetapowym rozwiązaniu układów równań $Lz = b$ i $Ux = z$. W obu etapach także korzystamy z funkcji wyznaczających zakresy niezerowych elementów w wierszach/kolumnach.

W takim razie, będziemy znowu mieli pętle uzależnione od wierszy/kolumn i stałej liczby operacji wewnątrz pętli. Osiągamy więc złożoność liniową.

3.3.2 Pseudokod

Algorytm 3: Zoptymalizowane wyznaczanie rozkładu LU bez częściowego wyboru

Dane: A - macierz współczynników , n - wielkość macierzy, l - wielkość bloku
Wynik: Rozkład LU zapisany w macierzy A

```
for k = 1 to n − 1 do
    for m = k + 1 to last_nonzero_row(n, l, k) do
        low ← a_mk / a_kk;
        a_mk = low;
        for j = k + 1 to last_nonzero_column(n, l, k) do
            a_mj ← a_mj − low · a_kj;
        end
    end
end
return A
```

Algorytm 4: Wyznaczanie rozwiązań z rozkładu LU bez częściowego wyboru

Dane: A - rozkład LU, b - wektor prawych stron,
n - wielkość macierzy, l - wielkość bloku
Wynik: x, takie, że $Ax = b$

```
for k = 1 to n − 1 do
    for m = k + 1 to last_nonzero_row(n, l, k) do
        b_m ← b_m − a_mk · b_k;
    end
end
x ← b;
for i = n to 1 do
    for j = i + 1 to last_nonzero_column(n, l, k) do
        x_i ← x_i − a_ij · x_j;
    end
    x_i ← x_i / a_ii;
end
return x
```

3.4 Zoptymalizowane wyznaczanie rozkładu LU z częściowym wyborem elementu głównego

Wyznaczanie rozkładu LU z częściowym wyborem elementu głównego przebiega tak samo jak eliminacja Gaussa z częściowym wyborem elementu głównego. Po prostu rozdzielamy fazy faktoryzacji

macierzy i wyznaczaniu rozwiązania układu równań na osobne algorytmy.

W tym przypadku także będziemy używać wektora permutacji, do optymalnego zamieniania wierszy macierzy. Musimy jednak zwracać go wraz z rozkładem LU, aby poprawnie wyznaczyć rozwiązanie układu $LUx = b$.

3.4.1 Złożoność obliczeniowa

Tak jak w przypadku poprzednich algorytmów, złożoność obliczeniowa wynosi $O(n^3)$ dla faktoryzacji i $O(n^2)$ dla wyznaczania rozwiązania.

Tutaj także, korzystając z rozważań poczynionych podczas omawiania poprzednich algorytmów, możemy osiągnąć złożoność liniową. Musimy także pamiętać, tak jak w przypadku eliminacji Gaussa z częściowym wyborem o modyfikacji zakresu pętli, która odpowiada za odejmowanie wierszy od siebie (mamy maksymalnie $2l$ niezerowych elementów do odjęcia).

3.4.2 Pseudokod

Algorytm 5: Zoptymalizowane wyznaczanie rozkładu LU z częściowym wyborem

Dane: A - macierz współczynników , n - wielkość macierzy, l - wielkość bloku
Wynik: Rozkład LU zapisany w macierzy A, P - wektor permutacji

$permutation \leftarrow [1..n]$;
for $k = 1$ **to** $n - 1$ **do**
    $max\_row \leftarrow k$;
    for $i = k + 1$ **to** $last\_nonzero\_row(n, l, k)$ **do**
        if $|a_{ik}| > |a_{max\_rowk}|$ **then**
            $max\_row \leftarrow i$;
        end
    end
    if $max\_row \neq k$ **then**
        $temp \leftarrow permutation[k]$;
        $permutation[k] \leftarrow permutation[max\_row]$;
        $permutation[max\_row] \leftarrow temp$;
    end
    for $m = k + 1$ **to** $last\_nonzero\_row(n, l, k)$ **do**
        $low \leftarrow \frac{a_{permutation[m]k}}{a_{permutation[k]k}}$;
        $a_{permutation[m]k} = low$;
        for $j = k + 1$ **to** $last\_nonzero\_column(n, l, k + l)$ **do**
            $a_{permutation[m]j} \leftarrow a_{permutation[m]j} - low \cdot a_{permutation[k]j}$;
        end
    end
end
return A, P

Algorytm 6: Wyznaczanie r ozwiązań z rozkładu LU z częściowym wyborem

Dane: A - rozkład LU, b - wektor prawych stron,
n - wielkość macierzy, l - wielkość bloku
$permutation$ - wektor permutacji
Wynik: x, takie, że $Ax = b$
for $k = 2$ **to** n **do**
  for $m = first\_nonzero\_column(permutation[k])$ **to** $k - 1$ **do**
    $b_{permutation[m]} \leftarrow b_{permutation[m]} - a_{permutation[m]k} \cdot b_{permutation[k]}$;
  end
end
$x \leftarrow [0, \ldots, 0]$;
for $i = n - 1$ **to** 1 **do**
  $x_i \leftarrow b_{permutation[i]}$;
  for $j = i + 1$ **to** $last\_nonzero\_column(n, l, k + l)$ **do**
    $x_i \leftarrow x_i - a_{permutation[i]j} \cdot x_j$;
  end
  $x_i \leftarrow \frac{x_i}{a_{permutation[i]i}}$;
end
return x

4 Sprawdzenie poprawności implementacji

4.1 Sposób testowania

Do sprawdzenia poprawności swojej implementacji wykorzystałem dane podane na stronie wykładowcy. Program testujący znajduje się w pliku *test.jl*, aby go uruchomić przygotowałem prosty skrypt *run_tests.sh*, który odpowiednio ustawia środowisko i uruchamia program.

Każda metoda testowana jest na macierzach podanych w plikach *A.txt* oraz wektorach prawych stron *b.txt*. Po wyznaczeniu wektora rozwiązań x układu równań liniowych $Ax = b$ korzystam z przeciążonej przeze mnie funkcji mnożenia macierzy przez wektor (*Base.:**) do wyliczenia Ax i przyrównuje wynik do wektora b.

W taki sposób dla danych o wielkościach kolejno $n = 16, 10000, 50000, 100000, 300000, 500000$ wykonujemy po 4 testy dla każdej z metod. Łącznie mamy więc 24 testy. Kiedy którykolwiek z nich zawodzi, program zwraca błąd.

Dodatkowo wykorzystałem funkcję do generowania macierzy z modułu *matrixgen.jl* podanego na stronie wykładowcy. Generuję macierz o zadanym n, l i stopniu uwarunkowania bloków ck oraz losowy wektor prawych stron b.

Sprawdzenie poprawności tak wygenerowanych macierzy i wektora b polega także na sprawdzeniu czy Ax jest bliskie b. Łącznie wykonuje zawsze 144 testów na losowych danych.

Uwaga: program testujący wymaga pobrania biblioteki *Test*, dlatego zalecam korzystać z załączonego skryptu, który czyta załączony *Project.toml* i pobiera wszystkie potrzebne biblioteki.

5 Główny program rozwiązujący układ równań liniowych

W pliku *main.jl* znajduje się program rozwiązujący układ równań liniowych dla podanych plików z danymi. Program sczytuje z linii komend, którą metodę chcemy wykorzystać do rozwiązania układu równań liniowych, ścieżkę do pliku z macierzą, ścieżkę pliku wynikowego oraz opcjonalnie ścieżkę do pliku z wektorem prawych stron.

Do uruchamiania programu przygotowałem skrypt *run_main.sh*, który odpowiednio ustawia środowisko *Julii* i uruchamia program. Program potrzebuje biblioteki *LinearAlgebra* oraz *BenchmarkTools*, aby

poprawnie zadziałał. Dlatego zalecam uruchamianie programu za pomocą podanego skryptu.

Przykładowe uruchomienie programu:

```
$ ./run_main.sh gauss ./dane/A.txt out.txt ./dane/b.txt
$ ./run_main.sh gauss_partial_pivoting ./dane/A.txt out.txt ./dane/b.txt
$ ./run_main.sh lu ./dane/A.txt out.txt ./dane/b.txt
$ ./run_main.sh lu_partial_pivoting ./dane/A.txt out.txt ./dane/b.txt
```

Program drukuje na standardowe wyjście po kolei jakie kroki wykonuje, zapisuje wynik do podanego pliku, a na końcu wykonuje benchmark wybranej metody. Dodatkowo drukowane jest błąd względny dla zadanej metody, statystyki odwołań do/modyfikacji elementów macierzy oraz średni czas dla zadanej liczby prób w benchmarku.

W przypadku gdy nie podamy ścieżki do pliku z wektorem b, program sam wygeneruje ten wektor zakładając, że $Ax = b, x = (1, \ldots, 1)^T$.

6 Wyniki

6.1 Tabelki dla danych podanych na stronie wykładowcy

Na stronie wykładowcy podane są pliki testowe zawierające macierze o wielkości $n = 16, 10000, 50000, 100000, 300000, 500000$. Wyniki testów dla tych danych znajdują się w poniższej tabeli:

n	Eliminacja Gaussa	Eliminacja Gaussa z częściowym wyborem	Rozkład LU	Rozkład LU z częściowym wyborem
16	$1.466850003875454e-15$	$3.433175098891678e-16$	$1.466850003875454e-15$	$3.433175098891678e-16$
10000	$2.6979457399889918e-14$	$3.283259680653731e-15$	$2.6979457399889918e-14$	$3.283259680653731e-15$
50000	$5.983696068447594e-14$	$1.6152360794648073e-15$	$5.98369606844759 4e-14$	$1.6152360794648073e-15$
100000	$7.422229851374492e-13$	$1.8959299688155e-14$	$7.422229851374492e-13$	$1.8959299688155e-14$
300000	$6.521306370065581e-14$	$1.0316656402525826e-13$	$6.521306370065581e-14$	$1.0316656402525826e-13$
500000	$8.329169633245345e-14$	$1.605049904978791e-14$	$8.329169633245345e-14$	$1.605049904978791e-14$

Tabela 1: Błąd względny dla metod rozwiązywania układu równań liniowych dla danych podanych na stronie wykładowcy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)

n	Eliminacja Gaussa	Eliminacja Gaussa z częściowym wyborem	Rozkład LU	Rozkład LU z częściowym wyborem
16	693.0	876	738	921
10000	552309	739692	584802	772185
50000	$2.762309e6$	$3.699692e6$	$2.924802e6$	$3.862185e6$
100000	$7.499692e6$	$1.0219479e7$	$7.879683e6$	$1.059947e7$
300000	$1.6574809e7$	$2.2199692e7$	$1.7549802e7$	$2.3174685e7$
500000	$2.7624809e7$	$3.6999692e7$	$2.9249802e7$	$3.8624685e7$

Tabela 2: Liczba odwołań do elementów macierzy dla metod rozwiązywania układu równań liniowych dla danych podanych na stronie wykładowcy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)

n	Eliminacja Gaussa	Eliminacja Gaussa z częściowym wyborem	Rozkład LU	Rozkład LU z częściowym wyborem
16	289	339	289	339
10000	234913	292371	234913	292371
50000	$1.174913e6$	$1.462371e6$	$1.174913e6$	$1.462371e6$
100000	$3.259858e6$	$4.179776e6$	$3.259858e6$	$4.179776e6$
300000	$7.049913e6$	$8.774871e6$	$7.049913e6$	$8.774871e6$
500000	$1.1749913e7$	$1.4624871e7$	$1.1749913e7$	$1.4624871e7$

Tabela 3: Liczba zmian elementów macierzy dla metod rozwiązywania układu równań liniowych dla danych podanych na stronie wykładowcy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)

6.2 Badanie złożoności obliczeniowej dla zaimplementowanych metod

W pliku *complexity.jl* znajduje się program, który bada jaką złożoność czasową osiągają zaimplementowane przeze mnie metody rozwiązywania układu równań liniowych.

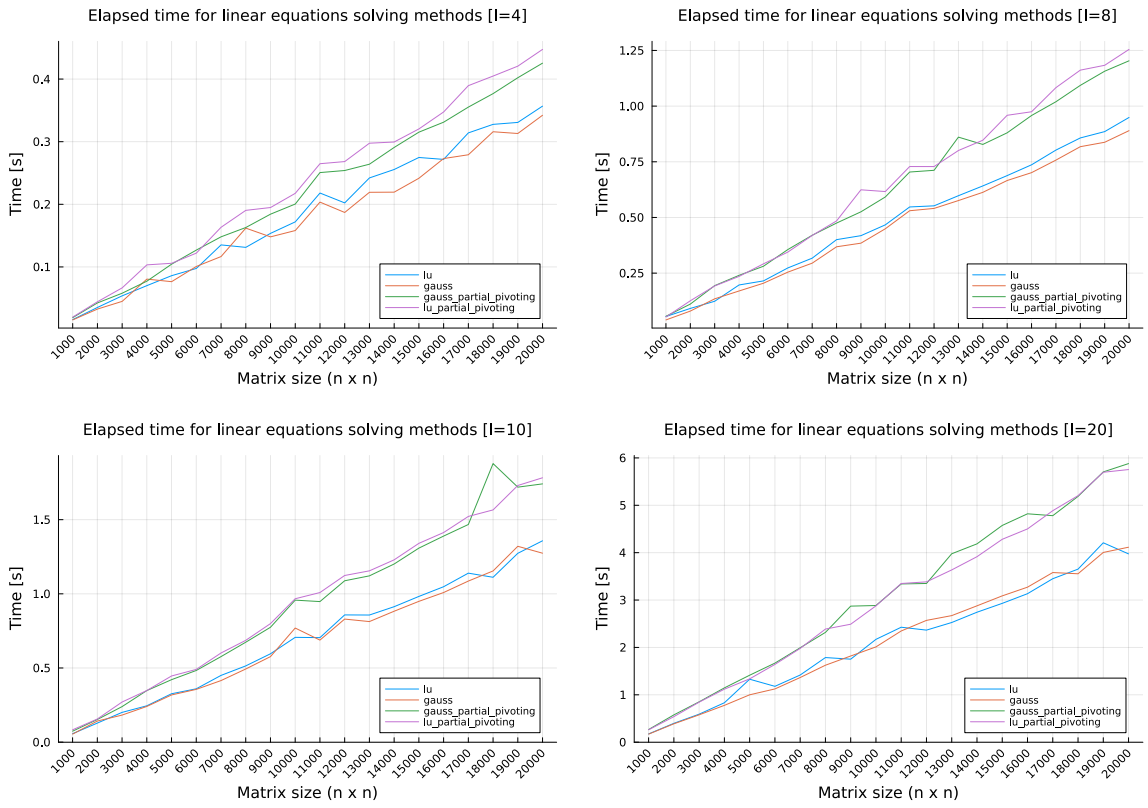
Program zawiera kilka trybów działania, które są opisane po uruchomieniu

```
$ julia complexity.jl --help
```

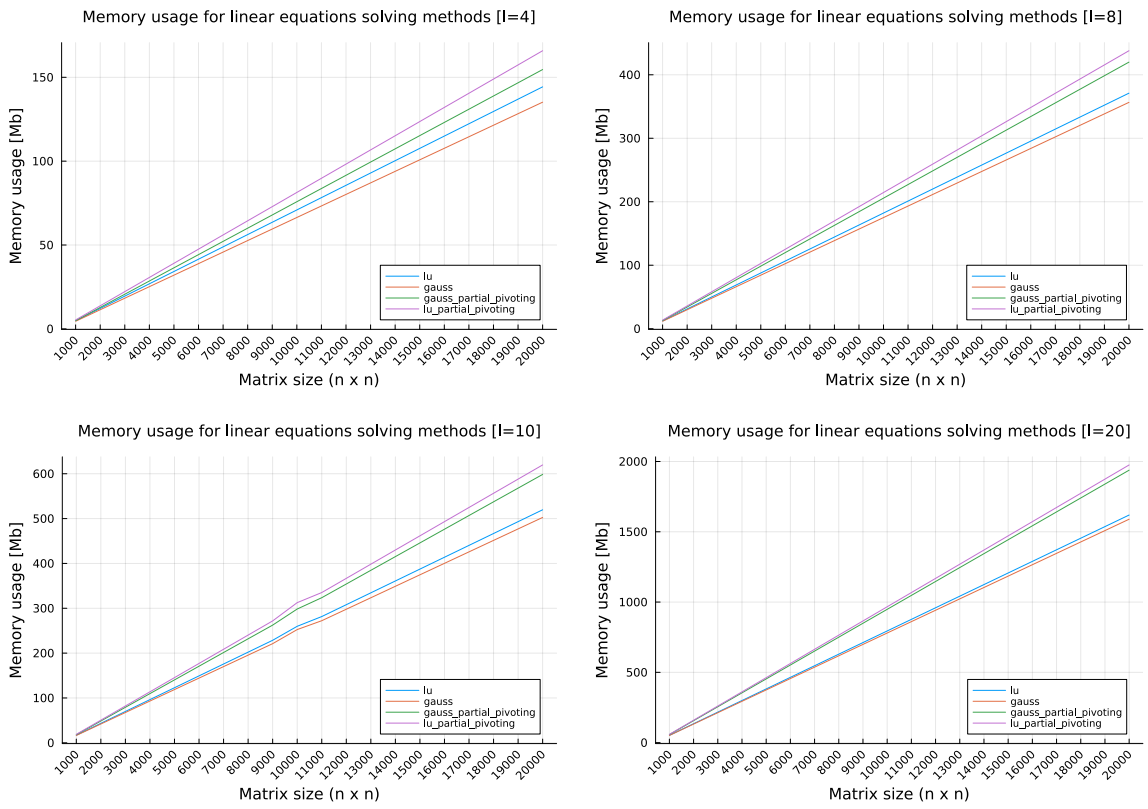
Tak samo jak w przypadku poprzednich programów, przygotowałem skrypt *run_complexity.sh*, który pobiera potrzebne biblioteki i uruchamia program *complexity.jl* z podanymi parametrami.

6.2.1 Wykresy

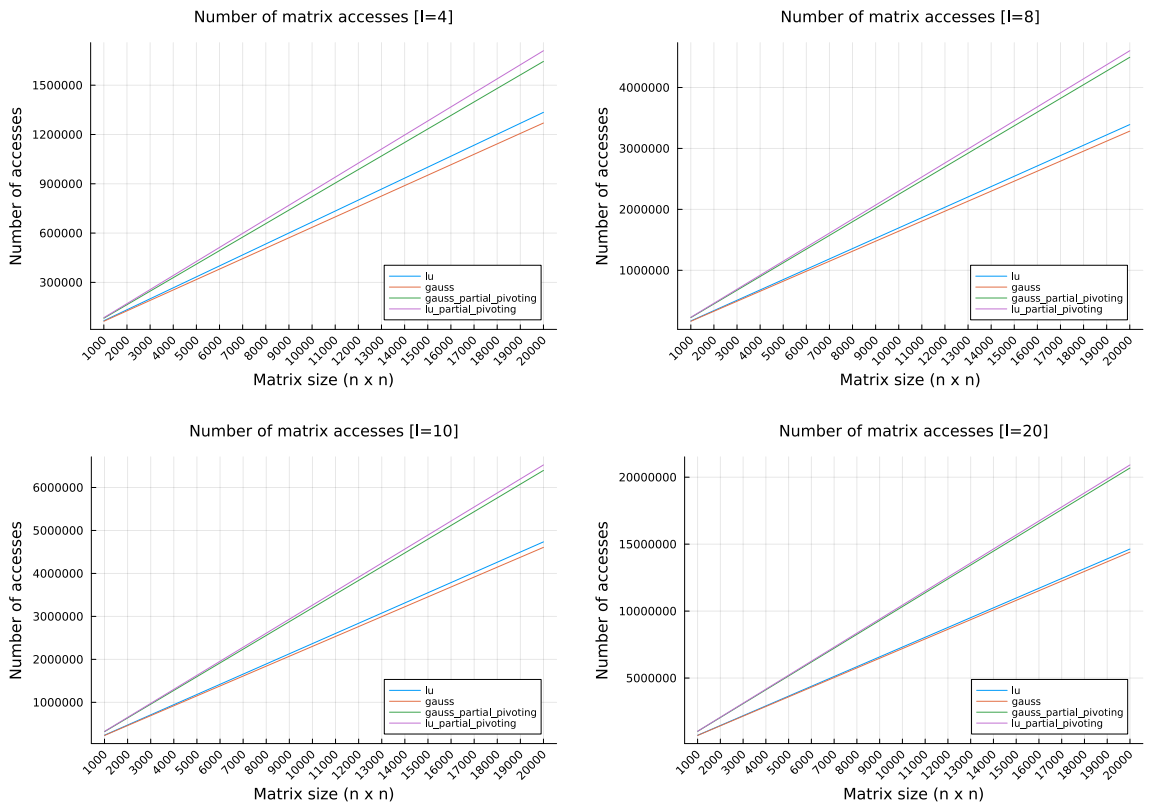
Wygenerowałem macierze dla $n = 1000, 2000, \ldots, 20000$, dla $l = 4, 8, 10, 20$ i dla $ck = 100.0$ (potrzebne dla funkcji *blockmat.jl* z modułu *matrixgen.jl*). Dla każdej zaimplementowanej metody wczytywałem macierz z wygenerowanego pliku, liczyłem wektor b z $b = Ax, x = (1, \ldots, 1)^T$ i zapisywałem statystyki do odpowiedniego pliku *.json*. Następnie dla każdego l generowałem wykresy porównujące zaimplementowane metody, które można zobaczyć poniżej:



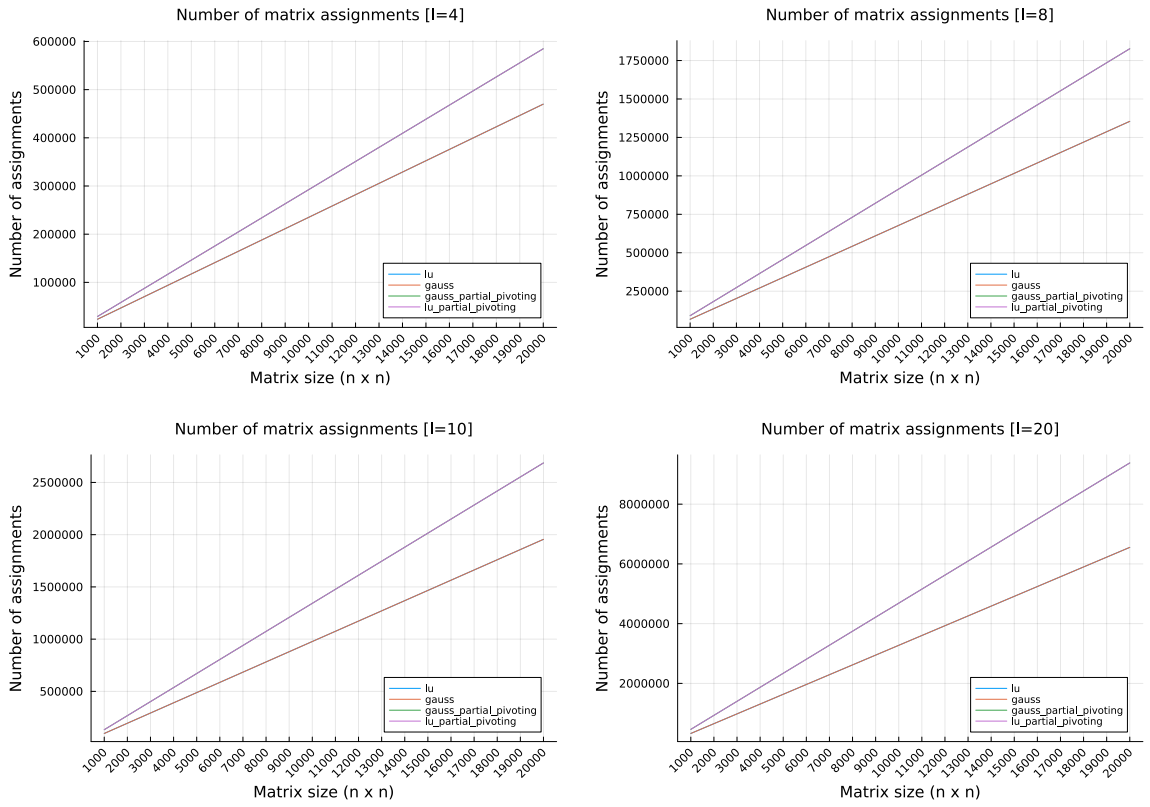
Rysunek 1: Czas wykonywania dla metod rozwiązywania układu równań liniowych dla losowo generowanych macierzy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)



Rysunek 2: Wykorzystanie pamięci dla metod rozwiązywania układu równań liniowych dla losowo generowanych macierzy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)



Rysunek 3: Liczba odwołań do wartości macierzy dla metod rozwiązywania układu równań liniowych dla losowo generowanych macierzy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)



Rysunek 4: Liczba modyfikacji wartości macierzy dla metod rozwiązywania układu równań liniowych dla losowo generowanych macierzy (wektor b obliczany z $b = Ax$, gdzie $x = (1, \ldots, 1)^T$)

7 Wnioski

Tak jak możemy spodziewać się po analizie złożoności obliczeniowej, poczynionej w opisie implementowanych metod, uzyskujemy liniowy czas wykonywania dla wszystkich metod. Poczynione przeze mnie optymalizacje są na tyle dobre, że nawet dla macierzy o wielkości ok. 20000 potrzebuję ok. 0.5 sekundy na rozwiązanie zadanego układu równań.

W każdym przypadku wykorzystujemy liniową liczbę bajtów pamięci, co także jest zgodne z oczekiwaniami.

Rozkład LU z częściowym wyborem elementu głównego oraz *eliminacja Gaussa z częściowym wyborem elementu głównego* wykonują się nieco dłużej od wersji bez częściowego wyboru, ale różnica jest niewielka.

Liczba odwołań do wartości macierzy oraz liczba modyfikacji wartości macierzy także zachowują się liniowo. Częściowy wybór elementu głównego zwiększa liczbę obu tych statystyk.

Zadanie pokazuje nam jak ważne jest odpowiednie zrozumienie struktury problemu. Stworzenie niestandardowej struktury oraz optymalizacja algorytmów pozwoliła na uzyskanie zaskakująco dobrych wyników dla wszystkich metod. Gdybyśmy nie poczynili opisanych usprawnień musielibyśmy czekać kilka minut na wyniki dla dużych macierzy.

Rozkład LU jest o tyle ciekawy, że pozwala nam na jeszcze większe zaoszczędzenie na czasie rozwiązywania danego układu, w przypadku gdy musimy rozwiązać ten układ dla różnych wektorów b. W *eliminacji Gaussa* musimy za każdym razem modyfikować macierz, aż osiągniemy postać górnotrójkątną, co jest bez sensu, kiedy musimy szukać wielu rozwiązań dla tego samego układu.

Dlatego warto jest wtedy skorzystać z rozkładu LU.

Dla każdego algorytmu osiągnęliśmy oczekiwane złożoności czasowe i pamięciowe, co potwierdza, że zaproponowana struktura przechowywania macierzy jest dobra oraz że metody rozwiąznywania układu równań liniowych są poprawnie zaimplementowane i zopytmalizowane.