Lista 5
Obliczenia naukowe

Patryk Majewski
250134

Załączone pliki:

- `blockmatrix.jl` – zawiera definicję typu do przechowywania macierzy wraz z metodami przydatnymi w implementowanych algorytmach
- `blocksys.jl` – zawiera moduł `blocksys` z implementacjami zadanych na liście algorytmów
- `program.jl` – zawiera program testujący zgodny ze specyfikacją z listy
- `utils.jl` – zawiera metody związane z czytaniem i tworzeniem plików
- `tests.jl` – zawiera testy jednostkowe algorytmów
- `complexity.jl` – zawiera testy algorytmów pod kątem złożoności

1 Problem

Potrzebujemy rozwiązać układ równań zadany jako $Ax = b$, gdzie $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, $n \geqslant 4$. Macierz A jest szczególnej postaci:

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

gdzie $v = n/l$, $A_k \in \mathbb{R}^{l \times l}$ są macierzami gęstymi, $0 \in \mathbb{R}^{l \times l}$ są macierzami zerowymi, $B_k \in \mathbb{R}^{l \times l}$ mają postać

$$B_k = \begin{bmatrix} 0 & \dots & 0 & b_1^{(k)} \\ 0 & \dots & 0 & b_2^{(k)} \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & b_l^{(k)} \end{bmatrix}$$

natomiast $C_k \in \mathbb{R}^{l \times l}$ mają postać

$$C_k = \begin{bmatrix} c_1^{(k)} & 0 & 0 & \dots & 0 \\ 0 & c_2^{(k)} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & c_{l-1}^{(k)} & 0 \\ 0 & \dots & 0 & 0 & c_l^{(k)} \end{bmatrix}$$

Symbole n i l używane dalej w opisach algorytmów oznaczać będą, jak powyżej, kolejno rozmiar macierzy i rozmiar pojedynczego bloku.

Celem listy jest wykorzystanie szczególnej struktury macierzy A do optymalizacji standardowych algorytmów rozwiązywania układów równań liniowych pod kątem złożoności obliczeniowej i pamięciowej.

2 Metoda eliminacji Gaussa

2.1 Wersja podstawowa

2.1.1 Idea

Najprostszy z implementowanych algorytmów podzielony jest na dwie fazy. Pierwsza z nich – faktyczna eliminacja – opiera się na fakcie, że układ równań powstały wskutek przeprowadzania elementarnych operacji na wierszach macierzy jest równoważny bazowemu układowi. Metoda polega na odejmowaniu kolejnych wierszy przemnożonych przez odpowiedni czynnik od tych znajdujących się pod nimi, tak aby doprowadzić do wyzerowania wszystkich wartości pod przekątną. Dokładniej, rozpatrując wiersze macierzy $[A|b]$ (z uwzględnieniem wektora prawych stron), w i-tym kroku wykonujemy

$$[a_{j,1} \;\; a_{j,2} \;\; \ldots \;\; a_{j,l} \;\; b_j] = [a_{j,1} \;\; a_{j,2} \;\; \ldots \;\; a_{j,l} \;\; b_j] - \frac{a_{j,i}}{a_{i,i}} \cdot [a_{i,1} \;\; a_{i,2} \;\; \ldots \;\; a_{i,l} \;\; b_i]$$

dla każdego $j > i$. Należy wspomnieć, że proces zawodzi w przypadku zerowej lub bardzo niewielkiej wartości leżącej na przekątnej, ponieważ występuje ona w mianowniku czynnika, przez który mnożony jest odejmowany wiersz, co może powodować błędy numeryczne. Odpowiedzią na ten problem jest wersja algorytmu z częściowym wyborem elementu głównego opisana w dalszej części sprawozdania.

Zwróćmy uwagę, że po i-tym kroku wszystkie wartości pod przekątną w i-tej kolumnie przyjmą wartość zero. Po $n-1$ krokach uzyskujemy zatem nietrudny do rozwiązania układ z macierzą górnotrójkątną.

Drugą fazą algorytmu jest rozwiązanie przekształconego układu. Można tego łatwo dokonać, iterując od n do 1 i korzystając ze wzorów

$$x_n = \frac{b_n}{a_{nn}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} x_j}{a_{ii}}$$

2.1.2 Złożoność i optymalizacje

Standardowa wersja procesu eliminacji ma złożoność $O(n^3)$, ponieważ k-ty wiersz (jeden z $n-1$, bo ostatniego nie mamy już od czego odejmować) musimy odjąć od $(n-k)$ wierszy poniżej niego, a każde takie odejmowanie polega na aktualizacji każdej z $(n+1)$ wartości w danym wierszu.

Zauważmy jednak, że w każdej kolumnie jest maksymalnie l niezerowych wartości pod diagonalą (jeżeli kolumna jest wielokrotnością l dokładnie tyle, bo ma pod sobą zawartość bloku B; w przeciwnym wypadku jest ich mniej, tylko do spodu bloku A). Od tych z zerowymi wartościami nie potrzebujemy odejmować, ponieważ cel wyzerowania został już osiągnięty. Co więcej, k-ty wiersz w k-tym kroku ma już same zera po lewej od diagonali, a po prawej (włącznie) ma dokładnie l leżących po sobie niezerowych wartości – ostatnią jest ta z bloku C, którego niezerowe współrzędne zawsze są postaci $(k, k\ +l)$. Możemy zatem w każdym kroku odejmować tylko od l wierszy, w których ma to sens, aktualizując tylko tych l wartości, które rzeczywiście ulegną zmianie. W ten sposób złożoność algorytmu spada do $O(l^2 \cdot (n-1)) = O(n)$.

Nietrudno dostrzec, że w podstawowej wersji faza rozwiązywania układu z macierzą trójkątną ma złożoność $O(n^2)$, ponieważ wyznaczamy wartość każdego z n x-ów, wyliczając przy tym sumę $n-(i+1)$ elementów.

Rozumując podobnie jak wyżej można zauważyć, że w każdym wierszu macierzy po dokonaniu eliminacji mamy maksymalnie l niezerowych wartości, co pozwala nam ograniczyć sumowanie w każdym przypadku do najwyżej $l-1$ składników. W ten sposób ograniczamy złożoność do $O(n)$.

Opisane dwie części procesu następują po sobie, zatem w niezoptymalizowanej wersji osiąga on złożoność $O(n^3)$. Wykorzystanie szczególnej struktury macierzy pozwoliło nam na zredukowanie jej do $O(n)$.

2.1.3 Pseudokod

Uproszczony przebieg procedury został umieszczony poniżej. Wykorzystane metody `last_row` oraz `last_column` zwracają kolejno pierwszy od dołu niezerowy wiersz w podanej kolumnie i pierwszą od prawej niezerową kolumnę w podanym wierszu macierzy A. Metoda `last_column` została już precyzyjnie określona w poprzednim podpunkcie – zwraca ona zawsze $(k+l)$. Jak zauważyliśmy, również `last_row` w najprostszej wersji może zwracać $(k+l)$, ponieważ jest to górne ograniczenie. Można natomiast poddać ją dalszym rozważaniom, żeby jeszcze bardziej ograniczyć zbędne iteracje. Rozważmy numer ostatniego wiersza dla pierwszych $2l$ kolumn macierzy:

$$\underbrace{l \quad l \quad \ldots \quad l \quad 2l}_{\text{kolumny 1 do } l} \qquad \underbrace{2l \quad \ldots \quad 2l \quad 3l}_{\text{kolumny } l+1 \text{ do } 2l}$$

Odejmując od każdej z tych wartości numery odpowiadających jej kolumn otrzymujemy

$$\underbrace{l-1 \quad l-2 \quad \ldots \quad l-(l-1) \quad 2l-l}_{\text{kolumny 1 do } l} \qquad \underbrace{2l-(l+1) \quad \ldots \quad 2l-(2l-1) \quad 3l-2l}_{\text{kolumny } l+1 \text{ do } 2l}$$

co po uproszczeniu sprowadza się do

$$\underbrace{l-1 \quad l-2 \quad \ldots \quad 1 \quad l}_{\text{kolumny 1 do } l} \qquad \underbrace{l-1 \quad \ldots \quad 1 \quad l}_{\text{kolumny } l+1 \text{ do } 2l}$$

Możemy zauważyć cykliczność, a ponieważ struktura macierzy jest jednolita, wzorzec ten powtarza się przez całą jej szerokość. Metodę `last_row` możemy zatem opisać wzorem

$$last\_row(k) = k \ + l \ - (k \bmod l)$$

jeśli wartość ta jest mniejsza od n, w przeciwnym wypadku funkcja zwraca n.

Algorithm 1: zoptymalizowana eliminacja Gaussa

Input: A – macierz współczynników, b – wektor prawych stron, n – rozmiar macierzy, l – rozmiar bloku
Output: x – wektor rozwiązań

```
for k from 1 to n − 1 do
    for i from k + 1 to A.last_row(k) do
        m = a_ik / a_kk
        a_ik = 0
        for j from k + 1 to A.last_column(k) do
            a_ij = a_ij − m · a_kj
        b_i = b_i − m · b_k
x = b
x_n = b_n / a_nn
for i from n − 1 down to 1 do
    for j from i + 1 to A.last_column(i) do
        x_i = x_i − a_ij · x_j
    x_i = x_i / a_ii
return x
```

2.2 Częściowy wybór elementu głównego

2.2.1 Idea

Uzupełnienie metody eliminacji Gaussa o częściowy wybór elementu głównego pomaga rozwiązać problem bardzo niewielkich wartości na przekątnej macierzy, które mogą powodować błędy w obliczeniach. Tytułowym elementem głównym jest wartość, której w k-tym kroku używamy do wyzerowania pozostałych w k-tej kolumnie. W przypadku podstawowej wersji algorytmu wartością tą zawsze była a_{kk}.

Częściowy wybór polega na wyznaczeniu takiego wiersza p, że

$$|a_{pk}| = \max_{k \leqslant i \leqslant n} |a_{ik}|$$

a następnie zamianę w macierzy $[A|b]$ wierszy p i k. Takie postępowanie pozwala nam na wybranie "lidera" każdej kolumny. Następnie procedura postępuje identycznie jak w przypadku podstawowym. Warto nadmienić, że w faktycznej implementacji wiersze nie są przestawiane, ale pamiętany jest wektor permutacji P, w którym k-ty element oznacza wiersz, w którym znajduje się k-ty element główny. Jeśli procedura częściowego wyboru w k-tym kroku wybiera p-ty wiersz, dokonujemy zamiany miejscami $P[k]$ i $P[p]$.

Efekt fazy eliminacji jest identyczny jak w podstawowej wersji – otrzymujemy układ z macierzą górnotrójkątną, który rozwiązujemy podobnie jak wcześniej.

2.2.2 Złożoność i optymalizacje

Nietrudno zauważyć, że procedura znalezienia elementu głównego, wykonywana raz w każdym z $(n-1)$ kroków eliminacji, jest $O(n)$, zatem złożoność niezoptymalizowanego algorytmu pozostaje $O(n^3)$. W fazie wyznaczania rozwiązania nie zachodzą większe zmiany, zatem również jego złożoność nie zmienia się.

Możemy zastosować analogiczne techniki optymalizacji jak poprzednio. Pod diagonalą mamy w każdym momencie maksymalnie l niezerowych wartości, zatem można zawęzić pole poszukiwań elementu głównego do stałej liczby iteracji. Podobnie, ponieważ wiersz z elementem głównym w k-tym kroku "staje się" k-tym wierszem, może być pod nim maksymalnie l niezerowych wartości, więc tylko od tych będziemy odejmować wybrany wiersz. Ograniczenie kolumn, od których odejmujemy musi jednak nastąpić łagodniej niż w wersji podstawowej. Z uwagi na to, że w k-tym kroku element główny może leżeć maksymalnie w $(k+l)$-tym wierszu, musimy wziąć pod uwagę, że wartości niezerowe (a więc mogące coś zmienić) w wybranym wierszu występują aż do $(k+2l)$-tej kolumny – nadal jednak jest to stała liczba iteracji. Rozważania te pozwalają nam ponownie ograniczyć złożoność fazy eliminacji do $O(n)$ W fazie wyznaczania rozwiązania również musimy zastosować "łagodniejsze" ograniczenie do iterowania od $(k+1)$ do $(k+2l)$, czym osiągamy złożoność $O(n)$.

Udało nam się zatem ograniczyć i tę wersję algorytmu rozwiązywania układu równań do złożoności $O(n)$.

2.2.3 Pseudokod

Algorithm 2: zoptymalizowana eliminacja Gaussa z częściowym wyborem

Input: A – macierz współczynników, b – wektor prawych stron, n – rozmiar macierzy, l – rozmiar bloku
Output: x – wektor rozwiązań

$P = [1, 2, ..., n]$
for k ***from*** *1* ***to*** $n-1$ **do**
  $j = i$, że $|a_{P[i],k}| = \max_{k \leqslant i \leqslant A.last\_row(k)} |a_{P[i],k}|$
  $P[k] \leftrightarrow P[j]$
  for i ***from*** $k+1$ ***to*** $A.last\_row(k)$ **do**
    $m = a_{P[i],k} / a_{P[k],k}$
    $a_{P[i],k} = 0$
    for j ***from*** $k+1$ ***to*** $A.last\_column(k+l)$ **do**
      $a_{P[i],j} = a_{P[i],j} - m \cdot a_{P[k],j}$
    $b_{P[i]} = b_{P[i]} - m \cdot b_{P[k]}$
$x = b$
$x_n = b_{P[n]} / a_{P[n]n}$
for i ***from*** $n-1$ ***down to*** 1 **do**
  for j ***from*** $i+1$ ***to*** $A.last\_column(i+l)$ **do**
    $x_{P[i]} = x_{P[i]} - a_{P[i],j} \cdot x_j$
  $x_{P[i]} = x_{P[i]} / a_{P[i],i}$
return x

3 Rozkład LU

Rozkładem LU nazywamy taki podział $A = LU$, że L jest macierzą dolnotrójkątną, a U – górnotrójkątną. Układ równań $Ax = LUx = b$ możemy wówczas rozwiązać jako dwa proste układy z macierzami trójkątnymi:

$$Lz = b$$

$$Ux = z$$

Ta dwustopniowość pozwala nam również na wielokrotne rozwiązywanie układów dla różnych wektorów prawych stron b, wyznaczając sam rozkład LU tylko raz.

3.1 Wersja podstawowa

3.1.1 Idea

Algorytm opiera się na spostrzeżeniu, że metoda eliminacji Gaussa w zasadzie konstruuje macierz U, a po niewielkiej modyfikacji również macierz L. Oznaczając przez $L^{(k)}$ macierz przekształcenia A w k-tym kroku eliminacji mamy

$$\begin{aligned} L^{(n-1)} \dots L^{(2)} L^{(1)} A &= U \\ A &= L^{(n-1)^{-1}} \dots L^{(2)^{-1}} L^{(1)^{-1}} U \end{aligned}$$

W k-tym kroku mamy zatem

$$L^{(k)} \cdot \begin{bmatrix} w_1^{(k)} \\ w_2^{(k)} \\ \vdots \\ w_n^{(k)} \end{bmatrix} = \begin{bmatrix} w_1^{(k)} \\ \vdots \\ w_k^{(k)} \\ w_{k+1}^{(k)} - m_{k+1,k} w_k^{(k)} \\ \vdots \\ w_n - m_{n,k} w_k^{(k)} \end{bmatrix}$$

gdzie $w_i^{(k)}$ jest i-tym wierszem macierzy A w k-tym kroku eliminacji, a $m_{i,k} = a_{i,k}/a_{k,k}$. Wówczas nietrudno zauważyć, że

$$L^{(k)} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & -m_{k+1,k} & \ddots & & \\ & & & \vdots & & \ddots & \\ & & & -m_{n,k} & & & 1 \end{bmatrix} \qquad L^{(k)^{-1}} = \begin{bmatrix} 1 & & & & & & \\ & 1 & & & & & \\ & & \ddots & & & & \\ & & & 1 & & & \\ & & & m_{k+1,k} & \ddots & & \\ & & & \vdots & & \ddots & \\ & & & m_{n,k} & & & 1 \end{bmatrix}$$

Złożenie wszystkich macierzy przekształcenia ma wówczas postać

$$L^{(n-1)^{-1}} \dots L^{(2)^{-1}} L^{(1)^{-1}} = L = \begin{bmatrix} 1 & & & & \\ m_{21} & 1 & & & \\ m_{31} & m_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ m_{n,1} & m_{n,2} & \dots & m_{n,n-1} & 1 \end{bmatrix}$$

i w istocie jest macierzą dolnotrójkątną.

Bazując na tym spostrzeżeniu, wprowadzimy drobne modyfikacje w stosunku do standardowego algorytmu eliminacji Gaussa:

- Zamiast "zerować" wartości pod diagonalą, będziemy zapisywać w ich miejscach mnożniki m, które służyły do ich wyzerowania. W ten sposób zapamiętamy cały rozkład LU w jednej macierzy, ponieważ L ma na diagonali same jedynki, których nie musimy fizycznie przechowywać.

- Nie będziemy rozpatrywać wierszy macierzy $[A|b]$, a jedynie A, ponieważ wektor prawych stron b potrzebny jest nam wyłącznie w fazie wyznaczania rozwiązań układu, i tam też jest modyfikowany.

Faza wyznaczania wektora x składa się z dwóch pętli rozwiązujących kolejno układy równań $Lz = b$ i $Ux = z$. Każda z nich jest analogiczna do tej opisanej przy okazji podstawowej eliminacji Gaussa.

3.1.2 Złożoność i optymalizacje

Z uwagi na bardzo niewielkie różnice między wyznaczaniem rozkładu LU a fazą eliminacji w podstawowym algorytmie (ta sama struktura i zakresy pętli), w niezoptymalizowanej formie mają one dokładnie tę samą złożoność $O(n^3)$. Z tego samego powodu możemy, wykorzystując identyczne jak wcześniej optymalizacje, zredukować ją do $O(n)$.

Pamiętając, że macierz L ma na przekątnej same jedynki, wiemy, że każdy z_i zawiera w sobie składnik b_i. Możemy zatem, celem zaoszczędzenia pamięci, nadpisywać wektor b, umieszczając w nim rozwiązania pierwszego układu. Co więcej, macierz przechowująca rozkład LU ma dokładnie tę samą strukturę, co macierz A (wynika to z tych samych spostrzeżeń, co optymalizacja pierwszej fazy), co powoduje, że rozwiązywanie obu układów możemy uprościć tak jak w przypadku zwykłej eliminacji Gaussa, zmniejszając ich złożoność z $O(n^2)$ do $O(n)$.

3.1.3 Pseudokod

Algorithm 3: zoptymalizowane wyznaczanie rozkładu LU

Input: A – macierz współczynników, n – rozmiar macierzy, l – rozmiar bloku
Output: LU – macierz zawierająca rozkład
for k ***from*** *1* ***to*** $n-1$ **do**
  for i ***from*** $k+1$ ***to*** $A.last\_row(k)$ **do**
    $m = a_{ik}/a_{kk}$
    $a_{ik} = m$
    for j ***from*** $k+1$ ***to*** $A.last\_column(k)$ **do**
      $a_{ij} = a_{ij} - m \cdot a_{kj}$
return A

Algorithm 4: zoptymalizowane wyznaczanie rozwiązań z rozkładem LU

Input: LU – macierz zawierająca rozkład, b – wektor prawych stron, n – rozmiar macierzy, l – rozmiar bloku
Output: x – wektor rozwiązań
`/* Lz=b idąc po kolumnach, potem od góry do dołu */`
for k ***from*** *1* ***to*** $n-1$ **do**
  for i ***from*** $k+1$ ***to*** $LU.last\_row(k)$ **do**
    $b_i = b_i - LU_{i,k} \cdot b_k$
`/* Ux=z idąc po wierszach, potem od lewej do prawej */`
$x = b$
for i ***from*** $n-1$ ***down to*** 1 **do**
  for j ***from*** $i+1$ ***to*** $LU.last\_column(i)$ **do**
    $x_i = x_i - LU_{i,j} \cdot x_j$
  $x_i = x_i / LU_{i,i}$
return x

3.2 Częściowy wybór elementu głównego

3.2.1 Idea

Podobnie jak w przypadku zwykłej eliminacji Gaussa, wariant z częściowym wyborem elementu głównego pozwala nam na uniknięcie błędów numerycznych w przypadku bardzo niewielkiej wartości na przekątnej macierzy A. Posiłkując się rozważaniami poczynionymi w sekcjach 2.2 i 3.1, dokonujemy drobnych zmian w algorytmie eliminacji z wyborem, ponownie nadpisując zera czynnikami m i pomijając operacje na wektorze b. Faza wyznaczania rozkładu musi dodatkowo zwracać wektor permutacji P, ponieważ jest on potrzebny przy obliczaniu rozwiązań.

3.2.2 Złożoność i optymalizacje

Tak jak w przypadku wariantu bez wyboru, w standardowym algorytmie eliminacji z wyborem nie zachodzą żadne zmiany związane z zakresami iteracji, co powoduje, że złożoność fazy wyznaczania rozkładu LU jest identyczna jak w sekcji 2.2.2. Niezoptymalizowana wersja ma zatem złożoność $O(n^3)$, optymalizacje redukują ją jednak do $O(n)$.

Idea fazy wyznaczania rozwiązań jest podobna jak w przypadku standardowego rozkładu LU, zatem jej niezoptymalizowana złożoność to $O(n^2)$. Jeśli jednak chodzi o optymalizacje, z uwagi na obecność wektora permutacji, musimy zastosować "łagodne" ograniczenie na liczbę iteracji rozważane wcześniej w sekcji 2.2.2. Ostatecznie, podobnie jak we wspomnianej sekcji, otrzymujemy złożoność $O(n)$.

3.2.3 Pseudokod

Wykorzystana w fazie rozwiązań metoda `first_column` zwraca w największym uproszczeniu $k-l$ – numer pierwszej kolumny dla wierszy, których numer k jest wielokrotnością l. W innych przypadkach można jeszcze bardziej ograniczyć liczbę iteracji, zauważając pewną cykliczność. W analizie podobnej do tej z sekcji 2.1.3 możemy otrzymać, że

$$first\_column(k) = k - 1 - ((k-1) \bmod l)$$

jeśli wartość ta jest większa od 0, w przeciwnym wypadku (dla pierwszych l wierszy) funkcja zwraca 1. W obu wariantach iterowanie od $first\_column(P[k])$ do $(k-1)$ daje nam stałą liczbę iteracji – najwyżej $2l$, bo $P[k]$ różni się od k o najwyżej l.

Algorithm 5: zoptymalizowane wyznaczanie rozkładu LU z częściowym wyborem

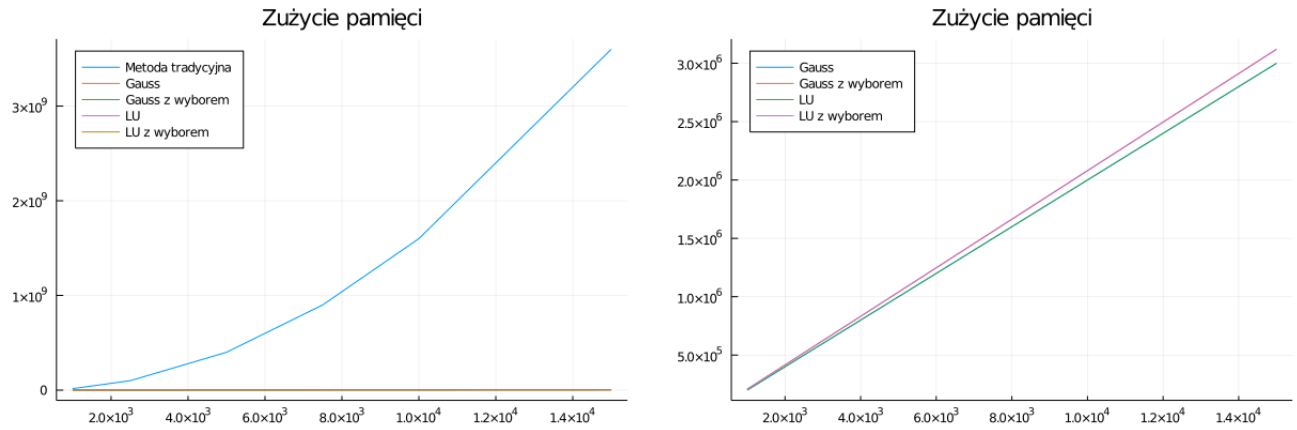
Input: A – macierz współczynników, n – rozmiar macierzy, l – rozmiar bloku
Output: LU – macierz zawierająca rozkład, P – wektor permutacji
$P = [1, 2, ..., n]$
for k ***from*** 1 ***to*** $n-1$ **do**
  $j = i$, że $|a_{P[i],k}| = \max_{k \leqslant i \leqslant A.last\_row(k)} |a_{P[i],k}|$
  $P[k] \leftrightarrow P[j]$
  for i ***from*** $k+1$ ***to*** $A.last\_row(k)$ **do**
    $m = a_{P[i],k} / a_{P[k],k}$
    $a_{P[i],k} = m$
    for j ***from*** $k+1$ ***to*** $A.last\_column(k+l)$ **do**
      $a_{P[i],j} = a_{P[i],j} - m \cdot a_{P[k],j}$
return A, P

Algorithm 6: zoptymalizowane wyznaczanie rozwiązań z rozkładem LU z częściowym wyborem

Input: LU – macierz zawierająca rozkład, P – wektor permutacji, b – wektor prawych stron,
n – rozmiar macierzy, l – rozmiar bloku
Output: x – wektor rozwiązań
`/* Lz=Pb idąc po wierszach, potem od lewej do prawej */`
for i ***from*** 2 ***to*** n **do**
  for j ***from*** $LU.first\_column(P[i])$ ***to*** $i-1$ **do**
    $b_{P[i]} = b_{P[i]} - LU_{P[i],k} \cdot b_{P[k]}$
`/* Ux=z idąc po wierszach, potem od lewej do prawej */`
for i ***from*** $n-1$ ***down to*** 1 **do**
  $x_i = b_{P[i]}$
  for j ***from*** $i+1$ ***to*** $LU.last\_column(i+l)$ **do**
    $x_i = x_i - LU_{P[i],j} \cdot x_j$
  $x_i = x_i / LU_{P[i],i}$
return x

4 Badania złożoności

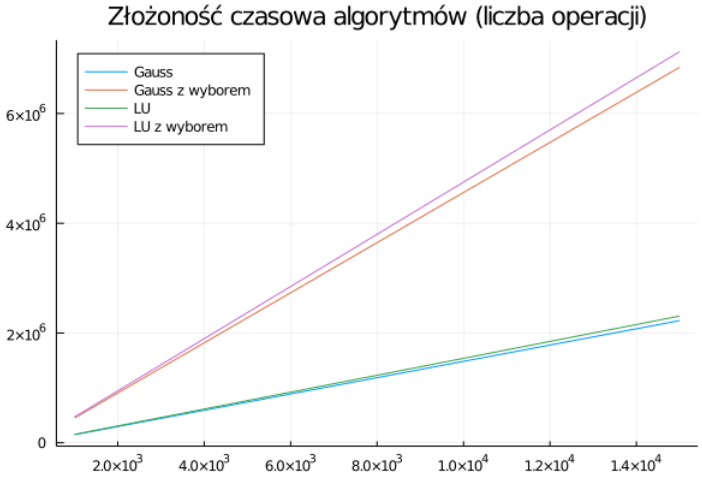
Korzystając z faktu, że macierze postaci opisanej na liście zawierają bardzo dużo elementów zerowych, złożoność pamięciowa zaimplementowanych algorytmów została znacząco ograniczona poprzez zastosowanie macierzy rzadkich z modułu `SparseArrays`. Ma to znaczenie szczególnie dla bardzo dużych n. Zaimplementowane algorytmy zostaną porównane z "metodą tradycyjną", to jest $A\backslash b$ dla A przechowywanego jako gęste. Wykresy porównujące złożoność pamięciową umieszczone zostały na rysunku 1.



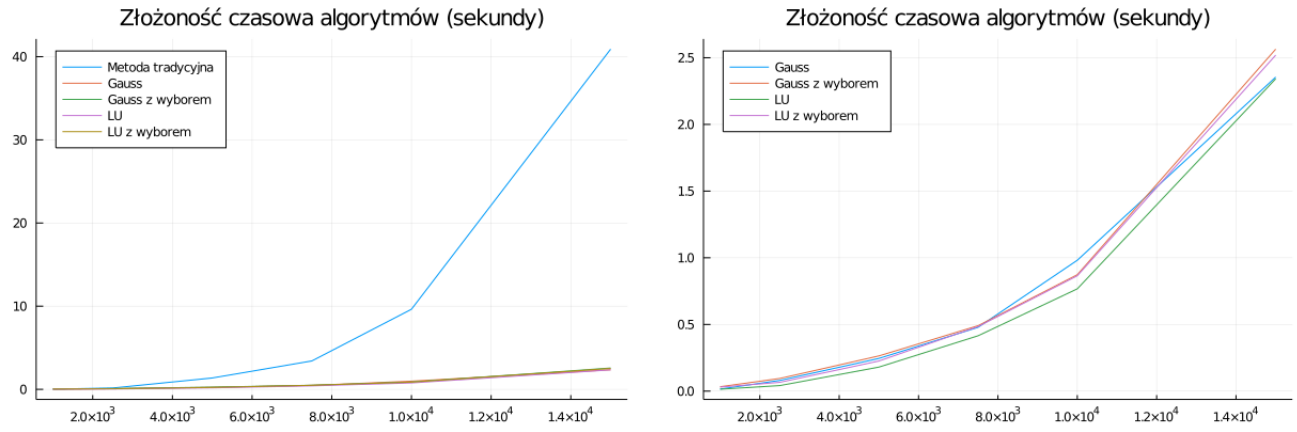
(a) (b)

Rysunek 1: Pomiary zużycia pamięci w bajtach w zależności od rozmiaru macierzy dla zaimplementowanych algorytmów, uzyskane za pomocą makra `@timed`. Jak widać, wykorzystanie macierzy rzadkich znacznie ogranicza zapotrzebowanie na pamięć. Różnice między wariantami z wyborem i bez wynikają z konieczności przechowywania wektora P.

Złożoność obliczeniowa zaimplementowanych algorytmów została zmierzona na dwa sposoby: z użyciem makra `@timed` oraz poprzez zliczanie odwołań do elementów macierzy A. W przeprowadzonych analizach złożoności zakładaliśmy bowiem, że dostęp do elementu macierzy odbywa się w czasie stałym. W rzeczywistości tak nie jest, co widać na wykresach z rysunku 3, jednak drugi sposób z rysunku 2 pozwala nam na zignorowanie tego kosztu, zasadniczo potwierdzając poprawność naszych analiz.



Rysunek 2: Pomiary liczby odwołań do elementów macierzy w zależności od jej rozmiaru dla zaimplementowanych algorytmów. Zgodnie z oczekiwaniami, wszystkie osiągają złożoność $O(n)$. Warianty z wyborem mają większe złożoności niż te bez, co spowodowane jest łagodniejszymi ograniczeniami na zakresy iteracji. Pozwalają one jednak na pracę z macierzami, w których na przekątnej występują niewielkie wartości. Warianty LU mają większe złożoności niż ich odpowiedniki, ponieważ w drugiej fazie rozwiązują dwa układy równań. Z drugiej jednak strony ich pierwsza faza może być przeprowadzona tylko raz i użyta wielokrotnie dla różnych wektorów b.



(a) (b)

Rysunek 3: Pomiary czasu pracy w sekundach w zależności od rozmiaru macierzy dla zaimplementowanych algorytmów, uzyskane za pomocą makra `@timed`. Ograniczenie zakresów iteracji do niezerowych kolumn i wierszy przynosi ogromne zyski czasowe w stosunku do naiwnego podejścia. Zaimplementowane algorytmy osiągają bardzo podobne do siebie wyniki.

5 Program testujący

W odpowiedzi na wymagania z listy, do przygotowanego modułu `blocksys` powstał prosty program testujący umieszczony w pliku `program.jl`. Użytkownik może wpisać następujące komendy:

- `read PLIK1 PLIK2` – wczytuje macierz A z pliku `PLIK1` oraz wektor b z pliku `PLIK2`
- `reada PLIK` – wczytuje macierz A z pliku `PLIK` i wyznacza wektor b przy $x = [1, 1, ..., 1]^T$
- `readb PLIK` – wczytuje wektor b z pliku `PLIK`
- `gauss` – rozwiązuje układ dla wczytanych A i b z użyciem metody Gaussa bez wyboru
- `gauss pivot` – rozwiązuje układ dla wczytanych A i b z użyciem metody Gaussa z częściowym wyborem
- `lu gen` – wyznacza rozkład LU dla wczytanej macierzy A
- `lu gen pivot` – wyznacza rozkład LU i wektor permutacji P dla wczytanej macierzy A z częściowym wyborem
- `lu solve` – rozwiązuje układ dla wyznaczonego rozkładu LU i wczytanego wektora b (jeśli w pamięci jest P, używa go)
- `write PLIK` – zapisuje do pliku `PLIK` wyznaczony wektor x
- `fin` – kończy pracę programu

Po wykonaniu zleconej komendy program wyświetla elementy (spośród A, P, LU, b, x) gotowe do użycia.

6 Wnioski

Odpowiednia analiza problemu może spowodować, że stosunkowo łatwo otrzymamy rozwiązanie, które w naiwnym podejściu byłoby kosztowne lub niemożliwe do uzyskania.

Drobne modyfikacje oryginalnych algorytmów umożliwiły nam znaczne ograniczenie ich złożoności obliczeniowej. Znając strukturę danych, udało nam się również ograniczyć zapotrzebowanie na pamięć, odpowiednio wybierając możliwie oszczędny sposób ich przechowywania.