Sprawozdanie laboratorium lista 5
Obliczenia naukowe

Zofia Tarchalska, indeks: 279699

1 Wstęp

Celem tej listy było zaimplementowanie trzech poniższych algorytmów o złożoności $O(n)$, gdzie n to rozmiar macierzy $n \times n$.

- funkcja rozwiązująca układ $Ax = b$ metodą eliminacji Gaussa
- funkcja wyznaczająca rozkład LU maciezry A metodą eliminacji Gaussa
- funkcja rozwiązująca układ $Ax = b$ jeśli już wcześniej został wyznaczony rozkład LU

Wszystkie te algorytmy mają działać dla macierzy A o specyficznej postaci, która została dokładnie opisana w poleceniu do zadania. Dodatkowo należało zaprogramować wersję podstawową oraz wersję z wybieraniem elementu głównego.

2 Metoda eliminacji Gaussa

Główną ideą metody eliminacji Gaussa jest doprowadzenie macierzy do postaci, w której pod przekątną znajdują się same zera. Uzyskujemy to poprzez mnożenie kolejnych wierszy macierzy przez odpowiednie czynniki i odejmowanie ich od wierszy następujących po nich. Poprawny schemat:
Eliminujemy zmienną x_k z wierszy od $k+1$ do n. Mnożymy $k-te$ równanie przez

$$l_{ik} = \frac{a_{ik}}{a_{kk}} \quad \text{dla } i \in \{k+1, ..., n\}$$

Kiedy jednak na przekątnej w miejscu a_{kk} będzie 0 metoda może powodować błąd numeryczny, ponieważ nastąpi dzielenie przez 0. W przeciwnym wypadku, po wykonaniu odpowiednich kroków kolejno dla wszystkich wierszy macierzy, otrzymujemy macierz górnotrójkątną.

2.1 Wariant pierwszy - bez wyboru elementu głównego

Ten wariant nie zabezpiecza nas przed opisanym powyżej błędem numerycznym.

2.1.1 Złożoność czasowa

W niezoptymalizowanej wersji algorytmu złożoność wynosiłaby $O(n^3)$, ponieważ mamy 3 pętle `for` przechodzące po elementach macierzy. Dla każdego wiersza w macierzy wyznaczamy czynnik przez który przemnażamy ten wiersz po czym odejmujemy go od kolejnych wierszy (musimy przejść po wszytskich elementach w danym wierszu, a macierz ma wymiary $n \times n$). Jednak wykorzystując blokową strukturę macierzy A da się sprowadzić ten algorytm do złożoności $O(n)$. Wiemy, że rozmiar pojedynczego bloku ma jakiś określony rozmiar. Oznaczmy go przez l. Zatem pod przekątną macierzy jest maksymalnie l elementów, które są niezerowe. Dodatkowo każdy wiersz, o indeksie większym niż 1 ma od lewej same zera, następnie l leżących po sobie niezerowych wartości i znów zera (chyba, że np. $k + l$ daje już indeks ostatniej kolumny, wtedy nie ma zar od końca). Możemy zatem wykonywać ten algorytm tylko dla wierszy i elementów, dla których ma to sens (oszczędzimy sobie przetwarzania ogromnej liczby samych zer). Możemy zatem odejmować tylko l wierszy i aktualizować w nich l wartości. Ponieważ l jest znacząco mniejsze niż n, złożoność spada do $O(n)$.

2.1.2 Złożoność pamięciowa

Zaimplementowana przeze mnie struktura `BlockMatrix` przechowuje naszą macierz blokową A. Korzysta przy tym ze `SparseArrays` dostępnych w języku Julia. `SparseArrays` nie przechowują niezerowych elementów, wobec tego, ze względu na budowę A, mamy tylko elementy zlokalizowane w otoczeniu przekątnej. Ponieważ jest to co najwyżej l elementów przypadających na każdy wiersz/kolumnę, których jest n, to ze względu, że l jest jakąś stałą, złożoność pamięciowa to $O(n)$.

2.1.3 Pseudokod

Zawarte w pseudokodzie metody `get_bottom_row` oraz `get_last_column` zwracają odpowiednio ostatni wiersz i ostatnią kolumnę, w której pojawia się niezerowy element. Dzięki temu dokonujemy optymalizowania opisanego we wcześniejszym paragrafie i ograniczamy nasze obliczenia jedynie do niezerowych elementów.

```
procedure GaussElimination(A, b)
    n ← size(A)
    for k ← 1 to n − 1 do
        for i ← k + 1 to get_bottom_row(A,k) do
            Jeżeli A[k, k] jest zerem - błąd
            m ← A[i, k]/A[k, k]
            A[i, k] ← 0
            for j ← k + 1 to get_last_column(A,k) do
                A[i, j] ← A[i, j] − m · A[k, j]
            end for
            b[i] ← b[i] − m · b[k]
        end for
    end for
    x ← 0 ∈ ℝ^n
    x[n] ← b[n]/A[n, n]
    for i ← n − 1 downto 1 do
        s ← 0
        for j ← i + 1 to get_last_column(A,i) do
            s ← s + A[i, j] · x[j]
        end for
        x[i] ← (b[i] − s)/A[i, i]
    end for
    return x
end procedure
```

2.2 Wariant drugi - z częściowym wyborem elementu głównego

Ten wariant ma zabezpieczać nas przed potencjalnym dzieleniem przez 0 gdy na przekątnej macierzy taka warotść się znajduje. Biorąc pod uwagę, że obliczenia wykonujemy na komputerze, liczby bardzo zbliżone do 0 również są tymi, które mogą nam zwrócić błąd.
Element główny to nic innego jak wybrana wartość, którą będziemy używać aby wyzerować pozostałe w kolumnie. Wybieramy ją poprzez znaleznienie największej wartości w kolumnie (patrząc na jej moduł). Następnie wyznaczamy czynnik, przez który będziemy mnożyć kolejne wiersze, korzystając już z tej największej wartości. Zapamiętujemy, które wiersze zostały zamienione za pomocą tablicy permutacji. Dzięki temu nie modyfikujemy macierzy A, a odwołując się do konkretnego wiersza zamiast używać jego indeksu wprost, używamy `p[indeks]`.

2.2.1 Złożoność czasowa

Algorytm z wyborem elementu głównego jest bardzo podobny do zwykłej eliminacji Gaussa. Jedyna różnica polega na tym, że w pierwszej pętli `for` gdy przechodzimy po kolumnach wybieramy największą wartość w danej kolumnie. Następnie wykonujemy, tak samo, resztę algorytmu na zmienionej kolejności wierszy. Ta jedna dodatkowa pętla nawet w niezoptymalizowanej wersji nie zwiększyłaby złożoności algorytmu. Wobec tego pozostaje, identycznie jak poprzednio, złożność czasowa równa $O(n)$.

2.2.2 Złożoność pamięciowa

W tym algorytmie mamy dodatkową tablicę permutacji. Jest ona rozmiaru n, który odpowiada liczbie wierszy w macierzy. Poza nią wszystko jest przechowywane tak samo jak poprzednio. Czyli, wszystkie modyfikacje wykonywane są in-place na macierzy A. Jak już wspomniałam wcześniej tablica permutacji ma n elementów, a więc nie zmienia to rzędu pamięci. Wobec tego złożoność pamięciowa to dalej $O(n)$.

2.2.3 Pseudokod

Require: A – `BlockMatrix` $n \times n$; b – wektor prawych stron dł. n

Ensure: x – rozwiązanie układu $Ax = b$

```
n ← A.size
p ← [1, 2, ..., n]                          ▷ wektor permutacji (mapowanie wierszy)
for k ← 1 to n − 1 do
    bound ← get_bottom_row(A, k)
    j ← k                  ▷ indeks wiersza z największym elementem w kolumnie
    Znajdź największy element w kolumnie i przypisz do j
    swap(p[k], p[j])                              ▷ zamiana w wektorze permutacji
    for i ← k + 1 to bound do
        z ← A[p[i], k]/A[p[k], k]
        A[p[i], k] ← 0
        last ← get_last_column(A, k + A.block_size)
        for j ← k + 1 to last do
            A[p[i], j] ← A[p[i], j] − z · A[p[k], j]
        end for
        b[p[i]] ← b[p[i]] − z · b[p[k]]
    end for
end for
x ← 0 ∈ ℝ^n
x[n] ← b[p[n]]/A[p[n], n]
for i ← n − 1 downto 1 do
    s ← b[p[i]]
    last ← get_last_column(A, i + A.block_size)
    for j ← i + 1 to last do
        s ← s − A[p[i], j] · x[j]
    end for
    x[i] ← s/A[p[i], i]
end for
return x
```

3 Rozkład LU

Rozkład LU to taki podział, w którym A = LU. Czyli zamieniamy macierz A na dwie macierze trójkątne. L jest macierzą dolnotrójkątną, a U górnotrójkątną. L zawiera mnożniki l_{ik} w miejscu zerowanych elementów.

$$L = \begin{pmatrix} 1 & \cdots & & & 1 \\ m_{21} & 1 & \cdots & & \\ m_{31} & m_{32} & 1 & \cdots & \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{n,n-1} & 1 \end{pmatrix}$$

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

Znając LU sprowadzamy nasz problem do rozwiązania

$$Ly = b$$

$$Ux = y$$

Podczas obliczeń, aby odszczędzać miejsce, będziemy modyfikować macierz A, tak aby pod przekątną znalazło się L a pozostałe miejsca były zajęte przez U.

3.1 Wariant pierwszy - bez wyboru elementu głównego

3.1.1 Złożoność czasowa

Po przyjrzeniu się algorytmowi na wyznaczenie LU zauważymy, że jest on bardzo podobny do procesu eliminacji w eliminacji Gaussa. Występują dokładnie te same obliczenia z tą różnicą, że macierz A jest modyfikowana i zapisywane są wewnątrz niej wartości L oraz U. Z tego powodu możemy stwierdzić, że złożoność czasowa tego algorytmu to również $O(n)$.

3.1.2 Złożoność pamięciowa

Jeśli chodzi o złożoność pamięciową to LU zapisujemy w A. Obydwie macierze mają tą samą strukturę. Więc złożoność pamięciowa wynosi $O(n)$.

3.1.3 Pseudokody

Algorithm 1 Generowanie rozkładu LU (in-place)

Require: A – macierz $n \times n$ typu BlockMatrix
Ensure: macierz A zmodyfikowana tak, że dolna część zawiera współczynniki L, a górna część zawiera U

$n \leftarrow A.size$
for $k \leftarrow 1$ **to** $n - 1$ **do**
    for $i \leftarrow k + 1$ **to** get_bottom_row(A, k) **do**
        Jeżeli A[k, k] jest zerem - błąd
        $l \leftarrow A[i, k]/A[k, k]$ ▷ współczynnik L
        $A[i, k] \leftarrow l$ ▷ zapisujemy l w dolnej części
        for $j \leftarrow k + 1$ **to** get_last_column(A, k) **do**
            $A[i, j] \leftarrow A[i, j] - l \cdot A[k, j]$ ▷ aktualizacja elementów U
        end for
    end for
end for

Algorithm 2 Rozwiązywanie układu przy użyciu rozkładu LU (in-place)

Require: LU – macierz zawierająca L (pod przekątną) i U (nad przekątną)
Require: b – wektor prawych stron długości n (nadpisywany)
Ensure: x – rozwiązanie układu $Ax = b$

$n \leftarrow LU.size$ ▷ 1. Rozwiązanie $Ly = b$ (podstawianie w przód), wynik zapisany w b
for $k \leftarrow 1$ **to** $n - 1$ **do**
    for $i \leftarrow k + 1$ **to** get_bottom_row(LU, k) **do**
        $b[i] \leftarrow b[i] - LU[i, k] \cdot b[k]$
    end for
end for
▷ 2. Rozwiązanie $Ux = y$ (podstawianie wsteczne)
$x \leftarrow \mathbf{0} \in \mathbb{R}^n$
for $i \leftarrow n$ **downto** 1 **do**
    $s \leftarrow b[i]$
    for $j \leftarrow i + 1$ **to** get_last_column(LU, i) **do**
        $s \leftarrow s - LU[i, j] \cdot x[j]$
    end for
    $x[i] \leftarrow s/LU[i, i]$
end for
return x

3.2 Wariant drugi - z częciowym wyborem elementu głównego