Bartosz Hanc, Paweł Surdyka

# Raport z zadania 2.

Celem zadania było zaimplementowanie dwóch algorytmów eliminacji Gaussa oraz dwóch algorytmów wykonujących LU faktoryzację macierzy. Eliminacja Gaussa jest podstawową bezpośrednią metodą (w przeciwieństwie do tzw. metod iteracyjnych) rozwiązywania równań liniowych postaci

$$Ax = b$$

gdzie $A \in \mathbb{R}^{n \times n}$ jest rzeczywistą, kwadratową macierzą współczynników natomiast $b \in \mathbb{R}^n$ to wektor wyrazów wolnych. Faktoryzacja LU macierzy to z kolei metoda pozwalająca dla dowolnej macierzy $A \in \mathbb{R}^{n \times n}$ obliczyć macierze $L$ i $U$ takie, że macierz $L$ jest macierzą trójkątną dolną, macierz $U$ jest macierzą trójkątną górną i zachodzi

$$A = LU$$

Faktoryzacja LU również może zostać wykorzystana do rozwiązania układu równań liniowych. Różnice między algorytmami polegały na zastosowaniu lub nie tzw. *pivoting*, czyli operacji polegającej na zamianie wierszy (lub kolumn) w taki sposób, aby dzielenie było wykonywane przez największą (co do wartości bezwzględnej) liczbę w kolumnie (wierszu). Ograniczyliśmy się do zaimplementowania jedynie tzw. *partial pivotingu* tj. szukania elementu wiodącego tylko wśród danej kolumny – taki też wariant został przedstawiony na wykładzie. Zastosowanie wyszukiwania elementu wiodącego w ogólności poprawia stabilność numeryczną algorytmów eliminacji Gaussa oraz faktoryzacji LU. Opiszemy teraz dokładniej poszczególne algorytmy.

## Algorytm eliminacji Gaussa

Zasadniczą ideą rozwiązania układu równań $Ax = b$ za pomocą eliminacji Gaussa jest sprowadzenie macierzy $A$ do postaci macierzy trójkątnej górnej za pomocą operacji elementarnych tj. dodania do wiersza kombinacji liniowej innych wierszy lub zamiany wierszy. Mając bowiem układ równań liniowych $A'x = b'$ dla macierzy $A'$ będącej macierzą trójkątną górną możemy łatwo znaleźć rozwiązanie $x$ w złożoności czasowej $O(n^2)$ korzystając z tzw. *backward substitution*

$$x_n = \frac{b'_n}{A'_{nn}}, \quad x_i = \frac{1}{A'_{ii}} \left( b'_i - \sum_{j=i+1}^{n} A'_{ij} x_j \right),$$

którego implementację w języku Python przedstawiono poniżej i wykorzystano w implementacji algorytmu eliminacji Gaussa. W zaproponowanej implementacji `A` jest macierzą rozszerzoną (ang. *augmented matrix*) układu równań tj. $[A \mid b]$.

```python
def backward_substitution(A: NDArray) -> NDArray:
    """
    The input matrix `A` is the augmented, upper-triangular matrix
    of the system of equations.
    """
    n = A.shape[0]
    x = np.empty(n)
    x[n - 1] = A[n - 1, -1] / A[n - 1, n - 1]
    for i in reversed(range(n - 1)):
        x[i] = 1 / A[i, i] * (A[i, -1] - np.sum(A[i, i+1:-1] * x[i+1:]))
    return x
```

Taki wybór był spowodowany zwięzłością kodu i wygodą przy implementacji samej eliminacji Gaussa. Pseudokod algorytmu eliminacji Gaussa można zapisać następująco. Zamieszczony pseudokod nie generuje jedynek na przekątnej, natomiast aby

**Algorithm 1:** Gaussian Elimination

**Data:** $A \leftarrow$ augmented matrix of the system of eqs. of shape $n \times (n+1)$
**for** $i = 1, \ldots, n-1$ **do**
  /* --- [Optional] Pivoting and rows swap */
  $p \leftarrow \underset{q \in \{i, i+1, \ldots, n\}}{\arg\max} |A_{qi}|$;
  $A_{p,:} \leftrightarrow A_{i,:}$;
  /* --- */
  **for** $j = i+1, \ldots, n$ **do**
    $A_{j,i:} \leftarrow A_{j,i:} - \frac{A_{ji}}{A_{ii}} A_{i,i:}$;
  **end**
**end**
/* Backward substitution */
$x_n \leftarrow \frac{A_{n,n+1}}{A_{n,n}}$;
**for** $i = n-1, \ldots, 1$ **do**
  $$x_i \leftarrow \frac{1}{A_{ii}} \left( A_{i,n+1} - \sum_{j=i+1}^{n} A_{ij} x_j \right);$$
**end**
**return** $x$;

tak było, wystarczy jedynie przed aktualizacją wartości w wierszach podzielić wiersz $A_{i,:}$ przez wartość $A_{ii}$ i potem przy aktualizacji nie wykonywać już dzielenia przez $A_{ii}$. Poniżej zamieszczono kod w języku Python implementujący algorytm eliminacji Gaussa w dwóch wariantach – prostej nieużywającej pivotingu oraz korzystającej z pivotingu. Poprawność implementacji została sprawdzona przez wygenerowanie wielu losowych macierzy o wartościach z przedziału $[0; 1]$ o rozmiarach $34 \times 35$

```python
def ge_simple(A: NDArray) -> NDArray:
    """
    The input matrix `A` is the augmented, matrix of the system of equations.
    This implementation does not use pivoting and can be unstable.
    """
    n = A.shape[0]
    A = A.copy().astype(np.float64)
    for i in range(n):
        A[i, :] /= A[i, i]
        for j in range(i + 1, n):
            A[j, i:] -= A[j, i] * A[i, i:]
    x = backward_substitution(A)
    return x
```

```python
def ge_pivot(A: NDArray) -> NDArray:
    """
    The input matrix `A` is the augmented, matrix of the system of equations.
    """
    n = A.shape[0]
    A = A.copy().astype(np.float64)
    for i in range(n - 1):
        p = max(range(i, n), key=lambda p: abs(A[p, i]))
        A[[i, p], :] = A[[p, i], :]
        for j in range(i + 1, n):
            A[j, i:] -= A[j, i] / A[i, i] * A[i, i:]
    x = backward_substitution(A)
    return x
```

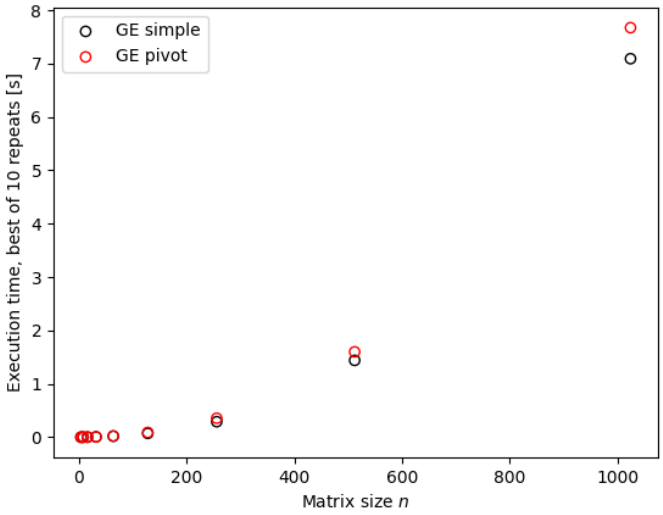
(co odpowiada układowi równań liniowych o rozmiarze równym miesiącowi + dniu urodzenia) i sprawdzenie, czy zwrócone rozwiązanie $x$ spełnia $Ax = b$. Losowe macierze o wartościach z przedziału $[0; 1]$ są dobrze uwarunkowane i wszystkie testy pokazały, że zwracane rozwiązania są poprawne. W celu pokazania jednak potrzeby wykorzystywania pivotingu przeprowadzono również taki sam test na małej macierzy $3 \times 4$

$$\begin{bmatrix} 1 & -1 & 1 & 3 \\ 2 & -2 & 4 & 8 \\ 3 & 0 & -9 & 0 \end{bmatrix}$$

dla której wyniki wyglądały następująco

```
>>> A = np.array([[1, -1, 1, 3], [2, -2, 4, 8], [3, 0, -9, 0]])
>>> x = ge_simple(A)
>>> x
array([nan, nan, nan])
>>> x = ge_pivot(A)
>>> x
array([3., 1., 1.])
>>> np.allclose(A[:, :-1] @ x, A[:, -1])
True
```

Fakt problemów numerycznych w przypadku naiwnej implementacji algorytmu eliminacji Gaussa wynika z faktu, iż w drugim kroku tego algorytmu element na diagonali, przez który dzielimy, wynosi 0. Na koniec zmierzono jeszcze czasy wykonania programu dla różnych wielkości macierzy, które przedstawiono na wykresie 1.




Rysunek 1: Czasy wykonania implementacji algorytmów eliminacji Gaussa w zależności od rozmiaru macierzy

## Algorytm faktoryzacji LU

Drugim zaimplementowanym algorytmem w ramach laboratorium był algorytm faktoryzacji LU zarówno bez pivotingu, jak i z pivotingiem. Tutaj celem jest znalezienie dwóch macierzy $L$, $U$ takich, że $L$ jest macierzą trójkątną dolną, $U$ jest macierzą trójkątną górną i zachodzi $A = LU$. Warto zauważyć, iż w przypadku algorytmu faktoryzacji LU z pivotingiem wynikiem, oprócz macierzy $L$ i $U$ jest macierz permutacji $P$ i wówczas zachodzi $PA = LU$. Pseudokod algorytmu faktoryzacji zamieszczono poniżej. W przypadku, gdy nie używamy pivotingu (tj. pomijamy odpowiedni blok kodu oznaczony jako *optional*) macierz permutacji $P$ jest macierzą identycznościową, więc możemy wówczas w ogóle ją pominąć. Poniżej zamieszczono również kod w języku Python implementujący obie wersje powyższego pseudokodu dla algorytmu faktoryzacji LU.

**Algorithm 2:** LU factorization

**Data:** $A \leftarrow$ matrix of shape $n \times n$
$L \leftarrow I_n$, $U \leftarrow A$, $P \leftarrow I_n$;
**for** $i = 1, \ldots, n-1$ **do**
  /* --- [Optional] Pivoting and rows swap */
  $p \leftarrow \underset{q \in \{i, i+1, \ldots, n\}}{\arg\max} |U_{qi}|$;
  $U_{i,i:} \leftrightarrow U_{p,i:}$, $L_{i,i:} \leftrightarrow L_{p,i:}$, $P_{i,:} \leftrightarrow P_{p,:}$;
  /* --- */
  **for** $j = i+1, \ldots, n$ **do**
    $L_{ji} \leftarrow \frac{U_{ji}}{U_{ii}}$;
    $U_{j,i:} \leftarrow U_{j,i:} - L_{ji} U_{i,i:}$;
  **end**
**end**
**return** $L, U, P$;

```python
def lu_simple(A: NDArray) -> tuple[NDArray, NDArray]:
    """
    The input matrix `A` is any real-valued square matrix of shape (n,n).
    """
    n = A.shape[0]
    L = np.eye(n)
    U = A.copy().astype(np.float64)
    for i in range(n - 1):
        for j in range(i + 1, n):
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]
    return L, U
```

```python
def lu_pivot(A: NDArray) -> tuple[NDArray, NDArray, NDArray]:
    """
    The input matrix `A` is any real-valued square matrix of shape (n,n).
    """
    n = A.shape[0]
    L = np.eye(n)
    U = A.copy().astype(np.float64)
    P = np.eye(n)
    for i in range(n - 1):
        p = max(range(i, n), key=lambda p: abs(U[p, i]))
        U[[i, p], i:] = U[[p, i], i:]
        L[[i, p], :i] = L[[p, i], :i]
        P[[i, p], ::] = P[[p, i], ::]
        for j in range(i + 1, n):
            L[j, i] = U[j, i] / U[i, i]
            U[j, i:] -= L[j, i] * U[i, i:]
    return L, U, P
```

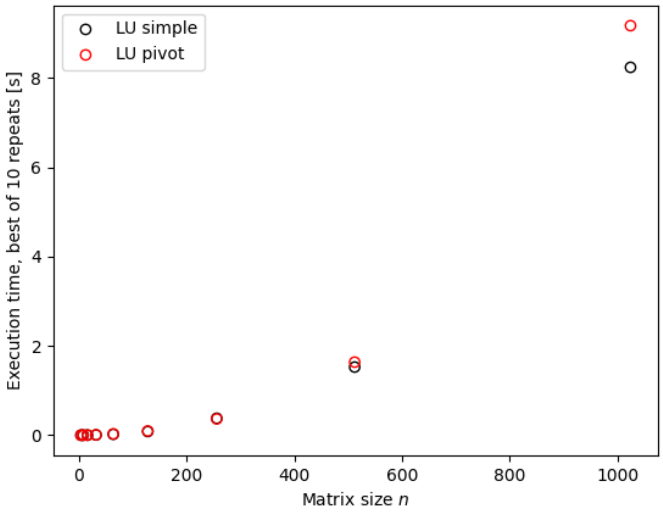
Implementacje przetestowano w sposób analogiczny jak implementacje algorytmu eliminacji Gaussa, tj. wygenerowano wiele losowych macierzy $A$ o wartościach z przedziału $[0; 1]$ i rozmiarach $34 \times 34$ (rozmiar jest równy miesiącowi + dniu urodzenia) i sprawdzano, czy zwrócone macierze $L$, $U$, $P$ spełniają $PA = LU$. Zgodnie z oczekiwaniami wszystkie testy zwróciły poprawny wynik. Ponownie porównano jeszcze algorytm z pivotingiem i bez pivotingu na małej, ale źle uwarunkowanej macierzy $3 \times 3$

$$\begin{bmatrix} 1 & -1 & 1 \\ 2 & -2 & 4 \\ 3 & 0 & -9 \end{bmatrix}$$

dla której wyniki prezentowały się następująco

```
>>> A = np.array([[1, -1, 1], [2, -2, 4], [3, 0, -9]])
>>> L, U = lu_simple(A)

array([[ 1.,  0.,  0.],
       [ 2.,  1.,  0.],
       [ 3., inf,  1.]])
array([[  1.,  -1.,   1.],
       [  0.,   0.,   2.],
       [  0.,  nan, -inf]])

>>> L, U, P = lu_pivot(A)

array([[1.        , 0.        , 0.        ],
       [0.66666667, 1.        , 0.        ],
       [0.33333333, 0.5       , 1.        ]])
array([[ 3.,  0., -9.],
       [ 0., -2., 10.],
       [ 0.,  0., -1.]])
array([[0., 0., 1.],
       [0., 1., 0.],
       [1., 0., 0.]])

>>> L @ U
array([[ 3.,  0., -9.],
       [ 2., -2.,  4.],
       [ 1., -1.,  1.]])
>>> P @ A
array([[ 3.,  0., -9.],
       [ 2., -2.,  4.],
       [ 1., -1.,  1.]])
```

Fakt problemów numerycznych w przypadku naiwnej implementacji algorytmu faktoryzacji LU wynika z faktu, iż w drugim kroku tego algorytmu element na diagonali, przez który dzielimy, wynosi 0. Widzimy zatem, że pivtoing jest kluczowy do uzyskania stabilnych numerycznie algorytmów eliminacji Gaussa oraz faktoryzacji LU. Na koniec zmierzono jeszcze czasy wykonania programu dla różnych wielkości macierzy, które przedstawiono na wykresie 2.




Rysunek 2: Czasy wykonania implementacji algorytmów faktoryzacji LU w zależności od rozmiaru macierzy