AGH

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA
W KRAKOWIE

Metody obliczeniowe w nauce i technice

Laboratorium 4

Jan Gołda, WIET 2018/2019

1. Wprowadzenie

Laboratorium miało na celu porównanie różnych realizacji rozkładu LU, używanego przy rozwiązywaniu układów równań.

Zadanie wykonano w Jupyter Notebook, pisząc w Pythonie i używając biblioteki do obliczeń naukowych o nazwie NumPy.

Wykorzystany kod oraz wyniki pomiarów są dostępne w repozytorium.

Dla uproszczenia przyjęto że macierz współczynników jest kwadratowa.

1.1. Funkcje pomocnicze

W programie wykorzystano następujące funkcje pomocnicze:

```
# tworzy rozszerzoną macierz [A|B]
def extended_matrix(A,B):
    return np.append(A, B, axis=1)
```

```
# macierz jest osobliwa <=> detA == 0
def is_singular(A):
    return np.isclose(0, np.linalg.det(A))
```

```
# sprawdza czy macierz jest dodatnio okreslona
def is_positive_definite(A):
    return np.all(np.linalg.eigvals(A) > 0)
```

```
# sprawdza czy macierz jest symetryczna
def is_symmetric(A):
    return np.all(A-A.T == 0)
```

1.2. Funkcje wyliczania układu równań

W celu wyliczenia rozwiązań układu równań przekształconego do LU wykorzystano poniższe metody:

```
# rozwiązuje układ równań w postaci LX=Y, L - trójkątna macierz dolna
def forward_substitution(L, Y):
    n = L.shape[0]
    X = np.zeros((n,1))

    for i in range(n):
        X[i,0] = Y[i,0] / L[i,i]
        for j in range(i):
            X[i,0] -= L[i,j] * X[j,0] / L[i,i]

    return X
```

```
# rozwiązuje układ równań w postaci UX=Y, U - trójkątna macierz górna
def backward_substitution(U, Y):
    n = U.shape[0]
    X = np.zeros((n,1))

    for i in range(n-1, -1, -1):
        X[i,0] = Y[i,0] / U[i,i]
        for j in range(i + 1, n):
            X[i,0] -= U[i,j] * X[j,0] / U[i,i]

    return X
```

2. Testowane algorytmy

2.1. Algorytm Doolittle'a

Polega na naprzemiennym wyliczaniu wierszy w U i kolumn w L przy użyciu następujących wzorów:

$$U_{ij} = A_{ij} - \sum_{k=0}^{i} L_{ik}U_{kj} \; dla\; j \in \{i, i+1, \dots, n\}$$

$$L_{ji} = \frac{1}{U_{ii}}\left(A_{ji} - \sum_{k=0}^{i} L_{jk}U_{ki}\right) \; dla\; j \in \{i+1, i+2, \dots, n\}$$

```
def doolittle_decomposition(A):
    n = A.shape[0]

    L = np.eye(n)
    U = np.zeros((n,n))

    for i in range(n):

        # row
        for j in range(i, n):
            U[i,j] = A[i,j]

            for k in range(i):
                U[i,j] -= L[i,k] * U[k,j]

        # col
        for j in range(i + 1, n):
            L[j,i] = A[j,i]

            for k in range(i):
                L[j,i] -= L[j,k] * U[k,i]

            L[j,i] /= U[i,i]

    return L, U
```

```
def doolittle(A, Y):
    L,U = doolittle_decomposition(A)
    Z = forward_substitution(L, Y)
    return backward_substitution(U, Z)
```

2.2. Algorytm Crouta

Metoda analogiczna do Doolittle'a, z tą różnicą że jedynki na przekątnej występują w U.

$$L_{ji} = A_{ji} - \sum_{k=0}^{i} L_{jk}U_{ki} \; dla\, j \in \{i+1, i+2, \dots, n\}$$

$$U_{ij} = \frac{1}{L_{ii}}\left(A_{ij} - \sum_{k=0}^{i} L_{ik}U_{kj}\right) dla\, j \in \{i, i+1, \dots, n\}$$

```python
def crout_decomposition(A):
    n = A.shape[0]

    L = np.zeros((n,n))
    U = np.eye(n)

    for i in range(n):

        # col
        for j in range(i, n):
            L[j,i] = A[j,i]

            for k in range(i):
                L[j,i] -= L[j,k] * U[k,i]

        # row
        for j in range(i + 1, n):
            U[i,j] = A[i,j]

            for k in range(i):
                U[i,j] -= L[i,k] * U[k,j]

            U[i,j] /= L[i,i]

    return L, U
```

```python
def crout(A, Y):
    L,U = crout_decomposition(A)
    Z = forward_substitution(L, Y)
    return backward_substitution(U, Z)
```

2.3. Algorytm Cholesky (w wersji Crouta)

Algorytm ten ma całkowicie inną zasadę działania niż poprzednie, ponieważ zamiast dokonywać rozkładu $A = LU$, wykonuje $A = LL^T$. Dzięki temu musimy obliczyć tylko L gdyż wyznaczenie L^T jest już szczegółem.

Metoda ta jest jednak o tyle problematyczna że aby działała macierz wejściowa A musi być:

1. Symetryczna
2. Dodatnio określona

A zatem dla większości układów nie możemy zastosować tej metody.

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=0}^{i} L_{jk} L_{jk}}$$

$$L_{ij} = \frac{1}{L_{ii}} \left(A_{ij} - \sum_{k=0}^{i} L_{ik} L_{jk} \right) dla\ i > j$$

```
def cholesky_decomposition(A):
    if not is_symmetric(A):
        raise ValueError("Matrix has to be symmetric to…")
    if not is_positive_definite(A):
        raise ValueError("Matrix has to be positive definite to…")

    n = A.shape[0]
    L = np.zeros((n,n))

    for j in range(n):
        for i in range(j, n):
            L[i,j] = A[i,j]

            for k in range(j):
                L[i,j] -= L[i,k] * L[j,k]

            if j == i:
                L[i,j] = math.sqrt(L[i,j])
            else:
                L[i,j] /= L[j,j]

    return L, L.transpose()
```

```
def cholesky(A, Y):
    L,U = cholesky_decomposition(A)
    Z = forward_substitution(L, Y)
    return backward_substitution(U, Z)
```

2.4. Algorytm eliminacji Gaussa

W celach porównawczych wykorzystano również implementację eliminacji Gaussa w poprzedniego laboratorium.

```
def gauss(A, B):
    if is_singular(A):
        raise ValueException("Cannot solve equation for singular matr
ix")

    U = extended_matrix(A, B)

    n = U.shape[0]

    factor = np.max(np.absolute(A), axis = 1)
    for i in range(n):
        for k in range(n+1):
            U[i,k] /= factor[i]

    for i in range(n):

        row = i
        for j in range(i+1, n):
            if abs(U[j,i]) > abs(U[row,i]):
                row = j
        U[[i,row]] = U[[row, i]]

        for j in range(i + 1, n):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    for i in range(n - 1, -1, -1):
        for j in range(i - 1, -1, -1):
            factor = U[j,i] / U[i,i]

            for k in range(n + 1):
                U[j,k] -= U[i,k] * factor

    for i in range(n):
        U[i,n] /= U[i,i]
        U[i,i] /= U[i,i]

    return U[:,n:n+1]
```

2.5. Funkcja z biblioteki NumPy

Jak punkt odniesienie użyto funkcji z biblioteki NumPy:

```
np.linalg.solve(A, Y)
```

3. Test manualny

W związku z wymaganiami algorytmu Choleskiego pierwszy test przeprowadzono dla ręcznie wprowadzonych danych, które spełniają obydwa warunki konieczne tej metody.

Układ równań:

$$A = \begin{bmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{bmatrix} \qquad Y = \begin{bmatrix} 4 \\ 10 \\ -6 \end{bmatrix}$$

Wyniki:

Gauss	Doolittle	Crout	Cholesky	NumPy
$\begin{bmatrix} 49.22222 \\ -13.1111145 \\ 2.22222 \end{bmatrix}$	$\begin{bmatrix} 49.22222222 \\ -13.11111111 \\ 2.22222222 \end{bmatrix}$	$\begin{bmatrix} 49.22222222 \\ -13.11111111 \\ 2.22222222 \end{bmatrix}$	$\begin{bmatrix} 49.22222222 \\ -13.11111111 \\ 2.22222222 \end{bmatrix}$	$\begin{bmatrix} 49.22222 \\ -13.111111 \\ 2.2222223 \end{bmatrix}$

3.1. Wnioski

Jak widać wszystkie metod dały poprawne wyniki.

Co ciekawe, metody oparte na rozkładzie UL (lub LL^T w przypadku Choleskye'go) osiągnęły nawet większą dokładność niż funkcja biblioteczna (może to być przypadek), a na pewno wykazały poprawę względem metody Gaussa.

4. Porównanie wydajności algorytmów

Pomiarów wydajności dokonano przy użyciu funkcji **%timeit** dostępnej w Jupyter Notebook.

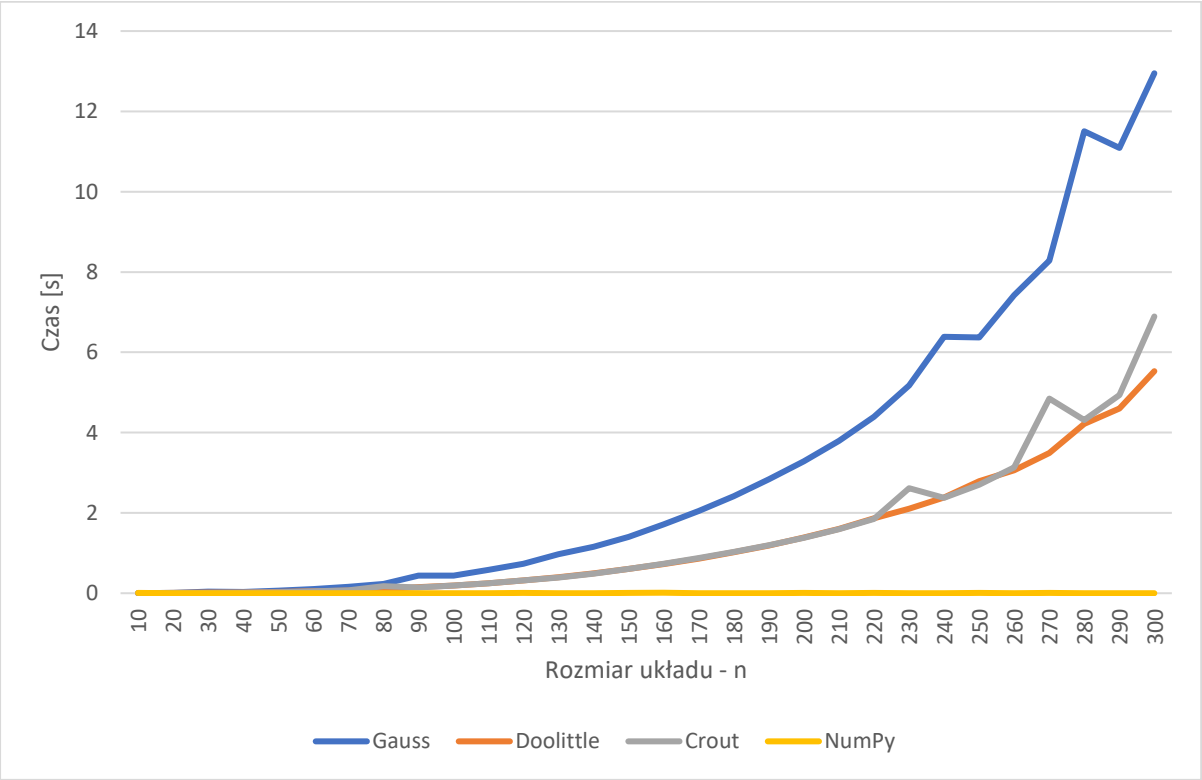
Pomiary wykonywane były dla losowego układu n równań o n nie wiadomych.

Każdy pomiar został powtórzony trzy razy, a wynik końcowy jest średnią z tych trzech pomiarów.

Pełne dane pomiarowe dostępne są tutaj.

Z racji problematycznych wymagań funkcji Choleskye'go pominięto ją w tych pomiarach.

4.1. Wyniki pomiarów



4.2. Wnioski

Jak widać na powyższym wykresie rozwiązania oparte o rozkład LU są ponad dwa razy szybsze od eliminacji Gaussa, ale nadal nawet nie zbliżają się do funkcji bibliotecznej.