Laboratorium 8 - Układy równań – metody bezpośrednie

Piotr Karamon

06.05.2024r.

Treści zadań

Napisz program, który:

1. Jako parametr pobiera rozmiar układu równań n.
2. Generuje macierz układu $A(nxn)$ i wektor wyrazów wolnych $b(n)$
3. Rozwiązuje układ równań $Ax = b$ na trzy sposoby:
   (a) poprzez dekompozycję LU macierzy A: $A = LU$;
   (b) poprzez odwrócenie macierzy A: $x = A^{-1}b$, sprawdzić czy $AA^{-1} = I$ i $A^{-1}A = I$
   (c) poprzez dekompozycję QR macierzy A: $A = QR$.
4. Sprawdzić poprawność rozwiązania (tj., czy $Ax = b$)
5. Zmierzyć całkowity czas rozwiązania układu.
6. Porównać czasy z trzech sposobów: poprzez dekompozycję LU, poprzez odwrócenie macierzy i poprzez dekompozycję QR.

Zadanie domowe: Narysuj wykres zależności całkowitego czasu rozwiązywania układu (LU, QR, odwrócenie macierzy) od rozmiaru układu równań. Wykonaj pomiary dla 5 wartości z przedziału od 10 do 100.

Uwaga: można się posłużyć funkcjami z biblioteki numerycznej dla danego języka programowania.

Zadanie

Zadanie zostało zrealizowane przy pomocy języka Python oraz bibliotek Numpy i Scipy.

Na początku tworzymy stałą `EPS` w celu sprawdzania poprawności rozwiązań. Tworzymy również funkcję do generacji macierzy i wektora wyrazów wolnych. Ustawienie `seed` na 0 na początku tych funkcji gwarantuje, że trzy metody otrzymają dokładnie te same macierze i wektory wyrazów wolnych.

```
import numpy as np
import scipy as sp
import time

EPS = 1e-10

def generate_matrix(n):
    np.random.seed(0)
    return np.random.rand(n, n) * 2


def generate_vector(n):
    np.random.seed(0)
    return np.random.rand(n) * 2
```

Metoda LU

Tworzymy funkcję która rozwiązuje układ metodą LU. Tworzymy również funkcję testującą tę metodę, która sprawdza poprawność wyniku i zwarca czas obliczenia rozwiązania. Korzystamy z funkcji `lu_factor` i `lu_solve` z biblioteki Scipy.

```
def solve_lu(A, b):
    lu, piv = sp.linalg.lu_factor(A)
    return sp.linalg.lu_solve((lu, piv), b)


def test_lu(n):
    A = generate_matrix(n)
    b = generate_vector(n)

    start = time.perf_counter()
    x = solve_lu(A, b)
    elapsed = time.perf_counter() - start

    if not np.allclose(A @ x, b, atol=EPS):
        raise ValueError("LU decomposition failed")
```

```
    return elapsed
```

Metoda macierzy odwróconej

Tworzymy analogiczne funkcje dla metody macierzy odwroconej jak dla metody LU.

```
def solve_inv(A, b):
    inv = np.linalg.inv(A)
    return inv, inv @ b


def test_inv(n):
    A = generate_matrix(n)
    b = generate_vector(n)

    start = time.perf_counter()
    inv, x = solve_inv(A, b)
    elapsed = time.perf_counter() - start

    if not np.allclose(A@inv, np.eye(n), atol=EPS):
        raise ValueError("A*inv(A) != I")

    if not np.allclose(inv@A, np.eye(n), atol=EPS):
        raise ValueError("inv(A)*A != I")

    if not np.allclose(A @ x, b, atol=EPS):
        raise ValueError("Inverse failed")

    return elapsed
```

Metoda QR

W przypadku metody QR używamy funkcji `solve_triangular` z biblioteki Sympy, by wykorzystać fakt, że macierz R jest trójkątna.

```
def solve_qr(A, b):
    Q, R = np.linalg.qr(A)
    return sp.linalg.solve_triangular(R, Q.T @ b)

def test_qr(n):
    A = generate_matrix(n)
    b = generate_vector(n)

    start = time.perf_counter()
```

```
    x = solve_qr(A, b)
    elapsed = time.perf_counter() - start

    if not np.allclose(A @ x, b, atol=EPS):
        raise ValueError("QR decomposition failed")

    return elapsed
```

Finalny program

Finalny kod programu wykorzystuje stworzone powyżej funkcje:

```
def main(n):
    try:
        lu_time = test_lu(n)
        inv_time = test_qr(n)
        qr_time = test_qr(n)
        print(f'LU = {lu_time:.6}s, INV = {inv_time:.6}s, QR =
        ↪  {qr_time:.6}s')
    except ValueError as e:
        print(f'PROCEDURE FAILED reason:{e}')

main(500)
```

Pomiar czasu

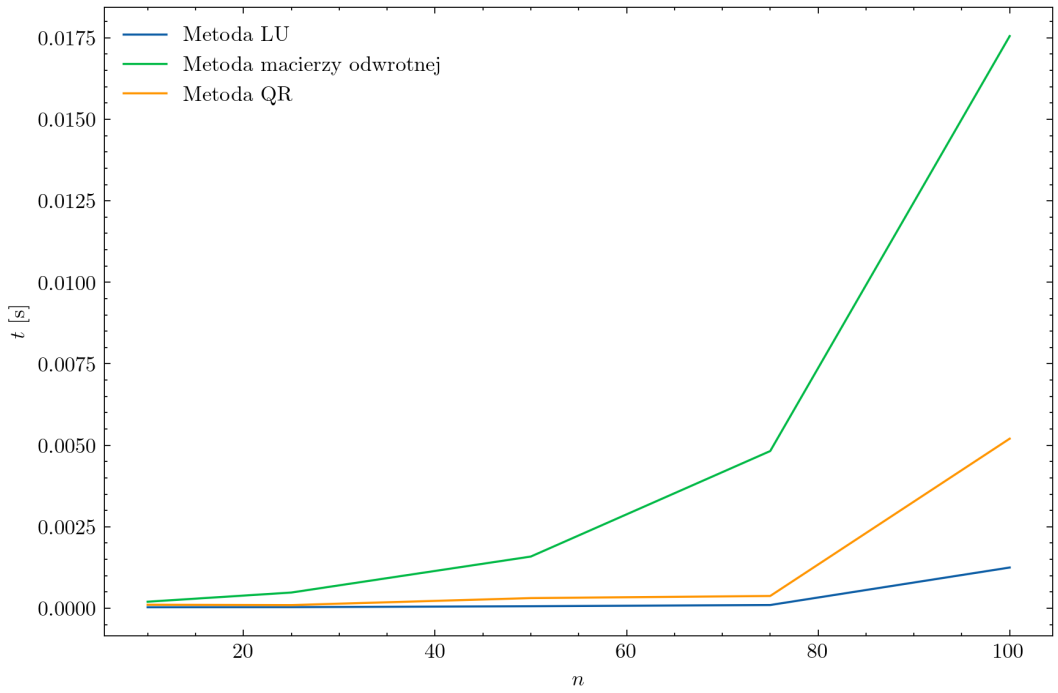
Dla trzech metod został przeprowadzony pomiar czasu dla pięciu różnych wartości n.

Tablica 1: Wyniki dla pięciu pomiarów dla trzech omawianych metod. Wartości są podane w sekundach.

n	Metoda LU	Macierz Odwrotna	Metoda QR
10	0.000031	0.000200	0.000108
25	0.000034	0.000482	0.000097
50	0.000061	0.001585	0.000311
75	0.000099	0.004823	0.000376
100	0.001249	0.017553	0.005203

Wykres

Z tabeli powyżej tworzymy wykres $t(n)$, czyli wykres zależności czasu rozwiązania równiania od ilości zmiennych w układzie.



Rysunek 1: Wykres zależności czasu rozwiązania równania od ilości zmiennych dla trzech metod: LU, macierzy odwrotnej i QR.

Wnioski

Spośród trzech testowanych metod rozwiązywania układów liniowych najszybsza okazała się być metoda dekompozycji LU. Trochę wolniejsza okazała się być metoda dekompozycji QR. Metoda macierzy odwrotnej wypadła najgorzej. Jest zdecydowanie wolniejsza od dwóch pozostałych metod, co w szczególności widać wraz ze wzrostem n. Różnice w szybkości metod widoczne są w szczególności dla dużych n, co oznacza, że w zależności od naszego problemu wybór metody może mieć małe znaczenie(gdy mamy doczynienia z małymi układami rzędu kilu/kilkunastu zmiennych) lub bardzo duże (układy 100 lub więcej zmiennych).

Bibliografia

- Marian Bubak, Katarzyna Rycerz: *Metody Obliczeniowe w Nauce i Technice. Metody bezpośrednie rozwiązywania układów liniowych*

- Metoda LU
- Metoda QR