

II zestaw zadań - Algorytmy macierzowe

Kacper Kozubowski, Mateusz Podmokły
III rok Informatyka WI

16 październik 2024

1 Treść zadania

Należy wygenerować macierze losowe o wartościach z przedziału otwartego $(10^{-8}, 1.0)$ i zaimplementować

1. Rekurencyjne odwracanie macierzy
2. Rekurencyjna eliminacja Gaussa
3. Rekurencyjna LU faktoryzacja
4. Rekurencyjne liczenie wyznacznika

Proszę zliczać liczbę operacji zmienna-przecinkowych wykonywanych podczas mnożenia macierzy.

2 Specyfikacja użytego środowiska

Specyfikacja:

- Środowisko: Jupyter Notebook,
- Język programowania: Python,
- System operacyjny: Microsoft Windows 11,
- Architektura systemu: x64.

3 Działanie algorytmów

3.1 Wykorzystane biblioteki

W realizacji rozwiązania wykorzystane zostały następujące biblioteki:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import time

```

3.2 Pseudokod

Algorithm 1 Rekurencyjne odwracanie macierzy

Input: A

Output: A_inv

function RECURSIVE_INVERSE(A)

 n = size(A)

if n = 1 **then**

return 1 / A[0, 0]

end if

 A11 = A[1:n/2, 1:n/2] // Górny lewy blok

 A12 = A[1:n/2, n/2+1:n] // Górny prawy blok

 A21 = A[n/2+1:n, 1:n/2] // Dolny lewy blok

 A22 = A[n/2+1:n, n/2+1:n] // Dolny prawy blok

 A11_inv = recursive_inverse(A11)

 S = A22 - A21 * A11_inv * A12

 S_inv = recursive_inverse(S)

 B11 = A11_inv + A11_inv * A12 * S_inv * A21 * A11_inv

 B12 = -A11_inv * A12 * S_inv

 B21 = -S_inv * A21 * A11_inv

 B22 = S_inv

 A_inv[1:n/2, 1:n/2] = B1

 A_inv[1:n/2, n/2+1:n] = B2

 A_inv[n/2+1:n, 1:n/2] = B3

 A_inv[n/2+1:n, n/2+1:n] = B4

return A_inv

end function

Algorithm 2 Rekurencyjna LU faktoryzacja

Input: A

Output: L, U

function LU_RECURSIVE(A)

 n = size(A)

if n = 1 **then**

return 1, A[0, 0]

end if

 A11 = A[1:n/2, 1:n/2] // Lewy górny blok

 A12 = A[1:n/2, n/2+1:n] // Prawy górny blok

 A21 = A[n/2+1:n, 1:n/2] // Lewy dolny blok

 A22 = A[n/2+1:n, n/2+1:n] // Prawy dolny blok

 L11, U11 = lu_recursive(A11)

 U11_inv = recursive_inverse(U11)

 L11_inv = recursive_inverse(L11)

 L21 = A21 * U11_inv

 U12 = L11_inv * A12

 S = A22 - L21 * U12

 Ls, Us = lu_recursive(S)

 L[1:n/2, 1:n/2] = L11 // Lewy górny blok

 L[1:n/2, n/2+1:n] = 0 // Prawy górny blok

 L[n/2+1:n, 1:n/2] = L21 // Lewy dolny blok

 L[n/2+1:n, n/2+1:n] = Ls // Prawy dolny blok

 U[1:n/2, 1:n/2] = U11

 U[1:n/2, n/2+1:n] = U12

 U[n/2+1:n, 1:n/2] = 0

 U[n/2+1:n, n/2+1:n] = Us

return L, U

end function

Algorithm 3 Rekurencyjne obliczanie wyznacznika macierzy

Input: A

Output: det

function RECURSIVE_DETERMINANT(A)

 L, U = lu_recursive(A)

 n = size(L)

 m = size(U)

 diagL = array(n)

 diagU = array(m)

for i **from** 0 **to** n - 1 **do**

 diagL[i] = L[i, i]

end for

for i **from** 0 **to** m - 1 **do**

 diagU[i] = U[i, i]

end for

 det = 1

for i **from** 0 **to** n - 1 **do**

 det = det * diagL[i]

end for

for i **from** 0 **to** m - 1 **do**

 det = det * diagU[i]

end for

return det

end function

Algorithm 4 Rekurencyjna eliminacja Gaussa

Input: A, b

Output: x

function RECURSIVE_GAUSSIAN_ELIMINATION(A, b)

 n = size(A)

if n = 1 **then**

 return b[0] / A[0, 0]

end if

 A11 = A[1:n/2, 1:n/2] // Lewy górny blok

 A12 = A[1:n/2, n/2+1:n] // Prawy górny blok

 A21 = A[n/2+1:n, 1:n/2] // Lewy dolny blok

 A22 = A[n/2+1:n, n/2+1:n] // Prawy dolny blok

 b1 = b[1:n/2]

 b2 = b[n/2+1:n]

 L11, U11 = lu_recursive(A11)

 L11_inv = recursive_inverse(L11)

 U11_inv = recursive_inverse(U11)

 S = A22 - A21 * U11_inv * L11_inv * A12

 Ls, Us = lu_recursive(S)

 Ls_inv = recursive_inverse(Ls)

 Us_inv = recursive_inverse(Us)

 RHS1 = L1_inv * b1

 RHS2 = Ls_inv * b2 - Ls_inv * A21 * U11_inv * RHS1

 x2 = Us_inv * RHS2

 x1 = U11_inv * RHS1 - U11_inv * L11_inv * A12 * x2

 x[1:size(x1)] = x1 // Lewa połowa

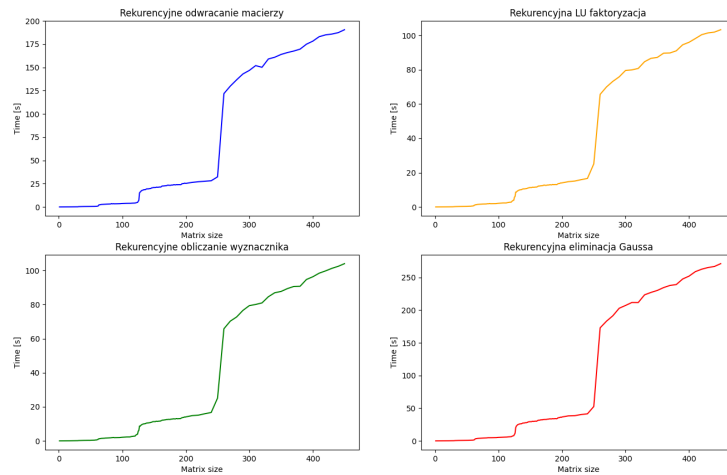
 x[size(x1) + 1:size(x1) + size(x2)] = x2 // Prawa połowa

return x

end function

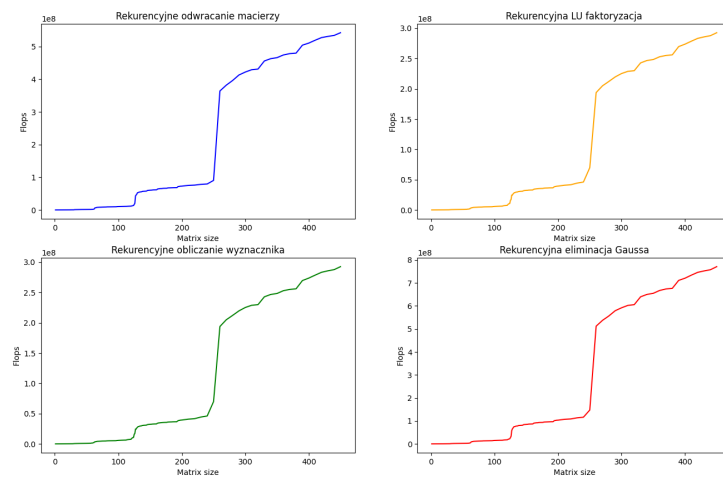
4 Porównanie działania algorytmów

4.1 Czas działania



Rysunek 1: Porównanie czasu działania każdego algorytmu.

4.2 Liczba operacji zmiennie-przecinkowych



Rysunek 2: Porównanie liczby operacji zmiennie-przecinkowych każdego algorytmu.

5 Oszacowanie złożoności obliczeniowej

W przypadku każdego algorytmu zmierzony został czas obliczeń dla

$$n \in [8, 100]$$

dla 40 równoodległych od siebie wartości oraz dopasowana krzywa złożoności obliczeniowej metodą `curve_fit` z pakietu `scipy.optimize`.

5.1 Rekurencyjne odwracanie macierzy

Dopasowana krzywa:

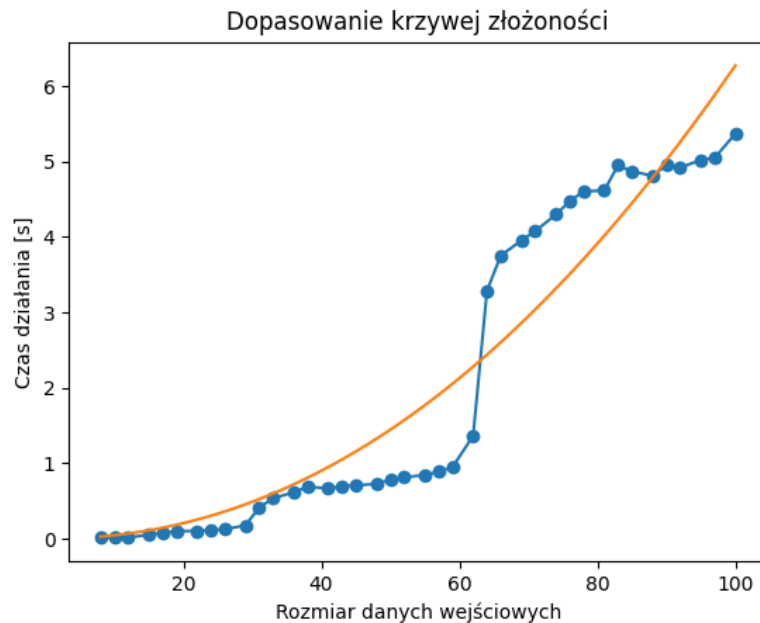
$$y = 3.7 \cdot 10^{-4} \cdot x^{2.12}$$

Zatem oszacowana złożoność obliczeniowa wynosi:

$$O(n) = n^{2.12}$$

a teoretyczna złożoność:

$$O(n) = n^{2.81}$$



Rysunek 3: Krzywa złożoności dla rekurencyjnego odwracania macierzy.

5.2 Rekurencyjna LU faktoryzacja

Dopasowana krzywa:

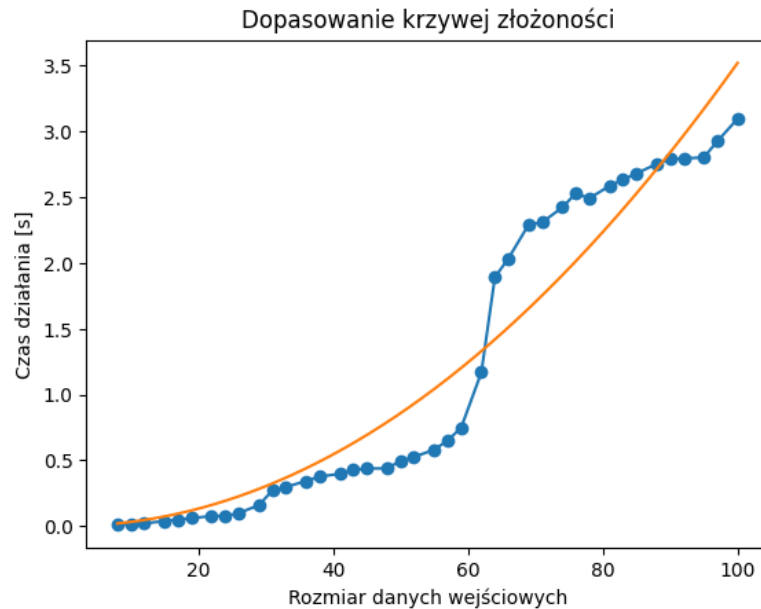
$$y = 3 \cdot 10^{-4} \cdot x^{2.04}$$

Zatem oszacowana złożoność obliczeniowa wynosi:

$$O(n) = n^{2.04}$$

a teoretyczna złożoność:

$$O(n) = n^3$$



Rysunek 4: Krzywa złożoności dla rekurencyjnej LU faktoryzacji.

5.3 Rekurencyjne liczenie wyznacznika

Dopasowana krzywa:

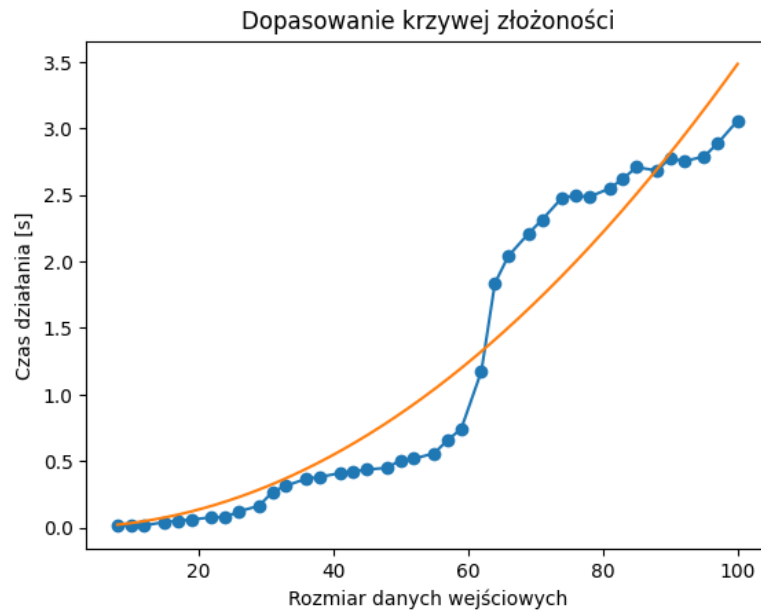
$$y = 3.1 \cdot 10^{-4} \cdot x^{2.02}$$

Zatem oszacowana złożoność obliczeniowa wynosi:

$$O(n) = n^{2.02}$$

a teoretyczna złożoność:

$$O(n) = n^3$$



Rysunek 5: Krzywa złożoności dla rekurencyjnego liczenia wyznacznika macierzy.

5.4 Rekurencyjna eliminacja Gaussa

Dopasowana krzywa:

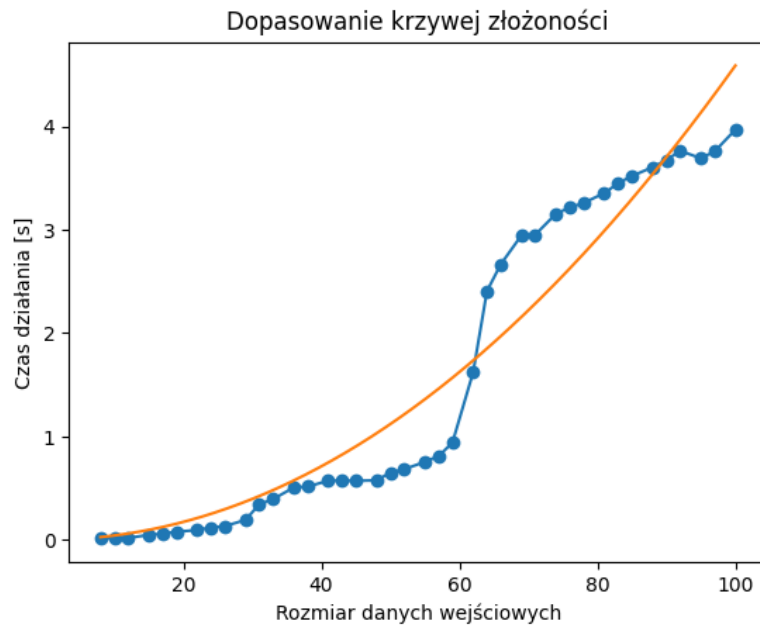
$$y = 3.9 \cdot 10^{-4} \cdot x^{2.03}$$

Zatem oszacowana złożoność obliczeniowa wynosi:

$$O(n) = n^{2.03}$$

a teoretyczna złożoność:

$$O(n) = n^3$$



Rysunek 6: Krzywa złożoności dla rekurencyjnej eliminacji Gaussa.

5.5 Podsumowanie

Oszacowane złożoności obliczeniowej prezentowanych algorytmów odbiegają od złożoności teoretycznych. Może być to spowodowane niewystarczająco stabilną platformą testową, zbyt młym rozmiarem danych wejściowych lub szeregiem usprawnień oferowanych przez biblioteki Pythona.

6 Porównanie obliczeń z biblioteką NumPy

Na wejściu mamy macierz A i wektor b z następującymi wartościami:

```
A =
[[0.37792419 0.07962609 0.98281711 0.18161286]
 [0.8118587  0.87496165 0.68841326 0.56949442]
 [0.16097145 0.46688003 0.34517206 0.22503997]
 [0.59251187 0.31226984 0.91630555 0.90963553]]
b = [0.25711829 0.1108913  0.19296273 0.49958417]
```

Rysunek 7: Wartości macierzy A i wektora b .

6.1 Rekurencyjne odwracanie macierzy

Wynik zaimplementowanego algorytmu:

```
Recursive matrix inverse:
[[ 0.25937966  2.24335044 -3.92362951 -0.48558875]
 [-0.36655885  0.13950977  2.33953436 -0.59294844]
 [ 1.1739274  -0.73117143  1.22300866 -0.07918347]
 [-1.22565182 -0.77261717  0.52062765  1.69895878]]
```

Rysunek 8: Wynik algorytmu.

Wynik uzyskany za pomocą biblioteki NumPy:

```
NumPy matrix inverse:
[[ 0.25937966  2.24335044 -3.92362951 -0.48558875]
 [-0.36655885  0.13950977  2.33953436 -0.59294844]
 [ 1.1739274  -0.73117143  1.22300866 -0.07918347]
 [-1.22565182 -0.77261717  0.52062765  1.69895878]]
```

Rysunek 9: Wynik NumPy.

6.2 Rekurencyjna LU faktoryzacja

Wynik zaimplementowanego algorytmu:

```
Recursive L:
[[ 1.          0.          0.          0.          ]
 [ 2.1482052   1.          0.          0.          ]
 [ 0.42593581  0.61508625  1.          0.          ]
 [ 1.56780618  0.26627265 -0.30643925  1.          ]]
Recursive U:
[[ 0.37792419  0.07962609  0.98281711  0.18161286]
 [ 0.          0.70390847 -1.42287958  0.17935273]
 [ 0.          0.          0.80174872  0.03736715]
 [ 0.          0.          0.          0.5885958 ]]
```

Rysunek 10: Wynik algorytmu.

Wbudowane biblioteki Pythona przy dekompozycji LU korzystają z zamiany wierszy przez co wyniki mogą się różnić. Dlatego w celu weryfikacji poprawności algorytmu pomnożymy macierze L i U , co w wyniku powinno dać macierz wejściową A . Wynik uzyskany z pomnożenia macierzy L i U :

```
L * U:  
[[0.37792419 0.07962609 0.98281711 0.18161286]  
 [0.8118587 0.87496165 0.68841326 0.56949442]  
 [0.16097145 0.46688003 0.34517206 0.22503997]  
 [0.59251187 0.31226984 0.91630555 0.90963553]]
```

Rysunek 11: Wynik mnożenia.

6.3 Rekurencyjne liczenie wyznacznika

Wynik zaimplementowanego algorytmu:

```
Recursive determinant:  
0.12553831950684474
```

Rysunek 12: Wynik algorytmu.

Wynik uzyskany za pomocą biblioteki NumPy:

```
NumPy determinant:  
0.12553831950684471
```

Rysunek 13: Wynik NumPy.

6.4 Rekurencyjna eliminacja Gaussa

Wynik zaimplementowanego algorytmu:

```
Recursive Gaussian elimination:  
[-0.68424742 0.07643672 0.41719395 0.54842062]
```

Rysunek 14: Wynik algorytmu.

Wynik uzyskany za pomocą biblioteki NumPy:

```
NumPy Gaussian elimination:  
[-0.68424742  0.07643672  0.41719395  0.54842062]
```

Rysunek 15: Wynik NumPy.

6.5 Podsumowanie

Wyniki uzyskane za pomocą zaimplementowanych algorytmów są prawie identyczne z wynikami uzyskanymi za pomocą biblioteki NumPy w Pythonie z dokładnością do bardzo błędu numerycznego. Można wnioskować, że przedstawione algorytmy działają poprawnie.