Algorytmy Macierzowe

Sprawozdanie z laboratorium nr.1

Władysław Jerzy Nieć, Paweł Surdyka

Zadania

1. Rekurencyjne odwracanie macierzy (10 punktów)
2. Rekurencyjna LU faktoryzacja (10 punktów)
3. Rekurencyjne obliczanie wyznacznika (10 punktów)

Algorytm odwracania macierzy

Dla rekurencyjnego odwracania macierzy zaczynamy od podział macierzy A na cztery podmacierze A11, A12, A21, A22 o tych samych rozmiarach. Następnie rekurencyjnie wywoływana jest funkcja odwracania macierzy na A11. W kolejnym krokach za pomocą algorytmu Strassen'a z poprzedniego laboratorium wykonujemy mnożenia na poszczególnych podmacierzach aby uzyskać podmacierze wynikowe B11,B12,B21,B22 i finalnie złożyć ją w jedną macierz inverse_A.

```
def inverse(A):
    X,Y <- rozmiary macierzy
    if X==1 and Y==1:
        return 1/A, 1, 1
    else:
        divide(A, X, Y) -> podzielenie macierzy na ćwiartki

        inverse(A11) -> odwracanie lewej górnej podmacierzy

        mul_sequence(A21,inv_A11,A12) -> przemnażanie macierzy przez siebie metodą Strassena
        S22 = A22 - S22_partly <- odejmowanie od prawej dolnej ćwiarki wynikowej macierzy powyższego mnożenia
        inverse(S22) -> odwracanie powyższej macierzy

        mul_sequence(inv_A11, A12, inv_S22, A21, inv_A11)  -> przemnażanie macierzy przez siebie metodą Strassena
        B11 = inv_A11 + B11_partly <- dodawanie do odwróconej lewej górnej ćwiarki wynikowej macierzy powyższego mnożenia

        mul_sequence(-inv_A11, A12, inv_S22) -> przemnażanie macierzy przez siebie metodą Strassena
        mul_sequence(-inv_S22, A21, inv_A11) -> przemnażanie macierzy przez siebie metodą Strassena
        B22 = inv_S22

        U = np.hstack((B11,B12)) <- scalanie macierzy wynikowej
        L = np.hstack((B21,B22)) <- scalanie macierzy wynikowej
        return np.vstack((U,L))
```

Pseudokod algorytmu

```python
def inverse(A):
    if A is None:
        return 0, 0, 0
    X,Y = A.shape[0],A.shape[1]
    if X != Y:
        raise ValueError
    if X==0 or Y==0:
        raise ArithmeticError
    if X==1 and Y==1:
        return 1/A, 1, 1
    else:
        A11,A12,A21,A22 = divide(A, X, Y)

        inv_A11, flops, mults = inverse(A11)

        S22_partly, f, m = mul_sequence(A21,inv_A11,A12)
        S22 = A22 - S22_partly
        flops += (A22.size+f)
        mults += m
        flops += f

        inv_S22, f, m = inverse(S22)
        mults += m
        flops += f

        B11_partly, B11f, B11m = mul_sequence(inv_A11, A12, inv_S22, A21, inv_A11)
        B11 = inv_A11 + B11_partly
        flops += B11.size

        B12, B12f, B12m = mul_sequence(-inv_A11, A12, inv_S22)
        B21, B21f, B21m = mul_sequence(-inv_S22, A21, inv_A11)
        B22 = inv_S22

        U = np.hstack((B11,B12))
        L = np.hstack((B21,B22))
        flops += sum((B12f, B21f, B11f))
        mults += sum((B12m, B21m, B11m))
        return np.vstack((U,L)), flops, mults
```

Algorytm w Pythonie

LU faktoryzacja

To technika faktoryzacji macierzy, która polega na dekompozycji macierzy na iloczyn dwóch macierzy trójkątnych: dolnej (Lower) i górnej (Upper) (tj. takich macierzy kwadratowych w których wszystkie współczynniki pod główną przekątną lub wszystkie współczynniki nad tą przekątną są równe zero). Faktoryzacja LU jest często używana w numerycznych metodach rozwiązywania układów równań liniowych.

```
def LU(A):
    X,Y <- rozmiary macierzy
    if X==1 and Y==1:
        return np.array([[1]]), A, 1, 1
    else:
        divide(A, X, Y) -> podzielenie macierzy na ćwiartki

        LU(A11) -> LU faktoryzacja na lewej górnej podmacierzy

        inverse(U11) -> odwracanie podmacierzy Upper otrzyamnej z wywołania LU(A11)
        inverse(L11) -> odwracanie podmacierzy Lower otrzyamnej z wywołania LU(A11)

        mul_sequence(A21, inv_U11, inv_L11, A12) -> przemnażanie macierzy przez siebie metodą Strassena
        S = A22 - S_partly <- odejmowanie od prawej dolnej ćwirtki macierzy wynikowej z powyższego mnożenia

        LU(S) -> LU faktoryzacja na macierzy S

        mul_sequence(inv_L11,A12) -> przemnażanie macierzy przez siebie metodą Strassena
        U21, L12 -> podmacierze wynikowe wypełnione zerami
        mul_sequence(A21, inv_U11) -> przemnażanie macierzy przez siebie metodą Strassena

        Uu = np.hstack((U11,U12)) \
        Ul = np.hstack((U21,Us))   -> scalanie macierzy wynikowej górnej
        U = np.vstack((Uu,Ul))    /

        Lu = np.hstack((L11,L12)) \
        Ll = np.hstack((L21,Ls))   -> scalanie macierzy wynikowej dolnej
        L = np.vstack((Lu,Ll))    /

        return L, U
```

Pseudokod algorytmu

```python
def LU(A):
    if A is None:
        return 0, 0, 0, 0
    X,Y = A.shape[0],A.shape[1]
    if X != Y:
        raise ValueError
    if X==0 or Y==0:
        raise ArithmeticError
    if X==1 and Y==1:
        return np.array([[1]]), A, 1, 1
    else:
        A11,A12,A21,A22 = divide(A, X, Y)
        L11, U11, flops, mults = LU(A11)

        inv_U11, fu, mu = inverse(U11)
        inv_L11, fl, ml = inverse(L11)
        flops += (fu + fl)
        mults += (mu + ml)

        S_partly, Sf, Sm = mul_sequence(A21, inv_U11, inv_L11, A12)
        S = A22 - S_partly
        flops += (Sf + A22.size)
        mults += Sm

        Ls, Us, f, m= LU(S)
        flops += f
        mults += m

        U12, f, m = mul_sequence(inv_L11,A12)
        flops += f
        mults += m
        U21 = np.zeros((Us.shape[0],U11.shape[1]))

        L12 = np.zeros((L11.shape[0],Ls.shape[1]))
        L21, f, m = mul_sequence(A21, inv_U11)
        flops += f
        mults += m

        Uu = np.hstack((U11,U12))
        Ul = np.hstack((U21,Us))
        U = np.vstack((Uu,Ul))

        Lu = np.hstack((L11,L12))
        Ll = np.hstack((L21,Ls))
        L = np.vstack((Lu,Ll))

        return L, U, flops, mults
```

Algorytm w Pythonie

Obliczanie wyznacznika

Do wyznaczania det(A) będziemy używać funkcji do odwracania macierzy oraz do LU faktoryzacji opisanych w wcześniej.

```
def det(A):
    X,Y = A.shape[0],A.shape[1]
    if X==1 and Y==1:
        return A[0,0], 0, 0
    else:
        divide(A, X, Y) -> podzielenie macierzy na ćwiartki
        LU(A11) -> LU faktoryzacja na lewej górnej podmacierzy

        inverse(U11) -> odwracanie podmacierzy Upper otrzyamnej z wywołania LU(A11)
        inverse(L11) -> odwracanie podmacierzy Lower otrzyamnej z wywołania LU(A11)

        mul_sequence(A21, inv_U11, inv_L11, A12) -> przemnażanie macierzy przez siebie metodą Strassena
        S = A22 - S_partly <- odejmowanie od prawej dolnej ćwirtki macierzy wynikowej z powyższego mnożenia

        det(S) -> obliczanie wyznacznika macierzy S
        diagonals_product <- iloczyn wszystkich elementów z przekątnych macierzy U11 oraz L11
        return diagonals_product*det_S
```

Pseudokod algorytmu

```
def det(A):
    if A is None:
        return 0, 0, 0
    X,Y = A.shape[0],A.shape[1]
    if X != Y:
        raise ValueError
    if X==0 or Y==0:
        raise ArithmeticError
    if X==1 and Y==1:
        return A[0,0], 0, 0
    else:
        A11,A12,A21,A22 = divide(A, X, Y)
        L11, U11, flops, mults= LU(A11)

        inv_U11, fu, mu = inverse(U11)
        inv_L11, fl, ml = inverse(L11)

        flops += fu + fl
        mults += mu + ml

        S_partly, Sf, Sm = mul_sequence(A21, inv_U11, inv_L11, A12)
        S = A22 - S_partly
        flops += (Sf + A22.size)
        mults += Sm


        det_S, Sf, Sm = det(S)
        diagonals_product = np.prod(np.diagonal(U11))*np.prod(np.diagonal(L11))
        return diagonals_product*det_S, flops + Sf + np.diagonal(U11).size, mults + Sm + np.diagonal(U11).size
```

Algorytm w Pythonie

Testy poprawności

Obliczanie różnic pomiędzy wynikami naszych funkcji oraz tymi z biblioteki numpy.

```
A = np.random.random((3,3))
```

```
array([[0.97236129, 0.5353034 , 0.65116603],
       [0.67783446, 0.20074432, 0.82562965],
       [0.01975174, 0.08753062, 0.90891652]])
```

- Odwracanie macierzy

```
inverse(A)
```

```
(array([[-0.61951111,  2.41497192, -1.74985029],
        [ 3.37206951, -4.89648321,  2.03198487],
        [-0.31127492,  0.41906195,  0.94255266]])
```

```
np.linalg.inv(A)
```

```
array([[-0.61951111,  2.41497192, -1.74985029],
       [ 3.37206951, -4.89648321,  2.03198487],
       [-0.31127492,  0.41906195,  0.94255266]])
```

Wyniki zgadzają się co do 8 miejsc po przecinku.

- LU faktoryzacja

```
L
```

```
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.69710145,  1.        ,  0.        ],
       [ 0.02031316, -0.44460322,  1.        ]])
```

```
U
```

```
array([[ 0.97236129,  0.5353034 ,  0.65116603],
       [ 0.        , -0.17241646,  0.37170087],
       [ 0.        ,  0.        ,  1.06094868]]))
```

```
L@U-A
```

```
array([[ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00],
       [-3.46944695e-18,  0.00000000e+00, -1.11022302e-16]])
```

Wyniki zgadzają się co do 16 liczb po przecinku albo i lepiej.

- Obliczanie wyznacznika

```
det(A)
```

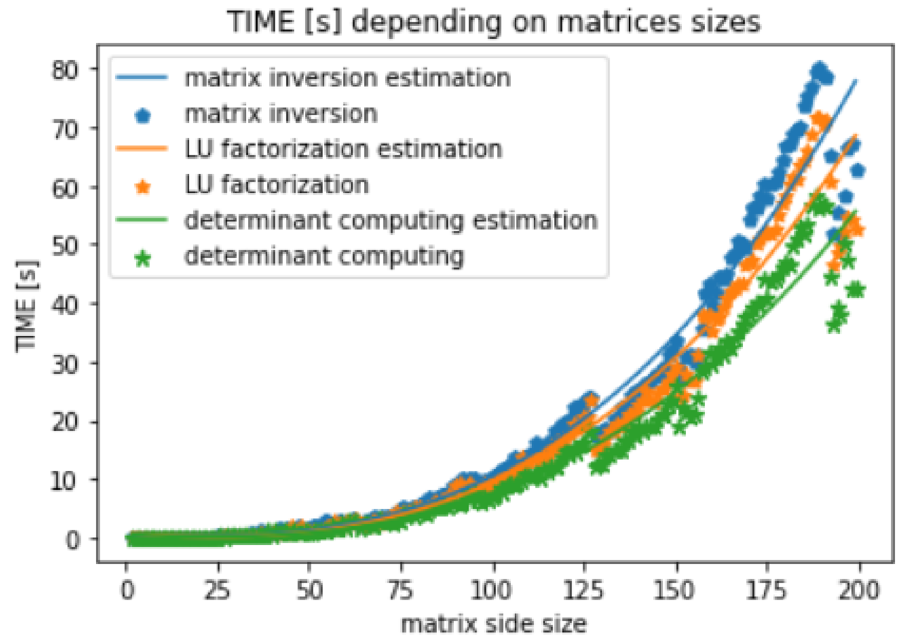
```
-0.17786920503418724
```

```
np.linalg.det(A)
```

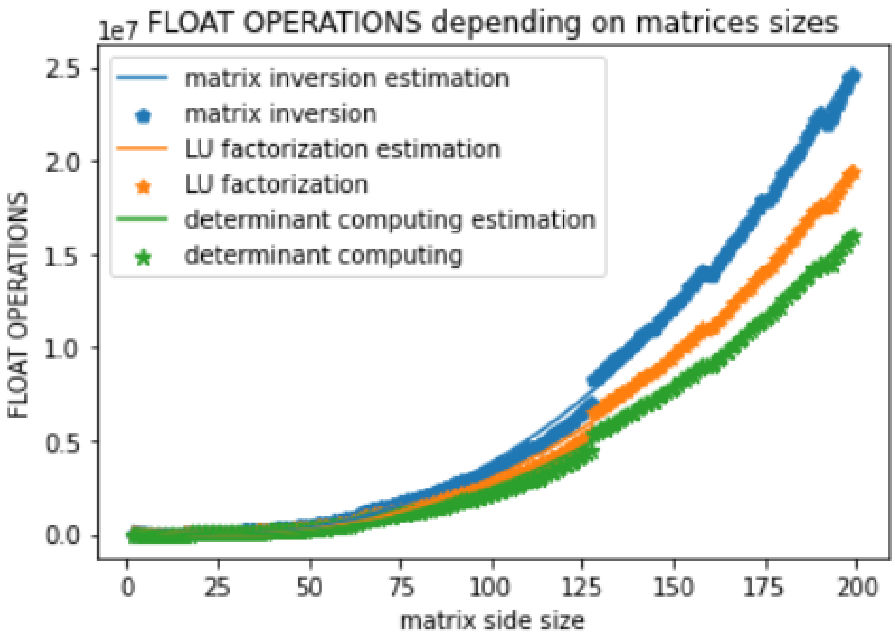
```
-0.17786920503418727
```

Wyniki zgadzają się co do 16 liczb po przecinku.

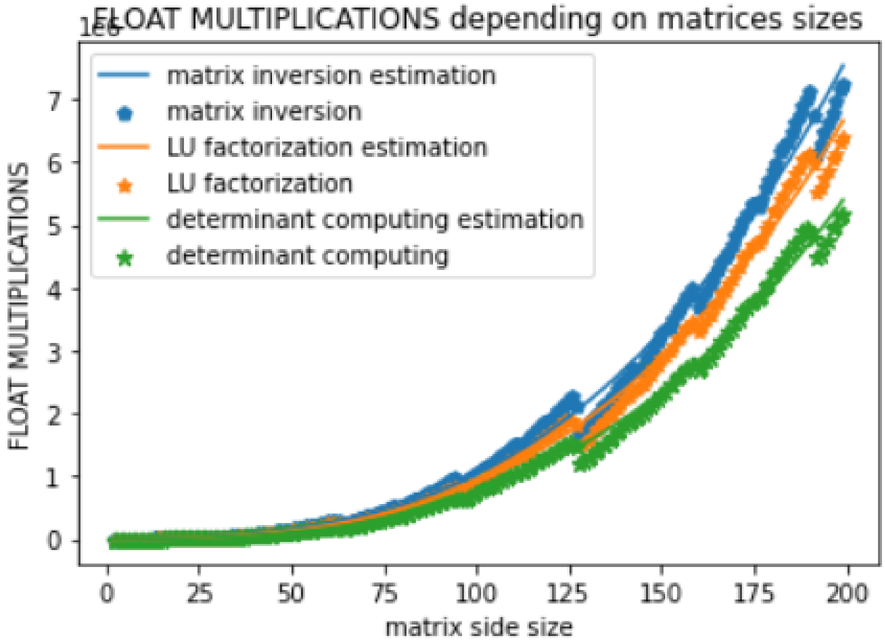
Wykresy



Wykres 1: wykres zależności czasu od rozmiaru macierz



Wykres 2: wykres zależności ilości operacji od rozmiaru macierzy



Wykres 3: wykres zależności ilości mnożeń od rozmiaru macierzy

Złożoność obliczeniowa

W przypadku wszystkich algorytmów złożoność powinna być zbliżona do złożoności algorytmu Strassena czyli w przybliżeniu O(n^2.8074).