lab1

November 8, 2021

1 Algorytmy macierzowe

Maciej Skoczeń, Kacper Kafara

grupa wtorek (A) 17:50

1.1 Mnożenie macierzy IGA i FEM

[3]:
```
import numpy as np
import re
import matplotlib.pyplot as plt
import scipy.sparse
import time
import os
```

1.1.1 Konwersja macierzy n x n wygenerowanej w Octave na macierz numpy nq x nq.

Wynikowa macierz, to macierz Octave powtórzona q razy w poziomie i pionie.

[4]:
```
def input_matrix(octave_matrix, n, m, q=1):
    result = np.zeros((n*q, m*q), dtype=np.double)

    for elem in octave_matrix:
        m = re.match(r"\s*\((\d+), (\d+)\) -> (\d+\.\d+)\s*", elem)
        if m is not None: # dla macierzy 256x256 dostaje m = None...
            x, y, value = m.groups()
        elif len(elem) > 0:
            coord, value = elem.strip().split(' -> ')
            value = float(value)
            x, y = coord.split(',')
            x, y = x[1:], y.strip()[:-1]
        else:
            continue

        for i in range(q):
            for j in range(q):
                result[i*n + int(x) - 1, j*n + int(y) - 1] = float(value)
```

```
    return result
```

[5]:
```
def load_octave_matrix(filename):
    with open(filename, "r") as file:
        return file.readlines()
```

[6]:
```
data_dir = "../output"

def resolve_path(matrix_type, width, height = None):
    if height is None: height = width
    path = f"{data_dir}/{matrix_type}-{width}x{height}.txt"
    if os.path.isfile(path): return path
    else: raise FileNotFoundError(f"Matrix file {path} not found")

resolve_matrix = lambda matrix_type, n, m, q = 1: input_matrix(
    load_octave_matrix(resolve_path(matrix_type, n, m)), n, m, q
)
```

[7]:
```
def timeit(times = 1, loops = 2):
    def timed_func(func):
        def wrapper(*args):
            tries_time = 0
            for n_try in range(times):
                loops_time = 0
                for loop in range(loops):
                    t_start = time.time()
                    func(*args)
                    loops_time += time.time() - t_start
                tries_time += loops_time / loops
            return tries_time / times
        return wrapper
    return timed_func
```

1.1.2 Sześć algorytmów mnożenia macierzy w zależności od kolejności pętli i, j oraz p.

A - macierz wejściowa n x m
B - macierz wejściowa m x k
C - macierz wynikowa n x k

[8]:
```
def dot_product(vect1, vect2, k):
    result = 0
    for p in range(k):
        result += vect1[p]*vect2[p]
    return result

def mul_and_add(val, vect, result, m):
```

```
    for j in range(m):
        result[j] += val*vect[j]
```

[9]:
```
def matrix_mul_ijp(A, B, C, n, m, k):
    for i in range(n):
        for j in range(m):
            C[i,j] = dot_product(A[i, 0:k], B[0:k, j], k)

def matrix_mul_ipj(A, B, C, n, m, k):
    for i in range(n):
        for p in range(k):
            mul_and_add(A[i, p], B[p,0:n], C[i, 0:n], m)

def matrix_mul_jip(A, B, C, n, m, k):
    for j in range(m):
        for i in range(n):
            C[i,j] = dot_product(A[i, 0:k], B[0:k, j], k)

def matrix_mul_jpi(A, B, C, n, m, k):
    for j in range(m):
        for p in range(k):
            mul_and_add(B[p,j], A[0:m, p], C[0:m, j], n)

def matrix_mul_pij(A, B, C, n, m, k):
    for p in range(k):
        for i in range(n):
            mul_and_add(A[i, p], B[p,0:n], C[i, 0:n], m)

def matrix_mul_pji(A, B, C, n, m, k):
    for p in range(k):
        for j in range(m):
            mul_and_add(B[p,j], A[0:m, p], C[0:m, j], n)

mmul = {
    "warmuprun": matrix_mul_ijp,
    "ijp": matrix_mul_ijp,
    "ipj": matrix_mul_ipj,
    "jip": matrix_mul_jip,
    "jpi": matrix_mul_jpi,
    "pij": matrix_mul_pij,
    "pji": matrix_mul_pji
}
```

1.1.3 Macierze wejściowe

[10]:
```
SIZE = 256
n = SIZE   # liczba wierszy macierzy A
m = SIZE   # liczba kolumn macierzy A / liczba wierszy macierzy B
k = SIZE   # liczba kolumn macierzy B
q = 3      # czynnik skalujący macierze
N_n = n * q
N_m = m * q
N_k = k * q
```

[11]:
```
A = resolve_matrix("iga", n, m, q)
B = resolve_matrix("fem", m, k, q)
print("Wymiary macierzy A:", A.shape)
print("Wymiary macierzy B:", B.shape)
```

```
Wymiary macierzy A: (768, 768)
Wymiary macierzy B: (768, 768)
```

1.1.4 Sprawdzanie najszybszego algorytmu

[10]:
```
stats = {}

min_loop_time = None
min_mul_type = "ijp"
mmul_optimal = mmul[min_mul_type]

for mul_type, mul_fun in mmul.items():
    print(f"{mul_type}:")
    C = np.zeros((N_n, N_k))
    loop_time = %timeit -n 1 -r 1 -o mul_fun(A, B, C, N_n, N_m, N_k)
    if mul_type == "warmuprun": continue
    stats[mul_type] = loop_time.average
    if min_loop_time is None or min_loop_time > loop_time.average:
        min_loop_time = loop_time.average
        min_mul_type = mul_type
        mmul_optimal = mul_fun

print(f"\nNajszybsza konfiguracja: {min_mul_type}")
```

```
warmuprun:
2min 28s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
ijp:
2min 27s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
ipj:
2min 53s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
jip:
2min 21s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
jpi:
2min 57s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
pij:
2min 55s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
pji:
2min 58s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

Najszybsza konfiguracja: jip
```

[11]:
```
from IPython.display import Markdown as md
table = """
| Kolejność pętli | Czas [s] |
|:---------------:|:--------:|
"""
for loop_order, loop_time in stats.items():
    table += f"|{loop_order}|{loop_time:>.6}|\n"

md(table)
```

[11]:

Kolejność pętli	Czas [s]
ijp	147.213
ipj	173.647
jip	141.856
jpi	177.66
pij	175.134
pji	178.625

1.1.5 Arbitralny wybór optymalnego algorytmu

Jeżeli ma zostać wykorzystany algorytm wyznaczony obliczeniowo (w komórce powyżej), nie wykonywać komórki poniżej.

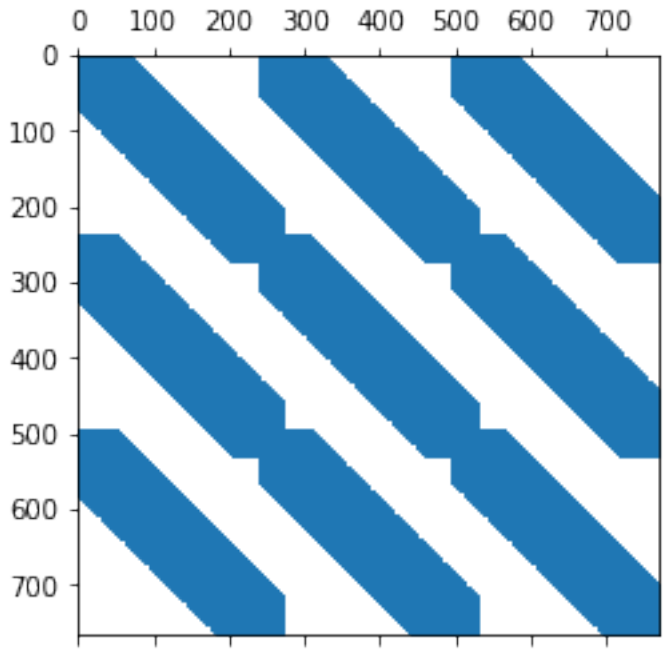
[ ]:
```
mmul_optimal = mmul["ijp"]
```

1.1.6 Sprawdzanie niezerowych miejsc macierzy A, B i C

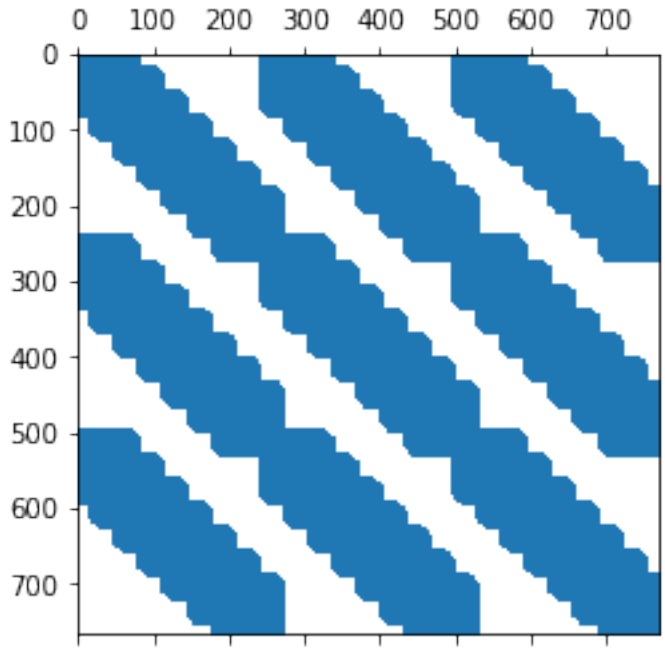
[12]:
```
C = np.zeros((N_n, N_k))
%timeit -n 1 -r 1 mmul_optimal(A, B, C, N_n, N_m, N_k)
```

```
2min 23s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

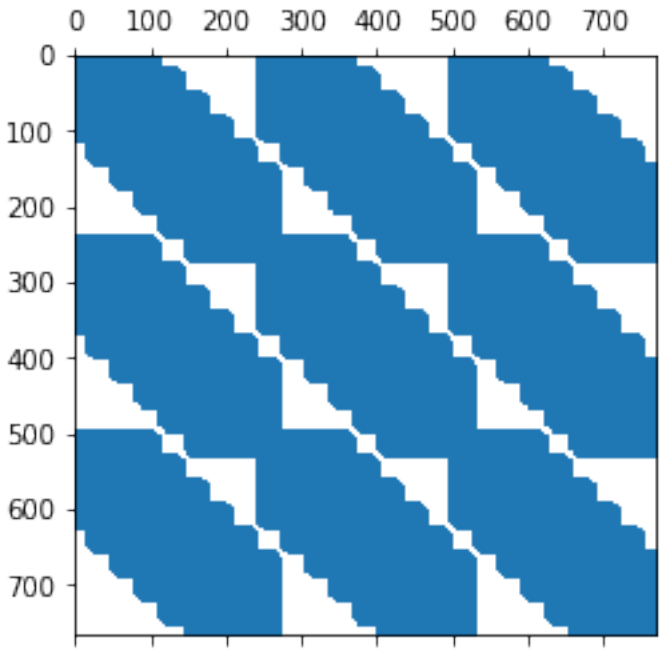
[13]:
```
plt.spy(scipy.sparse.csr_matrix(A))
plt.show()
```



[14]:
```
plt.spy(scipy.sparse.csr_matrix(B))
plt.show()
```



[15]:
```
plt.spy(scipy.sparse.csr_matrix(C))
plt.show()
```



1.1.7 Mnożenie blokowe macierzy przyjmując wielkość bloków: block_size x block_size.

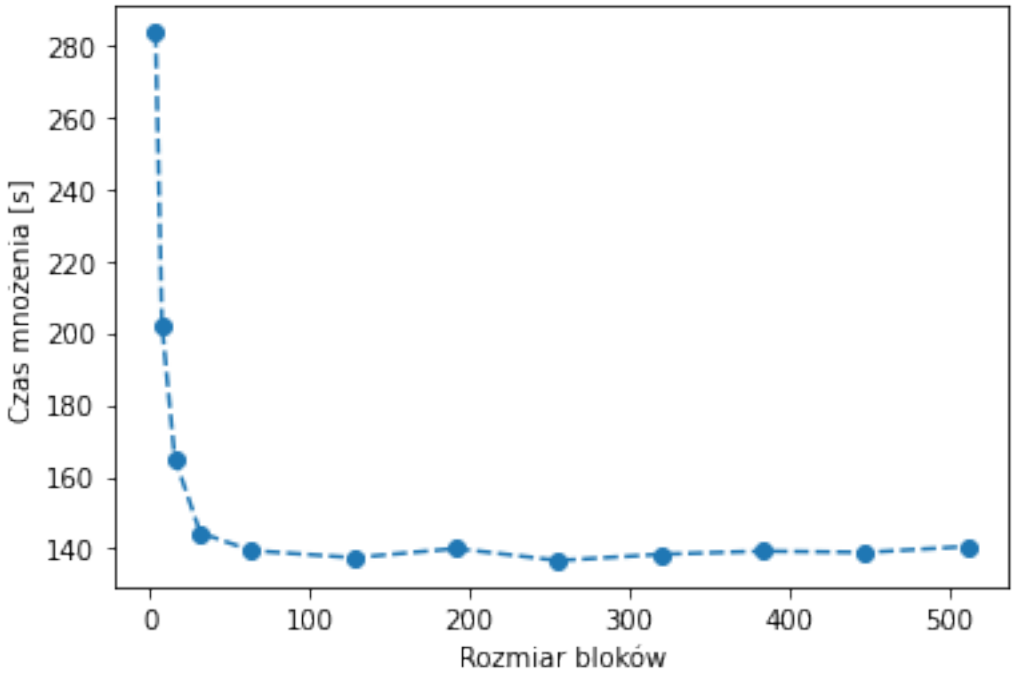
matrix_mul_fun to jedna z wybranych funkcji mnożenia macierzy zaimplementowanych wyżej

[12]:
```
def matrix_block_mul(A, B, C, n, m, k, block_size, matrix_mul_fun):
    for j in range(0, m, block_size):
        j_block = min(m - j, block_size)
        for i in range(0, n, block_size):
            i_block = min(n - i, block_size)
            for p in range(0, k, block_size):
                p_block = min(k - p, block_size)
                matrix_mul_fun(A[i:i + i_block, p:p + p_block],
                               B[p:p + p_block, j:j + j_block],
                               C[i:i + i_block, j:j + j_block],
                               i_block, j_block, p_block)
```

[23]:
```
from math import log
block_sizes = [2**i for i in range(2, int(log(N_n * q, 2))) if 2**i <= N_n]
block_sizes.extend([128 + 64, 256 + 64, 256 + 2 * 64, 256 + 3 * 64])
block_sizes.sort()

mul_time = []
```

```
runs = 1

for block_size in block_sizes:
    print("Obliczenia dla rozmiaru bloku", block_size, end='  ')
    C = np.zeros((N_n, N_k))
    t0 = time.time()
    for j in range(runs):
        matrix_block_mul(A, B, C, N_n, N_m, N_k, block_size, mmul_optimal)
    run_time = time.time() - t0 / runs
    mul_time.append(run_time)
    print(f"{run_time:.5}")

plt.plot(block_sizes, mul_time, linestyle='--')
plt.scatter(block_sizes, mul_time)
plt.xlabel("Rozmiar bloków")
plt.ylabel("Czas mnożenia [s]")
plt.show()
```

```
Obliczenia dla rozmiaru bloku 4  283.43
Obliczenia dla rozmiaru bloku 8  202.3
Obliczenia dla rozmiaru bloku 16  165.1
Obliczenia dla rozmiaru bloku 32  144.41
Obliczenia dla rozmiaru bloku 64  139.49
Obliczenia dla rozmiaru bloku 128  137.5
Obliczenia dla rozmiaru bloku 192  140.06
Obliczenia dla rozmiaru bloku 256  136.72
Obliczenia dla rozmiaru bloku 320  138.45
Obliczenia dla rozmiaru bloku 384  139.34
Obliczenia dla rozmiaru bloku 448  138.97
Obliczenia dla rozmiaru bloku 512  140.64
```



1.1.8 Liczba operacji zmiennoprzecinkowych w standardowym algorytmie mnożenia macierzy

W każdej iteracji najbardziej wewnętrznej pętli wykonujemy jedno dodawanie i jedno mnożenie -> $2m$ operacje zmienno przecinkowe. Wchodzimy w tą pętlę dokładnie $n * k$ razy (dwie zewnętrzne pętle). Sumarycznie $2nmk$ operacji zmiennoprzecinkowych.

1.1.9 Liczba operacji zmiennoprzecinkowych w algorytmie blokowego mnożenia macierzy

Niech $A = [a_{ij}]_{nxm} = [A_{ij}]_{NxM}$, $B = [b_{ij}]_{mxk} = [B_{ij}]_{MxK}$, $C = AB = [c_{ij}]_{nxk} = [C_{ij}]_{NxK}$, gdzie przez A_{ij}, B_{ij}, C_{ij} rozumiemy odpowiednie macierze blokowe rozmiarów, odpowiednio $\frac{n}{N}x\frac{m}{M}$, $\frac{m}{M}x\frac{k}{K}$, $\frac{n}{N}x\frac{k}{K}$.

Obliczenie C_{ij} wymaga M mnożeń macierzy. Każde mnożenie macierzy, korzystając z poprzedniego rezultatu, kosztuje $2\frac{nmk}{NMK}$ operacji zmienno przecinkowych.

Finalnie otrzymamy $N * K * M * 2 * \frac{nmk}{NMK} = 2nmk$

Zatem w przypadku naszych macierzy, mnożenie ich będzie wymagać następującej liczby operacji zmiennoprzecinkowych:

[24]:
```
n, m = A.shape
_, k = B.shape

print(2 * n * m * k)
```

905969664