CIE6002/CSC6119 Spring 2024 Assignment 1

Strassen's algorithm for matrix multiplication

This file is an example solution for reference. `The standards of grading` are listed for each question.

In [ ]:
```
import numpy as np
import pandas as pd
import time
```

1. Code implementation and verification with 2x2 matrices

`Grading keys`:

- K1: 10 points for correct functions
- K2, K3: 5 points for each printed result

In [ ]:
```
def standard_mul(A,B):

    p,m1 = A.shape
    m2,n = B.shape
    assert (m1 == m2), print(f'The number of rows of the first matrix (here : {m
    C = np.zeros((p,n))
    for i in range(p):
        for j in range(n):
            C[i,j] = sum([A[i,k]*B[k,j] for k in range(m1)])
    return C

def even_split(A):
    p,m = A.shape
    assert (p ==m), print('The function only works for square matrices of even s
    A11, A12, A21, A22 = A[:p//2, :p//2], A[:p//2, p//2:], A[p//2:,:p//2], A[p//
    return A11, A12, A21, A22

def strassen_mul_2n(A,B):
    assert (B.shape == A.shape), print('The function only works for square matri
    if (A.shape[0]>2):
        A11, A12, A21, A22 = even_split(A)
        B11, B12, B21, B22 = even_split(B)
        H1 = strassen_mul_2n(A11+A22, B11+B22)
        H2 = strassen_mul_2n(A21+A22, B11)
        H3 = strassen_mul_2n(A11, B12-B22)
        H4 = strassen_mul_2n(A22, B21-B11)
        H5 = strassen_mul_2n(A11+A12, B22)
        H6 = strassen_mul_2n(-A11+A21, B11+B12)
        H7 = strassen_mul_2n(A12-A22, B21+B22)
        C1 = np.concatenate([H1+H4-H5+H7, H3+H5],1)
        C2 = np.concatenate([H2+H4, H1-H2+H3+H6],1)
        return np.concatenate([C1,C2],0)
    else:

        # either:
```

```
        a11,a12,a21,a22 = A.ravel()
        b11,b12,b21,b22 = B.ravel()
        h1 = (a11+a22)*(b11+b22)
        h2 = (a21+a22)*(b11)
        h3 = (a11)*(b12-b22)
        h4 = (a22)*(b21-b11)
        h5 = (a11+a12)*(b22)
        h6 = (-a11+a21)*(b11+b12)
        h7 = (a12-a22)*(b21+b22)
        return np.array([[h1+h4-h5+h7, h3+h5], [h2+h4, h1-h2+h3+h6]])
        # or :
        # return standard_mul(A,B)
```

In [ ]:
```
A = np.array([[1,2],[0,1]])
B = np.array([[0,1],[3,1]])
print(strassen_mul_2n(A,B))
print(standard_mul(A,B))
```

```
[[6 3]
 [3 1]]
[[6. 3.]
 [3. 1.]]
```

2. Implementation and comparison for matrices $2^n \times 2^n$

`Grading keys`:

- K4: showed recorded time for 2 methods * 4 degrees = 2*4*1.5 = 12 points
- K5: showed recorded errors for 2 methods * 4 degrees = 2*4*1.5 = 12 points
- K6: divided computation time by the theoretical complexity, and the quotient stays approximately a constant throughout all n = 2*4*1.5 = 12 points
- K7: explicitly stated that the Strassen's method is less accurate = 4 points.

In [ ]:
```
n_list = [6, 7, 8, 9]
df = pd.DataFrame(columns=[ 'time strassen', 'time standard', 'max err strassen'
df.index.name = 'exp_n'

for exp in n_list:

    size = 2**exp
    A = np.random.random([size,size])
    B = np.random.random([size,size])

    t0 = time.perf_counter()
    myC = strassen_mul_2n(A,B)
    t1 = time.perf_counter()
    df.loc[exp]['max err strassen'] = f'{np.max(np.abs(myC-A@B)):.03e}'
    df.loc[exp]['time strassen'] = f'{t1-t0:.03f}'
    fac =(t1-t0)/(7**exp)
    df.loc[exp]['time quotient strassen'] = f'{1e6*fac:.02f}e-6'

    t0 = time.perf_counter()
    myC = standard_mul(A,B)
    t1 = time.perf_counter()

    df.loc[exp]['max err standard'] = f'{np.max(np.abs(myC-A@B)):.03e}'
    df.loc[exp]['time standard'] = f'{t1-t0:.03f}'
```

```
    fac = (t1-t0)/(8**exp)
    df.loc[exp]['time quotient standard'] = f'{1e6*fac:.02f}e-6'
```

In [ ]:
```
df.head()
```

Out[ ]:

exp_n	time strassen	time standard	max err strassen	max err standard	time quotient strassen	time quotient standard
6	0.111	0.057	5.969e-13	1.421e-14	0.95e-6	0.22e-6
7	0.796	0.423	2.256e-12	4.263e-14	0.97e-6	0.20e-6
8	5.402	3.397	8.811e-12	1.279e-13	0.94e-6	0.20e-6
9	37.559	31.590	6.327e-11	3.695e-13	0.93e-6	0.24e-6

The Strassen's method is less accurate.

With n = 5,6,7,8, the Strassen's time divied by 7^n stays approximatly 0.95e-6 throughout all n; the Standard's time divied by 8^n stays approximatly 0.22e-6 throughout all n.

3. Implementation and comparison for any matrix

`Grading keys` :

- K8 : correct Strassen's function = 16 points
- K9 : showed recorded time for 2 methods * 4 degrees = 2*4*1 = 8 points (if only Strassen's, no point is removed)
- K10 : showed recorded errors for 2 methods * 4 degrees = 2*4*1 = 8 points (if only Strassen's, no point is removed)
- K11: divided computation time by the theoretical complexity, and the quotient stays approximately a constant throughout all n = 2*4*1 = 8 points (if only Strassen's, no point is removed)

In [ ]:
```
def strassen_mul(A,B):
    p,m1 = A.shape
    m2,n = B.shape
    assert (m1 == m2), print(f'The number of rows of the first matrix (here : {m
    if (p<=5) or (m1<=5) or (n<=5):
        return standard_mul(A,B)
    else:
        nn = np.min([p,m1,n])
        k = 2*2**int(np.log(nn)/np.log(4))

        A11_, A12_, A21_, A22_ = A[:k, :k], A[:k, k:], A[k:,:k], A[k:,k:]
        B11_, B12_, B21_, B22_ = B[:k, :k], B[:k, k:], B[k:,:k], B[k:,k:]
        A11, A12, A21, A22 = even_split(A11_)
        B11, B12, B21, B22 = even_split(B11_)
        H1 = strassen_mul(A11+A22, B11+B22)
        H2 = strassen_mul(A21+A22, B11)
        H3 = strassen_mul(A11, B12-B22)
        H4 = strassen_mul(A22, B21-B11)
```

```
        H5 = strassen_mul(A11+A12, B22)
        H6 = strassen_mul(-A11+A21, B11+B12)
        H7 = strassen_mul(A12-A22, B21+B22)
        C1 = np.concatenate([H1+H4-H5+H7, H3+H5], 1)
        C2 = np.concatenate([H2+H4, H1-H2+H3+H6], 1)
        C = np.concatenate([C1,C2], 0)
        C1_ = np.concatenate([C + strassen_mul(A12_,B21_), strassen_mul(A11_,B12
        C2_ = np.concatenate([strassen_mul(A21_,B11_) + strassen_mul(A22_,B21_),
        return np.concatenate([C1_,C2_], 0)


n_list = [1,2,4,6]
df = pd.DataFrame(columns=[ 'time strassen', 'time standard', 'max err strassen'
df.index.name = 'n'

p0 = 50
m0 = 60
n0 = 70
for a in n_list:
    p = a* p0
    n = a* n0
    m = a* m0
    # print('p:', p)

    A = np.random.random([p,m])
    B = np.random.random([m,n])

    t0 = time.perf_counter()
    myC = strassen_mul(A,B)
    t1 = time.perf_counter()
    df.loc[a]['max err strassen'] = f'{np.max(np.abs(myC-A@B)):.03e}'
    df.loc[a]['time strassen'] = f'{t1-t0:.03f} sec'
    fac = (t1-t0)/(7**(np.log(p)/np.log(2)))
    df.loc[a]['time quotient strassen'] = f'{fac*1e6:.02f}e-6'

    t0 = time.perf_counter()
    myC = standard_mul(A,B)
    t1 = time.perf_counter()

    df.loc[a]['max err standard'] = f'{np.max(np.abs(myC-A@B)):.03e}'
    df.loc[a]['time standard'] = f'{t1-t0:.03f} sec'
    fac = (t1-t0)/(8**(np.log(p)/np.log(2)))
    df.loc[a]['time quotient standard'] = f'{fac*1e6:.02f}e-6'
```

In [ ]:
```
df.head()
```

Out[ ]:

n	time strassen	time standard	max err strassen	max err standard	time quotient strassen	time quotient standard
1	0.063 sec	0.043 sec	1.421e-14	7.105e-15	1.07e-6	0.34e-6
2	0.453 sec	0.353 sec	4.263e-14	3.908e-14	1.10e-6	0.35e-6
4	3.373 sec	2.728 sec	1.137e-13	1.066e-13	1.17e-6	0.34e-6
6	11.192 sec	9.744 sec	1.990e-13	2.274e-13	1.24e-6	0.36e-6

Take n $\approx$ 50, 100, 200, 300 respectively (you can also take $\times$ 60 or $\times$ 70), the Strassen's time divied by $7^{log_2(n)}$ stays approximatly 1.1e-6 throughout all n; the Standard's time divied by $8^{log_2(n)}$ stays approximatly 0.35e-6 throughout all n.