Lab 1 : Matrix Multiplication Algorithms

Course: CS20009.04 Data Structure
Name: Yijia Chen
Student Number: 24300240127
Date: October 10, 2025

Abstract

This lab implements and compares the performance of two matrix multiplication algorithms: the ordinary algorithm and Strassen's algorithm. Their theoretical complexities are $O(n^3)$ and $O(n^{2.81})$ respectively. Empirical results show that the running time of Strassen's algorithm grows slower than the ordinary algorithm for larger matrix sizes.

Introduction

Matrix multiplication is a fundamental operation in linear algebra and computer science. The complexity of ordinary matrix multiplication algorithm is $O(n^3)$, where n is the size of the matrices. This complexity is quite high, especially for large matrices. However, there are more efficient algorithms for matrix multiplication, such as Strassen's algorithm, which has a complexity of $O(n^{2.81})$. This lab evaluates and compares the performance of Strassen's algorithm and the ordinary algorithm through empirical testing.

Implementation

Ordinary Algorithm

Given two $n \times n$ matrices A and B, we want to figure out the matrix product $C = AB$. The ordinary algorithm follows the definition of matrix multiplication. The entry c_{ij} for $i, j = 1, 2, \ldots, n$ is computed by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Because of the triply-nested loops, the ordinary algorithm takes $O(n^3)$ time.

Strassen's Algorithm

Strassen's algorithm accelerates matrix multiplication by recursively compute 7 subproducts instead of 8, as in the naive divide-and-conquer algorithm which breaks the $n \times n$ matrix into 8 $\frac{n}{2} \times \frac{n}{2}$ matrices.

Suppose each of the $n \times n$ matrices A, B, and C are divided as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

To compute C, we recursively compute the following 7 $\frac{n}{2} \times \frac{n}{2}$ matrix products.

$$
\begin{aligned}
P_1 = &\quad A11 \cdot (B_{12} - B_{22}), && (1)\\
P_2 = &\quad (A_{11} + A_{12}) \cdot B_{22}, && (2)\\
P_3 = &\quad (A_{21} + A_{22}) \cdot B_{11}, && (3)\\
P_4 = &\quad A_{22} \cdot (B_{21} - B_{11}), && (4)\\
P_5 = &\quad (A_{11} + A_{22}) \cdot (B_{11} + B_{22}), && (5)\\
P_6 = &\quad (A_{12} - A_{22}) \cdot (B_{21} + B_{22}), && (6)\\
P_7 = &\quad (A_{11} - A_{21}) \cdot (B_{11} + B_{12}). && (7)
\end{aligned}
$$

Then, it can be verified that

$$
\begin{aligned}
C_{11} = &\quad P_5 + P_4 - P_2 + P_6, && (8)\\
C_{12} = &\quad P_1 + P_2, && (9)\\
C_{21} = &\quad P_3 + P_4, && (10)\\
C_{22} = &\quad P_5 + P_1 - P_3 - P_7. && (11)
\end{aligned}
$$

gives the result of $C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$.

```
#pragma once
#include <iostream>
#include <vector>
#include <iomanip>
#include <stdexcept>

using std::cout; using std::endl; using std::vector;
using mat = vector<vector<double>>; // matrix

bool isSquare(const mat& m) {
    if (m.empty()) return false;
    for (const auto& row : m) {
        if (row.size() != m.size()) return false;
    }
    return true;
}

bool isValid(const mat& m) {
    if (m.empty()) return false;
    int n = m[0].size();
    for (const auto& row : m) {
        if (row.size() != n) return false; // every row.size() should be the same
    }
    return true;
}

mat createMat(int n, double val = 0.0) {
    return mat(n, vector<double>(n, val));
}

void printMat(const mat& m) {
    for (auto& row : m) {
        for (auto elem : row)
            cout << std::fixed << std::setprecision(2) << std::setw(8) << elem << ' ';
        cout << '\n';
    }
    cout << '\n';
}

bool isPowerOf2(int n) {
    return n > 0 && (n & (n - 1)) == 0;     // n = 100...00 iff isPowerOf2
}

int nextPowerOf2(int n) {
    if (n < 1) return 0;
    int res = 1;
    while (res < n) {       // until res larger than n for the first time
        res *= 2;
    }
    return res;
}

mat extendMat(const mat& m) {
    int n0 = m.size();              // original size
    int n = nextPowerOf2(n0);       // extended size
    mat res = createMat(n);
    for (int i = 0; i < n0; ++i) {
```

```
        for (int j = 0; j < n0; ++j)
            res[i][j] = m[i][j];
    }
    return res;
}

mat matAdd(const mat& a, const mat& b) {        // for ordiinary algorithm
    if (!isValid(a) || !isValid(b) || a.size() != b.size() || a[0].size() != b[0].size()){
         throw std::invalid_argument("error: matrix a & b cannot add!");
    }
    mat c = a;
    for (int i = 0; i < a.size(); ++i)
        for (int j = 0; j < a[0].size(); ++j)
            c[i][j] += b[i][j];
    return c;
}

mat matAdd(const mat& a, int ra, int ca, const mat& b, int rb, int cb, int n) {
    mat c = createMat(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            c[i][j] = a[ra+i][ca+j] + b[rb+i][cb+j];
    return c;
}

mat matSub(const mat& a, int ra, int ca, const mat& b, int rb, int cb, int n) {
    mat c = createMat(n);
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            c[i][j] = a[ra+i][ca+j] - b[rb+i][cb+j];
    return c;
}

// ordinary matrix multiply

mat matMul(const mat& a, const mat& b) {
    if (!isValid(a) || !isValid(b) || a[0].size() != b.size())
        throw std::invalid_argument("Matrix a & b cannot multiply!");

    mat c(a.size(), vector<double>(b[0].size(), 0.0));
    for (int i = 0; i < a.size(); ++i) {
        for (int j = 0; j < b[0].size(); ++j) {
            for (int k = 0; k < b.size(); ++k)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
    return c;
}



// Strassen's algorithm



mat matMulStrassenRange(const mat& a, int ra, int ca, const mat& b, int rb, int cb, int n) {
    if (n == 1) return {{a[ra][ca] * b[rb][cb]}};
    int k = n / 2;
    mat p1 = matMulStrassenRange(a, ra, ca, matSub(b, rb, cb+k, b, rb+k, cb+k, k), 0, 0, k);
    mat p2 = matMulStrassenRange(matAdd(a, ra, ca, a, ra, ca+k, k), 0, 0, b, rb+k, cb+k, k);
```

```
    mat p3 = matMulStrassenRange(matAdd(a, ra+k, ca, a, ra+k, ca+k, k), 0, 0, b, rb, cb, k);
    mat p4 = matMulStrassenRange(a, ra+k, ca+k, matSub(b, rb+k, cb, b, rb, cb, k), 0, 0, k);
    mat p5 = matMulStrassenRange(matAdd(a, ra, ca, a, ra+k, ca+k, k), 0, 0,
                                 matAdd(b, rb, cb, b, rb+k, cb+k, k), 0, 0, k);
    mat p6 = matMulStrassenRange(matSub(a, ra, ca+k, a, ra+k, ca+k, k), 0, 0,
                                 matAdd(b, rb+k, cb, b, rb+k, cb+k, k), 0, 0, k);
    mat p7 = matMulStrassenRange(matSub(a, ra, ca, a, ra+k, ca, k), 0, 0,
                                 matAdd(b, rb, cb, b, rb, cb+k, k), 0, 0, k);
    mat c = createMat(n);
    for (int i = 0; i < k; ++i) {
        for (int j = 0; j < k; ++j) {
            c[i][j] = p5[i][j] + p4[i][j] - p2[i][j] + p6[i][j];
            c[i][j+k] = p1[i][j] + p2[i][j];
            c[i+k][j] = p3[i][j] + p4[i][j];
            c[i+k][j+k] = p5[i][j] + p1[i][j] - p3[i][j] - p7[i][j];
        }
    }
    return c;
}

mat matMulStrassen(const mat& a, const mat& b) {
    if (!isValid(a) || !isValid(b))
        throw std::invalid_argument("error: matrices not valid");
    if (!isSquare(a) || !isSquare(b))
        throw std::invalid_argument("error: matrices not square");
    if (a.size() != b.size())
        throw std::invalid_argument("error: a.size() != b.size()");
    int n = a.size();
    if (isPowerOf2(n)) {
        return matMulStrassenRange(a, 0, 0, b, 0, 0, n);
    } else {
        return matMulStrassenRange(extendMat(a), 0, 0, extendMat(b), 0, 0, nextPowerOf2(n));
    }
}
```

Experiment

To approximate the empirical complexity, a benchmarking program is implemented, which runs the algorithms with varied matrix sizes from 5 to 120. The inputs of the matrix multiplication procedures are generated randomly throughout the experiment, but remain the same for both algorithms each run. The test of each input size is carried out 100 or 500 times to eliminate the impact of randomness of an individual run.

```
#include <iostream>
#include <vector>
#include <random>
#include <fstream>
#include <chrono>
#include "matrixMult.h"

using namespace std;
using namespace std::chrono;

using Matrix = vector<vector<double>>;


// generate random n*n matrix
Matrix randomSquareMatrix(int n) {
    static std::mt19937 gen(std::random_device{}()); // Mersenne Twister engine
    std::uniform_real_distribution<double> dist(0, 10000);
    Matrix M(n, vector<double>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            M[i][j] = dist(gen);
    return M;
}

int main() {
    ofstream fout("matrix_times.csv");

    int totalRuns = 200;

    // Write header row
    fout << "InputSize";
    for (int run = 1; run <= totalRuns; ++run) {
        fout << ",Original_Run" << run;
    }
    for (int run = 1; run <= totalRuns; ++run) {
        fout << ",Strassen_Run" << run;
    }
    fout << "\n";

    vector<int> sizes(20);
    for (int i = 0; i < 20; ++i) sizes[i] = (i + 1) * 10;

    for (int n : sizes) {
        fout << n;
        vector<long long> originalTimes;
        originalTimes.reserve(totalRuns);
        vector<long long> strassenTimes;
        strassenTimes.reserve(totalRuns);

        for (int run = 1; run <= totalRuns; ++run) {
            mat A = randomSquareMatrix(n);
            mat B = randomSquareMatrix(n);

            auto start = high_resolution_clock::now();
            matMul(A, B);
            auto end = high_resolution_clock::now();
            originalTimes.push_back(duration_cast<microseconds>(end - start).count());
```

```
            start = high_resolution_clock::now();
            matMulStrassen(A, B);
            end = high_resolution_clock::now();
            strassenTimes.push_back(duration_cast<microseconds>(end - start).count());

            if (run % 5 == 0) {
                cerr << "Run " << run << " done\n";
            }
        }
        for (auto t : originalTimes) fout << "," << t;
        for (auto t : strassenTimes) fout << "," << t;
        fout << "\n";

        cerr << "======== Done n = " << n << " ========\n"; // progress display
    }

    fout.close();
    cerr << "Finished! Results written to matrix_times.csv\n";
}
```

The output of each run is saved in an electronic table file. Due to runtime and hardware limitations, the experiments were carried out in separate runs. Input sizes $5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60$ were run 500 times each, recorded in `matrix_times_5_60.csv`. Input sizes $70, 80, 90, 100, 110, 120$ were run 200 times each, recorded in `matrix_times_70_120.csv`. Input size 200 was run 100 times for the ordinary algorithm and 55 times for Strassen's algorithm, recorded in `matrix_times_200.csv`. Note that the average running time, which appear in the second and third columns of the tables, were calculated using Excel separately after the benchmarking program terminates.

Matrix multiplication is quite slow for large matrix sizes. To ensure the stability of my personal laptop computer and the benchmarking program, input size 500, 1000, 1500, 2000 were run separately, for 50, 8, 4, and 4 times each. Correspondingly, the benchmarking program is revised to output the timing data immediately. The results were saved in `matrix_times_500.csv`, `matrix_times_1000.csv`, and `matrix_times_1500.csv`.

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <fstream>
#include <chrono>
#include "matrixMult.h"

using namespace std;
using namespace std::chrono;

using Matrix = vector<vector<double>>;


// generate random n*n matrix
Matrix randomSquareMatrix(int n) {
    static std::mt19937 gen(std::random_device{}()); // Mersenne Twister engine
    std::uniform_real_distribution<double> dist(0, 10000);
    Matrix M(n, vector<double>(n));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            M[i][j] = dist(gen);
    return M;
}

int main() {
    ofstream fout("matrix_times_1500.csv");

    int totalRuns = 4;

    // Write header row
    fout << "InputSize,Original,Strassen\n";
    cout << "InputSize,Original,Strassen" << endl;

    vector<int> sizes(20);
    for (int i = 0; i < 20; ++i) sizes[i] = (i + 3) * 500;

    for (int n : sizes) {
        vector<long long> originalTimes;
        originalTimes.reserve(totalRuns);
        vector<long long> strassenTimes;
        strassenTimes.reserve(totalRuns);

        for (int run = 1; run <= totalRuns; ++run) {
            mat A = randomSquareMatrix(n);
            mat B = randomSquareMatrix(n);

            auto start = high_resolution_clock::now();
            matMul(A, B);
            auto end = high_resolution_clock::now();
            originalTimes.push_back(duration_cast<microseconds>(end - start).count());

            start = high_resolution_clock::now();
            matMulStrassen(A, B);
            end = high_resolution_clock::now();
            strassenTimes.push_back(duration_cast<microseconds>(end - start).count());

            fout << n << "," << originalTimes.back() << "," << strassenTimes.back() << "\n";
```

```
            fout.flush();
            cout << n << "," << originalTimes.back() << "," << strassenTimes.back() << endl;

            if (run % 5 == 0) {
                cerr << "Run " << run << " done\n";
            }
        }
        cerr << "======== Done n = " << n << " ========\n"; // progress display
    }

    fout.close();
    cerr << "Finished! Results written to matrix_times.csv\n";
}
```

Analysis

We want to show that both of the running time of these two algorithms follow a polynomial growth, which is

$$T(n) = c \cdot n^a.$$

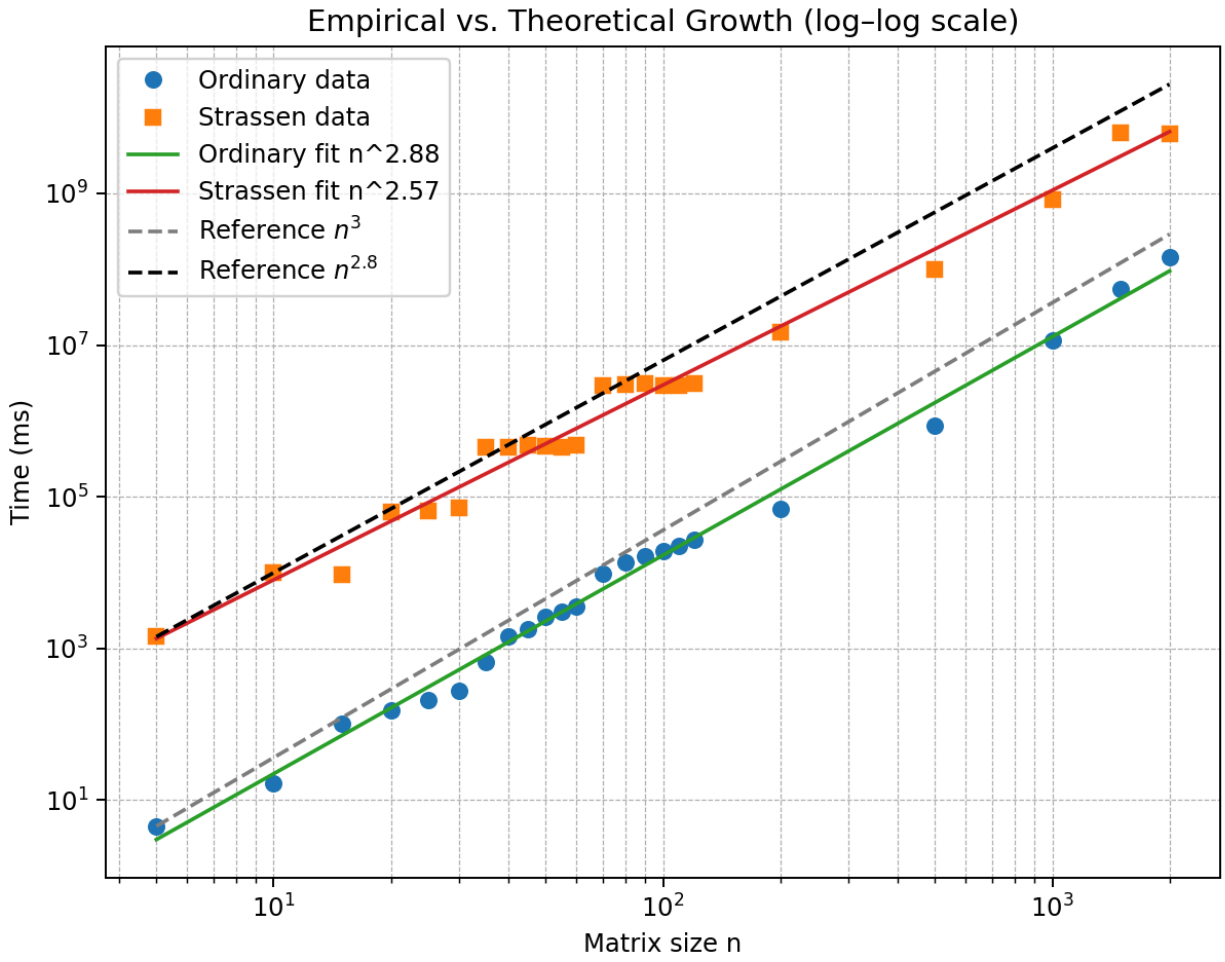
Take logs of both sides,

$$\lg T(n) = \lg c + a \lg n,$$

and we get a linear relationship between $\lg T(n)$ and $\lg n$.

The following code in Python can be used to plot the empirical growth against the theoretical growth. Log-log plot is adopted to visualize the relationship in straight lines. The slopes of the straight lines correspond to the desired a in the exponent. The theoretical growth of n^3 and $n^{2.81}$ are exhibited in the same plot for comparison.

```python
import numpy as np
import matplotlib.pyplot as plt

# Experimental data
n = np.array([5,10,15,20,25,30,35,40,45,50,55,60,70,80,90,100,110,120,200,500,1000,1500,2000])
ordinary = np.array([4.522,16.846,103.048,150.556,207.532,273.354,657.926,1454.63,
                     1811.388,2642.514,3028.104,3578.038,9621.615,13831.32,
                     16495.85,19044.37,22551.68,26728.385,69852.47,851557.4,
                     11452375.5,54580416.8,143505110])
strassen = np.array([1425.35,9959.524,9430.688,63808.25,64851.16,71175.02,453375.3,
                     451203.3,470522.8,469929.2,448538.6,476238.2,2950251,3028525,
                     3086257,2899294,2914468,3083276,14839065,98522252.3,
                     833197238.6,6241155812,6046228300])

# Fit log(time) = a + b*log(n)
def fit_powerlaw(n, t):
    b, a = np.polyfit(np.log(n), np.log(t), 1)
    return a, b  # log(k), exponent

a1, b1 = fit_powerlaw(n, ordinary)
a2, b2 = fit_powerlaw(n, strassen)

# Predicted curves from fitted exponents
n_fit = np.linspace(min(n), max(n), 200)
ordinary_fit = np.exp(a1) * n_fit**b1
strassen_fit = np.exp(a2) * n_fit**b2

# Theoretical reference curves (normalized for visibility)
n3_curve = (n_fit / n_fit[0])**3 * ordinary[0]
n28_curve = (n_fit / n_fit[0])**2.8 * strassen[0]

print(f"Ordinary fit exponent ≈ {b1:.2f}")
print(f"Strassen fit exponent ≈ {b2:.2f}")

# Plot experimental and theoretical curves (log-log)
plt.figure(figsize=(8,6))
plt.loglog(n, ordinary, 'o', label='Ordinary data')
plt.loglog(n, strassen, 's', label='Strassen data')
plt.loglog(n_fit, ordinary_fit, '-', label=f'Ordinary fit n^{b1:.2f}')
plt.loglog(n_fit, strassen_fit, '-', label=f'Strassen fit n^{b2:.2f}')
plt.loglog(n_fit, n3_curve, '--', color='gray', label=r'Reference $n^3$')
plt.loglog(n_fit, n28_curve, '--', color='black', label=r'Reference $n^{2.8}$')

plt.xlabel("Matrix size n")
plt.ylabel("Time (ms)")
plt.title("Empirical vs. Theoretical Growth (log-log scale)")
plt.legend()
plt.grid(True, which="both", ls="--", lw=0.5)
plt.show()
```



According to the plot, the ordinary algorithm fits in $n^{2.88}$, and Strassen's algorithm fits in $n^{2.57}$.

$2.57 < 2.88$ indicates that Strassen's algorithm runs faster than the ordinary algorithm for larger matrix sizes, which aligns with theoretical growth trend. However, the fitted exponents are lower than the theoretical 3 and 2.81.

Discussion

The empirical result does not fit in the theoretical growth very closely, which is attributed to many reasons.

Firstly, the scale of the test is limited in comparison with the crossover point of Strassen's algorithm. Although the lab specification says that Strassen's algorithm outperforms the conventional cubic-time algorithm for input sizes $n \geq 32$ on contemporary computing machinery, it differs greatly from the experiment data. For example, when input size is 1000, the ordinary algorithm takes 11452375.5 microseconds on average, while the average running time of Strassen's algorithm is 833197238.6 microseconds. According to D'Alberto and Nicolau, the crossover points range from $n = 400$ to $n = 2150$ on different systems, and may not exist on some machines. Because of hardware and time constraints, the crossover point was not reached. Strassen's algorithm performs 7 recursive multiplications and multiple additions and subtractions of matrices. Function call overheads and the frequent memory allocation result in larger constant factors and low order terms. They are likely to dominate the running time when the input size is small, leading to the poor fit.

The log-log plot exhibits a "staircase" pattern arouund input size 50 and 100 instead of a smooth curve. This might be caused by the memory hierarchy. With the growth of input matrix size, data may exceed certain memory levels like L1, L2, or L3 cache. This guess remains to be further investigated.

In addition, other uncontrolled factors exist. On a modern operating system such as Windows 11, background processes and scheduling policies can introduce timing noise that undermines microsecond-level accuracy. Since program optimization and

scheduling are largely opaque to the user, the measured runtime reflects only a general trend rather than an exact performance profile.

Conclusion

Although Strassen's algorithm did not outperform the ordinary matrix multiplication algorithm with small input size, the experimental trend aligns with its lower complexity. With larger matrices and optimized implementation, its advantage would become more evident.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009).
Introduction to Algorithms (3rd ed.). MIT Press.

Acknowledgement

ChatGPT-5 offered helpful guidance on generating random numbers, measuring the running time in C++, ploting results with Matplotlib and refining the writing for clarity and correctness.

> https://chatgpt.com/s/t_68d1f697dc9c8191aab672cd50b08c5f
> https://chatgpt.com/s/t_68d1f613548c81918a8bf7a13fd3881b
> https://chatgpt.com/s/t_68e720126b908191b321b0e0da699558

Appendix : Lab specification

Abstract

In the present laboratory exercise, you are instructed to implement two distinct methods for **matrix multiplication**: **Strassen's algorithm** and **ordinary algorithm**. You are encouraged to undertake a comprehensive analysis of both algorithms, encompassing both theoretical frameworks and empirical validation through experimentation.

1 Introduction

The Strassen algorithm, attributed to Volker Strassen, represents an enhanced approach to matrix multiplication, utilizing a divide-and-conquer strategy for optimization. The algorithm exhibits a computational complexity of $\Theta(n^2.81)$ as opposed to $\Theta(n^3)$. While the difference between 2.81 and 3 may appear negligible, it is important to note that this variation occurs in the exponent, thereby leading to a substantial impact on the algorithm's running time. In practical terms, Strassen's algorithm outperforms the conventional cubic-time algorithm for input sizes $n \geq 32$ on contemporary computing machinery.

2 The Tasks

2.1 Implementation

Write code for Strassen's and ordinary algorithms.

2.2 Experiment

Evaluate the empirical cost associated with the implementation of the algorithms to determine its alignment with the theoretical computational complexity.

2.3 Documentation

All document for the answers of the above questions.

3 Points for Attention

(1) For the implementation of these algorithms, you are free to select a programming language of your choice.
(2) Kindly upload the source code files along with their associated documentation in a compressed ZIP format to the elearning system for assessment.
(3) The deadline of this lab is *23:59:59 on October 10*.
(4) If you have any questions please feel free to contact teaching assistants.