AMS 326: Exam 2 Report

AMS 326-1: Numerical Analysis, Spring 2025
Joe Martinez

February 25, 2025

Attached to this report is the source code under the name *exam2-matrix.py* and the matrix calculations in *matrix-result.txt*. All calculations were done in Python with the help of libraries like NumPy, Math, and Random.

1 Problem T2.2

1.1 Generating Uniform Distributed Matrices

We were tasked to generate 2 matrices $A^{n\times n}$ and $B^{n\times n}$ with all of their elements being sampled from a uniform distribution from the range -2, 2.

$$a_{ij}, b_{ij} \sim U(-2, 2)$$

Generating the matrix was done by repeatedly sampling from a uniform distribution (see Algorithm 1). The sampling was done using Python's random number library, *Random*. The function *random::uniform(a, b)* samples from a uniform distribution of the (inclusive) range $[a, b]$. The resulting matrices are shown in *matrix-result.txt*.

1.2 Multiplying Matrices

We were tasked to develop algorithms that compute $A \times B$ and estimate the number of floating-point operations by the naive and Strassen approach. The number of floating-point operations for the naive approach was estimated to be 2146435072, since it's $n * n * n$ multiplications and $n * n * (n-1)$ additions, where $n = 2^{10}$. Meanwhile, the number of operations for the Strassen approach was estimated to be 1971035287, where it takes 7 * the number of operations for each resursive call of the Strassen method on each sub matrix, plus $(18 * (n/2))^2$ additions, which is 18 additions for each recursive call.

The naive approach multiples each pair of elements in each row and column combination to produce the final result, hence an $O(n^3)$ algorithm. If the product matrices were split into sections of 4 and multiplies, there would be 8 matrix products to add for the final result. The Strassen approach in this case shines by making the calculation using only 7 matrix products, and 18 matrix additions, lowering complexity to $O(n^{2.807})$

Algorithm 1: Generate Uniform Matrix

Input: A dimension N to construct a $N \times N$ matrix, and a range $[a, b]$ to sample uniformly from

Output: A $N \times N$ matrix

UniformMatrix (N, a, b)
- $matrix \leftarrow$ new empty array;
- **for** $i \leftarrow$ *0* **to** $N - 1$ **do**
  - $row \leftarrow$ new empty array;
  - **for** $j \leftarrow$ *0* **to** $N - 1$ **do**
    - $value \leftarrow$ sample $X \sim U(a, b)$;
    - $row.append(value)$;
  - **end**
  - $matrix.append(row)$;
- **end**
- **return** *matrix*

```python
def FPO_operations_naive(n):
    multiplications = n * n * n  # Each element requires n multiplications
    additions = n * n * (n - 1)  # Each element requires (n-1) additions
    return multiplications + additions

def FPO_operations_strassen(n):
    if n == 1:
        return 1  # Base case: single multiplication

    multiplications = 7 * FPO_operations_strassen(n // 2)  # Strassen uses 7 recursive c
    additions = 18 * (n // 2) ** 2  # Strassen requires 18 matrix additions/subtractions

    return multiplications + additions
```

Figure 1: Estimation functions for floating point operations for a given method

```python
def naive_matrix_multiplication(A, B):
    rows_A = len(A)
    cols_A = len(A[0])
    rows_B = len(B)
    cols_B = len(B[0])

    if cols_A != rows_B:
        raise ValueError("Columns of A must be equal to rows of B")

    # Initialize result matrix with zeros
    result = [[0 for _ in range(cols_B)] for _ in range(rows_A)]

    # Naive triple loop for matrix multiplication
    for i in range(rows_A):
        print(f'Row {i} or {rows_A}')
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result
```

Figure 2: Naive implementation in Python

```python
def strassen_matrix_multiplication(A, B):
    if (A.shape != B.shape):
        raise ValueError("Matrices must be square and of the same size")

    n = A.shape[0]
    # print(n);

    if n == 1:
        return (A @ B)

    mid = n // 2
    A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:, mid:]
    B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:, mid:]

    M1 = strassen_matrix_multiplication(A11 + A22, B11 + B22)
    M2 = strassen_matrix_multiplication(A21 + A22, B11)
    M3 = strassen_matrix_multiplication(A11, B12 - B22)
    M4 = strassen_matrix_multiplication(A22, B21 - B11)
    M5 = strassen_matrix_multiplication(A11 + A12, B22)
    M6 = strassen_matrix_multiplication(A21 - A11, B11 + B12)
    M7 = strassen_matrix_multiplication(A12 - A22, B21 + B22)

    C11 = np.array(M1) + np.array(M4) - np.array(M5) + np.array(M7)
    C12 = np.array(M3) + np.array(M5)
    C21 = np.array(M2) + np.array(M4)
    C22 = np.array(M1) - np.array(M2) + np.array(M3) + np.array(M6)

    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
    return C.tolist()
```

Figure 3: Strassen implementation in Python