Comparing the Efficiency of Python, Matlab and C using Matrix Multiplication

Oisin Mc Laughlin – 22441106
MA203 Optional Project
30/03/2023

Introduction
Matrix multiplication is one of the core operations in linear algebra, computer science and electrical engineering. In computer science, matrix multiplication becomes particularly useful with computer graphics as a digital image is more less an image to begin with or even being used when decoding digital video. Efficiency is key when it comes to computers and as a current computer science student, I wanted to find out which language was the most efficient when it came to matrix multiplication. I've chosen to compare the languages Python, Matlab and C as I've been using a lot of Matlab this semester when it comes to my introduction to modelling module and I spent two semesters last year using C in a programming module. Python as well as being one of the most used languages in computer science, was the first programming language I properly started to learn back when I was in TY and is one of the reason's that I fell in love with computer science.

Objective
This project aims to compare the performance of matrix multiplication across three programming languages: Python, Matlab and C. Python offering its NumPy library that enables numeric computing as well as being renowned for its ease of use. Matlab being literally named 'matrix laboratory' and is very famous for the built in matrix operations that it offers and is a staple of scientific and engineering research. C, known for its efficiency being a lower level programming language and also my experience of spending long hours hating but also loving it last year, also known for its rich history when it comes to programming! By comparing these languages, I will hopefully be able to gather insights into the efficiency of each of these languages and then at the end be able to rank them. The comparison is based on the execution time of each matrix multiplication operation, each matrix will have a fixed size and the operation will be repeated a fixed amount of times (100,000) and an average will be found.

Environment

This project does not take into account things such as hardware performance or scalability, for reference I am using a 13 inch 2017 MacBook Pro with a 3.1GHz Duel-Core Intel Core i5, 8GB 2133 MHz LPDDR3 Memory and I don't think it's important but for graphics it is a Intel Iris Plus Graphics 650 1536 MB. It was also a bad time for my Matlab license to run out and as a result, I had to resort to using Matlab Online. I tried to do some research into what kind of system would it be executed on but I didn't seem to have much luck to find out what the hardware is but you could try reading through references 5 and 6 to see if you can see anything I missed. What I know is that it is ran on cloud-based environments and multiple things such as the current server load, network and how much resources your session is allocated. During execution of the Matlab script my internet speeds were 78.5 Mbps download and 44.4 Mbps upload to the Dublin server with a latency of 7ms on Google's speed test. Python 3.12.2 and NumPy 1.26.4 was used. Matlab Online on the day I was using it was version 24.1.0.2554230 (R2024a) and for C, I used the GCC Compiler.

Implementation

All code has extensive comments for what everything does, all the code will be pasted at the end of this document and also uploaded on canvas as well as screenshots for the outputs. Python code uses the NumPy library for matrix generation (numpy.random.rand) and matrix multiplication (numpy.dot). Matlab code uses the built in functions to generate random matrices (rand) and also for performing the matrix multiplication. C code did not have any built in libraries so functions were made to generate a random matrix and multiply those matrices. For timing in Python I used the time.perf_counter() as it's meant to be better from what I read. Matlab timing was done using the tic and toc functions and clock() was used in C. Each matrix size was 2x3 and 3x2 and was run 100,000 times to get a good average.

Results

Python:

Average execution time over 100000 runs: 0.001252 milliseconds

Matlab:

Average execution time over 100000 runs: 0.000302 milliseconds

C:
```
Average execution time over 100000 runs: 0.000277 milliseconds
```

From these results we can see that Python had the longest average time per operation, Matlab's execution time is significantly shorter than Python's and C has the shortest average execution time just ever so slightly outperforming Matlab.

Python's Performance:
Using NumPy, the performance is good but just nothing compared to Matlab or C. From research, this may be to the fact that it has to call NumPy's library functions which cannot match the speed of directly compiled C code. Despite the performance difference, the ease of use, nice syntax and portability of Python makes it one of the most used and rapidly growing languages.

Matlab's Performance:
I was actually very impressed at Matlab's performance due to me having to use Matlab Online and also it being considered a high level language. This possibly could be due to optimization for matrix operations over the years of development.

C's Performance:
I expected C's execution time to be the shortest due to it being a low level language. C programs are compiled to machine code which allows for optimization that is not possible with higher level languages.

Conclusion
This optional project allowed me to do a comparative analysis of matrix multiplication across Python, Matlab and C and revealed insightful differences in performance across these very popular languages. My findings indicate that while Python offers ease of use and flexibility (using NumPy), it's execution time was the slowest. Matlab's performance was surprisingly efficient and has great strengths overall when it comes to numerical computing where it likely benefited from years of optimization. C being the lowest level among the three and also the fastest was no surprise, although the code was definitely more complicated to write it still continues to be one of the best low level languages out there. Overall Matlab provides a great balance between speed and utility and is definitely the most ideal when it comes to numerical computing.

Code

Python

```python
# This python code demonstrates matrix multiplication.
# Wrote by Oisin Mc Laughlin (22441106) and finished on the 30/03/24.

# Step 1: Import libraries, initialise variables and create for loop
import numpy
import time

execution = 0
runs = 100000

for i in range(runs):
    # Step 2: Generate two random matrices of size 3x2 and 2x3
    A = numpy.random.rand(3,2)
    B = numpy.random.rand(2,3)

    # Step 3: Do the matrix multiplication and measure how long it takes to execute
    start = time.perf_counter()  # Start timer (used time perf since I've read it's more accurate)
    C = numpy.dot(A, B)  # Multiplication of matrix A and B and save as C
    end = time.perf_counter()     # End timer

    # Accumulate execution time
    execution += (end - start)

# Calculate the average execution time and convert to miliseconds, round to 6 places
avg_execution = round(execution / runs * 1000, 6)

# Step 4: Display the average execution time
print("\nAverage execution time over " + str(runs) + " runs: " + str(avg_execution) + " milliseconds\n")



# Old code

# # Step 2: Generate two random matrices of size 3x2 and 2x3
# A = numpy.random.rand(3,2)
# B = numpy.random.rand(2,3)

# # Step 3: Do the matrix multiplication and measure how long it takes to execute
# start = time.time() # Start timer

# C = numpy.dot(A, B) # Multiplication of matrix A and B and save as C

# end = time.time() # End timer

# # Step 4: Display results and how long it took
# print("\nMatrix A:")
# print(A)
# print("\nMatrix B:")
# print(B)
# print("\nResult of A x B:")
# print(C)

# execution = round((end - start) * 1000, 6) # Total time of execution in ms
# print("\nExecution time: " + str(execution) + " seconds")
```

Matlab

```
% This MATLAB code demonstrates matrix multiplication.
% Written by Oisin Mc Laughlin (22441106) and finished on the 30/03/24.

% Step 1: Initialize the sum of execution times
execution = 0;
runs = 100000;

for i = 1:runs
    % Step 2: Generate two random matrices of size 3x2 and 2x3
    A = rand(3, 2);
    B = rand(2, 3);

    % Step 3: Do the matrix multiplication and measure how long it takes to execute
    tic;  % Start timer
    C = A * B;  % Multiplication of matrix A and B and save as C
    execution = execution + toc;  % End timer and accumulate execution time
end

% Calculate the average execution time and convert to milliseconds, round to 6 places
avg_execution = round((execution / runs) * 1000, 6);

% Step 4: Display the average execution time
disp(['Average execution time over ', num2str(runs), ' runs: ', num2str(avg_execution), ' milliseconds']);
```

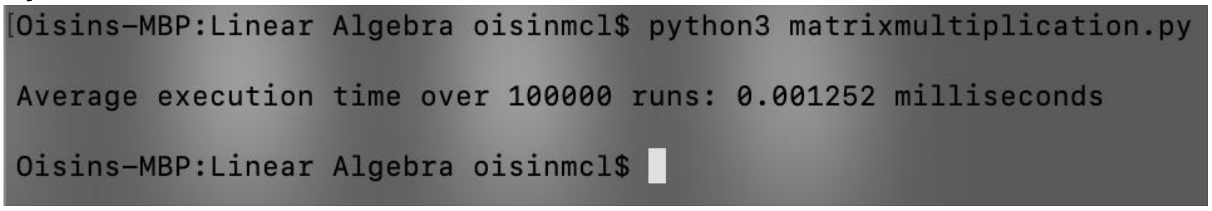
C

```
// This C code demonstrates matrix multiplication.
```

```c
// Written by Oisin Mc Laughlin (22441106) and finished on the 30/03/24.

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define the dimensions of the matrices A, B, and the number of runs for the experiment.
#define ROWS 3
#define COLS 2
#define COLS_B 3
#define RUNS 100000

// Function to generate a random matrix of a given size.
void generateRandomMatrix(double matrix[ROWS][COLS], int rows, int cols) {
    for(int i = 0; i < rows; i++) {
        for(int j = 0; j < cols; j++) {
            // Each element in the matrix is assigned a random double value between 0.0 and 1.0.
            matrix[i][j] = (double)rand() / RAND_MAX;
        }
    }
}

// Function to multiply two matrices A and B, storing the result in a third matrix C.
void multiplyMatrices(double A[ROWS][COLS], double B[COLS][COLS_B], double result[ROWS][COLS_B]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS_B; j++) {
            result[i][j] = 0.0;  // Initialize the result element to 0.
            for (int k = 0; k < COLS; k++) {
                // Accumulate the product of A's row and B's column elements in the result matrix.
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    srand(time(NULL)); // Initialize the random number generator.

    // Declare the matrices A, B, and C to hold the results of the multiplication.
    double A[ROWS][COLS], B[COLS][COLS_B], C[ROWS][COLS_B];
    // Start timing the experiment.
    clock_t start = clock();

    // Perform the matrix multiplication RUNS times to measure the average execution time.
    for(int i = 0; i < RUNS; i++) {
        generateRandomMatrix(A, ROWS, COLS);  // Generate random matrices A and B.
        generateRandomMatrix(B, COLS, COLS_B);
        multiplyMatrices(A, B, C);  // Multiply A and B, storing the result in C.
    }

    // Stop timing the experiment.
    clock_t end = clock();

    // Calculate the total time taken and convert it to milliseconds.
    double totalTime = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nAverage execution time over %d runs: %f milliseconds\n\n", RUNS, (totalTime / RUNS) * 1000);
}
```

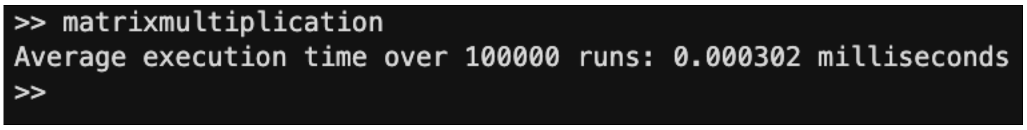
Outputs

Python



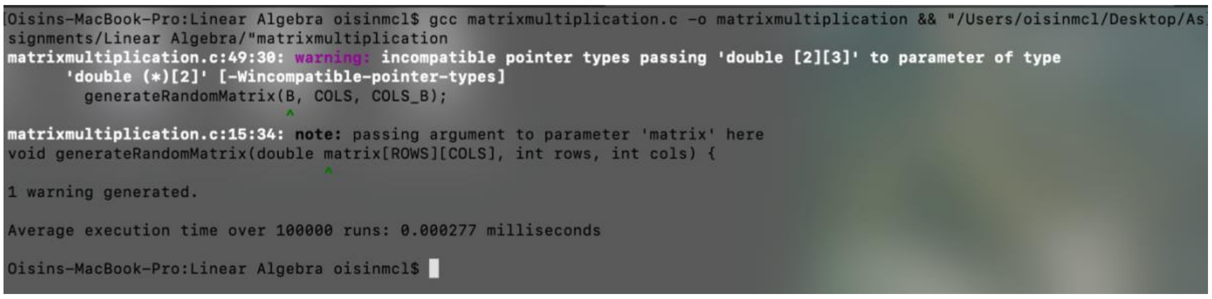
```
Oisins-MBP:Linear Algebra oisinmcl$ python3 matrixmultiplication.py

 Average execution time over 100000 runs: 0.001252 milliseconds

 Oisins-MBP:Linear Algebra oisinmcl$ 
```

Matlab



```
>> matrixmultiplication
Average execution time over 100000 runs: 0.000302 milliseconds
>>
```

C



```
Oisins-MacBook-Pro:Linear Algebra oisinmcl$ gcc matrixmultiplication.c -o matrixmultiplication && "/Users/oisinmcl/Desktop/As
signments/Linear Algebra/"matrixmultiplication
matrixmultiplication.c:49:30: warning: incompatible pointer types passing 'double [2][3]' to parameter of type
      'double (*)[2]' [-Wincompatible-pointer-types]
        generateRandomMatrix(B, COLS, COLS_B);
                             ^
matrixmultiplication.c:15:34: note: passing argument to parameter 'matrix' here
void generateRandomMatrix(double matrix[ROWS][COLS], int rows, int cols) {
                                 ^
1 warning generated.

Average execution time over 100000 runs: 0.000277 milliseconds

Oisins-MacBook-Pro:Linear Algebra oisinmcl$ 
```

Thank you.

References

1. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-matrix-data-structure/
2. https://news.mit.edu/2013/explained-matrices-1206
3. https://numpy.org/doc/stable/reference/generated/numpy.matmul.html
4. https://numpy.org/about/
5. https://uk.mathworks.com/matlabcentral/answers/371137-what-is-the-computation-power-of-matlab-online
6. https://uk.mathworks.com/help/matlab-web/matlab/matlab_env/matlab-online.html
7. https://superfastpython.com/time-time-vs-time-perf_counter/
8. https://stackoverflow.com/questions/3033329/why-are-python-programs-often-slower-than-the-equivalent-program-written-in-c-or
9. https://github.blog/2023-03-02-why-python-keeps-growing-explained/