Loc Nguyen

CS.4080

Professor Raheja

Project 1 – Matrix Operations in C, C++, and Java

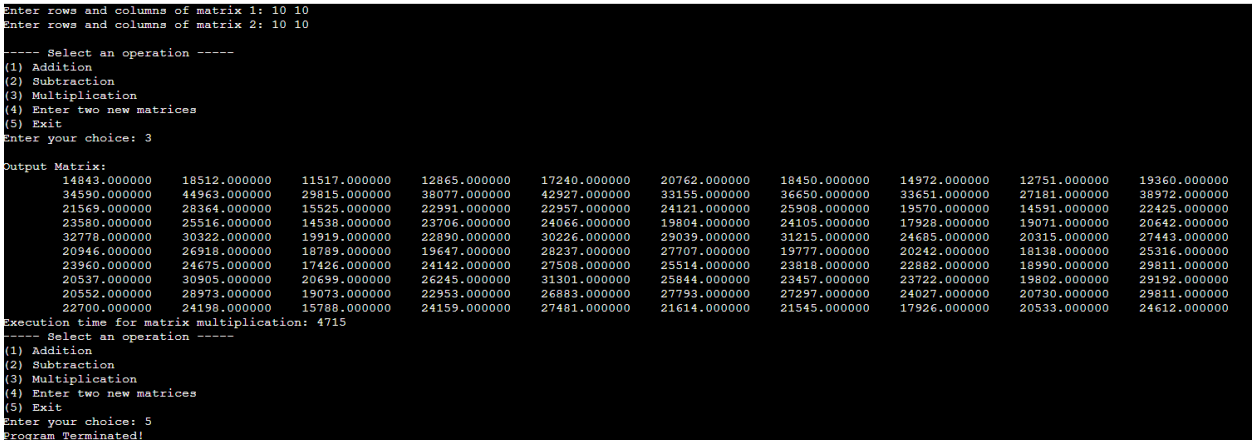
Methodology:

Matrices elements are populated with random float numbers starting from 1 to 100. Normally, users can select the size of matrices and the program will check for operation validity based on the two input matrices. However, square matrices size up to 100x100 are used to compute execution time for matrix multiplication in three programming languages. The goal is to compare execution time or speed of these programs for the purpose of optimization through language selection.

Experimentation:

- **C version**

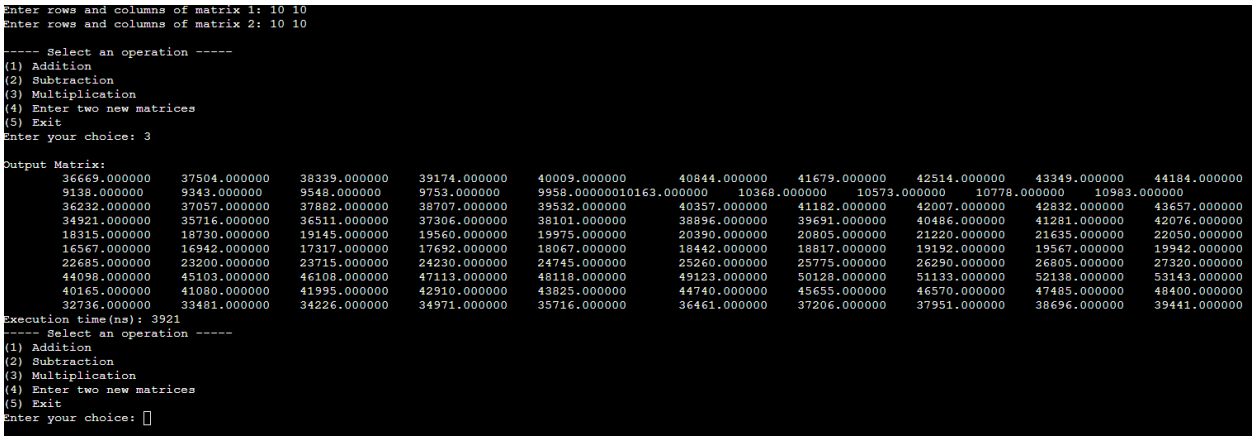
  This version uses a regular stack dynamic 2D array with a size limit of 100x100 to conduct addition, subtraction, and multiplication of matrices. Examples shown below:



```
Enter rows and columns of matrix 1: 10 10
Enter rows and columns of matrix 2: 10 10

----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 3

Output Matrix:
       14843.000000    18512.000000    11517.000000    12865.000000    17240.000000    20762.000000    18450.000000    14972.000000    12751.000000    19360.000000
       34590.000000    44963.000000    29815.000000    38077.000000    42927.000000    33155.000000    36650.000000    33651.000000    27181.000000    38972.000000
       21569.000000    28364.000000    15525.000000    22991.000000    22957.000000    24121.000000    25908.000000    19570.000000    14591.000000    22425.000000
       23580.000000    25516.000000    14538.000000    23706.000000    24066.000000    19804.000000    24105.000000    17928.000000    19071.000000    20642.000000
       32778.000000    30322.000000    19919.000000    22890.000000    30226.000000    29039.000000    31215.000000    24685.000000    20315.000000    27443.000000
       20946.000000    26918.000000    18789.000000    19647.000000    28237.000000    27707.000000    19777.000000    20242.000000    18138.000000    25316.000000
       23960.000000    24675.000000    17426.000000    24142.000000    27508.000000    25514.000000    23818.000000    22882.000000    18990.000000    29811.000000
       20537.000000    30905.000000    20699.000000    26245.000000    31301.000000    25844.000000    23457.000000    23722.000000    19802.000000    29192.000000
       20552.000000    28973.000000    19073.000000    22953.000000    26883.000000    27793.000000    27297.000000    24027.000000    20730.000000    29811.000000
       22700.000000    24198.000000    15788.000000    24159.000000    27481.000000    21614.000000    21545.000000    17926.000000    20533.000000    24612.000000
Execution time for matrix multiplication: 4715
----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 5
Program Terminated!
```

- **C version with pointers**

This version uses pointer to pointer that dynamically allocate memory using *malloc* to create the 2D matrices. Functions will take in pointers to the matrices to compute matrix addition, subtraction, and multiplication to return a pointer to a pointer of resultant matrix. Example shown below:



- **C++ version with pointers**

This version is similar to the C version using pointer to pointer to dynamically allocate memory using *new* to create 2D matrices. Private data class are rows, columns, and pointers to the Matrix class. Operations are overloaded with operator overload functions using +, -, and * for matrix addition, subtraction, and multiplication. Example shown below:

```
Enter rows and columns of matrix 1: 10 10
Enter rows and columns of matrix 2: 10 10

----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 3

Output Matrix:
        48.4544 45.4019 82.0025 33.3862 74.0199 19.6282 109.076 18.8073 11.9096 81.8934
        26.6786 126.842 98.3965 28.7655 65.8264 21.9186 151.415 19.1231 47.1694 30.7568
        55.036  118.763 255.629 31.5969 894.224 33.1995 93.2694 22.9034 54.3041 56.2197
        35.6483 43.2278 239.298 23.6554 65.1015 9.52437 33.6759 18.4888 15.7505 27.3769
        19.8431 22.8923 84.2732 12.8224 47.3461 3.54906 19.6363 9.85395 9.86539 16.4295
        13.9696 28.8117 78.5728 5.90051 41.6153 10.2051 31.1764 6.87895 13.3471 17.3725
        23.7948 31.3029 114.752 14.0459 161.564 9.92917 23.1372 12.5359 18.0553 21.5554
        22.9316 44.3668 59.1355 34.4328 83.7013 9.74558 64.3421 13.2894 18.0883 22.7789
        99.2963 109.113 164.834 64.4575 107.272 41.1672 205.789 40.0807 25.7886 177.911
        15.0323 42.5264 72.4052 29.4597 74.9486 7.82818 62.6648 10.1802 20.752  9.65332
Execution time(ns): 26000

----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 5
Program Terminated!
```

- **Java version with classes**

This version is similar to the C++ version but matrices are stack dynamic with no pointers. Functions are defined to carry out matrix addition, subtraction, and multiplication. Example shown below:

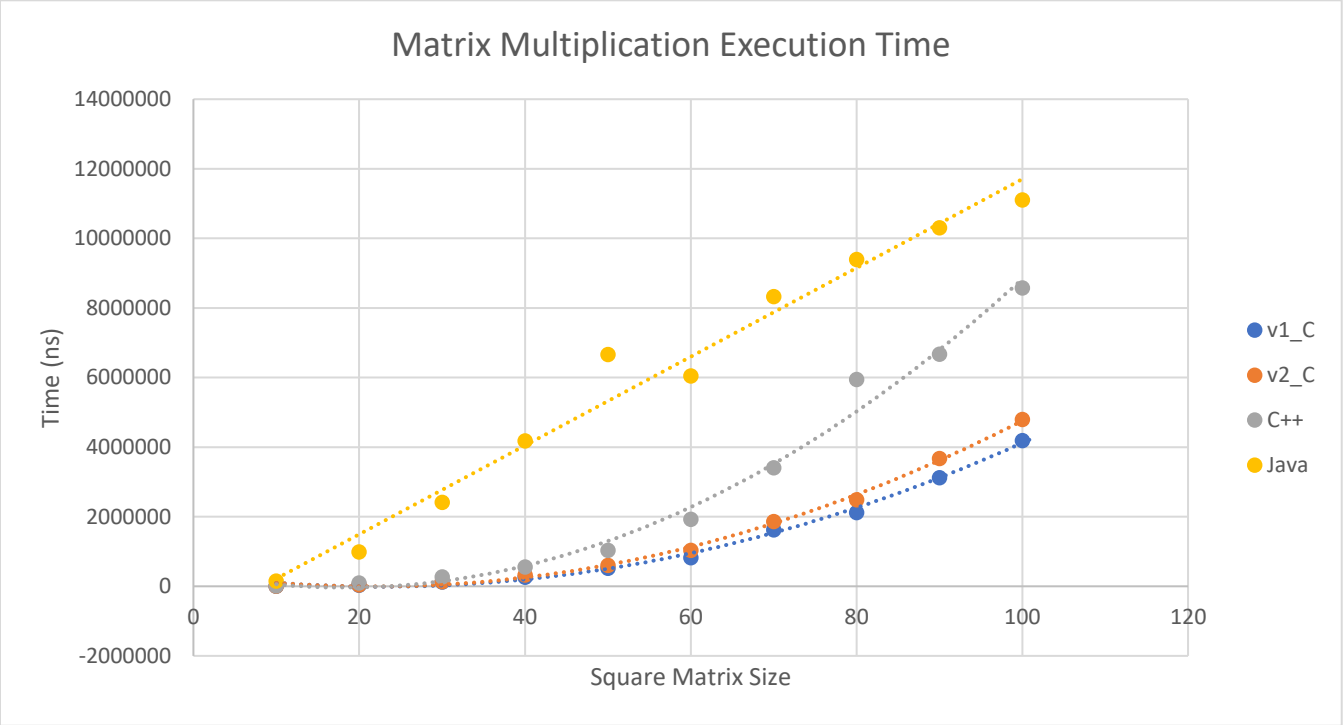
```
Enter rows and columns for matrix 1:
10 10
Enter rows and columns for matrix 2:
10 10

----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 3

Output Matrix:
        21747.635       34537.27        29469.207       50817.87        44348.22        38072.49
5206.637        29550.613       40519.133
        17537.016       25565.074       15429.108       34704.793       27173.162       22202.984
0954.629        18178.01        26013.223
        23171.756       30650.43        23500.613       44428.58        36921.438       32174.96
8654.992        20006.922       33842.598
        15811.596       24425.945       20793.055       34221.367       33010.973       29101.29
0979.64 19245.219       27197.203
        19796.738       27038.172       25118.264       41930.477       37446.387       33452.332
5149.086        19562.445       28603.037
        11664.187       22605.453       17770.062       29261.676       29089.719       26098.742
4768.34 15560.845       27299.96
        16674.512       18945.738       15657.428       27545.393       23838.918       18682.027
5632.719        15469.927       20795.777
        21892.086       33160.086       25436.027       40818.117       40184.81        36829.004
4683.79 18561.547       35444.273
        20017.475       26462.27        15230.642       33662.316       30500.898       24882.363
1794.367        16255.481       27761.14
        18763.564       27325.816       24796.986       35034.656       33401.29        25446.71
1967.281        24709.55        29302.525
Execution time(ns): 141400

----- Select an operation -----
(1) Addition
(2) Subtraction
(3) Multiplication
(4) Enter two new matrices
(5) Exit
Enter your choice: 5
Program terminated!
```

Testing:

I'm measuring execution time of multiplication operation in nanoseconds and using square matrices of 10x10, 20x20, 30x30, 40x40, 50x50, 60x60, 70x70, 80x80, 90x90, and 100x100.

Technical Specification: 16 GB RAM, 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz, 2803 Mhz, 4 Core(s), 8 Logical Processor(s)

I modified the original program to optimize testing by automating matrix size selection. All execution time are logged and charted below with Excel.



	Execution Time (ns)			
Matrix Size (nxn)	C Stack Dynamic	C Pointer to Pointer	C++ Classes with Pointer to Pointer	Java Classes
10x10	6100	6980	12760	155570
20x20	29850	39570	93860	993300
30x30	121530	134030	273440	2419000
40x40	258490	325470	556111	4177519
50x50	519190	606640	1038360	6663629
60x60	825310	1032270	1930560	6046159
70x70	1620220	1865950	3412349	8333260
80x80	2124120	2492500	5944710	9400500
90x90	3124950	3673519	6671419	10305378
100x100	4185989	4795690	8578209	11109520

Analysis:

The first two programs in C has the similar and fastest execution time when multiplying matrices. Stack dynamic is slightly faster than heap dynamic. The program in C++ boasted similar performance to C programs until matrix size is larger than 40, at which point the execution time gets much worse. The program in Java has the worst execution time but it's notable that the execution time growth is linear versus exponential in C and C++. This might be due to factors such as Java running on JVM while C and C++ are executables.

All programs theoretical performance could be optimized, especially with C++, when classes were used unnecessarily.

Conclusion and Improvements:

The methodology could be improved to consider variation on the host machine computer. Instead, we can take hundreds of trials of varying matrix sizes and select the median to decrease variability & noise while increasing "truthfulness" of the real execution time.