University of Puerto Rico - Mayaguez
Department of Electrical and Computer Engineering
ICOM5015 Inteligencia Artificial
Dr. J. Fernando Vega Riveros
21 de Febrero de 2025

Is Python fast or slow?

ALEJANDRO J RODRIGUEZ-BURGOS - ICOM Undergrad
Miguel A. Maldonado Maldonado - ICOM Undergrad

Purpose

In this assignment, we will be testing how fast python is as a programming language. This is due to the fact that python is widely used in the realm of Artificial Intelligence. We will be observing how we can use pre-existing libraries to optimize the time it takes for us to run algorithms.

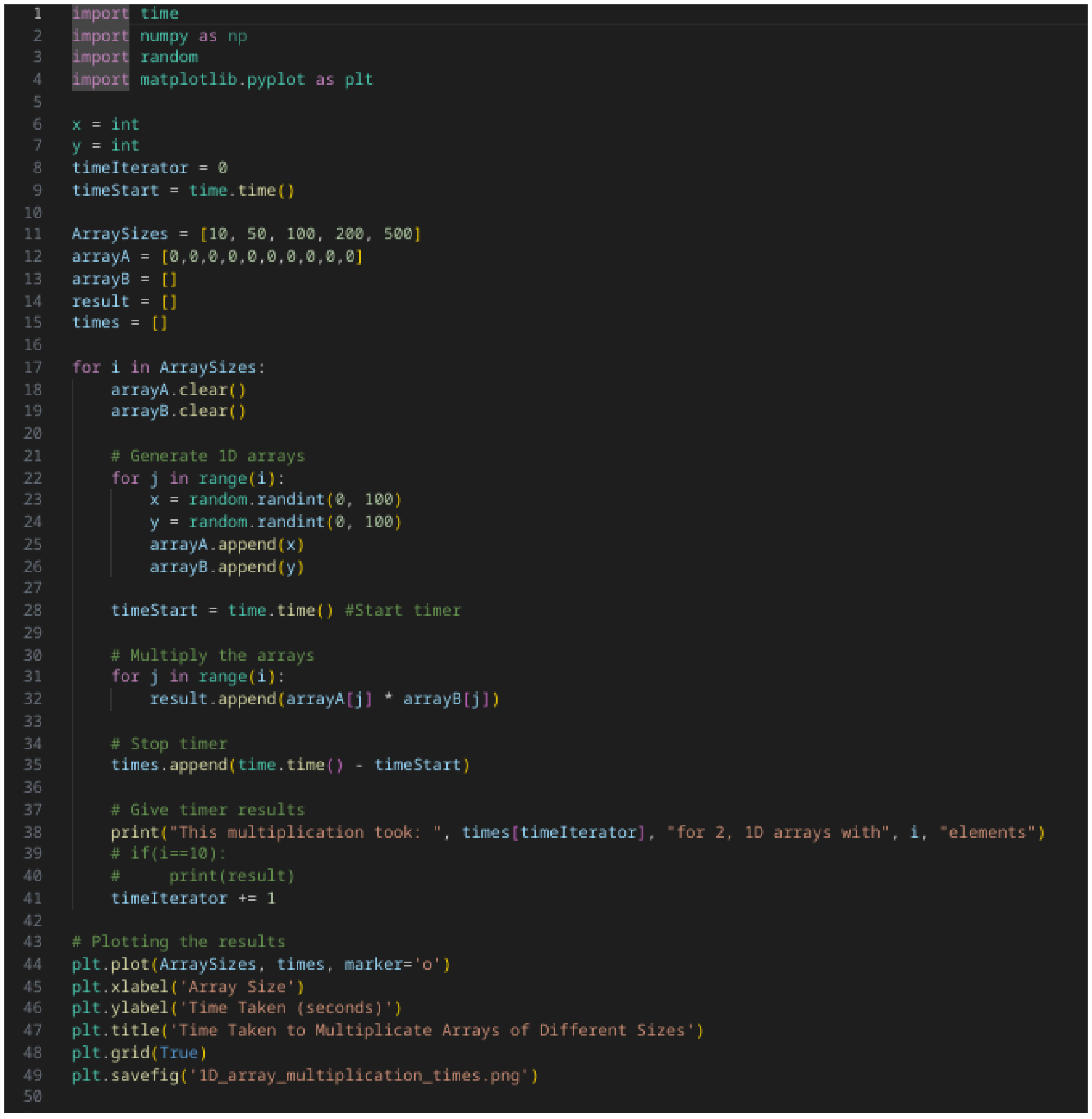
Concepts

The concepts we utilize in this assignment were:

- Array Dimensions
- Element wise - Matrix operations
- Time Complexity
- Memory Layout
- Nested for Loops

Information and Methods

We both utilized VSCode for this assignment. For the first part of the project, we needed to multiply two, one dimensional arrays using iterations. Without libraries, one way to approach this task is doing a nested for loop, where we iterate through the 5 different given sizes, (10, 50, 100, 200, 500), then do an inner loop for making the array and assigning it values. Then we started a timer using the time library and did another inner for loop to multiply the values of the arrays. At the end of that for loop, we ended the timer and plotted the amount of elements over the time it took to complete the task in a graph. For the 2-dimensional multiplication algorithm without NumPy, we used the same fundamental logic of nested for loops. So we create an outer loop to go through the given sizes, then we create an inner nested loop to create matrices, and another one to multiply them. We took the time it took to multiply the two matrices and plotted the elements over time.



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

x = int
y = int
timeIterator = 0
timeStart = time.time()

ArraySizes = [10, 50, 100, 200, 500]
arrayA = [0,0,0,0,0,0,0,0,0,0]
arrayB = []
result = []
times = []

for i in ArraySizes:
    arrayA.clear()
    arrayB.clear()

    # Generate 1D arrays
    for j in range(i):
        x = random.randint(0, 100)
        y = random.randint(0, 100)
        arrayA.append(x)
        arrayB.append(y)

    timeStart = time.time() #Start timer

    # Multiply the arrays
    for j in range(i):
        result.append(arrayA[j] * arrayB[j])

    # Stop timer
    times.append(time.time() - timeStart)

    # Give timer results
    print("This multiplication took: ", times[timeIterator], "for 2, 1D arrays with", i, "elements")
    # if(i==10):
    #     print(result)
    timeIterator += 1

# Plotting the results
plt.plot(ArraySizes, times, marker='o')
plt.xlabel('Array Size')
plt.ylabel('Time Taken (seconds)')
plt.title('Time Taken to Multiplicate Arrays of Different Sizes')
plt.grid(True)
plt.savefig('1D_array_multiplication_times.png')
```

Figure 1 - Code for array multiplication using iterators

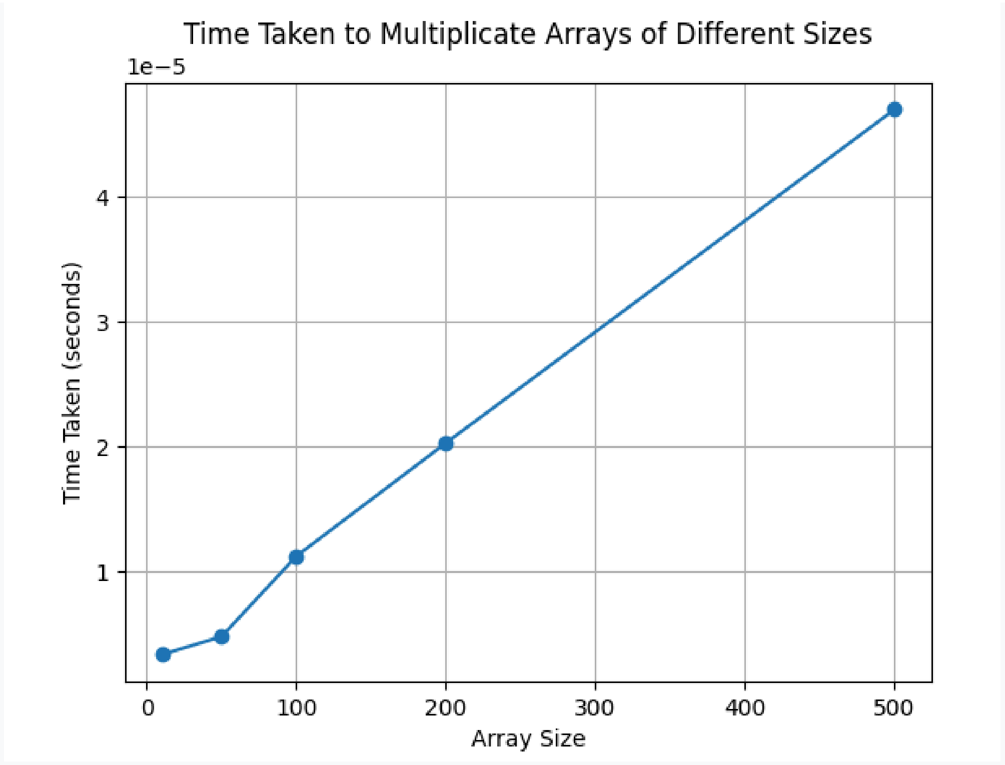
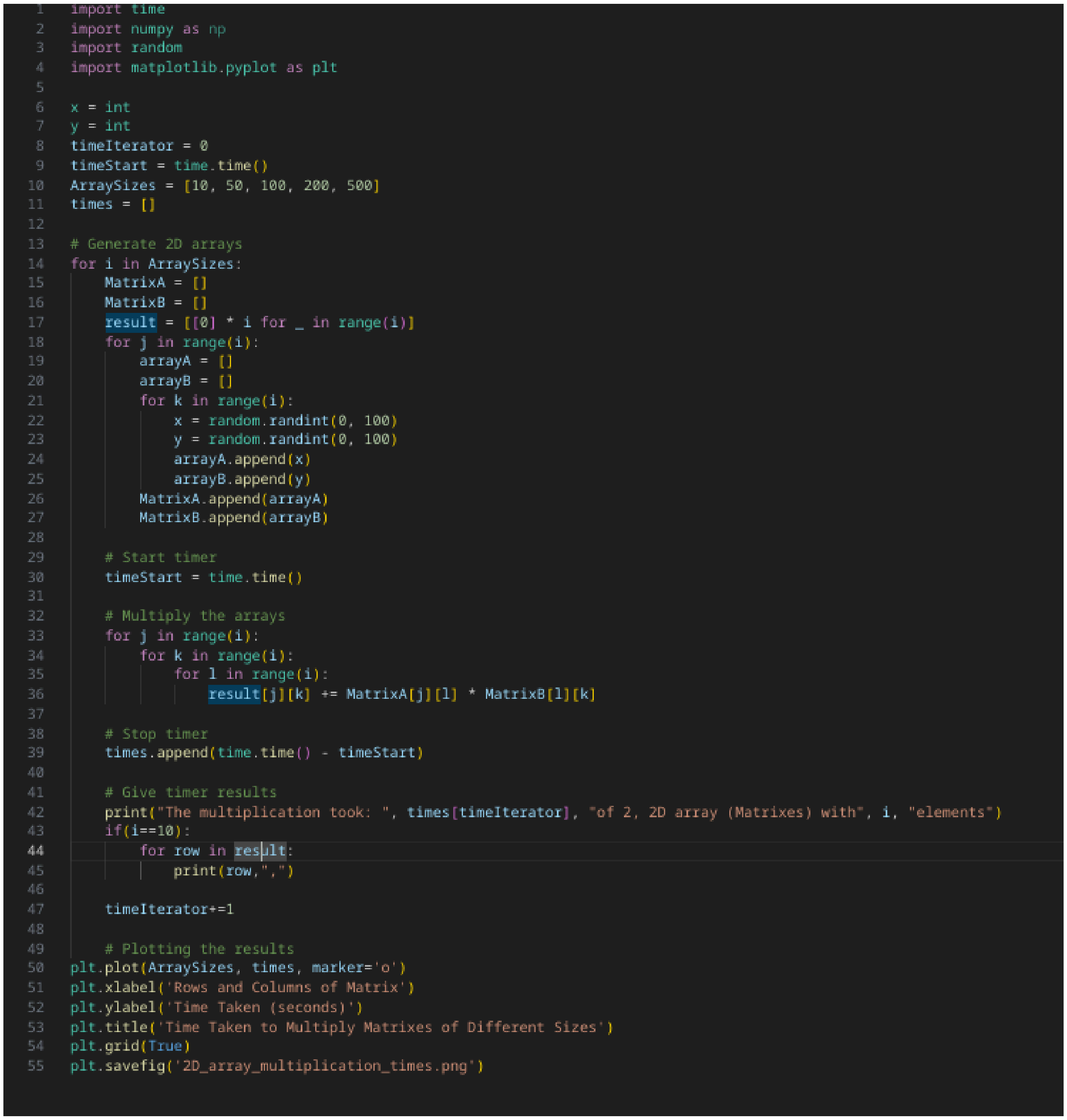


Figure 2 - Graph of Array Multiplication Using Iterators



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

x = int
y = int
timeIterator = 0
timeStart = time.time()
ArraySizes = [10, 50, 100, 200, 500]
times = []

# Generate 2D arrays
for i in ArraySizes:
    MatrixA = []
    MatrixB = []
    result = [[0] * i for _ in range(i)]
    for j in range(i):
        arrayA = []
        arrayB = []
        for k in range(i):
            x = random.randint(0, 100)
            y = random.randint(0, 100)
            arrayA.append(x)
            arrayB.append(y)
        MatrixA.append(arrayA)
        MatrixB.append(arrayB)

    # Start timer
    timeStart = time.time()

    # Multiply the arrays
    for j in range(i):
        for k in range(i):
            for l in range(i):
                result[j][k] += MatrixA[j][l] * MatrixB[l][k]

    # Stop timer
    times.append(time.time() - timeStart)

    # Give timer results
    print("The multiplication took: ", times[timeIterator], "of 2, 2D array (Matrixes) with", i, "elements")
    if(i==10):
        for row in result:
            print(row,",")

    timeIterator+=1

    # Plotting the results
plt.plot(ArraySizes, times, marker='o')
plt.xlabel('Rows and Columns of Matrix')
plt.ylabel('Time Taken (seconds)')
plt.title('Time Taken to Multiply Matrixes of Different Sizes')
plt.grid(True)
plt.savefig('2D_array_multiplication_times.png')
```

Figure 3 - Code for Matrix multiplication using iterators

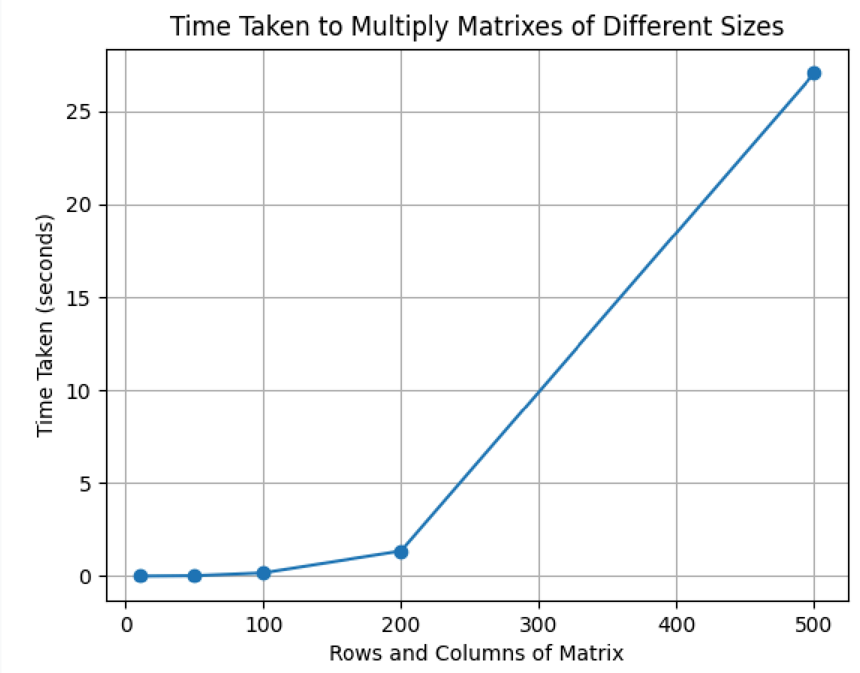
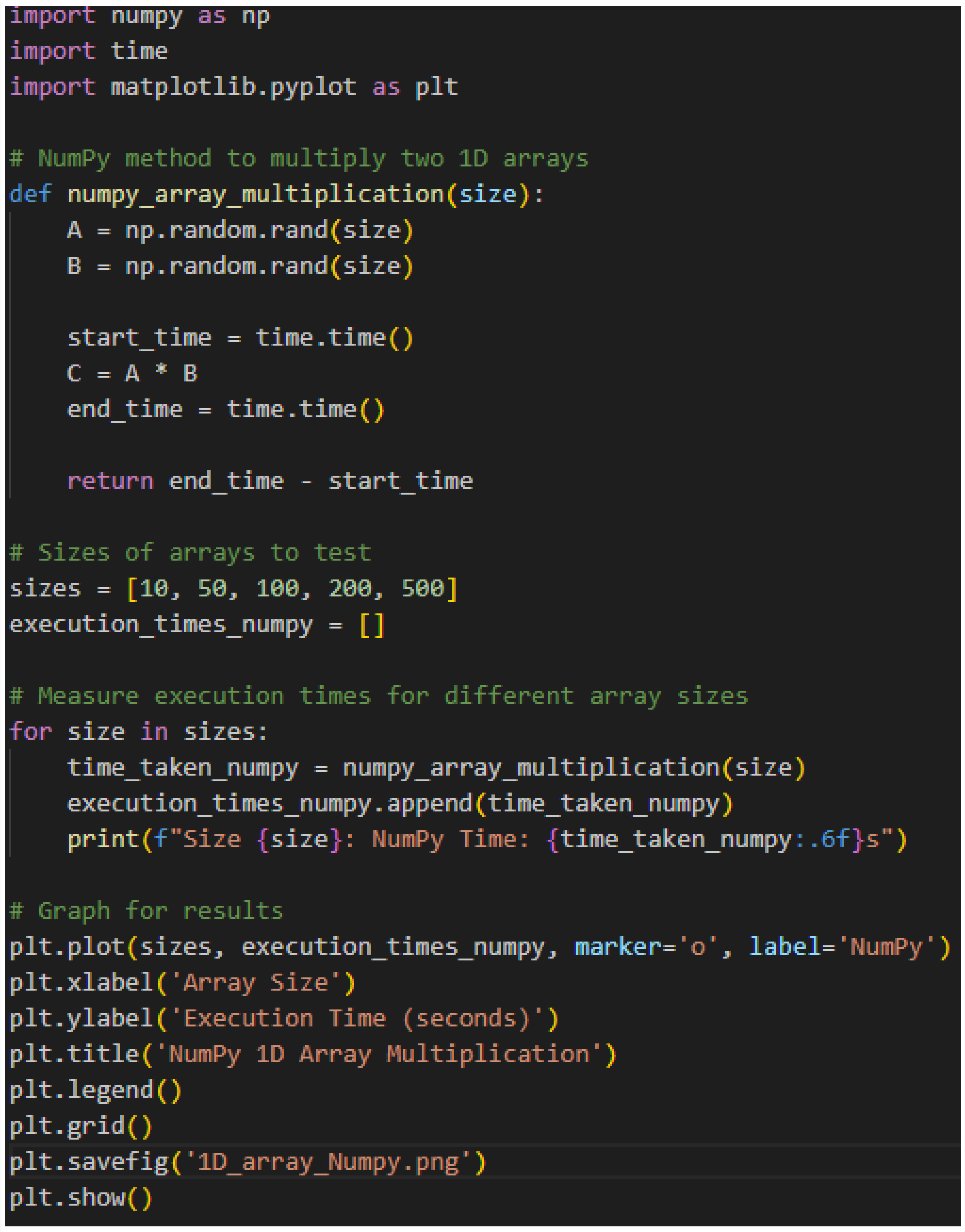


Figure 4 - Graph of Matrix multiplication using Iterators

For the NumPy part of this assignment, the multiplication of one dimensional arrays we just generated the arrays with NumPy and multiplied them as if we were multiplying variables, but NumPy actually goes through the elements on both arrays and multiplied them. Then we plotted our elements over time. We plotted this against what would be our python results, and we got much better times using NumPy above the 10 elements. Then, when doing the matrix multiplication, we generated the matrices using NumPy. Then we took the time it took for NumPy to multiply the matrices and plotted the elements over the time taken.



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

# NumPy method to multiply two 1D arrays
def numpy_array_multiplication(size):
    A = np.random.rand(size)
    B = np.random.rand(size)

    start_time = time.time()
    C = A * B
    end_time = time.time()

    return end_time - start_time

# Sizes of arrays to test
sizes = [10, 50, 100, 200, 500]
execution_times_numpy = []

# Measure execution times for different array sizes
for size in sizes:
    time_taken_numpy = numpy_array_multiplication(size)
    execution_times_numpy.append(time_taken_numpy)
    print(f"Size {size}: NumPy Time: {time_taken_numpy:.6f}s")

# Graph for results
plt.plot(sizes, execution_times_numpy, marker='o', label='NumPy')
plt.xlabel('Array Size')
plt.ylabel('Execution Time (seconds)')
plt.title('NumPy 1D Array Multiplication')
plt.legend()
plt.grid()
plt.savefig('1D_array_Numpy.png')
plt.show()
```

Figure 5 - Code for array multiplication NumPy

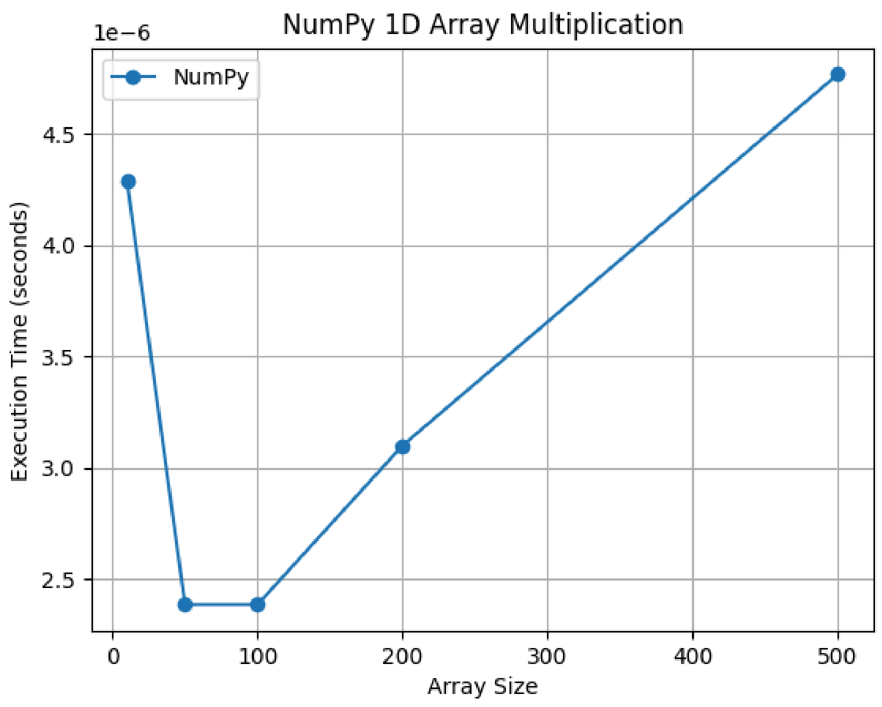
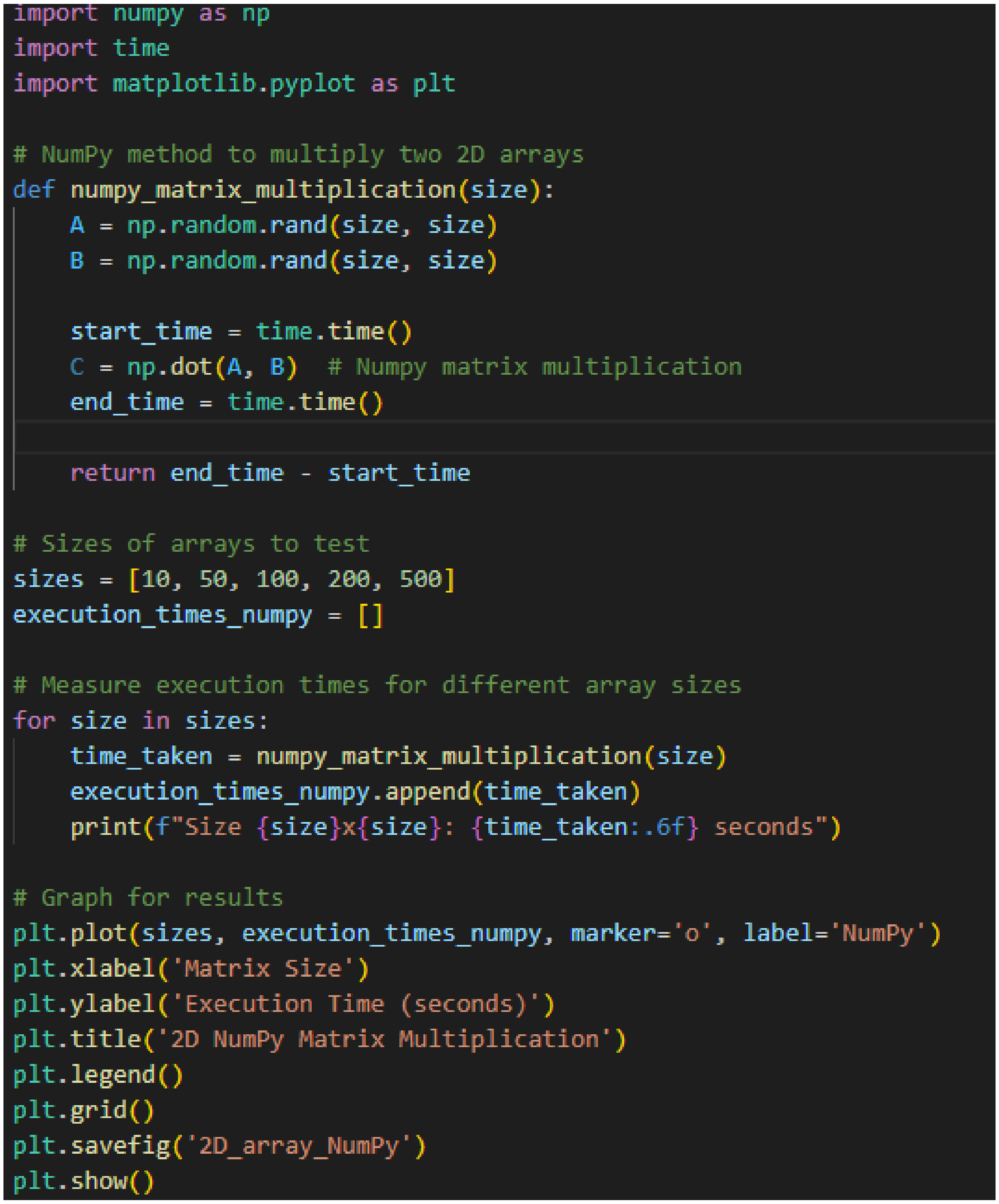


Figure 6 - Graph for array multiplication using NumPy



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

# NumPy method to multiply two 2D arrays
def numpy_matrix_multiplication(size):
    A = np.random.rand(size, size)
    B = np.random.rand(size, size)

    start_time = time.time()
    C = np.dot(A, B)  # Numpy matrix multiplication
    end_time = time.time()

    return end_time - start_time

# Sizes of arrays to test
sizes = [10, 50, 100, 200, 500]
execution_times_numpy = []

# Measure execution times for different array sizes
for size in sizes:
    time_taken = numpy_matrix_multiplication(size)
    execution_times_numpy.append(time_taken)
    print(f"Size {size}x{size}: {time_taken:.6f} seconds")

# Graph for results
plt.plot(sizes, execution_times_numpy, marker='o', label='NumPy')
plt.xlabel('Matrix Size')
plt.ylabel('Execution Time (seconds)')
plt.title('2D NumPy Matrix Multiplication')
plt.legend()
plt.grid()
plt.savefig('2D_array_NumPy')
plt.show()
```

Figure 7 - Code for Matrix multiplication using NumPy

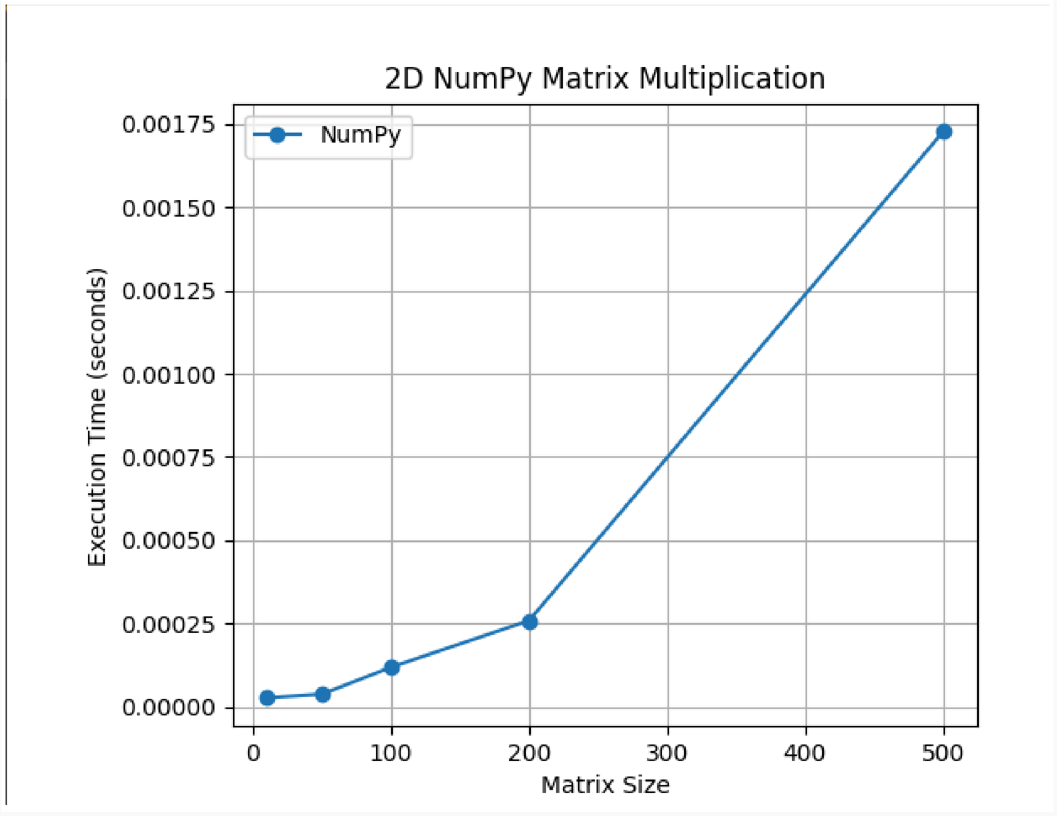


Figure 8 - Graph for Matrix multiplication using NumPy

Conclusion

We discover when visualizing the graph of plt.plot NumPy consistently outperformed Python list, especially as data size increased. This aligns with technical literature which explain that NumPy contiguous memory allocation and vectorized operations make it more efficient.

Based on the result, we conclude that NumPy significantly outperformed pure Python in both 1D and 2D array multiplication due to its optimized, low level operations. While 1AD multiplication with loops is relatively slow, NumPy element wise operations execute much faster. In 2D matrix multiplication, the naive triple loop from O(n^3) complexity, making it inefficient for large matrices, whereas the NumPy np.dot () function leverages optimized linear algebra libraries for superior performance. As arrays sizes grow, python loops become impractical, reinforcing the importance of using NumPy for efficient numerical computations.

Throughout this assignment, we learned that even if the fundamentals in coding of a programming language can be slow, we can always optimize code for better performance. In this case, using an optimized library, practically took operations with time complexity $O(n^2)$ to $O(\log n)$.

Work Distribution:

Numpy related code was done by Alejandro J Rodriguez Burgos
Iteration part of the code was done by Miguel A. Maldonado Maldonado
Report Prepared by Both
Slides Prepared by Both
Video Prepared by Both

Reference:

Python Software Foundation, "Python Documentation," [Online]. Available: https://docs.python.org/3/. [Accessed: Feb-21-2025].

NumPy Developers, "NumPy Documentation," [Online]. Available: https://numpy.org/doc/. [Accessed: Feb-21-2025].

J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://matplotlib.org/stable/contents.html. [Accessed: Feb-21-2025].