University of Puerto Rico Mayagüez Campus

Department of Engineering

Is Python fast or slow?

Víctor Y. Cruz Muñiz
Juan Díaz Díaz
Alex Strubbe Martinez
ICOM5015-001D Group E
Dr. José Fernando Vega Riveros
Assignment Report

Introduction

The purpose of this assignment is to visualize the importance of using Python libraries, specifically NumPy, for numerical computation tasks. We aim to compare the performance of array multiplication operations implemented in native Python versus those utilizing the NumPy library. The rapid advancement of computational technology has ushered in an era where numerical computations play a pivotal role in various scientific and engineering domains. Python, with its simplicity and versatility, has emerged as a preferred programming language for these applications. However, the performance of numerical operations in Python, particularly array manipulations, is a subject of ongoing scrutiny. This assignment aims to delve into this aspect by comparing the performance of array multiplication operations implemented in native Python with those utilizing NumPy, a widely acclaimed library for numerical computations in Python.

The key question driving this investigation is whether the utilization of NumPy significantly enhances the performance of array multiplication operations compared to their implementation in native Python. To address this, we designed an experiment focusing on the multiplication of 1D and 2D arrays of varying sizes, ranging from small arrays with dimensions less than 10 to larger arrays with dimensions in the hundreds. The execution times for these operations were meticulously recorded and analyzed to spot any differences in performance.

This report presents a comprehensive analysis of the experiment, guided by key concepts such as array operations, iteration, libraries, performance optimization, and memory management. By integrating insights from authoritative sources on Python and NumPy, we aim to provide a nuanced interpretation of the data obtained from our experiments. The findings from this investigation not only shed light on the performance implications of using NumPy but also offer valuable lessons on the importance of leveraging specialized libraries for numerical computations in Python. In this report, we all worked as a group, we all worked on the code, Juan Diaz worked on the errors and edited/fixed the code, Victor Cruz worked with the report and helped with the code, and Alex Strubbe worked on the presentation and edited the video.

Assignment's Purpose

The purpose of this assignment is to demonstrate the significance of using Python libraries, particularly in this case NumPy, for numerical computation tasks. The assignment entails conducting experiments to compare the performance of implementing array multiplication operations using basic Python (without libraries) versus utilizing NumPy, a powerful numerical computing library for Python.

Key question or Hypothesis

Hypothesis: Does utilizing the NumPy library significantly improve the performance of array multiplication operations compared to implementing them in native Python? Our hyphothesis is that utilizing libraries will affect the excecution time of our phython code.

Key concepts

The key concepts considered for the design of the experiment include array operations, iteration, the use of libraries, performance optimization, and memory management. Furthermore, considering dimensions of the arrays used in the mathematical operations. The experiment was designed to test the processing time for small arrays with dimensionalities less than 10, arrays with dimensionalities in the range of several tens, and arrays with dimensionalities in the range of several hundreds. The experiment was also designed to compare the processing time of mathematical operations on vectors like 1D arrays and a 2D arrays using native iteration and NumPy.

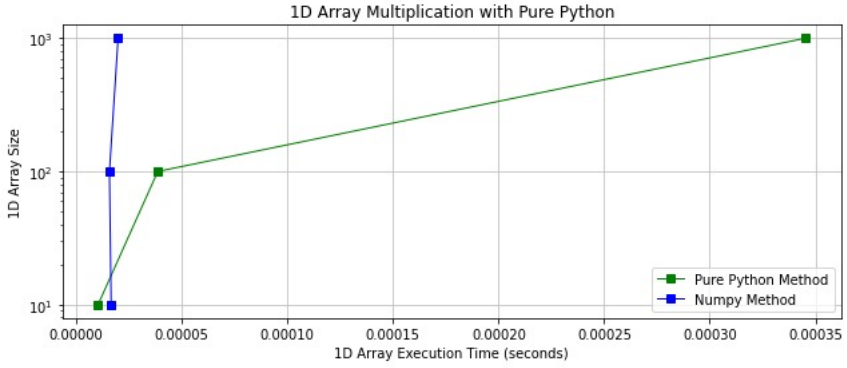
Experiment Set-up

The experiment was set up by creating 1-D and 2-D arrays of varying sizes (small arrays with dimensions less than 10, arrays with dimensions in the range of several tens, and arrays with dimensions in the range of several hundreds). Array multiplication operations were performed using both native Python and NumPy, and the execution time for each operation was recorded.

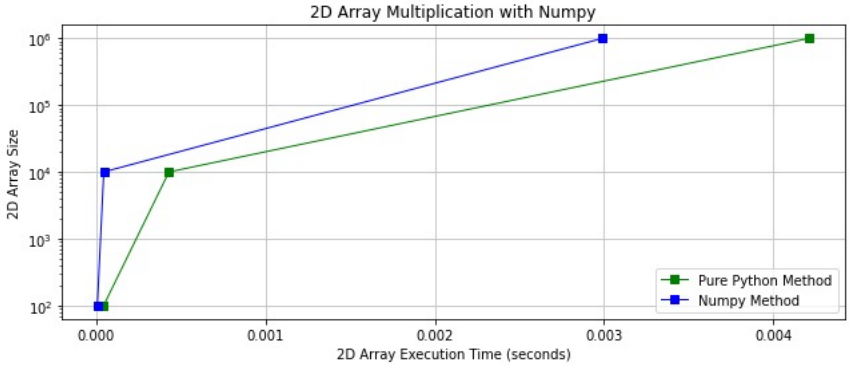
Analysis and Interpretation of Data Received and graphed.

The concepts identified, along with authoritative information sources, guided the analysis by providing a framework for understanding how NumPy optimizes array operations and manages memory more efficiently than native Python. This knowledge helped in interpreting the data, explaining the observed differences in execution times, and understanding the impact of array size and dimensionality on performance. The experiment results demonstrated that utilizing the NumPy library for array multiplication operations significantly improved processing time compared to Python iteration, especially for two-dimensional arrays. As the dimensions of the arrays increased, the processing time also escalated. However, the disparity in processing time between native Python and NumPy became more pronounced with larger arrays, showcasing NumPy's efficiency in handling complex numerical operations.When comparing the performance for one-dimensional arrays, the difference in processing time was less dramatic, but NumPy still outperformed Python iteration.

Graphs



1D Array Multiplication with Pure Python.



2D Array Multiplication with Numpy.

Code:

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

plt.close('all')

Y_1D = np.array([10, 100, 1000])
Y_2D = np.array([100, 10000, 1000000])
arrX_1D = np.zeros(3)
arrX_Numpy1D = np.zeros(3)
arrX_2D = np.zeros(3)
arrX_Numpy2D = np.zeros(3)

# 1D array of 10  ================================================================

start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []

counter = 0
for i in range(1, 11):       # This iteration creates the first array.
    counter += 2
    arr1.append(counter)

counter = 0
for i in range(1, 11):       # This iteration creates the second array.
    counter += 3
    arr2.append(counter)

for i in range(0,len(arr1)):        # This iteration multiplies the two arrays
  result.append(arr1[i] * arr2[i])  # to get a results array.

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_1D[0] = execution_time


# Numpy 1D array of 10 ===========================================================

start_time = time.perf_counter()

arr1 = np.arange(2, 21, 2)
arr2 = np.arange(3, 31, 3)

result = np.multiply(arr1, arr2)
```

```
end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy1D[0] = execution_time


# 1D array of 100 ============================================================
start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []

counter = 0
for i in range(1, 101):       # This iteration creates the first array.
    counter += 2
    arr1.append(counter)

counter = 0
for i in range(1, 101):       # This iteration creates the second array.
    counter += 3
    arr2.append(counter)

for i in range(0,len(arr1)):        # This iteration multiplies the two arrays
  result.append(arr1[i] * arr2[i])  # to get a results array.

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_1D[1] = execution_time


# Numpy 1D array of 100 ======================================================
start_time = time.perf_counter()

arr1 = np.arange(2, 201, 2)
arr2 = np.arange(3, 301, 3)

result = np.multiply(arr1, arr2)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy1D[1] = execution_time


# 1D array of 1000 ===========================================================
start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []

counter = 0
for i in range(1, 1001):       # This iteration creates the first array.
    counter += 2
    arr1.append(counter)

counter = 0
for i in range(1, 1001):       # This iteration creates the second array.
    counter += 3
    arr2.append(counter)

for i in range(0,len(arr1)):        # This iteration multiplies the two arrays
  result.append(arr1[i] * arr2[i])  # to get a results array.

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_1D[2] = execution_time


# Numpy 1D array of 1000 =====================================================
```

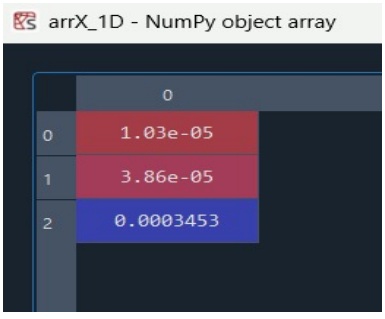
```
start_time = time.perf_counter()

arr1 = np.arange(2, 2001, 2)
arr2 = np.arange(3, 3001, 3)


result = np.multiply(arr1, arr2)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy1D[2] = execution_time


# 2D array of 10x10 ============================================================

start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []

counter = 0
for i in range(1, 11):          # This iteration creates the first array.
    row = []
    for j in range(1, 11):
        counter += 2
        row.append(counter)
    arr1.append(row)

counter = 0
for i in range(1, 11):          # This iteration creates the second array.
    row = []
    for j in range(1, 11):
        counter += 3
        row.append(counter)
    arr2.append(row)

for i in range(0,len(arr1)):    # This iteration multiplies the two arrays
    row = []                    # to get a results array.
    for j in range(0, len(arr1[i])):
        row.append(arr1[i][j]*arr2[i][j])
    result.append(row)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_2D[0] = execution_time


# Numpy 2D array of 10x10 ======================================================

start_time = time.perf_counter()

arr1 = np.arange(2, 201, 2).reshape(10,10)
arr2 = np.arange(3, 301, 3).reshape(10,10)

result = np.multiply(arr1, arr2)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy2D[0] = execution_time


# 2D array of 100x100 ==========================================================

start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []
```

```
counter = 0
for i in range(1, 101):         # This iteration creates the first array.
    row = []
    for j in range(1, 11):
        counter += 2
        row.append(counter)
    arr1.append(row)

counter = 0
for i in range(1, 101):         # This iteration creates the second array.
    row = []
    for j in range(1, 11):
        counter += 3
        row.append(counter)
    arr2.append(row)

for i in range(0,len(arr1)):    # This iteration multiplies the two arrays
    row = []                    # to get a results array.
    for j in range(0, len(arr1[i])):
        row.append(arr1[i][j]*arr2[i][j])
    result.append(row)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_2D[1] = execution_time


#2D array of 100x100 ============================================================

start_time = time.perf_counter()

arr1 = np.arange(2, 20001, 2).reshape(100,100)
arr2 = np.arange(3, 30001, 3).reshape(100,100)

result = np.multiply(arr1, arr2)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy2D[1] = execution_time


# 2D array of 1000x1000 ========================================================

start_time = time.perf_counter()

arr1 = []
arr2 = []
result = []

counter = 0
for i in range(1, 1001):          # This iteration creates the first array.
    row = []
    for j in range(1, 11):
        counter += 2
        row.append(counter)
    arr1.append(row)

counter = 0
for i in range(1, 1001):          # This iteration creates the second array.
    row = []
    for j in range(1, 11):
        counter += 3
        row.append(counter)
    arr2.append(row)

for i in range(0,len(arr1)):       # This iteration multiplies the two arrays
    row = []                      # to get a results array.
    for j in range(0, len(arr1[i])):
        row.append(arr1[i][j]*arr2[i][j])
    result.append(row)
```

```
end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_2D[2] = execution_time


# Numpy 2D array of 1000x1000 ================================================

start_time = time.perf_counter()

arr1 = np.arange(2, 2000001, 2).reshape(1000,1000)
arr2 = np.arange(3, 3000001, 3).reshape(1000,1000)

result = np.multiply(arr1, arr2)

end_time = time.perf_counter()
execution_time = end_time - start_time
arrX_Numpy2D[2] = execution_time

# ============================================================================
plt.figure()
plt.figure(figsize=(9,4))
plt.title('1D Array Multiplication with Pure Python')
plt.semilogy(arrX_1D, Y_1D, 's-', lw=1, color='green', label='Pure Python Method')
plt.semilogy(arrX_Numpy1D, Y_1D, 's-', lw=1, color='blue', label='Numpy Method')
plt.grid(color='silver')
plt.legend(loc='lower right')
plt.xlabel('1D Array Execution Time (seconds)')
plt.ylabel('1D Array Size')
plt.tight_layout()

plt.figure()
plt.figure(figsize=(9,4))
plt.title('2D Array Multiplication with Numpy')
plt.semilogy(arrX_2D, Y_2D, 's-', lw=1, color='green', label='Pure Python Method')
plt.semilogy(arrX_Numpy2D, Y_2D, 's-', lw=1, color='blue', label='Numpy Method')
plt.grid(color='silver')
plt.legend(loc='lower right')
plt.xlabel('2D Array Execution Time (seconds)')
plt.ylabel('2D Array Size')
plt.tight_layout()
```

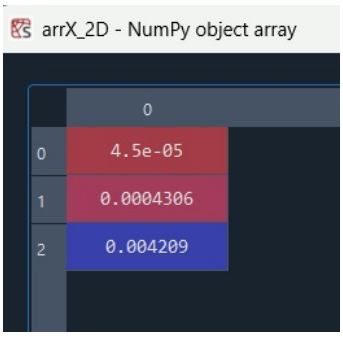
Link to access the Python code:
https://github.com/Juan-Dz/AIAssignment/blob/main/AIFullAssignment.py

Output Data

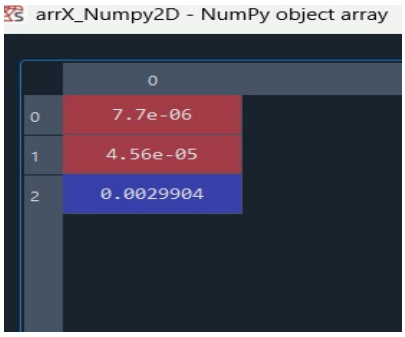


Execution Times de Pure Python 1D array multiplication of 10, 100, and 1000.

Execution times using Numpy en 1D array multiplication of 10, 100, and 1000.



Execution times de Pure Python 2D array multiplication of 100, 10,000, and 1,000,000.



Execution times de Numpy 2D array multiplication of 100, 10,000 and 1,000,000.

```
Pure Python Multiplication of 1D arrays with 10 values


Array_1:
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]


Array_2:
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]


Result of Multiplication of Array_1 with Array_2:
[6, 24, 54, 96, 150, 216, 294, 384, 486, 600]
```

10 1D array.

```
Numpy Multiplication of 1D arrays with 10 values


Array_1:
[ 2  4  6  8 10 12 14 16 18 20]


Array_2:
[ 3  6  9 12 15 18 21 24 27 30]


Result of Multiplication of Array_1 with Array_2:
[  6  24  54  96 150 216 294 384 486 600]
```

Numpy method 10 1D array.

```
Pure Python Multiplication of 1D arrays with 100 values


Array_1:
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62,
64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118,
120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166,
168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198, 200]


Array_2:
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93,
96, 99, 102, 105, 108, 111, 114, 117, 120, 123, 126, 129, 132, 135, 138, 141, 144, 147, 150, 153, 156, 159, 162, 165, 168,
171, 174, 177, 180, 183, 186, 189, 192, 195, 198, 201, 204, 207, 210, 213, 216, 219, 222, 225, 228, 231, 234, 237, 240,
243, 246, 249, 252, 255, 258, 261, 264, 267, 270, 273, 276, 279, 282, 285, 288, 291, 294, 297, 300]


Result of Multiplication of Array_1 with Array_2:
[6, 24, 54, 96, 150, 216, 294, 384, 486, 600, 726, 864, 1014, 1176, 1350, 1536, 1734, 1944, 2166, 2400, 2646, 2904, 3174,
3456, 3750, 4056, 4374, 4704, 5046, 5400, 5766, 6144, 6534, 6936, 7350, 7776, 8214, 8664, 9126, 9600, 10086, 10584, 11094,
11616, 12150, 12696, 13254, 13824, 14406, 15000, 15606, 16224, 16854, 17496, 18150, 18816, 19494, 20184, 20886, 21600,
22326, 23064, 23814, 24576, 25350, 26136, 26934, 27744, 28566, 29400, 30246, 31104, 31974, 32856, 33750, 34656, 35574,
36504, 37446, 38400, 39366, 40344, 41334, 42336, 43350, 44376, 45414, 46464, 47526, 48600, 49686, 50784, 51894, 53016,
54150, 55296, 56454, 57624, 58806, 60000]
```

Pure Python 100 1D array multiplication.

```
Numpy Multiplication of 1D arrays with 100 values


Array_1:
[  2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36
  38  40  42  44  46  48  50  52  54  56  58  60  62  64  66  68  70  72
  74  76  78  80  82  84  86  88  90  92  94  96  98 100 102 104 106 108
 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144
 146 148 150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180
 182 184 186 188 190 192 194 196 198 200]


Array_2:
[  3   6   9  12  15  18  21  24  27  30  33  36  39  42  45  48  51  54
  57  60  63  66  69  72  75  78  81  84  87  90  93  96  99 102 105 108
 111 114 117 120 123 126 129 132 135 138 141 144 147 150 153 156 159 162
 165 168 171 174 177 180 183 186 189 192 195 198 201 204 207 210 213 216
 219 222 225 228 231 234 237 240 243 246 249 252 255 258 261 264 267 270
 273 276 279 282 285 288 291 294 297 300]


Result of Multiplication of Array_1 with Array_2:
[    6    24    54    96   150   216   294   384   486   600   726   864
  1014  1176  1350  1536  1734  1944  2166  2400  2646  2904  3174  3456
  3750  4056  4374  4704  5046  5400  5766  6144  6534  6936  7350  7776
  8214  8664  9126  9600 10086 10584 11094 11616 12150 12696 13254 13824
 14406 15000 15606 16224 16854 17496 18150 18816 19494 20184 20886 21600
 22326 23064 23814 24576 25350 26136 26934 27744 28566 29400 30246 31104
 31974 32856 33750 34656 35574 36504 37446 38400 39366 40344 41334 42336
 43350 44376 45414 46464 47526 48600 49686 50784 51894 53016 54150 55296
 56454 57624 58806 60000]
```

Numpy multiplication of 100 1D array.

For the code not using libraries:

- Speeds: $1.03 \times 10^{-5}, 3.86 \times 10^{-5}, 0.0003453$

Average of speeds: $\frac{1.03\times10^{-5}+3.86\times10^{-5}+0.0003453}{3} = 0.0001292633$

For the code using libraries:

- Speeds: $1.65 \times 10^{-5}, 1.57 \times 10^{-5}, 1.98 \times 10^{-5}$

Average of speeds: $\frac{1.65\times10^{-5}+1.57\times10^{-5}+1.98\times10^{-5}}{3} = 0.00001667$

Now, plug these values into the percent difference formula:

$$\text{Percent Difference} = \left|\frac{0.0001292633 - 0.00001667}{(0.0001292633 + 0.00001667)/2}\right| \times 100$$
$$= \left|\frac{0.0001125933}{0.00007296665}\right| \times 100$$
$$= |1.5414| \times 100$$
$$= 154.14\%$$

So, the corrected percent difference between the speeds of the two codes is approximately 154.14%.

Calculations for 1-D arrays.

For the code not using libraries:

- Speeds: $4.5 \times 10^{-6}, 0.0004306, 0.004209$

Average of speeds: $\frac{4.5\times10^{-6}+0.0004306+0.004209}{3} = 0.0012140333$

For the code using libraries:

- Speeds: $7.7 \times 10^{-6}, 4.5 \times 10^{-5}, 0.0029904$

Average of speeds: $\frac{7.7\times10^{-6}+4.5\times10^{-5}+0.0029904}{3} = 0.0016636333$

Now, plug these values into the percent difference formula:

$$\text{Percent Difference} = \left|\frac{0.0012140333 - 0.0016636333}{(0.0012140333 + 0.0016636333)/2}\right| \times 100$$
$$= \left|\frac{-0.0004496}{0.0014388333}\right| \times 100$$
$$= |-0.3120672| \times 100$$
$$= 31.20672\%$$

So, the percent difference between the speeds of the two codes is approximately 31.21%.

Calculations for 2-D arrays.

Conclusion

In conclusion, our experiment clearly showed that using the NumPy library can greatly improve how efficiently and quickly our code runs. This is especially true when we compare it to times when we don't use the library. By using NumPy's powerful tools for working with arrays and doing calculations, we saw big improvements in how well and how fast our code worked. These results really underline how important it is to use special libraries like NumPy when we're doing calculations, especially in areas where being fast and efficient is really important for getting great results and being productive. This is super clear in areas like scientific computing, data analysis, and machine learning, where making things run faster and more efficiently is key.

The conclusion from this experiment is that utilizing the NumPy library significantly improves the performance of array multiplication operations compared to native Python, especially as the size and dimensionality of the arrays increase. This improvement is due to NumPy's optimized array operations and efficient memory management.From this assignment, we learned about the importance of using specialized libraries like NumPy for numerical computations in Python. We gained insights into how these libraries can optimize performance and manage memory more effectively than native Python, especially for large-scale array operations. This knowledge is valuable for developing efficient and scalable code in scientific computing, data analysis, and machine learning applications.

References

[1] J. Brownlee, “5 ways to measure execution time in Python,” Super Fast Python, https://superfastpython.com/benchmark-execution-time/#Measure_Execution_Time_With_timeperf_counter (accessed Feb. 27, 2024).

[2] K. Willems, “NumPy cheat sheet: Data Analysis in python,” DataCamp, https://www.datacamp.com/cheat-sheet/numpy-cheat-sheet-data-analysis-in-python (accessed Feb. 27, 2024).

[3] Github, “NumPy,” GitHub, https://github.com/numpy (accessed Feb. 27, 2024).

[4] N. Developers, “What is numpy?,” What is NumPy? - NumPy v1.26 Manual, https://numpy.org/doc/stable/user/whatisnumpy.html (accessed Feb. 27, 2024).

[5] S. Team, “Home - spyder ide,” Home - Spyder IDE, https://www.spyder-ide.org/ (accessed Feb. 27, 2024).

[6] P. Software, “Welcome to Python.org,” Python.org, https://www.python.org/ (accessed Feb. 28, 2024).