The exercises below require handling multi-dimensional data, complex calculations, and leverage NumPy's core strengths (vectorization, broadcasting, and optimized routines) against Python's native loop structures.

Advanced Performance & Linear Algebra Exercises

General Requirement: For each exercise, use the ***timeit*** module to measure and compare the execution time of the **Python List/Loop implementation** versus the **NumPy implementation**. Use matrix/array sizes large enough (e.g., N >= 10^3 for linear algebra, N >= 10^6 for basic array operations) to highlight the performance gap.

Code snippet for timeit

```
import timeit
import numpy as np

# 1. Setup Parameters
N = 5000000  # 5 million elements

# 2. Setup Data Structures (Code that runs once)
SETUP_CODE = f"""
import numpy as np
# Create lists and NumPy arrays with random data
list_a = [np.random.rand() for _ in range({N})]
list_b = [np.random.rand() for _ in range({N})]
np_a = np.array(list_a)
np_b = np.array(list_b)
"""

# 3. Define the Code Snippets to Time

# Implementation A: Python List Comprehension (Uses loops)
LIST_CODE = f'[list_a[i] + list_b[i] for i in range({N})]'
list_run = 5 # Number of times to run the statement

# Implementation B: NumPy Vectorized Operation (Optimized C code)
NUMPY_CODE = 'np_a + np_b'
numpy_run = 10 # NumPy is faster, so we run it more times for accuracy

# 4. Execute Time Measurements
print(f"--- Timing Element-wise Addition (N = {N}) ---")

# Measure Python List performance
list_time = timeit.timeit(
```

```
    stmt=LIST_CODE,
    setup=SETUP_CODE,
    number=list_run)

# Measure NumPy Array performance
numpy_time = timeit.timeit(
    stmt=NUMPY_CODE,
    setup=SETUP_CODE,
    number=numpy_run)

# 5. Output Results
print(f"1. Python List (Loop)      : {list_time /  list_run:.6f}
seconds (average of {list_run} runs)")
print(f"2. NumPy Array (Vectorized): {numpy_time/ numpy_run:.6f}
seconds (average of {numpy_run} runs)")

# 6. Speedup
if numpy_time > 0 and list_time > 0:
    speed_factor = (list_time / list_run) / (numpy_time / numpy_run)
    print(f"\t NumPy is approximately {speed_factor:.1f}x faster.")
```

Performance Comparison Exercises (Focusing on Vectorization)

1. Dot Product (Large Vector/Array)

Concept	**Vectorization, Linear Algebra (Inner Product)**
Task	Calculate the dot product of two large 1D vectors, ***A*** and ***B***, each of length $N=5 \times 10^6$.
Python List	Use a for loop and the accumulator pattern: ***sum(A[i] * B[i])*** for all ***i***.
NumPy Array	Use the highly optimised function: ***np.dot(A, B)***

2. Broadcasting Simulation

Concept	**Vectorization, Broadcasting**
Task	Given a large 2D matrix ***M*** (e.g., 1000 x 1000) and a 1D vector ***V*** (length 1000), add ***V*** to **every row** of ***M***.
Python List	Use nested for loops. The outer loop iterates through rows of ***M***, and the inner loop iterates through columns, adding ***V[j]*** to ***M[i][j]***.
NumPy Array	Use NumPy's **broadcasting** feature: ***M + V.***

3. Logarithmic Transformation and Sum (Chained Operations)

Concept	**Vectorization, Function Application, Chaining**
Task	Given a large array ***A*** of positive numbers (***N=10^6***), calculate ***B(i) = Log(sqrt(A(i) + 1))*** for every element ***i***, and then find the total sum of ***B***.
Python List	Use a list comprehension or for loop combined with functions from Python's built-in math module (***math.log, math.sqrt***).
NumPy Array	Use NumPy's vectorized mathematical functions: ***np.sum, np.log, np.sqrt***.

4. K-Nearest Neighbor (KNN) Distance Metric

Concept	**Vectorization, Distance Calculation, Root Mean Square**
Task	Calculate the Squared Euclidean Distance between one query vector ***Q*** and every

	data vector ***D(i)*** in a dataset ***D.*** The formula for the Squared Euclidean Distance between two vectors, ***Q*** and ***D(i)***, each having ***M*** features, is: ***Distance(Q,D(i)) = Sum_{j=1}^{m}(Q(j) - D(i,j))²***
Python List	Use a loop over all N data points. Inside the loop, use a second loop to calculate the element-wise squared difference and sum them up.
NumPy Array	Use a single vectorized operation: np.sum((D - Q)**2, axis=1).

Linear Algebra Algorithm Exercises (Focusing on NumPy Routines)

5. Matrix Multiplication

Concept	**Linear Algebra, Matrix Product**
Task	Multiply two large square matrices, ***A*** and ***B*** (e.g., N x N, with N=500). ***C = A * B***.
Python List	Implement the standard triple-nested for loop matrix multiplication algorithm ($O(N^3)$ complexity).
NumPy Array	Use the dedicated matrix multiplication operator.

6. Matrix Transposition and Subsetting

Concept	**Data Manipulation, Array Views**
Task	Given a matrix ***M*** (1000 x 500): 1) ***Transpose the matrix***. 2) ***Extract the first 100 rows and first 100 columns*** from the

	transposed result.
Python List	Implement transposition using nested loops or zip() (less common for large matrices). Implement subsetting via list slicing on the list of lists.
NumPy Array	Use the .T attribute for transposition and NumPy's advanced slicing: M.T[:100, :100].

7. Cumulative Sum and Conditional Reset

Concept	**Vectorization, Cumulative Operations, Conditional Logic**
Task	Given a large 1D array A of numbers (e.g., $N=2 \times 10^6$), calculate the running total (cumulative sum). However, if the current element A(i) is negative, the running total should **reset to zero** before adding A(i).
Python List	Use a for loop with an if condition and a state variable to track the running sum.
NumPy Array	This is a "scan" operation that is slightly more complex in NumPy (it breaks simple vectorization), but can be solved efficiently using specialized functions like np.cumsum combined with conditional masks, or using numba (advanced extension) or a highly optimized routine (if one exists). This highlights the limits of simple NumPy vectorization.

8. Histogram Calculation

Concept	**Binning, Data Aggregation**

Task	Given a large array A ($N=10^7$) of random data between 0 and 10, calculate the frequency of the data falling into 10 fixed-size bins (i.e., generate a histogram).
Python List	Use a for loop. Inside the loop, use an if/elif structure to check which bin the number falls into and increment a counter list/dictionary.
NumPy Array	Use the highly efficient and optimized function: counts, bins = np.histogram(A, bins=10).