Introduction

This project of MiniNumPy is to design and implement a simplified version of NumPy. In the project some methods of array creation, elementwise operations and linear algebra are designed. The design prioritizes clarity, maintainability, and scalability, allowing the system to be extended or optimized in future iterations.

Part 1: Core Array Class

1.1 Store attributes

Array Class automatically computes self.data, self.shape, self.ndim, self.size using len() function in Python, where shape describes the structure, ndim gives the number of nested list levels and size gives the total number of elements.

This step is essential as they attribute further operations like validating reshape operations, supporting matrix operations and so on.

1.2 Methods

The reshape(new_shape) method is designed to modify the structural layout of an array without altering the order of its elements. The implementation first flattens the original nested list into a one-dimensional list, then rebuilds a new nested structure according to new_shape. This “flatten → rebuild” strategy ensures that the total number of elements remains consistent while supporting arbitrary dimensions.

The purpose of transpose() is to perform a dimension (axis) swap. The method constructs a new nested list by interchanging row and column indices, effectively generating the mathematical transpose of a 2D array. This direct reconstruction approach avoids flattening and preserves the relative positions of elements along each axis, providing a clear and reliable implementation of matrix transposition.

The __str__() method aims to produce a clean and readable representation of the array. It displays metadata such as the array’s shape and formats the internal data with consistent indentation. This enhances the usability of the class, making debugging, testing, and presenting results significantly more intuitive.

Part 2: Array Creation

The array() function serves as a user-facing constructor that converts standard Python lists or nested lists into an Array object. Its primary role is to validate the input structure and delegate the construction to the Array class, which automatically infers attributes such as shape, ndim, and size.

The zeros() function generates a multi-dimensional array where all elements are initialized to zero. Because the target shape may consist of arbitrarily many dimensions, a recursive list-building algorithm is used. This approach is both natural and general: each dimension corresponds to one level of recursion, producing a properly nested list structure.

ones() uses the same underlying recursive construction mechanism as zeros(), differing only in the fill value. This uniform design makes the system easy to maintain and extend.

The eye(n) function constructs an n × n identity matrix, placing ones on the main diagonal and zeros elsewhere. The implementation uses simple nested loops to populate each row accordingly.

arange() creates a one-dimensional array containing a sequence of evenly spaced values. It uses an iterative loop to generate elements according to the provided start, stop, and step parameters. It supports both positive and negative step sizes ensures flexibility.

The linspace(start, stop, num) function complements arange by generating a specific number of evenly spaced samples over a specified interval. Unlike arange, which determines the number of elements based on a step size, linspace calculates the precise step required $step = \frac{stop-start}{num-1}$ to generate exactly num points. This implementation ensures that both the start and stop endpoints are included (closed interval). Special handling was implemented for the edge case where num=1 to prevent division-by-zero errors.

Part 3: Elementwise Operations

3.1 Overload Python operators

To provide an intuitive user interface consistent with standard mathematical notation, Python's built-in operators (e.g., +, -, *, /) were overloaded using magic methods such as __add__ and __mul__. This design choice allows users to perform arithmetic directly on Array objects without invoking verbose method calls, ensuring that the code remains readable and concise.

3.2 Elementwise Functions

The library implements essential mathematical transformations: exp, log, sqrt, and abs. A dedicated helper method, _unary_op, was designed to handle these operations. It accepts a mathematical function and recursively maps it to every element within the nested list structure, preserving the original shape.

To enhance robustness, functions sensitive to domain errors (log and sqrt) are wrapped with safety logic. Instead of raising immediate exceptions for invalid inputs, these wrappers return NaN and issue a runtime warning. This ensures that experimental workflows are not interrupted by isolated data anomalies.

3.3 Reductions

Statistical aggregation methods were implemented, including sum, mean, min, max, argmin, and argmax. To manage the complexity of multi-dimensional reductions, the implementation leverages the _flatten() helper function. By first converting the nested data structure into a linear one-dimensional list, the system can utilize Python's optimized built-in functions (like sum and max) directly.

Part 4: Linear Algebra Module

4.1 Matrix and Vector Operations

The dot function serves as a context-aware dispatcher: it computes the scalar inner product for 1D inputs and delegates 2D inputs to the matmul function. The matmul function implements the standard row-by-column multiplication.

The current implementation utilizes a triple-nested loop structure with a time complexity of $O(n^3)$. In numerical linear algebra theory, asymptotically faster algorithms exist, most notably Strassen's Algorithm. Strassen's method employs a divide-and-conquer strategy to reduce the recursive multiplications required for block matrices, achieving a complexity of approximately $O\left(n^{\log_2 7}\right) \approx 0(n^{2.81})$.

However, the standard $O(n^3)$ approach was chosen for MiniNumPy because in a pure Python environment, the recursion overhead and memory allocation required by Strassen's algorithm often outweigh its theoretical benefits for small-to-medium sized matrices. Theoretically, when the size of matrix n reaches over 100 or over 1000, the Strassen method will overspeed the standard method. But taken into account that most cases of matrix multiplication the size of matrix is much lower than 100, standard $O(n^3)$ approach was chosen.

4.2 Determinant

The traditional recursive approach to calculate the determinant of a matrix cost $O(n!)$. To improve, Gaussian Elimination method is introduced, which performs row operations to transform the input matrix into an Upper Triangular Matrix. Once transformed, the determinant is efficiently calculated as the product of the main diagonal elements.

To ensure numerical stability and prevent division-by-zero errors, Partial Pivoting is implemented. For each column, the algorithm swaps rows to move the element with the largest absolute value to the diagonal. A sign variable tracks these swaps, as each row swap reverses the sign of the determinant.

This implementation achieves a time complexity of $O(n^3)$, a significant improvement over the $O(n!)$ complexity of recursive expansion. This optimization allows the library to handle larger matrices with negligible runtime.

4.3 Inverse

The initial design considered the Adjugate Matrix method, but its complexity $O(n^5)$ proved impractical for matrices larger than 10×10. The final implementation utilizes Gauss-Jordan Elimination on an augmented matrix.

The process begins by constructing an augmented matrix [A|I] where the input matrix A is

concatenated with an identity matrix. The algorithm then iterates through each column, employing partial pivoting to ensure numerical stability by swapping rows to place the largest absolute value on the diagonal. Row operations are subsequently applied to normalize the pivot row and eliminate all other non-zero elements in the current column, effectively transforming the left side into the identity matrix. Once the transformation $[A|I] \rightarrow [I|A^{-1}]$ is complete, the right half of the augmented matrix is extracted as the computed inverse, reducing the computational complexity from $O(n^5)$ to a highly efficient $O(n^3)$.