**Scientific computing with Python**

**Chapter 4:**

**Linear Algebra – Arrays**

**Linear algebra** is a fundamental pillar of **computational mathematics**, playing a key role in fields such as data science, machine learning, and physics simulations.

Its core elements are **vectors** and **matrices**, which are used to represent data and perform transformations.

In Python, the **NumPy** library provides powerful and efficient tools for working with these mathematical objects. It allows users to easily manipulate vectors and matrices, perform operations on them, and solve complex linear algebra problems.

1. **Overview of the array type**

**NumPy** is already imported with:

**from numpy import \***

This gives you access to NumPy’s powerful **ndarray** data type, which we’ll explore in more detail in the next sections.

* **Vectors and matrices:** Creating vectors is as simple as using the function array to convert a list into an array:

**v = array([1.,2.,3.])**

Here are some illustrations of the basic linear algebra operations on vectors:

# two vectors with three components

v1 = array([1., 2., 3.])

v2 = array([2, 0, 1.])

# scalar multiplications/divisions

2\*v1 # array([2., 4., 6.])

v1/2 # array([0.5, 1., 1.5])

# linear combinations

3\*v1 # array([ 3., 6., 9.])

3\*v1 + 2\*v2 # array([ 7., 6., 11.])

# norm from numpy.linalg import norm

norm(v1) # 3.7416573867739413

# scalar product

dot(v1, v2) # 5.

v1 @ v2 # 5 ; alternative formulation

Many mathematical functions like **cos()** apply to each element:

**cos(v1) # array([0.5403, -0.4161, -0.9899])**

Matrices are created using lists of lists:

**M = array([[1., 2.], [0., 1.]])**

Even with the same data, a vector, row matrix, and column matrix are different objects:

v = array([1., 2., 1.])

R = v.reshape((1, 3)) # Row matrix → shape: (1, 3)

C = v.reshape((3, 1)) # Column matrix → shape: (3, 1)

* **Indexing and slices:** Indexing and slicing are similar to the corresponding operations for lists. The main difference is that there may be several indexes or slices when the array is a matrix.
* **Linear algebra operations:** The core function for performing most linear algebra operations in NumPy is dot().

dot(M, v)

# or equivalently:

M @ v

* **Solving a linear system :**

If A is a matrix and b is a vector, you can solve the linear system **Ax = b** using:

**from numpy.linalg import solve**

**x = solve(A, b)**

1. **Mathematical preliminaries**

To fully grasp how arrays work in NumPy, it's helpful to recognize the mathematical analogy between:

* Accessing elements of vectors or matrices (tensors) using indexes, and
* Evaluating functions by passing arguments.

This perspective clarifies how NumPy arrays operate internally and how indexing relates to mathematical function application.

* **Arrays as functions:** Arrays in NumPy can be understood from multiple perspectives. One particularly helpful approach is to think of arrays as functions of several variables.

This functional viewpoint will also be useful later when we discuss broadcasting.

A **vector** of size n can be thought of as a function:

f:{0,1,...,n−1}→R

In Python, the set {0,1,...,n−1}\{0, 1, ..., n-1\}{0,1,...,n−1} is generated using range(n).

A **matrix** can be seen as a function of **two variables** (row and column indices):

f:{0,1,...,m−1}×{0,1,...,n−1}→R

Selecting an element M[i, j] corresponds to evaluating this two-variable function at the point (i,j)(i, j)(i,j).

* **Operations are elementwise:** NumPy arrays can be naturally understood and treated as mathematical functions, especially when performing operations on them.

For example, consider two functions f and g defined on the same domain, both taking real values:

f,g:D→R

The **product** of these functions is defined **pointwise**:

(f⋅g)(x)=f(x)⋅g(x)for all x∈D

* **Shape and number of dimensions :** In NumPy, arrays can be understood as functions, where the number of arguments corresponds to the number of dimensions:

**Scalar** → Function with **no arguments**

**Vector** → Function with **one argument**

**Matrix** → Function with **two arguments**

**Higher-order tensor** → Function with **more than two arguments**

* **The dot operation:**

While treating **arrays as functions** is a powerful and intuitive perspective it **does not capture** the **linear algebra structure** we're often interested in.

This includes key operations such as:

**Matrix-vector multiplication**

**Matrix-matrix multiplication**

**Dot products**

**Linear transformations**

1. **The array Type :** In NumPy, the primary data structure used to represent and manipulate vectors, matrices, and higher-dimensional tensors is the ndarray (short for *N-dimensional array*), commonly referred to simply as an array.

* **Array properties:**

**Shape**: The shape attribute of a NumPy array defines how the data is structured order tensor.It also directly reflects the array’s dimensions. **array.shape**

**Dtype:** This gives the type of the underlying data (float, complex, integer, and so on).

**Strides:** The strides attribute of a NumPy array defines how many bytes must be skipped in memory to move: To the next element along each axis.

* **Creating arrays from lists :** The general way to create a NumPy array is using the array() function.

The general way to create an array is by using the function array. The syntax to create a real-valued vector would be:

V = array([1., 2., 1.], dtype=float)

To create a complex vector with the same data, you use:

V = array([1., 2., 1.], dtype=complex)float

1. **Accessing array entries**

In NumPy, matrix elements are accessed using a single pair of brackets with a comma-separated list of indices. This is different from accessing elements in a list of lists, where you'd use two sets of brackets.

**M = array([[1., 2.],[3., 4.]])**

**M[0, 0] # first row, first column: 1.**

**M[-1, 0] # last row, first column: 3**

* **Basic array slicing :** Slices are similar to those of lists except that they might now be in more than one dimension:

**M[i,:]** is a vector filled by the row of .

**M[:,j]** is a vector filled by the column of .

**M[2:4,:]** is a slice of 2:4 on the rows only.

**M[2:4,1:4]** is a slice of rows and columns

* **Altering an array using slices**: NumPy allows direct modification of array elements using slicing and indexing, including full rows, submatrices, and columns.

NumPy **distinguishes** between column matrices and 1D vectors during assignment:

Column matrix assignment : M[1:4, 2:3] = np.array([[1.], [0.], [-1.]])

Vector assignment : M[1:4, 2:3] = np.array([1., 0., -1.]) # Error!

1. **Functions to construct arrays**

* **zeros((n,m))** Matrix filled with zeros
* **ones((n,m))** Matrix filled with ones
* **diag(v,k)**  (Sub-, super-) diagonal matrix from a vector v
* **random.rand(n,m**) Matrix filled with uniformly distributed random numbers in (0,1)

1. **Accessing and changing the shape**

* **The function shape :** The shape of a matrix in NumPy is a tuple representing its dimensions: **(n,m)⇒n rows and m columns** While the .shape **attribute** is commonly used with NumPy arrays, the **shape() function** offers extra flexibility: it also works with **scalars**, **lists**, and **other sequences**, not just ndarray objects.
* **Number of dimensions :** The number of dimensions (also called rank) of a NumPy array is accessed using either:

The **ndim()** function

The array’s **.ndim** attribute

**from numpy import ndim, shape, zeros**

**A = zeros((2, 3))** # 2D array (matrix)

**print(ndim(A))** # Output: 2

**print(A.ndim)** # Output: 2

1. **Stacking**

The universal method to build matrices from a couple of (matching) submatrices is concatenate. Its syntax is: concatenate((a1, a2, ...), axis = 0)

**hstack:** Used to stack arrays horizontally

**vstack:** Used to stack arrays vertically

**columnstack:** Used to stack vectors in columns

1. **Functions acting on arrays**

There are different types of functions that operate on arrays. Some work **elementwise**—that is, they apply to each individual element and return an array with the **same shape** as the input. These are called **universal functions**, or **ufuncs**.

Other array functions may return an output with a **different shape**. In this section, we’ll explore both types of functions. We’ll also learn how to turn regular scalar functions into universal functions that can work elementwise on arrays.

* **Universal functions :** Universal functions are functions that operate elementwise on arrays. This means that they return an output array with the same shape as the input. Universal functions make it easy to apply a scalar operation to an entire array at once.

np.cos(np.pi) # Returns -1.0

np.cos(np.array([[0, np.pi/2, np.pi]])) # Returns array([[ 1., 0., -1.]])

**Creation of universal functions**

Your function will behave like a universal function automatically—as long as it only uses other universal functions internally. However, if your function relies on non-universal (scalar-only) functions or control structures like if, it may not behave as expected when applied to arrays.

vheaviside = np.vectorize(heaviside)

vheaviside(np.array([-1, 2])) # Returns array([0., 1.])

* **Array functions :** There are many array functions that do not operate elementwise. Instead, they perform computations across multiple elements of the array as a whole. Common examples include max, min, and sum.

These functions can operate in different ways:

On the entire array (default behavior)

Row-wise or column-wise, depending on the specified axis

If no axis argument is provided, the function applies to all elements in the array.

1. **Linear algebra methods in SciPy**

**SciPy** provides a wide range of **numerical linear algebra** tools through its **scipy.linalg** module. Many of these functions are **Python wrappers** around **LAPACK**—a well-established library of high-performance FORTRAN routines used for solving systems of linear equations, eigenvalue problems, and more.

Since SciPy uses these highly optimized compiled routines rather than pure Python code, its linear algebra operations are **extremely fast and reliable**, making them a core part of any scientific computing workflow.

In this section, we’ll explore two key linear algebra problems and demonstrate how SciPy can be used to solve them, giving you a sense of the module’s capabilities.

**sl.det** Determinant of a matrix

**sl.eig** Eigenvalues and eigenvectors of a matrix

**sl.inv** Matrix inverse

**sl.pinv** Matrix pseudoinverse

**sl.norm** Matrix or vector norm