Article Image

Introduction to Python for Numerical Linear Algebra

Table of Contents

Introduction

Numerical Linear Algebra (NLA) is the study of how matrix operations can be used to create computer algorithms which efficiently and accurately provide approximate answers to questions in continuous mathematics.

Consequently, it is mandatory to be able to efficiently implement matrix operations, i.e. operations regarding matrices (that we will represent with uppercase letters $A, B, \dots$) and vectors (that we will represent with lowercase letters $v, w, \dots$). The main library in Python implementing all the main NLA operations is `numpy`.

In this course, we will make massive use of `numpy`, together with its add-ons libraries, such as `scipy` and `pandas`.

`numpy` can be imported into Python by typing

```
import numpy as np
```

at the beginning of your code. If `numpy` is not installed on your Python environment, please follow numpy.org for informations on how to install it.

Documentation

At numpy.org it is possible to find a complete documentation of all the `numpy` functions with application examples.

Numpy and Numpy-Arrays

Creating a Numpy array

The basic object of `numpy` is the so-called `ndarray`, which defines the concept of vectors, matrices, tensors, … The simplest way to create a numpy array is to cast it from a Python `list` or `tuple`. This can be simply done as

```
a = [1, 2, 3]
a_vec = np.array(a)
```

producing a `numpy` array, `a_vec`. This can be checked by running the command

```
print(type(a_vec))
```

A basic propriety of a numpy array is the `shape`, representing its dimension. For example, a 5-dimensional vector $a = (1,2,3,4,5)^T$ will have shape $(5,)$, while a 3×3 matrix

$$A = \begin{bmatrix} 1 & 1 & -1 \\ 2 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix} \tag{1}$$

have `shape` $(3,3)$. A working example creating the matrix A defined above and checking its dimension is

```
A = [[1, 1, -1], [2, 0, 0], [0, 1, 2]]
A = np.array(A)

print(A.shape) # Use .shape to print the shape
```

Other functions to create arrays

In real applications, we will usually make use of huge matrices and vectos, with dimension that can easily reach a couple of millions. Clearly, it is not possible to define those kind of array by manually typing them and then converting them to numpy arrays with the `np.array` function. Luckily, this can be avoided when the array we need to create has a specific pattern. We will now list some functions we can use to simply create specific high-dimensional arrays.

- `np.linspace(a, b, n)`: Creates a vector of length `n`, containing `n` elements uniformely distributed in the interval $[a,b]$.
- `np.arange(start, end, step)`: Creates a vector containing all the integer numbers from `start` to `end-1`, skipping `step` numbers every time.
- `np.zeros((m, n))`: Creates an $m \times n$ matrix full of zeros. Clearly, to create a vector instead of a matrix, simply use `np.zeros((m, ))`.
- `np.ones((m, n))`: Creates an $m \times n$ matrix full of ones.
- `np.zeros_like(a)`: Creates an array full of zeros of the same shape of `a`. This is equivalent to `np.zeros(a.shape)`.
- `np.diag(v)`: Given a vector `v` of shape $(n,)$, returns an $n \times n$ diagonal matrix with `v` as diagonal.
- `np.random.randn(m, n)`: Creates an $m \times n$ matrix of normally distributed elements (i.e. sampled from $\mathcal{N}(0, I)$).

For example, if we want to create a vector of length 10 containing all the even numbers between 0 and 18, we can use

```
 # Create the vector
a = np.arange(0, 20, 2)

# Visualize the vector
print(a)
```

Exercise: Create an visualize the following matrix: $A = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$ Moreover, print its shape.

▶ Visualize the solution

Arrays operations

Now that we are able to create arrays, we need to understand how to use them. To simplify the implementation of NLA algorithms, the operations between numpy arrays basically follows the same syntax you can find in every math textbook. In particular, almost every operations is applied *element-wise*.

> A scalar operation between n-dimensional arrays a and b is said to be element-wise if it is applied to a and b element by element.

For example, if

$$a = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ 2 \\ 2 \end{bmatrix} \tag{2}$$

then, since

$$a + b = \begin{bmatrix} 1+0 \\ 0+2 \\ -1+2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \tag{3}$$

then we say that the $+$ operation is element-wise.

We'll list now the most important element-wise operations between arrays in numpy. When one of the two elements of the operators is a scalar number, it is threated as an array of the correct shape, where each element is equal to the number itself. In the following, we will use `a`, `b` to indicate generic arrays (vectors, matrices, …), `v`, `w` to indicate vectors and `A`, `B` to indicate matrices.

- `a + b`: Returns the element-wise sum of `a` and `b`. Requires the two arrays to have the same `shape`.
- `a - b`: Returns the element-wise difference of `a` and `b`. Requires the two arrays to have the same `shape`.
- `a * b`: Returns the element-wise multiplication of `a` and `b`. Requires the two arrays to have the same `shape`.
- `a / b`: Returns the element-wise division between `a` and `b`. Requires the two arrays to have the same `shape`.
- `a ** b`: Returns the element-wise exponentiation of `a` to the power of `b`. Requires the two arrays to have the same `shape`.
- `np.exp(a)`: Returns the element-wise result of e^a.
- `np.sin(a), np.cos(a), np.tan(a), np.log(a)`: Returns the corresponding element-wise operation applied to `a`.

Other than element-wise operations, important operators widely used in NLA are the matrix-by-vector product, the matrix-by-matrix product and the inner product between vectors. Since those operations are mathematically similar, numpy implements them in the same way:

- `a @ b`: Returns the matrix-by-matrix product between `a` and `b`. It requires the shapes of `a` and `b` to be compatible, e.g. shape of `a` (m, n), shape of `b` (n, k). The shape of the result is (m, k).

Clearly, when either `a` or `b` are vectors of the correct shape, then `@` returns the matrix-by-vector multiplication, while if both of them are vectors, then `a @ b` returns the inner product between the two vectors. The inner product can be equivalently written as `np.dot(a, b)`.

Example

To understand the basic operations between arrays, we will list an example code where we construct two vectors x_1, x_2 of the same dimension n and a matrix A of shape $n \times n$. Then, we compute $y_i = Ax_i$ for $i = 1, 2$. Finally, we check that $y_1 + y_2 = A(x_1 + x_2)$ i.e. we check the linearity of A.

```
import numpy as np

# Dimension of the problem
n = 10

# Create the vectors
x1 = np.linspace(0, 1, n)
x2 = np.random.randn(n)

# Create the matrix
A = np.random.randn(n, n)

# Compute y1 and y2
y1 = A @ x1
y2 = A @ x2

# Compute y = A(x1 + x2)
y = A @ (x1 + x2)

# Check the equality
print(y)
print(y1 + y2)
```

Exercise: Create two vectors x_1 and x_2 of dimension n and check that $e^{x_1}e^{x_2} = e^{x_1+x_2}$

▶ Visualize the solution

Logic operations between vectors

Clearly, it is also possible to define element-wise logical operations between arrays. The results will always be a boolean array of the same dimension of the input arrays, where the logic is applied element by element. Here we report a table of the main logic operations:

Operator	Meaning
==	EQUAL
!=	NOT EQUAL
>, >=	GREATER THAN
<, <=	LOWER THAN
&&	AND
||	OR
!	NOT

Slicing

An important operation we will often use in practice, is the so-called *slicing*. Slicing is extracting a portion of an array, indexed by a given index array. For example, consider

$$v = [0, 1, -1, 2, 1, -1]^T \tag{4}$$

and assume we want to extract the first three elements of v and assign them to a new vector w. This can be easily done by

```
# Create the array
v = np.array([0, 1, -1, 2, 1, -1])
```

```
# Slicing
w = v[0:3]
```

The notation `v[start:end]` returns the elements of `v` from `start` to `end-1`. When `start`=0 as in the example above, it can be emitted (e.g. `v[0:3]` is equivalent to `v[:3]`).

Slicing can also be performing by passing a numpy array of indices inside of the square brackets. For example, assume we want to extract the elements in even position of `v`. Then

```
# Create the array
v = np.array([0, 1, -1, 2, 1, -1])

# Slicing
idx = np.arange(0, len(v), 2)
w = v[idx]
```

does the job.

Finally, we can also slice by using boolean arrays. When this is the case, the elements in the position of the `True` values are returned. For example

```
# Create arrays
v = np.array([0, 1, -1, 2, 1, -1])
w = np.array([0, 0, -1, 1, 2, -1])

# Slicing
t = v[v == w]
```

is how we extract the elements that `v` and `w` have in common.

Slicing matrices

Slicing matrices works the same way as slicing vectors. The sole difference is that we need to use a 2-dimensional indexing array. For example, if

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{5}$$

and we want to extract the 2×2 principal submatrix of A (that is, the left upper-most 2×2 submatrix of A), then we can do

```
# Create the matrix
A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Slicing
B = A[:2, :2]
```

> ***Exercise:*** Create an $n \times n$ matrix A of normally distributed values. Then, create a second matrix B with the same shape of A such that it is equal to A when $a_{i,j}$ is positive, while it is equal to 0 when $a_{i,j} < 0$.

▶ Visualize the solution

Matrix and Vector Manipulation

Numpy also implements the basic operations on matrix and vectors. In particular, the following functions can be useful in this course:

- `np.linalg.norm(a, p)`: Computes the p-norm of a vector or a matrix a;
- `np.linalg.cond(A, p)`: Computes the condition number in p-norm of a matrix A;

- `np.linalg.matrix_rank(A)`: Computes the rank of the matrix A;
- `np.linalg.inv(A)`: When invertible, compute the inverse matrix of A. *Warning:* Very slow;
- `np.transpose(A)`: Compute the transpose matrix of A. It is equivalent to `A.T`;
- `np.reshape(a, new_shape)`: Reshape an array `a` to a given shape.

Read data with pandas

Since we will frequently work with data, it will be important to be able to manipulate them. In this class, we will learn how to load a dataset into Python by using a library called `pandas`, whose documentation can be found here.

As an example, download the data from Virtuale, which is taken by Kaggle at the following link: www.kaggle.com/mysarahmadbhat/us-births-2000-to-2014.

Then, place it in the same folder as the Python file on which you are working and use the following code to load it in memory.

```
import pandas as pd

# Read data from a csv file
data = pd.read_csv('./data/US_births_2000-2014_SSA.csv')
```

Pandas uses similar function name as numpy to keep everything coherent. For example, we can check the shape of `data` by using the function `print(data.shape)`. Moreover, a pandas dataframe can be casted into a numpy array by simply

```
import numpy as np

# Cast into numpy array
np_data = np.array(data)

# Check that the dimension didn't change
print(f"{data.shape} should be equal to {np_data.shape}")
```

Going on

An application of Numpy will be on Linear Systems. An introduction to plots in Python with matplotlib can be found here.

Written by

Davide Evangelista

Published 16 Sep 2022