# PHOT 110: Introduction to programming

# LECTURE 08: Arrays, vectors & linear algebra (Ch. 5)

Michaël Barbier, Spring semester (2023-2024)

# MORE ON ARRAYS

# ARRAY DEFINITION

- Arrays have a fixed length
- Array elements have the same type
- We will use arrays from the Numpy library
- Optimized for numerical calculations

```
1  # Load the Numpy library
2  import numpy as np
3
4  a = np.array([1, 2, 4, 6, 5, 9])
5  print(a)
```

```
[1 2 4 6 5 9]
```

# ARE ARRAYS FASTER THAN LISTS?

- Accessing array elements is easy & fast:
- Example: array “a” of N integers of size 64-bit
  - i$^{th}$ element is at memory-address: (address of a[0]) + $i \times 64$ bits
  - Copy/change M integers: memory chunk of $M \times 64$ bits

For a list:

- Size can **grow/shrink** & element **types vary** - Elements’ address is in a look-up-table
- i$^{th}$ element is at memory-address at i$^{th}$ table row
- Reading in memory/data chunks is difficult

# ARRAYS ARE FASTER !

**If you do not grow them !**

$$[a_1, .., a_N] \rightarrow [a_1, .., a_N, a_{N+1}]$$

- Extending an array of $N$ elements with an extra element:
  - Creates a new array of $N + 1$ elements
  - Copies the old array into the new
  - Puts the extra element value in the last position

# ARRAY INITIALIZATION BY CASTING

```
1  import numpy as np
2
3  a = np.array([1, 2, 4, 6, 5, 9])
4  print(a)
```

```
[1 2 4 6 5 9]
```

```
1  a = np.array([[6, 0], [7, 4], [2, 9]])
2  print(a)
```

```
[[6 0]
 [7 4]
 [2 9]]
```

# ARRAY PROPERTIES

- Shape of the array
- Array dimensions
- Type of the elements
  - Default: 64-bit floats
  - Other options: “uint32”: unsigned 32-bit integers
- Remember: all elements are of same type !

# GET ARRAY PROPERTIES

```
1 print(a)
```

```
[[6 0]
 [7 4]
 [2 9]]
```

Shape of an array $\longrightarrow$ (rows, columns):

```
1 print(a.shape)
```

```
(3, 2)
```

Type of the elements:

```
1 print(a.dtype)
```

```
int32
```

# ADAPT ARRAY PROPERTIES: ELEMENT TYPE

```
1 print(a)
2 print(f"Type of the elements is: {a.dtype}")
```

```
[[6 0]
 [7 4]
 [2 9]]
Type of the elements is: int32
```

## Change the type to complex 128-bit:

```
1 b = a.astype(np.complex128)
2 print(b)
3 print(f"Type of the elements is: {b.dtype}")
```

```
[[6.+0.j 0.+0.j]
 [7.+0.j 4.+0.j]
 [2.+0.j 9.+0.j]]
Type of the elements is: complex128
```

# ARRAY INITIALIZATION

```
1  a = np.zeros((4, 3))          # 2D array with zeros
2  print(a)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
1  a = np.ones(4)                # 1D array with ones
2  print(a)
```

```
[1. 1. 1. 1.]
```

# ARRAY INITIALIZATION

```
1  a = np.full((5, 3), 1.2)    # 2D array filled with a value
2  print(a)
```

```
[[1.2 1.2 1.2]
 [1.2 1.2 1.2]
 [1.2 1.2 1.2]
 [1.2 1.2 1.2]
 [1.2 1.2 1.2]]
```

```
1  a = np.full((3, 3), [0.2, 0.3, 1.7])    # 2D array with va
2  print(a)
```

```
[[0.2 0.3 1.7]
 [0.2 0.3 1.7]
 [0.2 0.3 1.7]]
```

# ARRAY INITIALIZATION: SIMILAR SHAPE

```
1  a = np.ones((2, 4))          # 2D array with zeros
2  print(a)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

```
1  a = np.zeros_like(a)          # 2D array with zeros
2  print(a)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

# ARRAY INITIALIZATION: SIMILAR SHAPE

```
1  b = np.full_like(a, 4.5)          # 2D array with zeros
2  print()
3  print(f"a = {a}")
4  print()
5  print(f"b = {b}")
```

```
a = [[0. 0. 0. 0.]
 [0. 0. 0. 0.]]

b = [[4.5 4.5 4.5 4.5]
 [4.5 4.5 4.5 4.5]]
```

# EMPTY ARRAY INITIALIZATION

```
1  a = np.empty((3, 2))        # empty is not empty !
2  print(a)
```

```
[[ 3.       0.34  ]
 [ 0.546   34.9   ]
 [ 5.3462   9.    ]]
```

**Fill it** before using !

```
1  nx, ny = a.shape
2  for i in range(nx):
3    for j in range(ny):
4      a[i, j] = i + 2*j
5  print(a)
```

```
[[0. 2.]
 [1. 3.]
 [2. 4.]]
```

# ARRAY INITIALIZATION: INTERVALS

```
1  a = np.arange(15)              # similar to range()
2  print(a)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

```
1  a = np.arange(5, 7, 0.3)   # decimal float step
2  print(a)
```

```
[5.  5.3 5.6 5.9 6.2 6.5 6.8]
```

```
1  a = np.linspace(0, 5, 11)   # linearly spaced interval
2  print(a)
```

```
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
```

# 2D DOMAINS/INTERVALS: MESHGRID

```
1  x = np.linspace(0, 5, 6)
2  y = 2 ** np.linspace(0, 3, 4)
3  xx, yy = np.meshgrid(x, y)
```

```
1  print(xx)
```

```
[[0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]]
```

```
1  print(yy)
```

```
[[1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2. 2.]
 [4. 4. 4. 4. 4. 4.]
 [8. 8. 8. 8. 8. 8.]]
```

For more on domains see the Numpy meshgrid howto.

# HIGHER DIMENSIONAL ARRAYS

- Multi-dimensional N-D arrays have N axes

```
1  tmp = np.array([[ 1,  2,  3],[ 4,  5,  6],[ 7,  8,  9]])
2  a = np.array([tmp, 2*tmp, 3*tmp])
3  print(a)
```

```
[[[ 1  2  3]
  [ 4  5  6]
  [ 7  8  9]]

 [[ 2  4  6]
  [ 8 10 12]
  [14 16 18]]

 [[ 3  6  9]
  [12 15 18]
  [21 24 27]]]
```

# ELEMENT SELECTION & ARRAY SLICING

- list element selection with `[i]`
- list slicing operator: `[start:end+1:step]`

```
1 a = np.array([[ 1,  2,  3],[ 4,  5,  6],[ 7,  8,  9]])
2 print(f"a[2, 0] = {a[2, 0]}")
3 print(f"a[0:2:2, :] = {a[0:2:2, :]}")
```

```
a[2, 0] = 7
a[0:2:2, :] = [[1 2 3]]
```

Reduce single dimensions:

```
1 print(f"Squeezed a[0:2:2, :] = {np.squeeze(a[0:2:2, :])}"
```

```
Squeezed a[0:2:2, :] = [1 2 3]
```

# ADAPTING THE ARRAY SHAPE

```
1  a = np.array([[1,5,6,2],[5,3,8,9],[1,1,1,0]])
2  print(a)
```

```
[[1 5 6 2]
 [5 3 8 9]
 [1 1 1 0]]
```

## Flattening the array ND $\longrightarrow$ 1D

```
1  print(a.flatten())
```

```
[1 5 6 2 5 3 8 9 1 1 1 0]
```

## Reshaping array

```
1  print(a.reshape((2,6)))
```

```
[[1 5 6 2 5 3]
 [8 9 1 1 1 0]]
```

# ARRAY ARITHMETIC OPERATIONS

- similar to **arithmetic** operations on numbers
- element-wise
- shapes need to be compatible

```
1  a = np.array([2.5, 3, 4])
2  b = np.array([1, 0.1, 10])
3  print(f"{a} + {b} = {a + b}")
4  print(f"{a} * {b} = {a * b}")
5  print(f"{a} / {b} = {a / b}")
```

```
[2.5 3.  4. ] + [ 1.   0.1 10. ] = [ 3.5  3.1 14. ]
[2.5 3.  4. ] * [ 1.   0.1 10. ] = [ 2.5  0.3 40. ]
[2.5 3.  4. ] / [ 1.   0.1 10. ] = [ 2.5 30.   0.4]
```

# ARRAY LOGIC OPERATIONS

- similar to **logic** operations on (boolean) numbers
- element-wise
- shapes need to be compatible

```
1  a = np.array([1, 3, 4])
2  b = np.array([1, 1, 10])
3  print(f"{a} > {b} = {a > b}")
4  print(f"{a} == {b} = {a == b}")
```

```
[1 3 4] > [ 1  1 10] = [False  True False]
[1 3 4] == [ 1  1 10] = [ True False False]
```

# ARRAY MATHEMATICAL FUNCTIONS

- Apply mathematical functions similar to math library
- Numpy functions, for example: `math.sin()` → `np.sin()`

```
1 print(f"sin({a}) = {np.sin(a)}")
```

```
sin([1 3 4]) = [ 0.84147098  0.14112001 -0.7568025 ]
```

```
1 print(f"sqrt({a}) = {np.sqrt(a)}")
```

```
sqrt([1 3 4]) = [1.         1.73205081 2.        ]
```

```
1 print(f"exp({a}) = {np.exp(a)}")
```

```
exp([1 3 4]) = [ 2.71828183 20.08553692 54.59815003]
```

# LINEAR ALGEBRA: VECTORS

# VECTORS

A vector is a 1D array, e.g. a column or a row vector:

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_1 & b_2 & \dots & b_N \end{pmatrix}$$

A column vector:

```
1  a = np.array([[1],[3],[5]])
2  print(a)
```

```
[[1]
 [3]
 [5]]
```

# VECTORS

A vector is a 1D array, e.g. a column or a row vector:

$$\vec{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_M \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_1 & b_2 & \dots & b_N \end{pmatrix}$$

A row vector:

```
1 b = np.array([2, 4, 6, 8])
2 print(b)
```

```
[2 4 6 8]
```

# THE SCALAR PRODUCT (DOT-PRODUCT)

The scalar product $\vec{a} \cdot \vec{b}$ is defined as **scalar** value

$$\vec{a} \cdot \vec{b} = |a||b| \cos(\theta)$$

with $\theta$ the angle between $a$ and $b$

**Alternative definition** of the scalar product $\vec{a} \cdot \vec{b}$:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{d} a_i \, b_i$$

where $d$ is the dimension: 2D, 3D, …, ND

# THE SCALAR PRODUCT (DOT-PRODUCT)

Scalar product $\vec{a} \cdot \vec{b}$ can be computed in Numpy using `np.dot`:

```
1  a = np.array([1, -1])
2  b = np.array([1.5, 2.5])
3  s = np.dot(a, b)
4  print(f"{a} . {b} = {s}")
```

```
[ 1 -1] . [1.5 2.5] = -1.0
```

Use `np.vdot` for complex-valued vectors, calculates $\vec{a}^* \cdot \vec{b}$:

```
1  a = np.array([-1 + 1j, 1 - 1j])
2  b = np.array([0, 2j])
3  print(f"{np.conj(a)} . {b} = {np.vdot(a, b)}")
```

```
[-1.-1.j  1.+1.j] . [0.+0.j 0.+2.j] = (-2+2j)
```

# THE SCALAR PRODUCT (DOT-PRODUCT)

Scalar product $\vec{a} \cdot \vec{b}$ can be computed in Numpy using `np.dot`:

```
1  a = np.array([1, -1])
2  b = np.array([1.5, 2.5])
3  s = np.dot(a, b)
4  print(f"{a} . {b} = {s}")
```

```
[ 1 -1] . [1.5 2.5] = -1.0
```

Higher dimensions than 3 possible:

```
1  a = np.array([-1, 2, 2, 2, 1])
2  b = np.array([0, 0, -3, 3, 2])
3  print(f"{a} . {b} = {np.dot(a, b)}")
```

```
[-1  2  2  2  1] . [ 0  0 -3  3  2] = 2
```

# NORM OF A VECTOR

- Norm of a vector is the length of a vector
- Defined as square root of scalar product:

$$|\vec{a}| = \sqrt{\vec{a} \cdot \vec{a}} = |\,(\,a_1 \quad a_1 \quad \ldots \quad a_d\,)\,| = \sqrt{(\sum_{i=1}^{d} a_i^2)}$$

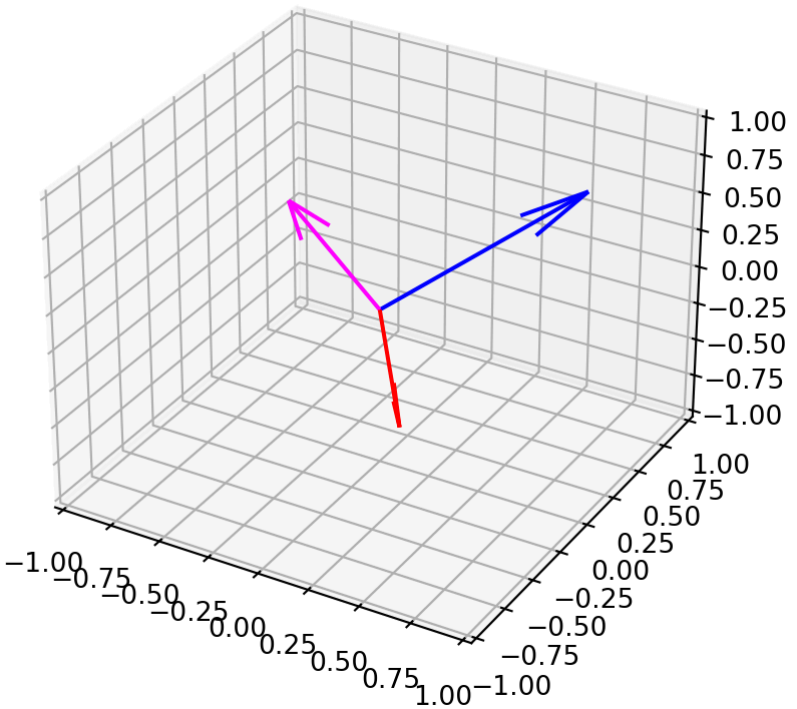
```
1  import numpy.linalg as la
2  a = np.array([1, 3, 0, 5])
3  print(f"The norm of {a} = {la.norm(a)}")
```

```
The norm of [1 3 0 5] = 5.916079783099616
```

# VECTOR PRODUCTS (CROSS-PRODUCT)

The vector product $\vec{a} \times \vec{b}$ is defined as vector $\vec{c}$ with

- Norm $|\vec{a} \times \vec{b}| = |a||b| \sin(\theta)$
- Perpendicular to surface spanned by $\vec{a}$ and $\vec{b}$


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

# VECTOR PRODUCTS

Alternative definition (suitable for computers):

$$\vec{a} \times \vec{b} = \det \begin{pmatrix} \overrightarrow{e_1} & \overrightarrow{e_2} & \overrightarrow{e_3} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$$

$$= \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \overrightarrow{e_1} - \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} \overrightarrow{e_2} + \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} \overrightarrow{e_3}$$

$$= (a_2 b_3 - a_3 b_2)\overrightarrow{e_1} - (a_1 b_3 - a_3 b_1)\overrightarrow{e_2} + (a_1 b_2 - a_2 b_1)\overrightarrow{e_2}$$

# VECTOR PRODUCTS

Calculate $\vec{a} \times \vec{b}$ using the Numpy function `np.cross(a, b)`:

```
1  a = np.array([1, -1/2, 1])
2  b = np.array([1/2, 1, 1/2])
3  c = np.cross(a, b)
4  print(f"{a} x {b} = {c}")
```

```
[ 1.  -0.5  1. ] x [0.5 1.  0.5] = [-1.25  0.    1.25]
```

# LINEAR ALGEBRA: MATRICES

# ARRAY INITIALIZATION: MATRICES

```
1  a = np.eye(3)                    # 2D unity matrix
2  print(a)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

```
1  # Constructing a matrix with "diagonals"
2  a = np.diagflat([6, 5, 4], 1) + np.diagflat([1, 2, 3], -1
3  print(a)
```

```
[[0 6 0 0]
 [1 0 5 0]
 [0 2 0 4]
 [0 0 3 0]]
```

# MATRIX PRODUCTS

The matrix product between matrices $A$ and $B$ is defined as

$$A \cdot B = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

$$= \sum_j a_{ij} b_{jk}$$

- Rows of $A$ are multiplied by columns of $B$.
- $A_{MN} \cdot B_{NK} \leftarrow$ No. columns of $A$ must equal No. rows of $B$

# MATRIX PRODUCTS

The matrix product $A \cdot B$ using Numpy `np.matmul()`:

```
1 A = np.array([[1,-2,3],[0,2,4]])
2 B = np.array([[1,-2,3],[0,2,4],[-1,1,0]])
3 print(A)
```

```
[[ 1 -2  3]
 [ 0  2  4]]
```

```
1 print(B)
```

```
[[ 1 -2  3]
 [ 0  2  4]
 [-1  1  0]]
```

```
1 print(np.matmul(A, B))
```

```
[[-2 -3 -5]
 [-4  8  8]]
```

# MATRIX PRODUCTS

Alternatives to `np.matmul(A, B)` for $A \cdot B$ using Numpy:

- `@` operator: `A @ B`, and
- `np.dot(A, B)`

```
1 print(np.matmul(A, B))
```

```
[[-2 -3 -5]
 [-4  8  8]]
```

```
1 print(A @ B)
```

```
[[-2 -3 -5]
 [-4  8  8]]
```

```
1 print(np.dot(A, B))
```

```
[[-2 -3 -5]
 [-4  8  8]]
```

# DETERMINANT OF A MATRIX

The determinant of a matrix $A$:

$$\det(A) = \det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

$$= \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} a_{11} - \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} a_{12} + \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} a_{13}$$

$$= (a_{22}a_{33} - a_{23}a_{32})a_{11} - \dots$$

# DETERMINANT OF A MATRIX

## Use Numpy `numpy.linalg.det()` function

```
1  import numpy.linalg as la
2  A = np.array([[1,-2,3],[0,2,4],[-1,1,0]])
3  print(A)
```

```
[[ 1 -2  3]
 [ 0  2  4]
 [-1  1  0]]
```

## Determinant is given by

```
1  print(f"Det(A) = {la.det(A)}")
```

```
Det(A) = 9.999999999999998
```

# INVERSE OF A MATRIX

The inverse $A^{-1}$ of a matrix $A$ is defined as:

$$A^{-1}A = A^{-1}A = 1$$

```
1  import numpy.linalg as la
2  A = np.array([[1,-2,3],[0,2,4],[-1,1,0]])
3  print(A)
4  print(f"Det(A) = {la.det(A)}")
```

```
[[ 1 -2  3]
 [ 0  2  4]
 [-1  1  0]]
Det(A) = 9.999999999999998
```

# INVERSE OF A MATRIX

The inverse $A^{-1}$ of a matrix $A$ is defined as:

$$A^{-1}A = A^{-1}A = 1$$

Since $\det(A) \neq 0 \rightarrow A^{-1}$ exists

```
1  Ainv = la.inv(A)
2  print(Ainv)
```

```
[[-0.4  0.3 -1.4]
 [-0.4  0.3 -0.4]
 [ 0.2  0.1  0.2]]
```

# INVERSE OF A MATRIX

The inverse $A^{-1}$ of a matrix $A$ is defined as:

$$A^{-1}A = A^{-1}A = 1$$

Verify $A^{-1}A = 1$:

```
1  np.dot(A, Ainv)
```

```
array([[ 1.00000000e+00, -2.77555756e-17, -5.55111512e-17],
       [ 0.00000000e+00,  1.00000000e+00,  0.00000000e+00],
       [ 1.11022302e-16,  5.55111512e-17,  1.00000000e+00]])
```

```
1  np.round(np.dot(A, Ainv), 5) # Round numerical errors
```

```
array([[ 1., -0., -0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Lecture 08: Arrays, vectors & linear algebra (Ch. 5)