Università
Ca' Foscari
Venezia

Computer Vision

Python / Numpy / OpenCV

Filippo Bergamasco (filippo.bergamasco@unive.it)
http://www.dais.unive.it/~bergamasco
DAIS, Ca' Foscari University of Venice
Academic year 2018/2019

Why Python

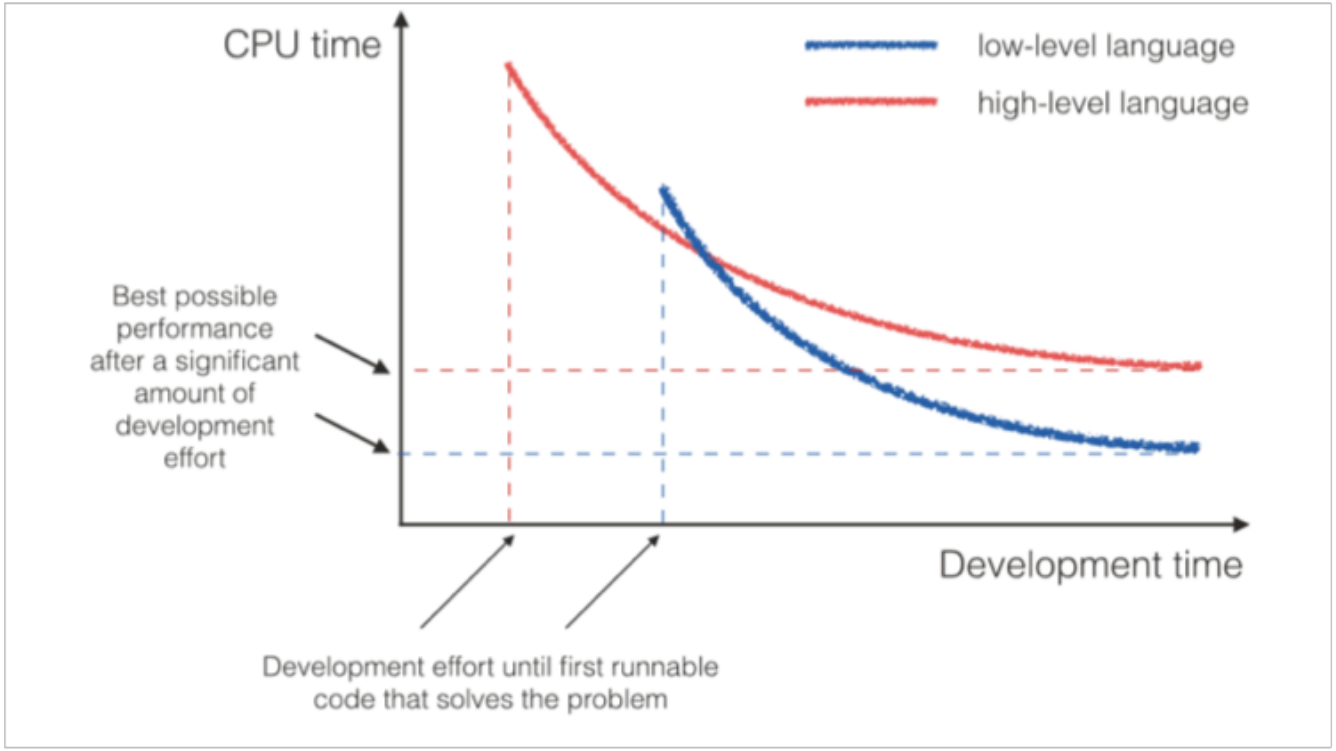
Python is a **high-level**, general-purpose **interpreted** programming language that is widely used in scientific computing and engineering.

- Known for clean and easy-to-read code syntax
- Was not designed for numerical computing… but it is well suited for this task

The struggle of computational problem-solving:

Low-level languages	VS	High-level languages
Needed to obtain the best performance out of the hardware that runs the code		Reduced development time, more flexible code, fast turnaround to test various ideas

Trade-off between low and high level languages



A solution to overcome the tradeoff is use a high-level language to interface libraries and packages written in low-level language.

Python excels at this type of integration

Python ecosystem

As a consequence of this "multi-language" model, Python for scientific and technical computing involves an entire ecosystem of software and libraries:

- Python interpreter (python2 / python3)
- Development tools and environments (Jupyter, VSC, Ipython, PyCharm, etc)
- Python packages (numpy, scipy, matplotlib)
- System libraries (os, blas, lapack, etc.)

Package managers

To have a complete ecosystem up and running, we must manage to:

- Install a python interpreter (problem: choose the correct version)
- Install all the required packages
- Compile/Install the low-level libraries required by the packages
- Optional: install an IDE or an interactive environment

To ease the process, there exist a number of prepackaged Python environments with automated installers (package managers)

Conda / Miniconda

"Conda is an open source **package management system** and environment management system that runs on Windows, macOS and Linux."

https://conda.io

It was created for Python but can now manage different languages.

With conda you can install the Anaconda **distribution** (a set of hundreds of packages including numpy, scipy, ipython, etc)

Miniconda is a smaller alternative containing just the required packages to run conda

Install miniconda

Go to https://conda.io/en/latest/miniconda.html

And download the package for your system.

After the installation, the "conda" package manager can be used in different ways:

- By opening the "Anaconda Prompt" in Windows
- By opening a terminal in OSX or Linux (please ensure that miniconda/bin directory is in your PATH)

Let's start by updating conda itself:

```
$ conda update conda
```

Virtual environments

The software ecosystem consist in multiple libraries and independent projects

Often, new releases of some libraries are not backward-compatible, complicating the problem of creating a stable and reproducible environment over in the long term and for different users.

Conda allows you to to create separate environments containing files, packages and their dependencies that will not interact with other environments.

We will create a new environment for our vision-related projects

Creating a new environment

Add conda-forge channel

```
$ conda config --add channels conda-forge
```

```
$ conda create --name cv python=3.6
```

To activate the environment:

```
$ conda activate cv
```

Installing some stuff

```
$ conda install jupyterlab
qtconsole ipywidgets
$ conda install numpy matplotlib
$ conda install opencv
```

Working with Python

You can use Python in two ways:

- Run the program directly through the Python interpreter
- Use an enhanced command-line REPL environment (IPython) by either:
  - Invoking ipython from a shell `$ ipython`
  - Using a local qt-based console `$ jupyter qtconsole`
  - Using a web-based environment (jupyter lab or jupyter notebook)

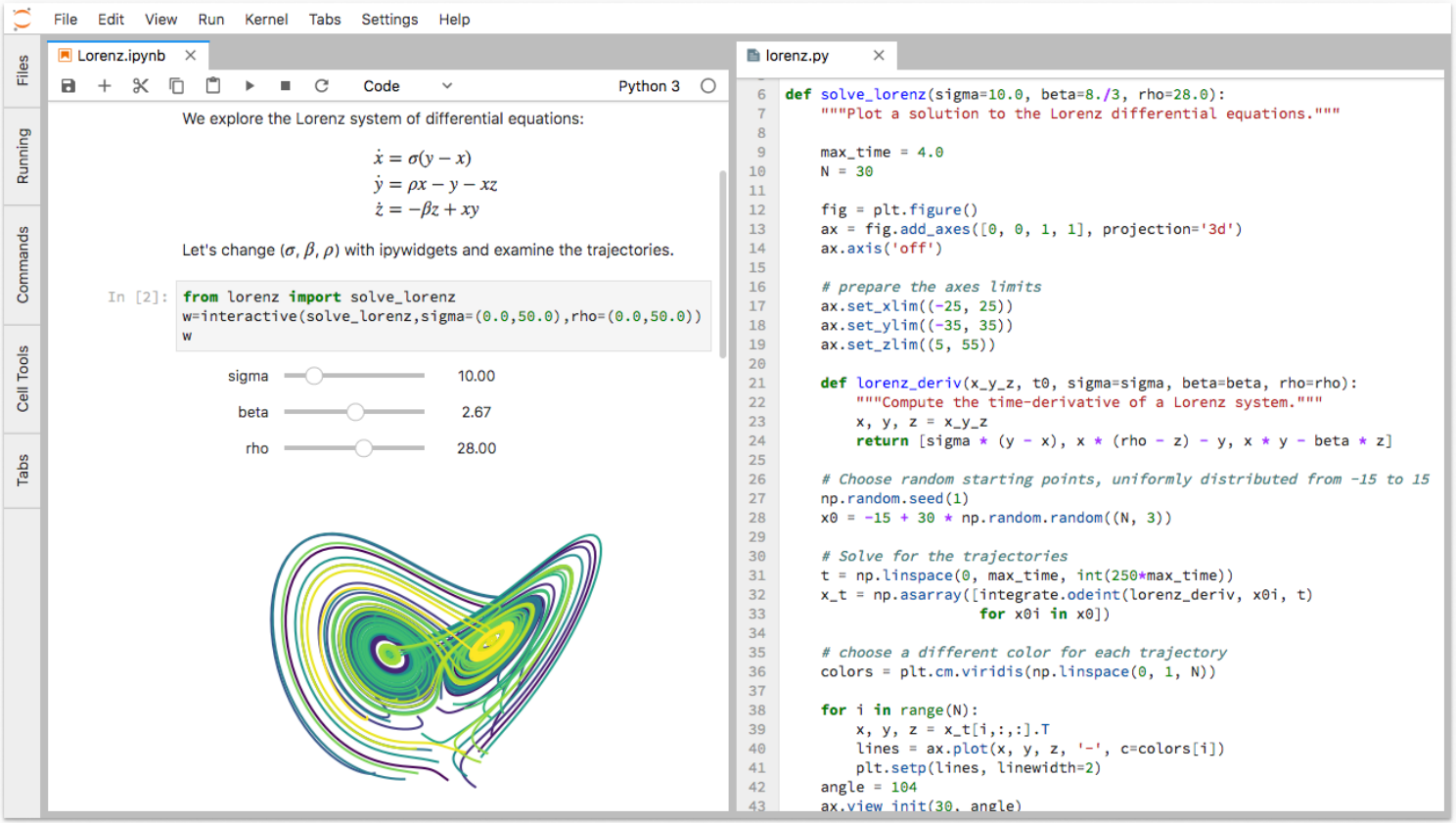
Jupyter is a powerful way to experiment new algorithms before programming a standalone python application

Università Ca' Foscari Venezia

Jupyter lab

To use Jupyter lab run the command

```
$ jupyter lab
```



Jupyter lab

Jupyter is a web interface based on a client-server architecture:

- A «notebook» represent a front-end to the user in which the python code can be grouped in different cells
- Each notebook is backed by a python process (called kernel) in which the cells are executed
- A kernel, unless manually restarted, remains active between different code executions (ie. variables declared in one cell are visible to the others after execution)
- An interactive console can refer to the same kernel used by a notebook to evaluate code on-the-fly
- A jupyter notebook can be converted back to a python file with: `$ jupyter nbconvert --to python Notebook.ipynb`

Working with data

Python stores data in several different ways, but the most popular methods are

- **Lists**

[1, [2, 'three'], 4.5] , list(range(10))

- **Dictionaries**

{'food': 'spam', 'taste': 'yum'} , dict(hours=10)

Both can store nearly any type of Python **object** as an element. But operating on the elements in a list can only be done through iterative **loops**, which is computationally **inefficient in Python**

numpy

Numpy package provide the **ndarray** data object representing a multidimensional array of homogeneous data (same type)

[mathematically known as tensor]

ndarray contains also metadata about the array:

Attribute	Description
ndim	Number of dimensions (axes)
shape	A tuple that contains the number of elements (the length) for each dimension (axis) of the array.
size	Total number of elements in the array
nbytes	Total number of bytes used by the array
dtype	Data type of the array elements

numpy

Basic numerical data types supported:

- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- Bool
- float16, float32, float64, float128
- complex64, complex128, complex256

Once created, the dtype of an array cannot be changed. The method astype(dtype) can be used to get a copy of the original array with a different datatype.

numpy

Multidimensional arrays are stored as **contiguous data** in memory. There is a freedom of choice in how to arrange the elements in the array memory segment

Example: 2-dimensional array

1	2	3	4
5	6	7	8
9	10	11	12

Row-major format

1	4	7	10
2	5	8	11
3	6	9	12

Column-major format

Strides

Row-major and column-major ordering are special cases of strategies (2D case) for mapping **the index** used to address an element, **to the offset** for the element in the array's memory

In the general case, such mapping is defined by the **strides** attribute:

A **tuple** of the same length as the number of axes of the array. Each value is the **factor** by which the index for the corresponding axis is **multiplied** when calculating the memory offset (in bytes) for a given index expression.

Strides example

1	2	3	4
5	6	7	8
9	10	11	12

```
A.shape = (3,4)
A.dtype = uint32
A.strides = (16,4)
```

Each increment of **m** in A[n,**m**] increase the memory offset by 4 bytes

Each increment of **n** in A[**n**,m] increase the memory offset by 16 bytes

1	4	7	10
2	5	8	11
3	6	9	12

```
A.shape = (3,4)
A.dtype = uint32
A.strides = (4,16)
```

Each increment of **m** in A[n,**m**] increase the memory offset by 16 bytes

Each increment of **n** in A[**n**,m] increase the memory offset by 4 bytes

Strides and views

Strides is a smart way to implement reshaping operations like the transpose without moving data in memory (numpy just changes the stride attribute)

Operations that only require changing the strides attribute result in new **ndarray object that refer to the same data as the original array**. Such arrays are called **views.**

- Modifying data on a view modifies data in the original array
- Data may not be contiguous in memory (important when interfacing with C libraries)

Creating arrays

```
import numpy as np
```

Function	Description
np.array	Creates an array for which the elements are given by an array-like object (ie. python list, a tuple, an iterable sequence, or another ndarray instance)
np.zeros	Creates an array filled with zeros
np.ones	Creates an array filled with ones
np.diag	Creates a diagonal array with specified values along the diagonal
np.linspace	Creates an array with evenly spaced values between specified start and end values, using a specified number of elements.
np.meshgrid	Generates coordinate matrices (and higher-dimensional coordinate arrays) from one-dimensional coordinate vectors
np.loadtxt	Creates an array from a text file (for example a CSV file)

https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html

Indexing and slicing

- **Integers** are used to select single elements
- **Slices** are used to select ranges and sequences of elements (m:n:s notation selects elements from m to n-1 with stride s).
- **Positive integers** are used to index elements from the beginning of the array (index starts at 0)
- **Negative integers** are used to index elements from the end of the array, where the last element is indexed with –1

Slices create a **view** on a specified array, not a copy!

Slicing examples

Expression	Description
A[m]	Selects an alement at index m
A[-m]	Select the m^{th} element from the end of the list.
A[m:n]	Select elements with index starting at m and ending at n−1
A[:]	Select all elements (in the given axis)
A[:n]	Select elements starting with index 0 and going up to index n − 1
A[m:]	Select elements starting with index m and going up to the last element in the array.
A[m:n:p]	Select elements with index m through n (exclusive), with increment p
A[::-1]	Select all the elements, in reverse order.

With multidimensional arrays, element selections can be applied on each axis.

Ex: A[:3,:3] is the upper-half diagonal block matrix of size 3

Fancy and boolean indexing

An array can be indexed with another NumPy array, a Python list, or a sequence of integers, whose values select elements in the indexed array

In Boolean indexing, each element (with values True or False) indicates whether or not to select the element from the list with the corresponding index

Arrays returned using fancy indexing and Boolean-valued indexing are not views but rather new independent arrays

Reshaping and Resizing

Function	Description
np.reshape	Reshape an N-dimensional array creating a **view**. The total number of elements must remain the same.
np.flatten	Creates a **copy** of an N-dimensional array, and reinterpret it as a one-dimensional array (i.e., all dimensions are collapsed into one)
np.ravel	Create a **view** (if possible, otherwise a copy) of an N-dimensional array in which it is interpreted as a one-dimensional array.
np.squeeze	Removes axes with length 1.
np.expand_dims	Add a new axis of length 1 to an array
np.hstack	Stacks a list of arrays horizontally (along axis 1)
np.vstack	Stacks a list of arrays vertically (along axis 0)
np.concatenate	Creates a new array by appending arrays after each other, along a given axis.
np.resize	Resizes an array. Creates a new copy of the original array, with the requested size. If necessary, the original array will be repeated to fill up the new array.

Vectorized expressions and broadcasting

NumPy implements functions and vectorized operations corresponding to most fundamental mathematical functions and operators

https://docs.scipy.org/doc/numpy/reference/routines.math.html

- Most of these functions and operations **act on arrays on an elementwise basis**

```
In [132]: x = np.array([[1, 2], [3, 4]])
In [133]: y = np.array([[5, 6], [7, 8]])
In [134]: x + y
Out[134]: array([[ 6,  8], [10, 12]])
```

Vectorized expressions and broadcasting

What if the input arrays have different shape?

Two arrays are called **broadcastable** if one of the following is true:

1. The arrays all have exactly the same shape.
2. The arrays have the same number of dimensions, and the length of each dimensions is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

Vectorized expressions and broadcasting

How broadcasting works:

- If the number of axes of the two arrays is not equal, **the array with fewer axes is padded with new axes of length 1 from the left** until the numbers of dimensions of the two arrays agree.
- If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stride will be 0 for that dimension and the numbers are hence repeated.
- Operations are then performed element-wise

Broadcasting examples

1	2	3
4	5	6
7	8	9

(3,3)

*

1	2	5
3	2	1
7	1	3

(3,3)

=

1	4	15
12	10	6
49	8	27

(3,3)

Broadcasting examples

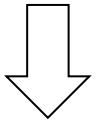
1	2	3
4	5	6
7	8	9

(3,3)

*

1
3
7

(3,1)



1	1	1
3	3	3
7	7	7

(3,3)

=

1	2	3
12	15	18
49	56	63

(3,3)

Broadcasting examples

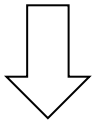
1	2	3
4	5	6
7	8	9

(3,3)

*

1	2	5

(3) -> prepended with a new axis to have (1,3)



1	2	5
1	2	5
1	2	5

(3,3)

=

1	4	15
4	10	30
7	16	45

(3,3)

Broadcasting examples

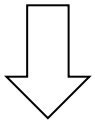
1	2	3
4	5	6
7	8	9

(3,3)

*

2

(1) -> prepended with a new axis to have (1,1)



2	2	2
2	2	2
2	2	2

(3,3)

=

2	4	6
8	10	12
14	16	18

(3,3)

Broadcasting examples

1	2	3
4	5	6
7	8	9

(3,3)

*

1	2

=

?

Broadcasting examples

1	2	3
4	5	6
7	8	9

(3,3)

*

1	2

=

X

The two arrays are not broadcastable!

Matrices

- All the common mathematical operations work in an element-wise fashon, even for 2D arrays.
- The **ndarray** class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while **matrix** is intended to facilitate **linear algebra computations specifically**

Matrix class is now **deprecated** since is less general with respect to ndarrays, even for linear algebra

Linear algebra with arrays

Function	Description
A * B Or multiply()	Element-wise multiplication
A @ B Or dot()	Matrix multiplication You can treat one-dimensional arrays as either row or column vectors. A @ v treats v as a column vector, while v @ A treats v as a row vector (useful to avoid transpose operations)
A.T	Gives the transpose of a 2D array. Conjugate transpose can be done with A.conj().T
linalg.solve(A, b)	Solve a linear matrix equation, or system of linear scalar equations

OpenCV Basics

OpenCV (3.4.5) library documentation:

https://docs.opencv.org/3.4.5/

Composed by different modules:

core

imgproc

calib3d

features2d

highgui

… and many others

OpenCV Basics

You can find useful examples for common computer vision topics in the python tutorials page:

https://docs.opencv.org/3.4.5/d6/d00/tutorial_py_root.html

cv::Mat

The cv::Mat is probably the most important data type in OpenCV

http://docs.opencv.org/3.4.5/d3/d63/classcv_1_1Mat.html

The class Mat represents an n-dimensional dense numerical single-channel or multi-channel array. It can be used to store real or complex-valued vectors and matrices, grayscale or color images, voxel volumes, vector fields, and so on

When working with python, a numpy ndarray can be used instead!

Loading/Saving images

The imgcodecs module contains two useful functions cv::imread and cv::imwrite to save and load images respectively

Imgcodecs documentation:

http://docs.opencv.org/3.4.5/d4/da8/group__imgcodecs.html

imread function:

http://docs.opencv.org/3.4.5/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56

imwrite function:

http://docs.opencv.org/3.4.5/d4/da8/group__imgcodecs.html#gabbc7ef1aa2edfaa87772f1202d67e0ce

Other useful functions

Color-space conversion:

http://docs.opencv.org/3.4.5/d7/d1b/group__imgproc__misc.html#ga397ae87e1288a81d2363b61574eb8cab

Thresholding:

https://docs.opencv.org/3.4.0/d7/d1b/group__imgproc__misc.html#gae8a4a146d1ca78c626a53577199e9c57

Histogram:

https://docs.opencv.org/3.4.0/d6/dc7/group__imgproc__hist.html#ga4b2b5fd75503ff9e6844cc4dcdaed35d

Displaying images

Images can also be displayed interactively with the function **imshow** implemented in the highgui module

highgui documentation:

http://docs.opencv.org/3.4.5/d7/dfc/group__highgui.html

imshow function:

http://docs.opencv.org/3.4.5/d7/dfc/group__highgui.html#ga453d42fe4cb60e5723281a89973ee563

NOTE: The function should be followed by waitKey function which displays the image for specified milliseconds. Otherwise, it won't display the image.

Images in Jupyter

When working with Jupyter, images and plots can be displayed with the matplotlib library:

https://matplotlib.org/

General usage:

```
import matplotlib.pyplot as plt
plt.figure()
plt.imshow( I )
plt.title("My image")
```

Interactive plots

Jupyter supports widgets to automatically create user interface (UI) controls for exploring code and data interactively.

Install extension first:

```
$ jupyter lab clean
$ jupyter labextension install
@jupyter-widgets/jupyterlab-manager
```

Then use the interact() function provided by ipywidgets

`https://ipywidgets.readthedocs.io/en/stable/examples/Using%20Interact.html`

Interactive plots

```
from ipywidgets import interact, widgets
import numpy as np

def plot_func(freq):
    x = np.linspace(0,
2*np.pi,int(100*freq))
    y = np.sin(x * freq)
    plt.plot(x, y)

interact(plot_func, freq = (1.0,10.0,0.5))
```