Python For Geosciences

Thursday 25/09/2025 9h30 OSERen room

Add slides with specific methods/applications to contribute !
Add *.py scripts to :
https://drive.google.com/drive/folders/1yh6mnoBtGPEm_eqGLHqjzezgdhfqeyn2?usp=sharing

General methods	Computing	Image analysis	Automation

Python Workshop
Thursday 25 September, Salle Oseren, 9h30-16h

Program :

9h30 - 10h30	Python Basics
10h30 - 11h	*Coffee break*
11h-12h45	Python for Computing, Analysis and Automation
12h45 - 14h	Vegan/Vegetarian Lunch offered by the TERA team (bring your own cup, plates, and cutlery)
14h-16h	Workshop in 3 small groups with mentors : Computing / Analysis / Automation. Bring your own problems !

Speakers : Stefano Ascione, Boris Gailleton, Quentin Courtois, Marc Lamblin , Pratyaksh Karan, Manuel Maeritz, Lisa Ringel, Clément Petitjean, Joris Heyman, hugo Blons

General methods - Getting started

www.python.org

- What is Python
- Distribution : Python, Anaconda
- IDE : Spyder, VS / VS Code, PyCharm
- Jupyter Notebook
- Installing Python
- Python versions :
  - **Python 2** (eol) :

    last : 2.7.2 on 11/06/2011,

    eol : 01/01/2020

  - **Python 3** (current) : Python 3.13.7 (14/08/2025)

Version	Latest micro version	Release date	End of full support	End of security fixes
0.9	0.9.9[2]	1991-02-20[2]	1993-07-29[a][2]	
1.0	1.0.4[2]	1994-01-26[2]	1994-07-14[a][2]	
1.1	1.1.1[2]	1994-10-11[2]	1994-11-10[a][2]	
1.2		1995-04-13[2]	Unsupported	
1.3		1995-10-13[2]	Unsupported	
1.4		1996-10-25[2]	Unsupported	
1.5	1.5.2[55]	1998-01-03[2]	1999-04-13[a][2]	
1.6	1.6.1[55]	2000-09-05[56]	2000–09[a][55]	
2.0	2.0.1[57]	2000-10-16[58]	2001-06-22[a][57]	
2.1	2.1.3[57]	2001-04-15[59]	2002-04-09[a][57]	
2.2	2.2.3[57]	2001-12-21[60]	2003-05-30[a][57]	
2.3	2.3.7[57]	2003-06-29[61]	2008-03-11[a][57]	
2.4	2.4.6[57]	2004-11-30[62]	2008-12-19[a][57]	
2.5	2.5.6[57]	2006-09-19[63]	2011-05-26[a][57]	
2.6	2.6.9[31]	2008-10-01[31]	2010-08-24[b][31]	2013-10-29[31]
3.0	3.0.1[57]	2008-12-03[31]	2009-06-27[64]	
3.1	3.1.5[65]	2009-06-27[65]	2011-06-12[66]	2012-04-06[65]
2.7	2.7.18[36]	2010-07-03[36]	2020-01-01[c][36]	
3.2	3.2.6[67]	2011-02-20[67]	2013-05-13[b][67]	2016-02-20[67]
3.3	3.3.7[68]	2012-09-29[68]	2014-03-08[b][68]	2017-09-29[68]
3.4	3.4.10[69]	2014-03-16[69]	2017-08-09[70]	2019-03-18[a][69]
3.5	3.5.10[71]	2015-09-13[71]	2017-08-08[72]	2020-09-30[71]
3.6	3.6.15[73]	2016-12-23[73]	2018-12-24[b][73]	2021-12-23[73]
3.7	3.7.17[74]	2018-06-27[74]	2020-06-27[b][74]	2023-06-06[74]
3.8	3.8.20[75]	2019-10-14[75]	2021-05-03[b][75]	2024-10-07[75]
3.9	*3.9.23*[76]	2020-10-05[76]	2022-05-17[b][76]	2025-10[76][77]
3.10	*3.10.18*[78]	2021-10-04[78]	2023-04-05[b][78]	2026-10[78]
3.11	*3.11.13*[79]	2022-10-24[79]	2024-04-02[b][79]	2027-10[79]
3.12	*3.12.11*[80]	2023-10-02[80]	2025-04-08[b][80]	2028-10[80]
3.13	*3.13.7*[81]	2024-10-07[81]	2026-05[81]	2029-10[81]
3.14	3.14.0rc3[82]	2025-10-07[82]	2027-05[82]	2030-10[82]
3.15	3.15.0a0[83]	2026-10-01[83]	2028-05[83]	2031-10[83]

Python 2

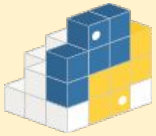
Python 3

Python 2 EOL

Phyton versions timeline

General methods - Managing packages

- Pip (linux-windows) :
  - Install from Python Package Index (PyPI)
  - Install from wheel file / setup file



https://pypi.org/

- Conda (python and more)
  - Install from Conda Forge
  - Other functionalities (e.g. environment setup)

CONDA

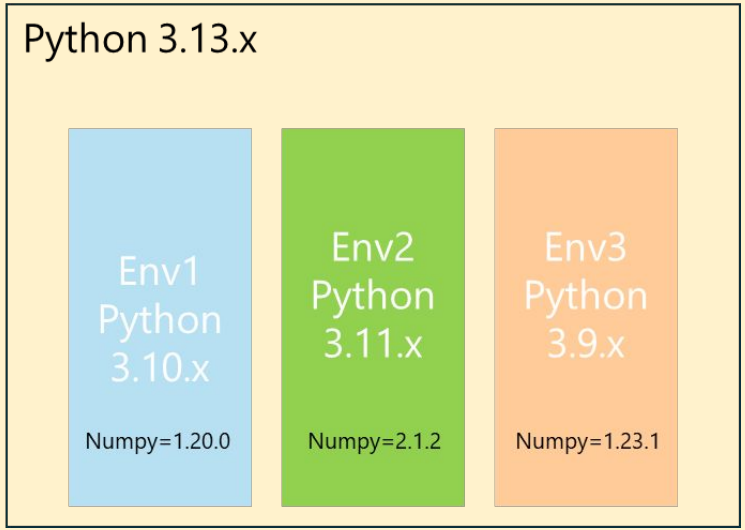
https://docs.conda.io

General methods - Python environment

Virtualenv : https://virtualenv.pypa.io/en/latest/
Venv : https://docs.python.org/3/library/venv.html
Conda : https://docs.conda.io/projects/conda/en/stable/user-guide/tasks/manage-environments.html

- What is a Python environment ?
  - Separate Python installation

- Why using a Python environment ?
  - Allows multiple Python installation to coexist
  - Freeze an installation
  - Prevents version conflicts
  - Older software



Example of multiple Python environments

General methods - Python environment : Venv

- Choose a **Library** : **venv**
- **Install** (local version) the **Python** version (Py 3.x.x) you want for your environment
- **Create** the environment by referencing Py 3.x.x

```
Py 3.x.x -m venv MyEnvironment
```

- **Activate** environment from ./MyEnvironment/Scripts/activate

- **Install** required packages

- (Export list of packages to a .txt file)

```
python -m pip freeze > requirements.txt
```

General methods - Python environment : Conda

- Choose a **Library** : **conda**
- **Install** conda distribution (miniconda / anaconda)
- **Create** the environment for chose Py 3.x.x

```
conda create -n MyEnvironment python=3.x.x
```

- **Activate** environment

```
conda activate MyEnvironment
```

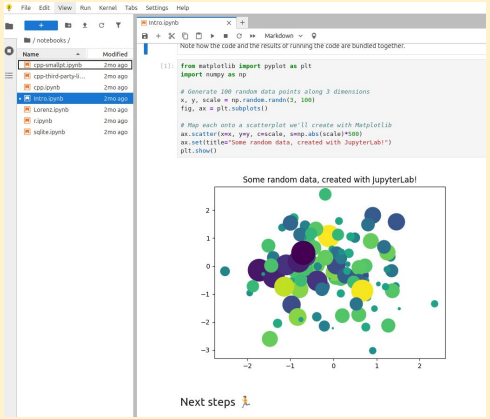
- **Install** required packages (pip or conda)

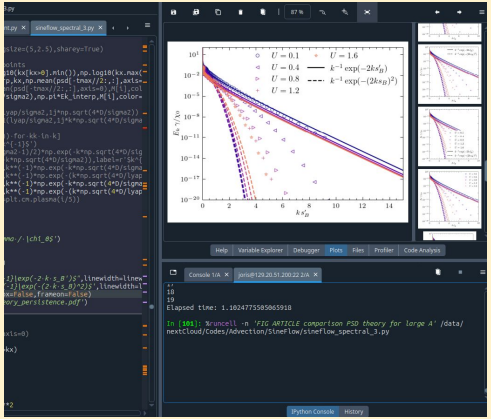
- (Export list of packages to a .txt file)

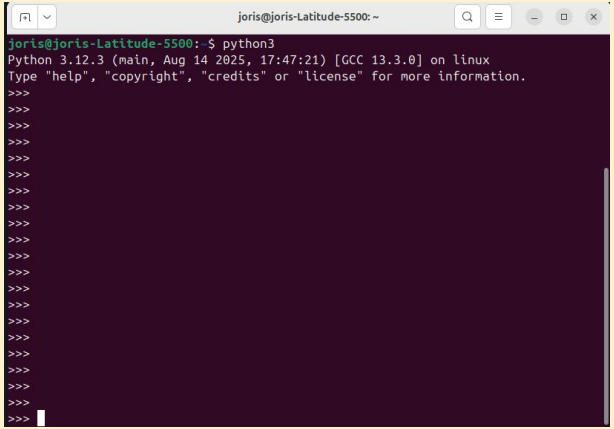
```
python -m pip freeze > requirements.txt
```

General methods

Jupyter notebook/lab versus ipython versus python







Try python on your browser: https://jupyter.org/try-jupyter/lab/

Convert your old codes with AI

Chatbot-style LLMs:

AIstudio (Gemini 2.5 pro) [aistudio.google.com]

Chatgpt	[chat.openai.com]
DeepSeek	[deepseek.com]
LLAMA (Local model)	[from huggingface.co]
Mistral (Le Chat)	[lechat.mistral.ai]
Claude	[claude.ai]
Perplexity	[perplexity.ai]

- Files upload
- Large context window
- Web search
- Repository connection
- Code execution

Convert your old codes with AI

AGENTIC AI:

Github Copilot

Codex GPT

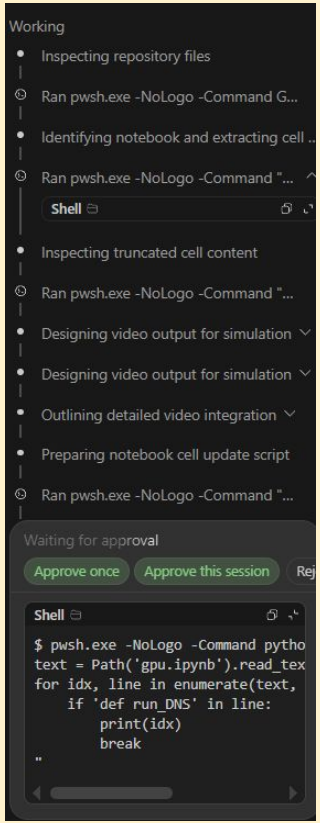
GEMINI 2.5

MANUS

Cursor

LLAMA + (LangChain, LlamaIndex, Haystack, AutoGPT, Open Interpreter)

Cell 17 produce one frame from a ADE simulation. Edit it to produce an entire video that last 10 seconds at 60 fps.



Finished working

Converted the ADE cell to render a 10 s, 60 fps video while keeping the final frame preview.

- Updated `run_DNS` to manage frame counts, downsampling, FFmpeg writing, and cleanup when `capture_video=True` (`gpu.ipynb:528`).
- Added a configured call that produces `ade_simulation.mp4`, reports its path, and still shows the concluding `plt.imshow` (`gpu.ipynb:717`).

FFmpeg must be available on PATH; otherwise the cell will raise the new runtime error.
Next steps:

1. Re-run cell 17 to generate and review `ade_simulation.mp4`.

INSTALL and IMPORT toolboxes

```
%pip install numpy,matplotlib
import numpy as np
import matplotlib.pyplot as plt

a=np.sin([0,1,2,3])
plt.plot(a,a**2,'r-')
```

In matlab, each toolbox cost around 200€/year

In python, all are free

General methods : **Connecting to a remote kernel in Spyder, hosted on a Geosciences server (with vpn access)**

Why ? To have access to the same environment and TOP computing capability at work, home or on the train, inside spyder with graphical capabilities (plot).

1/ Set up a ssh tunnel

2/ Make a python virtual env on server and client, with same spyder version

3/ Start the server kernel and find connection file PID :
ssh joris@129.20.51.200
nohup ~/venv_spyder/bin/python3.12 -m spyder_kernels.console &
ls -allt /home/joris/.local/share/jupyter/runtime/

4/ Copy the kernel connection file to the client
scp joris@129.20.51.200:/home/joris/.local/share/jupyter/runtime/kernel-1389548.json ~/

5/ Connect to the kernel with the connection file in spyder

6/ Kill the kernel
Kill 1389548

Common objects

Numpy Array : A=np.array([1,2,3])
Methods : A.shape, numpy operations …

List : A=[1,2,3]
Methods : len(A), indexing A[i]

Dictionaries: dic ={‘var1’:1,’var2’: A ,’var3’:3}
Methods : dic.keys(), dic[‘var1’]

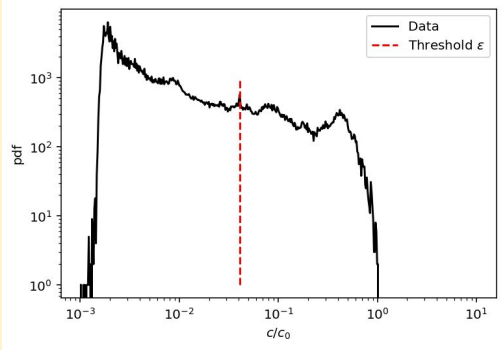
…

General methods : Plotting

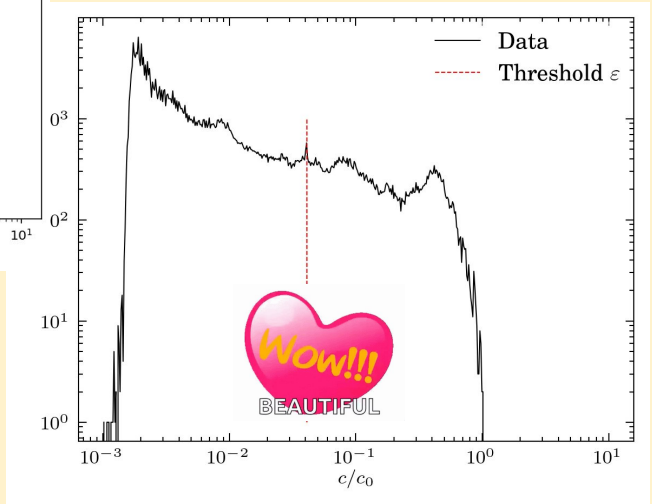
Use a matplotlib config file

Why ? plot publishable figures with only one click. The config file have default figure size, font size, colors, …

1/ without :



2/ plt.style.use('joris.mplstyle') :



Indexing rules in Python (list and arrays)

```
A[:,:,:]
A[1:,:-2,::2]
A.reshape(A.shape[0],-1)
```

For Loops and indents

```
A=np.arange(10)
for a in A:
    print(a)
```

```
A=np.arange(10)
for i in range(len(A)):
    print(A[i])
```

```
A=np.arange(10)
for i,a in enumerate(A):
    print(a,A[i])
```

Appening values in a loop : 3 ways

```
a= []
For i in range(10):
        a.append([i,i**2])
a=np.array(a)
```

using an intermediate list

```
a= np.array([]).reshape(-1,2)
For i in range(10):
        a=np.hstack((a,np.array([i,i**2])))
```

using a nparray

```
a=np.zeros(10,2)+np.nan
For i in range(10):
        a[i,:]=[i,i**2]
```

using a nparray preconstructed

General methods : Use **optional** parameters with **default values** in function definition

```
def my_function(par1, par2=1.2, par3='essai', par4=True):
    print(par1,par2,par3,par4)
    return
```

Will work :

```
my_function(par1)
my_function(par1, par2=3)
```

Wont work :

```
my_function(par2=3)
my_function(par8=3)
```

Very useful when incrementally improving a code with new features while keeping old calls working !

General methods :Using **exceptions** to avoid program stop

```
try:
        a=[1,2]
        b=[2,3]
        c=a*b
except :
        print('Warning, there was a problem somewhere')
        c = np.nan
```

Cons : can lead to unexpected outcomes, difficult to predict and debug.

General methods for files : glob file search

```
import os, glob

os.mkdir(‘/home/joris/mydir’)

files = glob.glob(‘/home/joris/*.txt’)

for file in files:
        print(file)
```

General methods for strings : .format(), re

```
Var = 1.6551348486
print('my var is {:1.1f} big'.format(Var))
```

```
import re
s = "My phone number is 123-456-7890."
pat = r"\d{3}-\d{3}-\d{4}"  # match phone number format
res = re.search(pat, s)  # search pattern

if res:
    print(res.group())  # matched number
    print(res.start())  # match start index
```

General methods: Object Oriented Python (a programming paradigm)

Why ?

- Organize your code (Stop the spaghetti approach)
- make your code more readable
- make your code modular

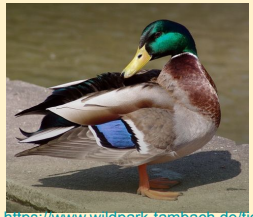
Examples for objects:

- Python built in classes (lists, dictionaries, …)
- Numpy ndarray
- Matplotlib objects (Figure, Axes, Artist, …)

> “Not everything needs to be object-oriented. Object-oriented design frequently does not add value in scientific computing.” https://nsls-ii.github.io/scientific-python-cookiecutter

Basics:

Object: Duck



https://www.wildpark-tambach.de/tiere/ente

Attributes
- Species
- Age
- Sex
- Is hungry

Methods can access and modify the attributes

Methods
- Waddle
- Swim
- Fly
- eat

General methods : Object Oriented Python (a programming paradigm)

Creating an object (duck.py)

```python
class Duck:
    """
    A simple Duck class.

    Attributes:
        name (string): Name of the duck
        age (int): Age of the duck
        sex (string): Gender of the duck. One of 'female', 'male', 'non-binary'
        isHungry (bool): Is duck hungry

    Class Attributes:
        species (string): Species of the duck
    """

    species: str = 'Duck'

    def __init__(self, name: str, age: int, sex: str):
        """
        Initialize a new Duck.

        Parameters:
        name (str): Name of duck
        age (int): age of duck
        sex (str): Gender of duck. One of 'female', 'male', 'non-binary'
        """
        self.name: str = name
        self.sex: int = sex
        self.age: str = age
        self.isHungry: bool = True

    def swim(self) -> None:
        """ Use feet to move in water """
        pass

    def eat(self, food) -> None:
        """ Consume some food """
        self.isHungry = False
        return

    def quack(self) -> None:
        """ Make noises with the peak """
        pass

    @staticmethod
    def tellDuckJoke() -> None:
        print('Why do ducks make great detectives?\n' +
              'Because they always quack the case.')
```

Executing code (main.py)

```python
from duck import Duck

name = 'Donald'
age = 2
sex = 'male'

donald = Duck(name, age, sex)

donald.waddle()
print(donald.name)
```

Inheritance (eiderDuck.py)

```python
from duck import Duck

class EiderDuck(Duck):

    species: str = 'Eider Duck'

    def __init__(self, name: str, age: int, sex: str, downThickness: float):
        super().__init__(name, age, sex)
        self.downThickness: float = downThickness

    def swim(self) -> None:
        """ Swim very gently to protect its down
        """
```

Further reading:

- Duck Typing
- Protocols

https://natur.gl/arter/common-eider/?lang=en

Python for computing

NumPy n-dimensional arrays:

- storing and retrieving data
- ndarray.ndim, ndarray.shape, ndarray.size
- same data type for each entry
- Initialization by lists, np.zeros, np.ones, np.arange, np.linspace,...
- Access elements by indexing, slicing, Boolean indexing,...

Python for computing

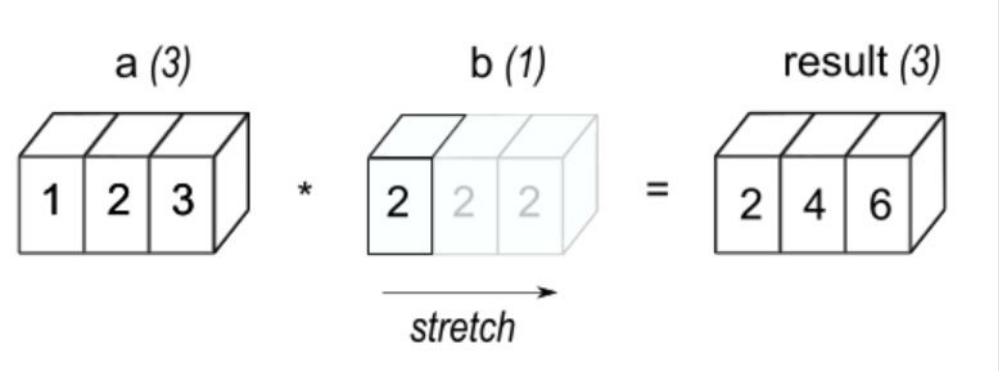
NumPy broadcasting rules:

The shapes of two arrays (np.shape) are compared element-wise from the right.

The dimensions are compatible, when

1: They are equal

2: One of them is 1.



Python for computing - Linear algebra

```
# system of linear equations
A1 = np.array([[3, 2, -1], [2, -2, 4], [-1, .5, -1]])
A2 = np.array([[1, 2], [2, 4]])

b1 = np.array([1, -2, 0])
b2 = np.array([3, 6])

x1 = np.linalg.solve(A1, b1)
print(x1)
print(np.abs(A1 @ x1 - b1))

#x2 = np.linalg.solve(A2, b2)
#print(x2)
```

```
[ 1. -2. -2.]
[2.22044605e-16 1.77635684e-15 2.22044605e-16]
```

```
A3 = np.array([[1, 2, -1], [6, -3, -1], [2, 1, -1]])
b3 = np.array([0, 0, 6])
x3 = np.linalg.solve(A3, b3)
print(x3)
print(np.abs(A3 @ x3 - b3))
```

```
[-2.70215978e+17 -2.70215978e+17 -8.10647933e+17]
[  0. 128.   6.]
```

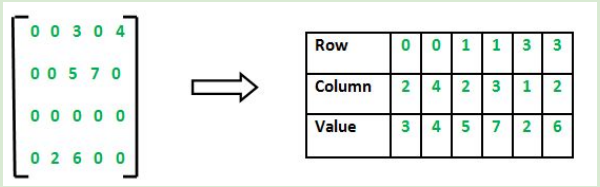
Python for computing - Linear algebra

```
import scipy.linalg

coeff = np.zeros((3, 4))
coeff[:,0:3] = A3
coeff[:,-1] = b3
print(coeff)

# check rank of coefficient matrix
print(np.linalg.matrix_rank(A3))
print(np.linalg.matrix_rank(coeff))

# determinant
print(np.linalg.det(A3))

# eigenvalues
print(np.linalg.eigvals(A3))

# condition number
print(np.linalg.cond(A3))


print(scipy.linalg.solve(A3, b3))
```

```
[[ 1.  2. -1.  0.]
 [ 6. -3. -1.  0.]
 [ 2.  1. -1.  6.]]
2
3
1.1102230246251573e-16
[-5.00000000e+00  2.00000000e+00  8.89048039e-17]
6.1733873453798584e+16
[-2.70215978e+17 -2.70215978e+17 -8.10647933e+17]
```

```
C:\Users\lisar\AppData\Local\Temp\ipykernel_32908\4091296081.py:22: LinAlgWarning: Ill-conditioned matrix (rcond=4.93432e-19): result may not be accurate.
  print(scipy.linalg.solve(A3, b3))
```

Python for computing - Sparse matrices



`bsr_array`(arg1[, shape, dtype, copy, ...])	Block Sparse Row format sparse array.
`coo_array`(arg1[, shape, dtype, copy, maxprint])	A sparse array in COOrdinate format.
`csc_array`(arg1[, shape, dtype, copy, maxprint])	Compressed Sparse Column array.
`csr_array`(arg1[, shape, dtype, copy, maxprint])	Compressed Sparse Row array.
`dia_array`(arg1[, shape, dtype, copy, maxprint])	Sparse array with DIAgonal storage.
`dok_array`(arg1[, shape, dtype, copy, maxprint])	Dictionary Of Keys based sparse array.
`lil_array`(arg1[, shape, dtype, copy, maxprint])	Row-based LIst of Lists sparse array.

For constructing sparse arrays; fast conversion to csr/csc formats; summation of duplicate entries

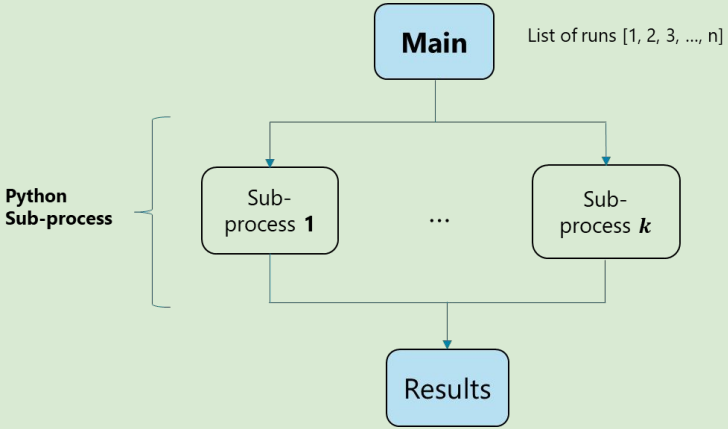
Fast column slicing; efficient arithmetic operations

Fast row slicing; efficient arithmetic operations; fast matrix vector products; system of linear equations

Python for computing - Subprocess

https://docs.python.org/3/library/subprocess.html

- List of n simulations to run
- Split the list in k chunks
- Open a Python process for each chunk
- Merge results for each sub-process



Parallel python code encapsulated within subprocess

Python for computing:

Need for speed? A short guide for HPC in python

Basic Python is ***optimised for flexibility, not for raw performances***

What if I need high performance computing?

- Execution speed
- processing/visualising large data

Domain specific functions

Python for computing:

Need for speed? A short guide for HPC in python

First :
What already exists ?

Table-like data?

Pandas polars

N-dimensional time series?

xarray

Visualisation ?

Paper like figures:
matplotlib seaborn
Big data:
Plotly datashaders
3D: pyvista blender
Performances + GUI:
Moderngl (steep learning curve)

Massive data?

More than your RAM, automation over TB of data

Processing: dask
Compressing: zarr
Organising: h5py netCDF

Machine Learning ?

Didactic: scikit-learn
Deep Learning: pytorch tensorflow
Boost: *XGBoost*/*CatBoost*
HuggingFace has python interfaces

Geospatial?

Raster manip: rasterio
Shapefiles: fiona/shapely/geopandas
Plotting: cartopy/folium

Geomorphology?

topotoolbox fastscapelib LANDLAB

High performance?

Numerical: numpy, scipy
JIT: numba, numexpr, jax
GPU: taichi, pycuda, kompute, cupy
PDE solvers: FEniCSx, pyCLAW, PETSc4Py
Binders: pybind11, cython, f2py

Python for computing:

Need for speed? A short guide for HPC in python

Your code is there, but slow ?

What part of it is slow?

Profiling.

Easy solutions:

Jupyter: `%timeit` (line) `%%timeit` (full cell)

```
[11]: %timeit np.sqrt(np.random.rand(512*512))
      1.17 ms ± 514 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Timestamp: `time.perf_counter()`

```
import time

start = time.perf_counter()
do_something()
stop = time.perf_counter()

print(f'do_something took {stop - start}')
```

Sophisticated solutions:

cProfile:

`python -m cProfile -s tottime your_script.py`
(from terminal)

`cProfile.run("my_function(x), sort='tottime'")`
(in code for a function)

```
Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     4    0.879    0.220    0.879    0.220 _pocketfft.py:51(_raw_fft)
     1    0.403    0.403    1.349    1.349 red_noise.py:22(generate_red_noise_spectral)
     2    0.031    0.015    0.031    0.015 {method 'copy' of 'numpy.ndarray' objects}
     2    0.029    0.014    0.029    0.014 {method 'astype' of 'numpy.ndarray' objects}
     2    0.007    0.004    0.007    0.004 {method 'reduce' of 'numpy.ufunc' objects}
     1    0.001    0.001    1.350    1.350 red_noise.py:86(red_noise)
     2    0.001    0.000    0.879    0.440 _pocketfft.py:747(_raw_fftnd)
     1    0.000    0.000    1.350    1.350 <string>:1(<module>)
     2    0.000    0.000    0.000    0.000 _pocketfft.py:710(_cook_nd_args)
     2    0.000    0.000    0.000    0.000 _helper.py:125(fftfreq)
     1    0.000    0.000    0.004    0.004 _methods.py:119(_mean)
     2    0.000    0.000    0.000    0.000 _stride_tricks_impl.py:350(_broadcast_to)
```

Python for computing:

Need for speed? A short guide for HPC in python

I manage to vectorise (numpy) it. Can I speed it up?

- Numexpr: speeding up complex numpy expression (x2 to x10+)

```python
# Step 1: Complex polynomial (creates temporary array)
step1 = a**3 + 2*b**2 - 3*c + d

# Step 2: Trigonometric operations (creates more temporaries)
step2 = np.sin(step1) * np.cos(b) + np.exp(-c/10)

# Step 3: Statistical transformations (normalization)
step3 = (step2 - step2.mean()) / step2.std()
```

x2 - x10+ (CPU)

```python
# All operations in single optimized expressions - no intermediate arrays
step1 = ne.evaluate("a**3 + 2*b**2 - 3*c + d")
step2 = ne.evaluate("sin(step1) * cos(b) + exp(-c/10)")

# Calculate mean and std using numpy (numexpr has limitations with reductions)
mean_val = np.mean(step2)
std_val = np.std(step2)

# Use broadcasting variables in numexpr expressions
step3 = ne.evaluate("(step2 - mean_val) / std_val")
```

- JAX: Google JIT compiler, can target GPUs and most-numpy code

```python
def numpy_computation(a, b, c):
    """
    Standard NumPy computation - interpreted Python with vectorized operations.
    Each operation is dispatched individually to optimized C code.
    """
    # Complex mathematical expression
    result = np.tanh(a) * np.exp(-b**2) + np.sin(c) * np.cos(a)
    result = result + np.sqrt(np.abs(a * b)) - np.log1p(np.abs(c))
    return result
```

x2 - x100+ (GPU)

```python
def jax_computation(a, b, c):
    """
    JAX computation - can be JIT compiled for optimization.
    Operations are fused and optimized by XLA compiler.
    """
    # Same mathematical expression as NumPy version
    result = jnp.tanh(a) * jnp.exp(-b**2) + jnp.sin(c) * jnp.cos(a)
    result = result + jnp.sqrt(jnp.abs(a * b)) - jnp.log1p(jnp.abs(c))
    return result

# Create JIT-compiled version
jax_computation_jit = jit(jax_computation)
```

Python for computing:

Need for speed? A short guide for HPC in python

I cannot vectorise my code… (theory)

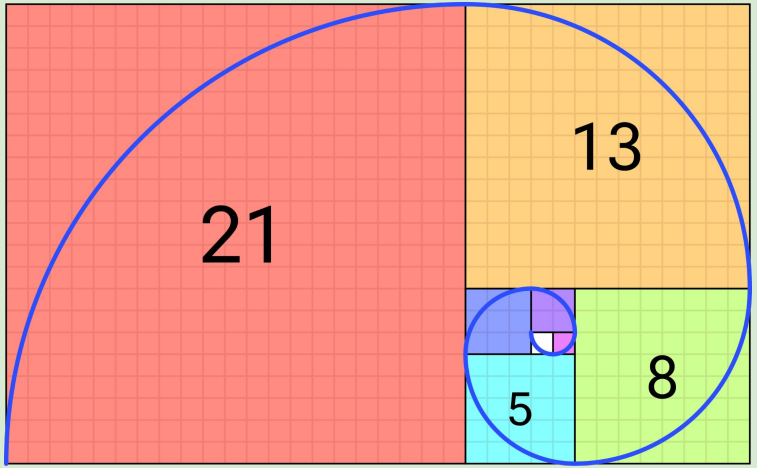
Sometimes, a loop is unavoidable

E.g. Fibonacci sequence:

$$F_0 = 0, \quad F_1 = 1,$$

and

$$F_n = F_{n-1} + F_{n-2}$$



F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181

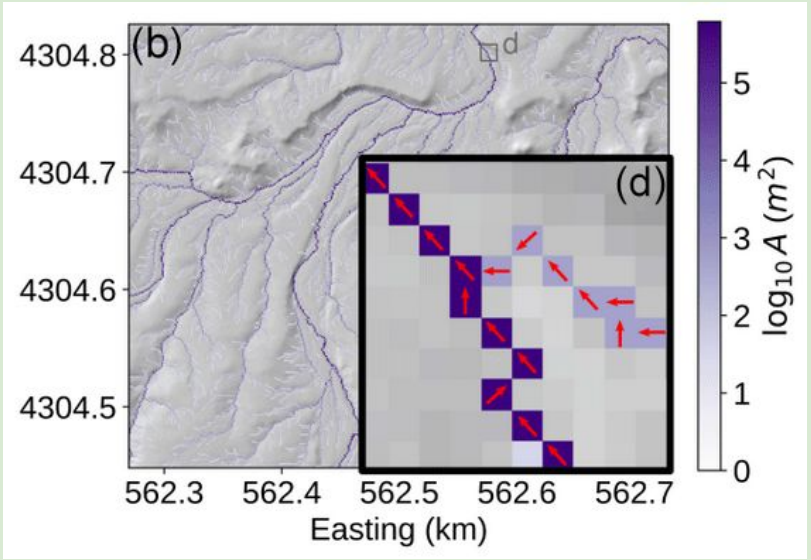
Data dependency = non vectorisable

Python for computing:

Need for speed? A short guide for HPC in python

Python loop is slow, use numba to speed it up

Real case study: Flow direction as steepest descent



Numpy: shift 8 arrays then reduce

```
# Test all directions and reduce (Z[i,j] < Z[i-1,j-1])...
for di, dj, d in zip(offs_i, offs_j, dists):
    nb = Z[1+di:n-1+di, 1+dj:m-1+dj]
    candidates.append((Zi - nb) / (dx * d))
return np.maximum.reduce(candidates)  # shape (n-2, m-2)
```

Speedup ~x23

Numba: one loop

```
# Loop manually through all pixels
for i in prange(1, n-1):
    for j in range(1, m-1):
        zc = Z[i, j]
        best = -1e30
        for k in range(8):
            val = (zc - Z[i+oi[k], j+oj[k]]) / (dx * dd[k])
            if val > best:
                best = val
        out[i-1, j-1] = best
return out
```

Flow accumulation: only achievable with numba

Python for computing:

Need for speed? A short guide for HPC in python

Numba = C-like code with the ease of python

How does numba work?

```python
@numba.njit(parallel=False, fastmath=True)
def flow_accumulation(stack, receivers, width, height, nodata_value):

    # Most of Numpy is accepted into numba
    Accumulation = np.zeros_like(stack)

    # iterating through all the pixels (nodes) ordered from upstream to downstream
    for node in stack:

        # Checking if the pixel is active
        if node == nodata_value:
            continue

        # Local source
        Accumulation[node] += 1

        # Checking the downstream node of node
        this_receiver = receivers[node]

        # Propagating
        Accumulation[this_receiver] += Accumulation[node]

    # Return 2D array
    return Accumulation.reshape(width,height)
```

1) *Write the function*
2) *Compile options*

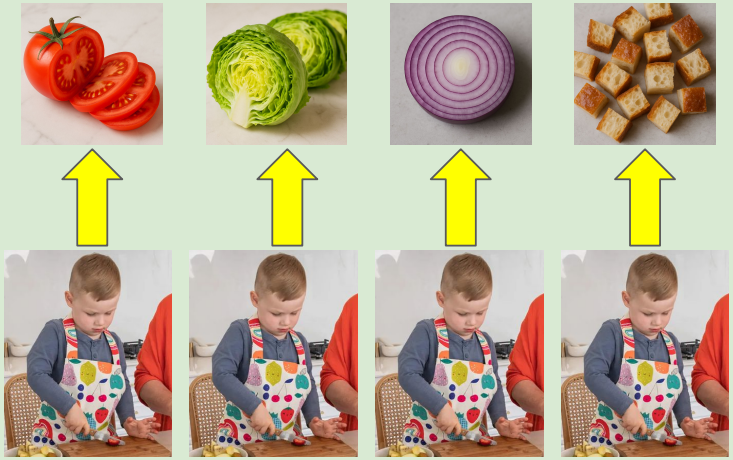
Limitations:

- Only “common” python and numpy accepted
- Compilation overhead at first launch

Still too slow? Mass parallelism.

Python for computing - GPU

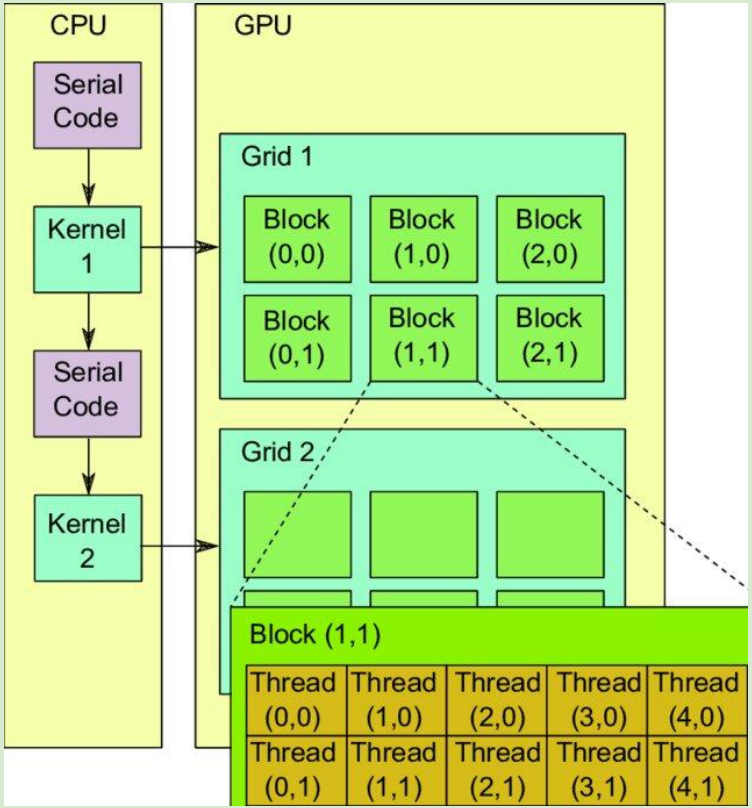
CPU: serial processing



GPU: parallel processing

Python for computing - GPU

Workflow:



Structure:

Grid – full kernel launch, share **global memory** (high latency, available to all threads)

Block – group of threads, condivide **shared memory** (medium latency, available to each block)

Thread – smallest execution unit with private **registers** (low latency, available only for thread)

Python for computing - GPU

CUDA C → low-level, full control of threads, memory, performance

```
sudo apt-get install nvidia-cuda-toolkit
```

CuPy → NumPy-like, high-level, quick GPU acceleration

```
pip install cupy-cuda12x
```

CuPy allow for custom kernels = Python ease + CUDA power!

Python for computing - GPU

CuPy → GPU-accelerated drop-in replacement for NumPy, but runs on CUDA GPUs

```
import cupy as cp
```

```
# 1) Create arrays on GPU
x = cp.array([1, 2, 3], dtype=cp.float32)          # shape (3,)
y = cp.arange(3, dtype=cp.float32)                 # [0,1,2], shape (3,)
M = cp.arange(9, dtype=cp.float32).reshape(3, 3)   # 3x3 matrix
```

```
# 2) Elementwise ops (same shape)
z_add = x + y
z_mul = x * y
z_uadd = cp.add(x, y)
```

```
# 3) Linear algebra
A = cp.random.rand(256, 256, dtype=cp.float32)
B = cp.random.rand(256, 256, dtype=cp.float32)
Cmm = A @ B                              # GEMM
u, svals, vt = cp.linalg.svd(A)          # SVD
w, V = cp.linalg.eig(M)                  # eigen
```

```
# 4) FFT
import cupy.fft as cufft
F = cufft.fft(cp.random.rand(1024))
x_time = cufft.ifft(F)
```

```
# 5) Host-GPU transfer
x_cpu = cp.asnumpy(x)                    # GPU -> CPU (NumPy)
y_gpu = cp.asarray(np.array([4,5,6]))# CPU -> GPU
```

Device memory (GPU) is different from Host memory!

Python for computing - GPU

CuPy → GPU-accelerated drop-in replacement for NumPy, but runs on CUDA GPUs

```
# 6) Raw kernels (advanced)
src = r'''
extern "C" __global__
void axpy(const float a, const float* __restrict__ x,
          float* __restrict__ y, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n){ y[i] = a * x[i] + y[i]; }
}'''
axpy = cp.RawKernel(src, "axpy")
n = x.size
y2 = y.copy()
block = 256
grid = (n + block - 1)//block
axpy((grid,), (block,), (cp.float32(2.0), x, y2, n))
```

custom kernels are
chunks of CUDA C
directly called by Python!

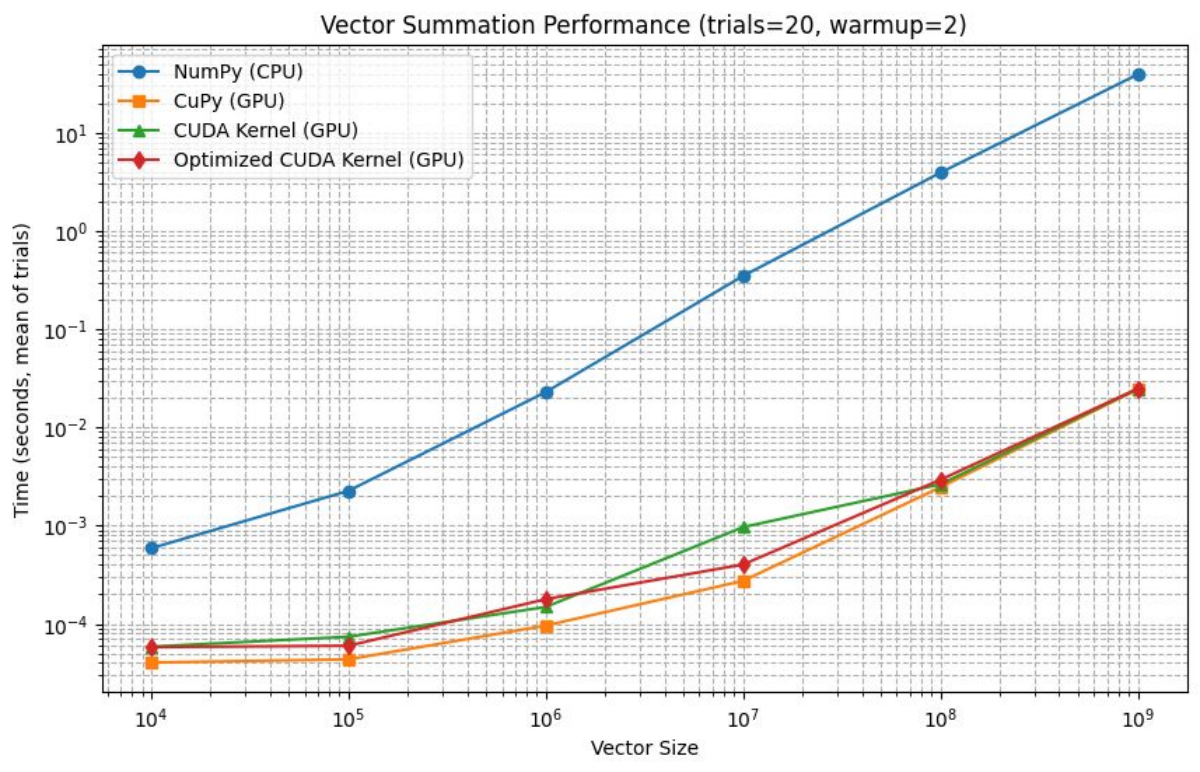
Python for computing - GPU

Vector sum comparison:

$$\sum_{i=1}^{n} x_i$$

```python
def numpy_sum(vector):
    if vector.size < 1000:
        number = 10000
    else:
        number = 100
    timer = timeit.Timer(lambda: np.sum(vector))
    total_time = timer.timeit(number=number)
    return total_time / number

def cupy_sum(vector):
    vector_gpu = cp.asarray(vector)
    start_time = cp.cuda.Event()
    end_time = cp.cuda.Event()
    start_time.record()
    result = cp.sum(vector_gpu)
    end_time.record()
    end_time.synchronize()
    return cp.cuda.get_elapsed_time(start_time, end_time) / 1000.0
```

```
sum_kernel = cp.RawKernel(r'''
extern "C" __global__
void sum_kernel(const float* x, float* y, int size) {
    __shared__ float partial_sum[256];
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    float temp_sum = 0;
    while (i < size) {
        temp_sum += x[i];
        i += gridDim.x * blockDim.x;
    }
    partial_sum[tid] = temp_sum;

    __syncthreads();

    for (int s = blockDim.x / 2; s > 0; s >>= 1) {
        if (tid < s) {
            partial_sum[tid] += partial_sum[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        y[blockIdx.x] = partial_sum[0];
    }
}
''', 'sum_kernel')
```



Python for computing - GPU

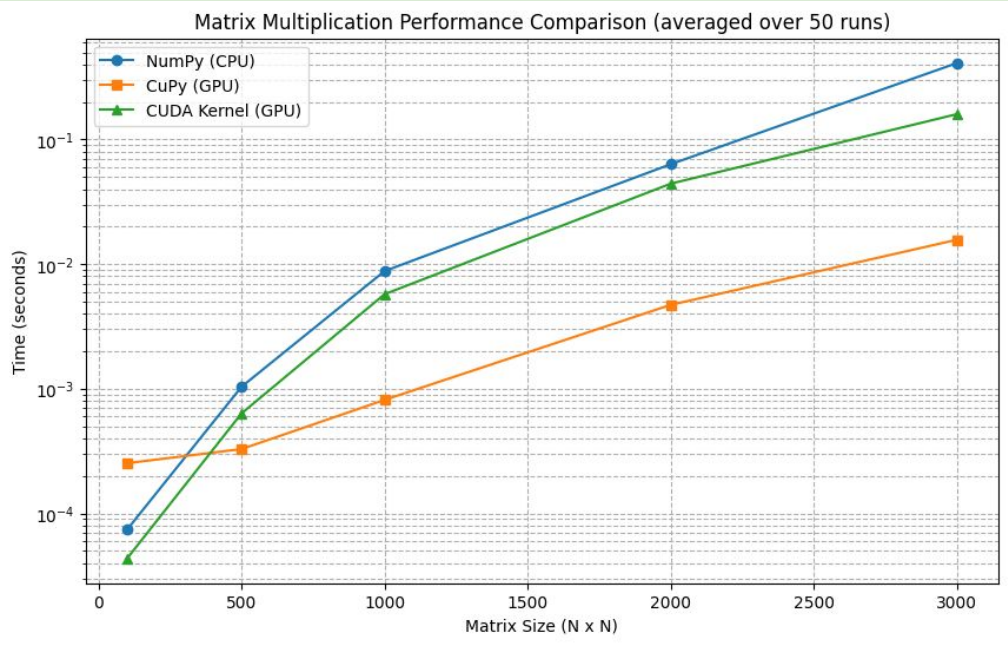
Matrix multiplication comparison:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

```python
def cupy_matmul(A, B):
    A_gpu = cp.asarray(A)
    B_gpu = cp.asarray(B)
    start_time = cp.cuda.Event()
    end_time = cp.cuda.Event()
    start_time.record()
    result = cp.matmul(A_gpu, B_gpu)
    end_time.record()
    end_time.synchronize()
    return cp.cuda.get_elapsed_time(start_time, end_time) / 1000.0
```

```python
matmul_kernel = cp.RawKernel(r'''
extern "C" __global__
void matmul_kernel(const float* A, const float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
''', 'matmul_kernel')

def cuda_kernel_matmul(A, B):
    N = A.shape[0]
    A_gpu = cp.asarray(A)
    B_gpu = cp.asarray(B)
    C_gpu = cp.zeros((N, N), dtype=cp.float32)

    block_size = (16, 16)
    grid_size = ((N + block_size[0] - 1) // block_size[0], (N + block_size[1] - 1) // block_size[1])

    start_time = cp.cuda.Event()
    end_time = cp.cuda.Event()
    start_time.record()

    matmul_kernel(grid_size, block_size, (A_gpu, B_gpu, C_gpu, N))

    end_time.record()
    end_time.synchronize()
    return cp.cuda.get_elapsed_time(start_time, end_time) / 1000.0
```

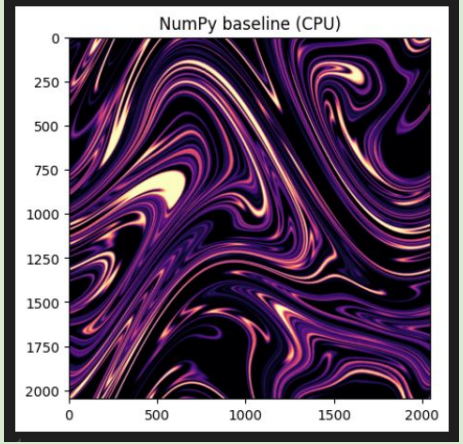


Python for computing - GPU

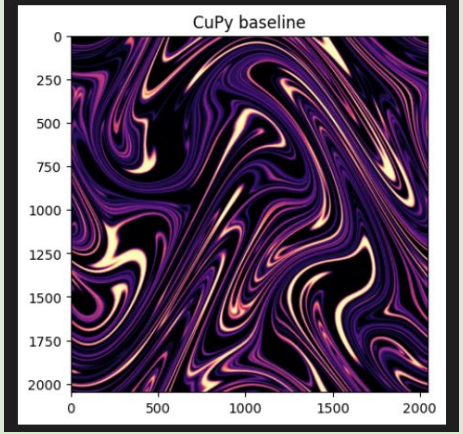
```python
for step in range(1, total_steps):
    phase_idx = min(step, phase_steps - 1)

    sin_arg_y = (x_idx / n_float) * freq + phases_y[phase_idx]
    displacement_y = amplitude * np.sin(sin_arg_y)
    y_targets = y_idx + displacement_y
    map_y = np.rint(np.mod(y_targets, n_float)).astype(np.int32)
    map_y = np.mod(map_y, n)

    sin_arg_x = (y_idx / n_float) * freq + phases_x[phase_idx]
    displacement_x = amplitude * np.sin(sin_arg_x)
    x_targets = x_idx + displacement_x
    map_x = np.rint(np.mod(x_targets, n_float)).astype(np.int32)
    map_x = np.mod(map_x, n)

    for _ in range(substeps):
        C = C[np.arange(n)[:,None], map_y]
        C = diffusion_fourier_cpu(C, sigma_step)
    for _ in range(substeps):
        C = C[map_x, np.arange(n)[None,:]]
        C = diffusion_fourier_cpu(C, sigma_step)

return C
```

```python
for step in range(1, total_steps):
    phase_idx = min(step, phase_steps - 1)

    sin_arg_y = (x_idx / n_float) * freq + phases_y[phase_idx]
    displacement_y = amplitude * cp.sin(sin_arg_y)
    y_targets = y_idx + displacement_y
    map_y = cp.rint(cp.mod(y_targets, n_float)).astype(cp.int32)
    map_y = cp.mod(map_y, n).astype(cp.int32)

    sin_arg_x = (y_idx / n_float) * freq + phases_x[phase_idx]
    displacement_x = amplitude * cp.sin(sin_arg_x)
    x_targets = x_idx + displacement_x
    map_x = cp.rint(cp.mod(x_targets, n_float)).astype(cp.int32)
    map_x = cp.mod(map_x, n).astype(cp.int32)

    for _ in range(substeps):
        C = cp.take_along_axis(C, map_y, axis=1)
        C = diffusion_fourier_gpu(C, sigma_step)
    for _ in range(substeps):
        C = cp.take_along_axis(C, map_x, axis=0)
        C = diffusion_fourier_gpu(C, sigma_step)

    VarC[step] = cp.var(C)
    MeanC[step] = cp.mean(C)

return C, MeanC, VarC
```

NumPy baseline time = 28.419 s



CuPy baseline time = 0.661 s

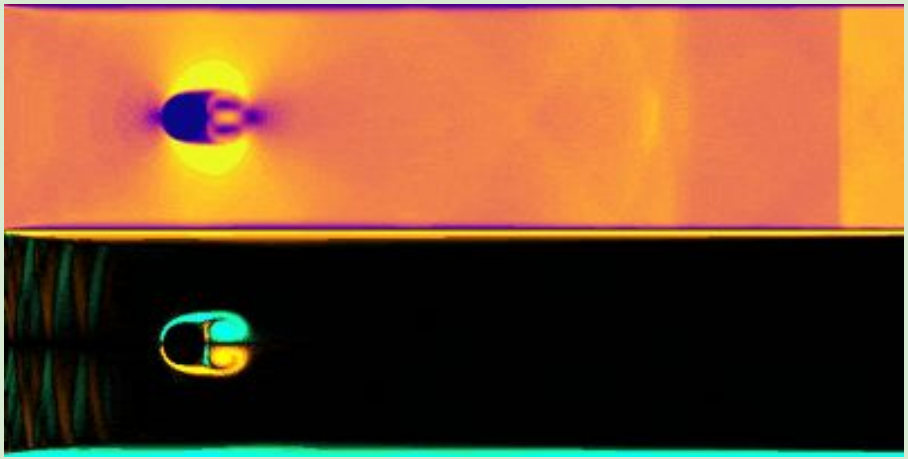


Python for computing:

Need for speed? A short guide for HPC in python

Portable custom kernels with taichi lang

Sometimes, you need to code your own kernel for GPU, but don't want to compile CUDA or even don't have a GPU. One solution: *taichi lang*



Real time LBM fluid simulation with taichi lang

Taichi lang:

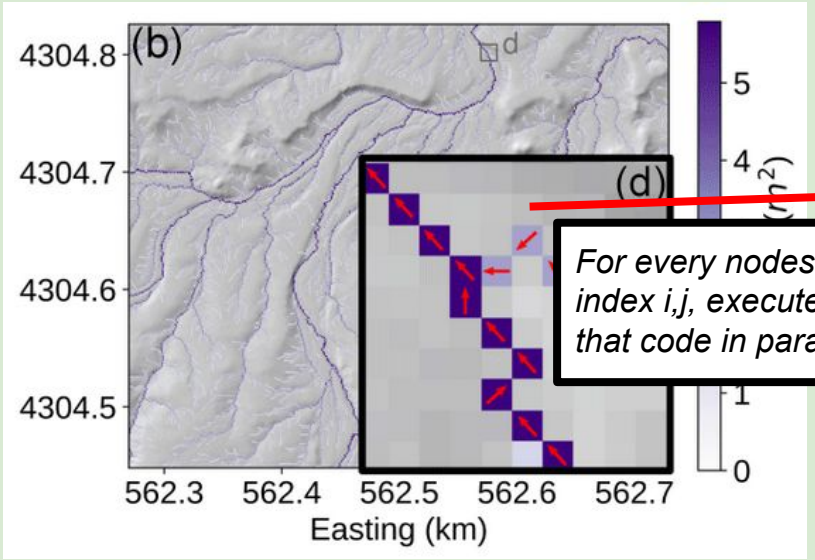
- Compiles python functions to massively parallel code
- Portable (CPU/GPU/all Oses)
- Made for simulation and CG
- Maintained by AI companies

Python for computing:

Need for speed? A short guide for HPC in python

Portable custom kernels with taichi lang

How to write a kernel?



For every nodes of index i,j, execute that code in parallel

Please compile this function to gpu

This input is a n-dimensional field

This input is a scalar

```
@ti.kernel
def gradient(dem: ti.template(), grad: ti.template(), dx: ti.f32):
    for i, j in dem:
        # central difference in x
        gx = (dem[i + 1, j] - dem[i - 1, j]) * 0.5 / dx
        # central difference in y
        gy = (dem[i, j + 1] - dem[i, j - 1]) * 0.5 / dx
        grad[i, j] = ti.math.sqrt(gx**2+ gy**2)
```

@ti.kernel: function is called from cpu, run on gpu

@ti.func: function is called from kernel, run on gpu

Python for computing:

Need for speed? A short guide for HPC in python

Portable custom kernels with taichi lang

```python
import taichi as ti

@ti.kernel
def gradient(dem: ti.template(), grad: ti.template(), dx: ti.f32):
    for i, j in dem:
        # central difference in x
        gx = (dem[i + 1, j] - dem[i - 1, j]) * 0.5 / dx
        # central difference in y
        gy = (dem[i, j + 1] - dem[i, j - 1]) * 0.5 / dx
        grad[i, j] = ti.math.sqrt(gx**2+ gy**2)


ti.init(ti.gpu) # or cpu

dem = ti.field(ti.f32, shape = (512,512))
dem.from_numpy(...) # loading existing data from numpy
grad = ti.field(ti.f32, shape = (512,512))

grad.fill(0.)

# Runs 500 times this kernel
for i in range(500):
    gradient(dem,grad,50.)
    # ... chain kernels here

res = dem.to_numpy()
```

Easy, But ⚠️⚠️⚠️:

- ⚠️ **Race conditions**
- ⚠️ **Dependency graph**
- ⚠️ **Specific algorithms**

Python for computing:

Need for speed? A short guide for HPC in python

Portable custom kernels with taichi lang

Table-like data?
Pandas polars

N-dimensional time series?
xarray

Visualisation ?
Paper like figures:
matplotlib seaborn
Big data:
Plotly datashaders
3D: pyvista blender
Performances + GUI:
Moderngl (steep learning curve)

Massive data?
More than your RAM, automation over TB of data

Processing: dask
Compressing: zarr
Organising: h5py netCDF

Machine Learning ?

Didactic: scikit-learn
Deep Learning: pytorch tensorflow
Boost: XGBoost/CatBoost
HuggingFace has python interfaces

Geospatial?
Raster manip: rasterio
Shapefiles: fiona/shapely/geopandas
Plotting: cartopy/folium

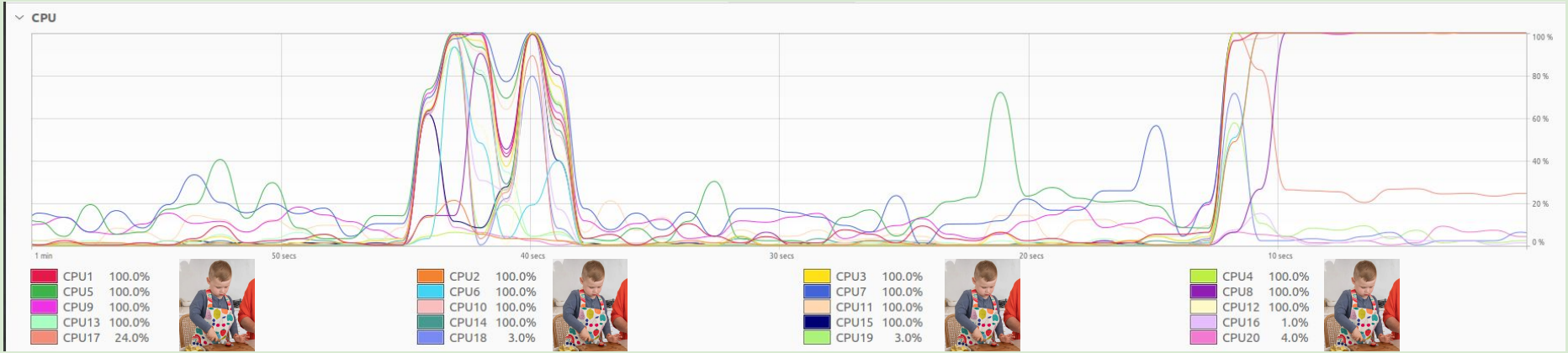
Geomorphology?
topotoolbox fastscapelib LANDLAB

High performance?
Numerical: numpy, scipy
JIT: numba, numexpr, jax
GPU: taichi, pycuda, kompute, cupy
PDE solvers: FEniCSx, pyCLAW, PETSc4Py
Binders: pybind11, cython, f2py

I started to gather many example ipynb in a repo:
https://github.com/bgailleton/workshop_python

Python for computing:

Using the multiprocessing package (level 0)




```
In [11]: runcell('main', '/home/hugoblons/Documents/py code/HugoCurentScripts/
Analysis_on_Network/Flow Analysis/Particle Tracking/
parallelParticleTracking_Network_Floriant.py')
Particle path generated after : 3.082950 s
Pathways stored after : 2.355672 s
```

15

```
In [12]: runcell('main', '/home/hugoblons/Documents/py code/HugoCurentScripts/
Analysis_on_Network/Flow Analysis/Particle Tracking/
parallelParticleTracking_Network_Floriant.py')
Particle path generated after : 5.138878 s
Pathways stored after : 7.096743 s
```

5

```
In [13]: runcell('main', '/home/hugoblons/Documents/py code/HugoCurentScripts/
Analysis_on_Network/Flow Analysis/Particle Tracking/
parallelParticleTracking_Network_Floriant.py')
Particle path generated after : 22.323298 s
Pathways stored after : 77.704291 s
```

1

Python for computing:

Using the multiprocessing package (level 0) : exemple

```python
# %% main

if __name__ == "__main__":

    totalParticuleNumber = 1.5*10**5  # Number of particles to simulate
    procs = 15                        # Number of processes to create
    particuleNumber = int(totalParticuleNumber/procs)
    args= list()
    for i in range(procs):
        args.append(particuleNumber)

    start = time.time()

    p = multiprocessing.Pool(procs)
    allParticles  = p.map(particleTacking,args)

    end = time.time()
    timetaken = end - start
    print(f"Particle path generated after : {timetaken:.6f} s ")
    start = time.time()

    pathWays  = p.map(PathwaysList,allParticles)

    end = time.time()
    timetaken = end - start
    print(f"Pathways stored after : {timetaken:.6f} s ")
    p.close()
```

1 Create a pool objects

- Control a pool of workers process to which jobs can be submitted

2 Use .map(f,args)

- Return an iterator applying a function to every item in args

3 Close pool

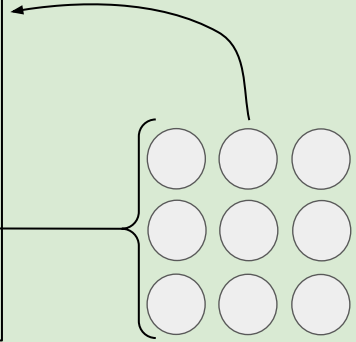
- It's the end

Python for computing:

Using the multiprocessing package (level 0) :

Asking for the name :

```
def f (args) :
        doSomething

if __name__ == '__main__' :
        Spawn 9 python interpreters
        Use f()
```



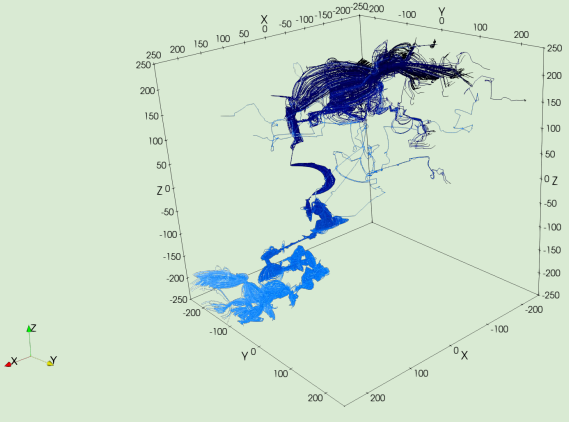
Closing :

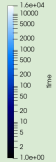
Process Name	User	% CPU
spyder	hugoblons	0.03
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00
python	hugoblons	0.00

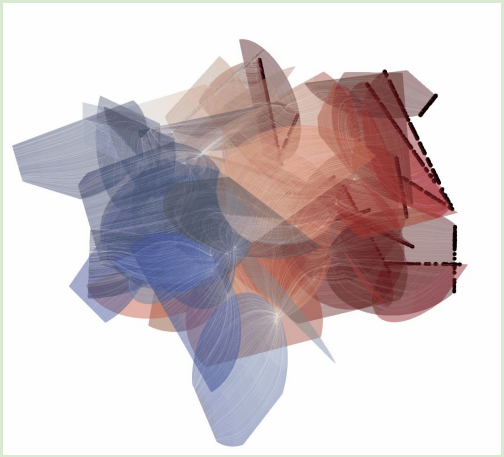
Python for computing - Vizualisation

https://docs.pyvista.org/index.html

- Python interface for vtk
- Used for 3D plots and animation (GIF / Videos)







Python for data analysis : read/write huge data with h5py

```
import h5py


Data=np.zeros((1000,1000,1000))


F=h5py.File('hugedata.h5','w')
F.create_dataset('3Dimage',data=Data,chunks=(20,50,50), compression="gzip")
F.attrs['Place']=’Rennes’
F.attrs['Times']=[10,100,1000]
F.close()

F=h5py.File('hugedata.h5','r')
Fmemory=F['3Dimage'][10:30,:,50:100]
```

Python for data analysis : read tabular CSV data using pandas

Name of columns become the variable names

```
import pandas as pd

fpath = ‘myfile.csv’

Data=pd.read_csv(fpath,delimiter=’;’,skiprows=5,encoding=’utf-8’,engine='python')

Value=pd.to_numeric(Data[‘col_Value’],downcast='float',errors='coerce').to_numpy()

Time=pd.to_datetime(Data[“col_time"],errors='coerce',dayfirst=True).fillna(value=pd.Timestamp('20000101'))
```

Python for image analysis : Use dictionary to read/write arbitrary parameter file

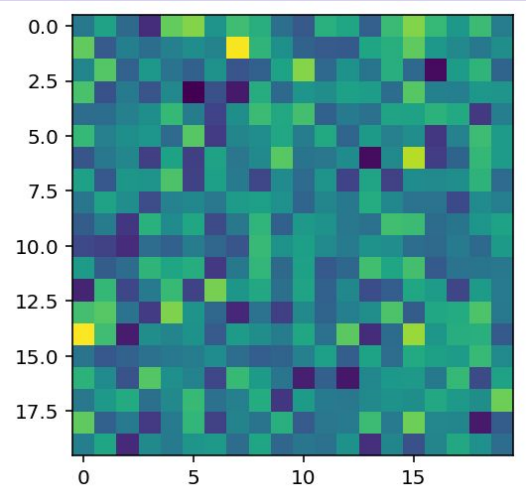
```python
def read_parameter_file(filename):
    par=[{}]
    if os.path.exists(filename):
        with open(filename) as f:
            for line in f:
                s=search('{varname:w} {varvalue:g}',line)
                if (s != None): par[0][str(s['varname'])]=s['varvalue']
        #   print('Parameter file read !')
    else:
        print('WARNING: no parameter file exists. Taking default values! ')
    return par
```

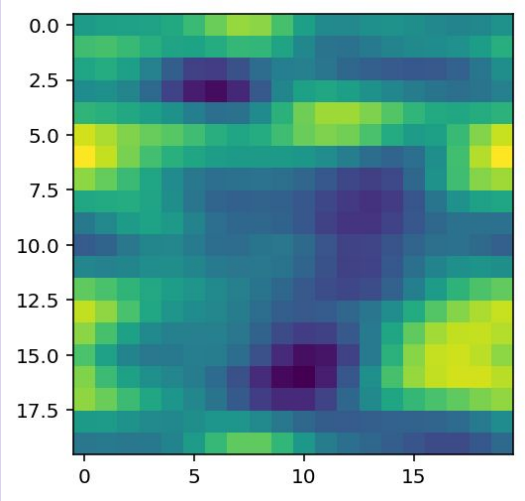
```
# Parameter file generated by TracTrac Python v3.0 (22/05/2021) |

# Video loops
vid_loop 0.0            # Number of loops over frames to train mo

# Image Processing
ROIxmin 0               # Region of interest for tracking (xmin)
ROIymin 0               # Region of interest for tracking (ymin)
ROIxmax 332             # Region of interest for tracking (xmax)
ROIymax 165             # Region of interest for tracking (ymax)
BG 0.0          # (0 or 1, default 0) Use background subtraction
BGspeed 0.02            # (0 to 1, default 0.01) adaptation speed
noise 0.0               # (0 or 1, default 0) Use median filterin
noise_size 3            # (3 or 11, default 3) Size of median ker

# Object Detection
peak_th_auto 0.0        # (0 or 1) Automatic object detection thr
```

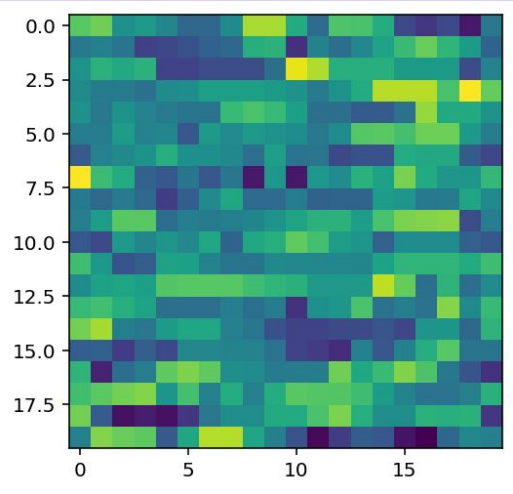
Par[‘myparameter’]

Python for image analysis : filtering with scipy.ndimage

Import numpy as np
Import scipy
C=np.random.randn(20,20,20)
Cfilt = scipy.ndimage.gaussian_filter(C, [1,2,5], mode='wrap')
Cfilt_med = scipy.ndimage.median_filter(C, [1,3,5], mode='wrap')







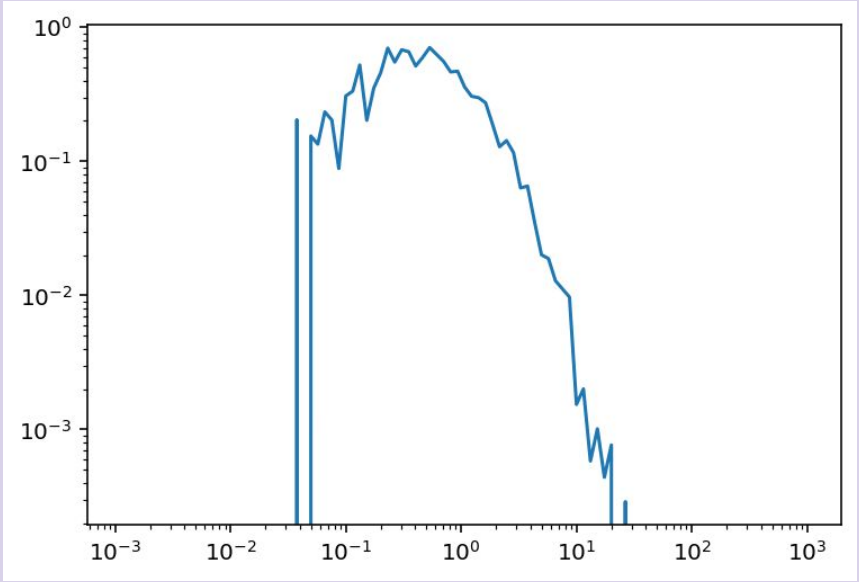
Python for signal analysis : fast average along axis

Flatten data :

```
C=np.random.randn(100,100,365)

# Mean per days

M=np.nanmean(C.reshape(-1,C.shape[2]),axis=0)

# Spatial Mean per year

M=np.nanmean(C,axis=2)
```

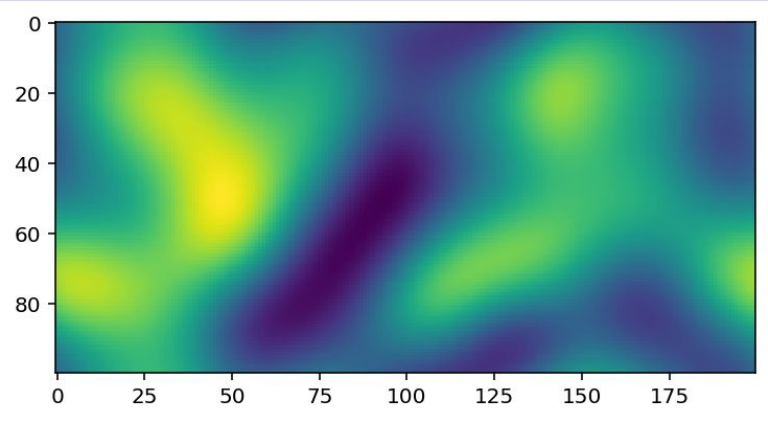
Python for signal analysis : Histogram / PDF

```
bins=np.logspace(-3,3,100)

var=np.exp(np.random.randn(1000))

x,h=np.histogram(var,bins,density=True)

plt.plot(x[1:],h)

plt.yscale('log')

plt.xscale('log')
```

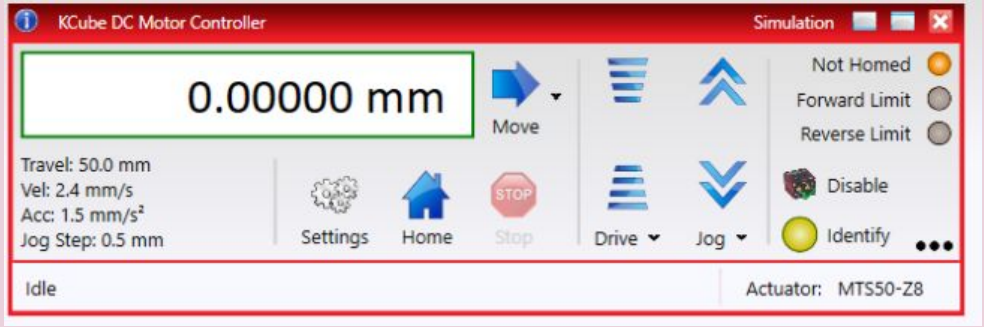


Python for signal analysis : ND fft

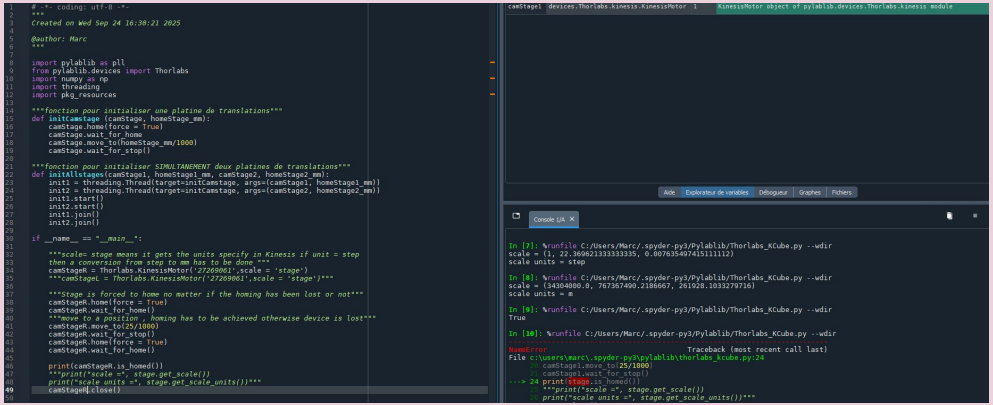
```
C = np.random.randn(100,200)

from scipy.fft import fft, ifft,fftfreq

ky=2*np.pi*np.tile(fftfreq(C.shape[0], d=1.0/C.shape[0]),(C.shape[1],1)).T
kx=2*np.pi*np.tile(fftfreq(C.shape[1], d=1.0/C.shape[0]),(C.shape[0],1))
k=np.sqrt(ky**2+kx**2)

fC=fft(fft(C,axis=0),axis=1)

sigma2=0.01
fC=fC*np.exp(-k**2*sigma2) # gaussian filter

C=np.real(ifft(ifft(fC,axis=1),axis=0))
plt.imshow(C)
```



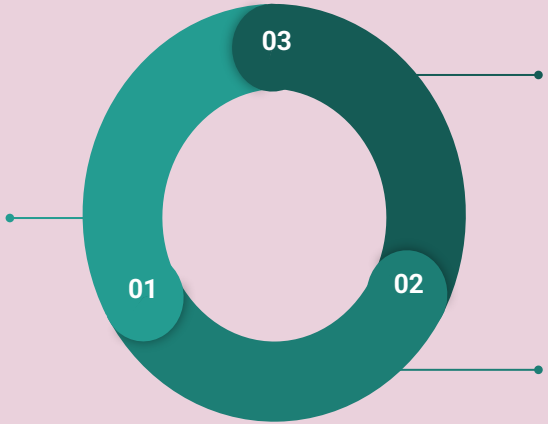
Python for automation



VS



Python for automation



Protocol (workflow)
Python +

Data management (storage, file format ...)

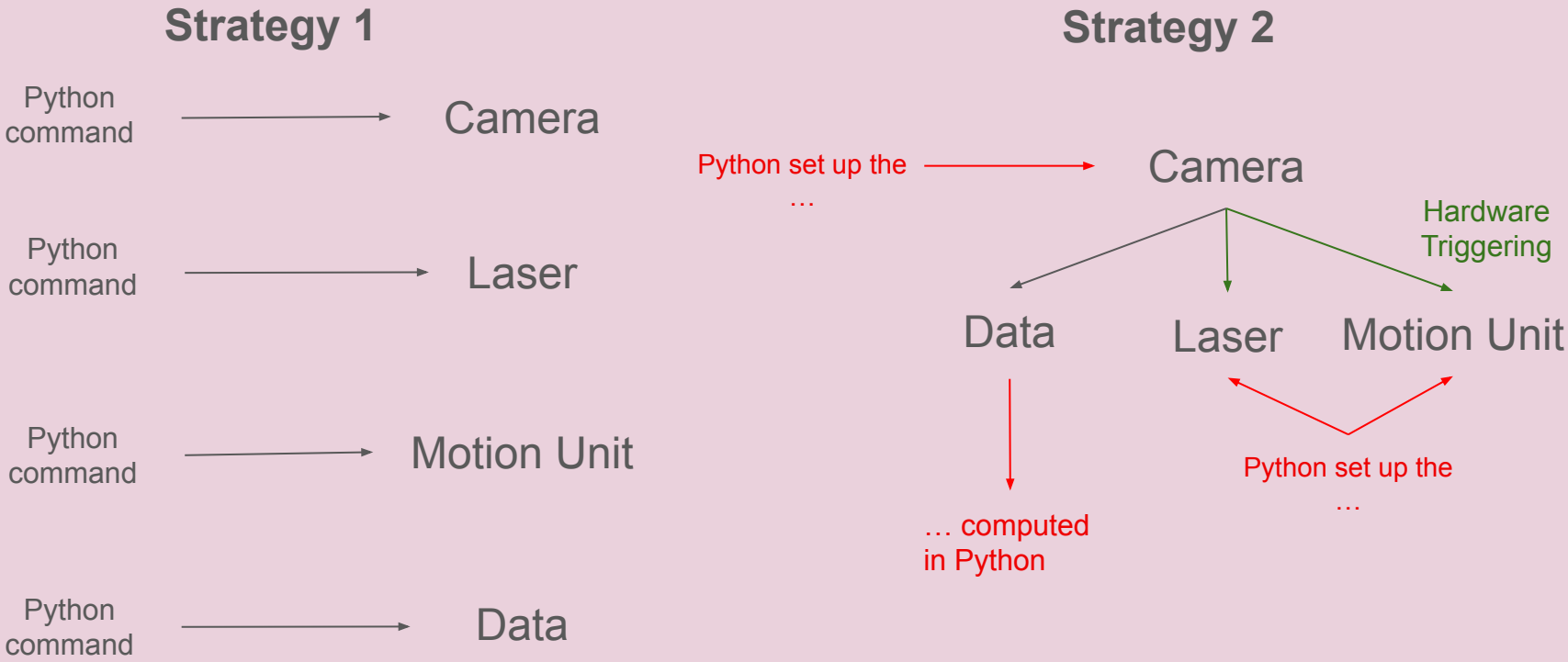
Python +

Control (Direct command of each equipment, timing ...)

Python ~

Trade-off between:
- Python compatibility
- Hardware specification
- Experiment complexity

Python for automation



Python for automation

What it is?

- Auto-control of equipment
- Automatic acquisition of data (continuous, discrete, …)

What are the advantage(s)?

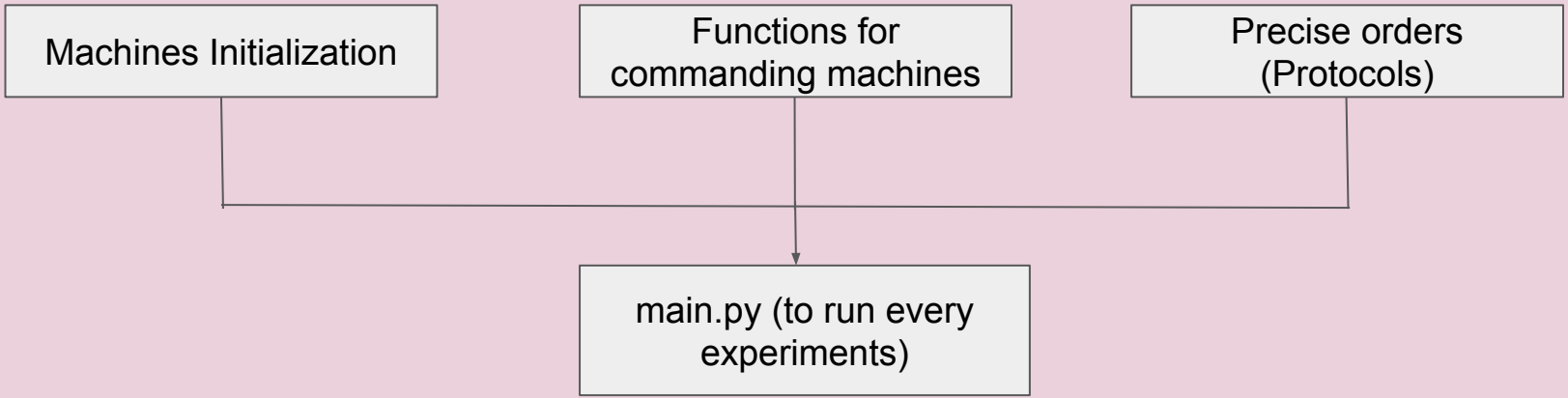
- Synchronization of data obtained from different devices
- Less failures/stoppages, no manual response time, … ← issues fixed by automation

Challenge(s)?

- Communication protocol varies for each device, device settings, PC system
- Scripting is specific to each experiment, should not terminate when in operation

Python for automation : Proposition for a (very) practical and simple framework

- 3 coding BLOCKS; 3 PURPOSES:
- 1 for initializing communications with devices
- 1 to setup all your functions to send orders to machines
- 1 per experimental protocols



Machines Initialization

Librairies to download :
pyserial

RS 232
Old standard but most common

- Requires physical COM ports
- Initialize by a SERIAL communication, specify BAUD RATE (e.g. 9600) and parity bits (e.g. 1)
- Used with passive USB adaptor but requires COM ports

- Emulates COM ports
- Still need to define BAUD RATE & Parity

Machines Initialization : The case of non-generic devices

- For complex systems (Cameras, LASER, Spectrophotometers, Acquisition cards, Pressure controllers…etc),
- Proprietary protocols,
- **READ THE DOC !!**
- API / DLL / SDK files (should be given by manufacturer)

Librairies to download :

- **Pylablib (Andor, Hamamatsu, Thorlabs)**
- **PyVisa (NI-MAX, NI-VISA driver based)**
- **PyMeasure**

!! READ THE DOC !!

Machines Initialization

- Define COM PORT (**string**)
- Define Baud rate (**int**)
- Use serial.Serial()

```
import serial
SCALE_COM_PORT = 'COM21'
LASER_COM_PORT = 'COM8'
PUMP_COM_PORT = 'COM15'
OB1_COM_PORT = 'COM12'
BAUD_RATE_SCALE = 9600
BAUD_RATE_LASER = 9600
BAUD_RATE_PUMP = 9600
BAUD_RATE_OB1 = 115200
 ser = serial.Serial(LASER_COM_PORT, BAUD_RATE_LASER, timeout=1)
 scale_ser = serial.Serial(SCALE_COM_PORT, BAUD_RATE_SCALE, timeout=1)
 pump_ser = serial.Serial(PUMP_COM_PORT, BAUD_RATE_PUMP, timeout=1)
 ob1_ser = serial.Serial(OB1_COM_PORT, BAUD_RATE_OB1, timeout=1)
```

To find the COM port associated, open the **device manager**, open **COM ports** and **plug & unplug** the device to see what COM port is associated to it

Function for commanding machines

Generic function

```
def send_command(ser, command, wait=0.1):
    full_command = f"{command}\r\n"
    ser.write(full_command.encode())
    time.sleep(wait)
    return ser.read_all().decode(errors='ignore').strip()
GREEN_LASER_ON = "L2 L=1"
GREEN_LASER_OFF = "L2 L=0"
RED_LASER_ON = "L1 L=1"
RED_LASER_OFF = "L1 L=0"
BLUE_LASER_ON = "L3 L=1"
BLUE_LASER_OFF = "L3 L=0"
PURPLE_LASER_ON = "L4 L=1"
PURPLE_LASER_OFF = "L4 L=0"
BLUE_LASER_POWER_ASK = "L3 ?C"
BLUE_LASER_POWER = "L3 C 100"
SHUTTER_OPEN = "SH1=1"
SHUTTER_CLOSE = "SH1=0"
```

- ser : opened serial communication,
- command : string object (**READ THE DOC**)
- eg : **IP? / L1 C 75 / SH1=0**
- “wait=0.1” is optional but good to prevent bugs
- Most machines return an answer to ser.write(). Need to .encode() the string

Precise orders (protocols)

```
send_command(ser, SHUTTER_OPEN)
send_command(ser, BLUE_LASER_ON)
send_command(ser, BLUE_LASER_POWER)
ob1_send(ob1_ser, command: "SET_PRESS 2 100")
pump_start(pump_ser, fluo_injection)
save_parameters()
j = 0
start_time = time.time()
while time.time() - start_time < temps:
    i = cam.grab(1)
```

Here, you just call for whatever you need. Few ideas :

- Saving parameters in TXT file in acquisition folder,
- Python TIME librairy for automation
- **ENJOY ONE CLICK experiments**

Paul Lafargue in "The Right to be lazy", 1883 :

They do not yet understand that the machine is the redeemer of humanity, the God who will redeem man from the sordidæ artes and from wage-labor, the God who will give him leisure and freedom.

Python for automation: Weight Acquisition Example

```
def acqwght(case, imag, pres, wght, lasr):
    global timewghts, wghtwghts
    fdump = []
    ...
    for iwght in range(wght['scale_count']):
        fdump.append(open(wght['raw_folder']+'wght_'+str(iwght).zfill(4)+'.txt',"w"))
        fdump[iwght].close()
        if wght['scale_manu'][iwght]=='ohaus':
            serlwght.append(serial.Serial(wght['serial_ports'][iwght], wght['baud_rates'][iwght], timeout=1))
        ...
    while time.time() < case['starttime s']+60*case['exptime_min']:
        for iwght in range(wght['scale count']):
            datastream0 = serlwght[iwght].readline()
                ...
                if not '?' in datastream:
                    wghtstream, timestream = datastream.split(" g    ACCEPT ")
                    ...
                    timestream = float(timestream[0])*3600+float(timestream[1])*60+float(timestream[2])
                    ...
                    timewghts[iwght].append(timestream)
                    ...
                if wght['dropmode'] == 'incr':
                    fstream = open(wght['raw_folder']+'wght_'+str(iwght).zfill(4)+'.txt',"a")
                    fstream.write(str(timestream)+","+str(wghtstream)+","+acttimestream+"\n")
                    fstream.close()
                ...
    for iwght in range(wght['scale_count']):
        serlwght[iwght].close()
```

← defining as function enables passing control parameters and simultaneous runs

← global variables with other functions (plotter in this case)

↙ initiation of output file (creates a new one)

↙ specification of serial port

← continuous data acquisition for experimentation time

← error contingency

↖ data parsing

↖ data dump to output file

← closing port

Python for automation: Parameter Specification

- Separate routine for specifying parameters
- Can be saved as pickle ← input for the 'main.py' run

Host specific parent folder, heirarchy maintained across systems →

```
if socket.gethostname() == "gs-lab-mobile-precision-3591":
    params_folder = "D:/InstrControlParams/"
...
os.makedirs(params_folder, exist_ok=True)
...
```

Dictionaries for parameters, 'case' dictionary has parameters common to all devices →

```
case = {}
case['caselabel'] = "test" # case label
case['exptime_min'] = 80.0 # experiment time in minutes
...
```

Example: Dictionary for imaging →

```
imag = {}
imag['do'] = False # whether to do imaging in experiment
imag['camera_manu'] = 'hama' # camera manufacturer
...
imag['exposuretime_ms'] = 10.0 # base exposure time
...
```

Dictionaries for other equipment in the experiment →

```
wght = {}
...
lasr = {}
...
```

All parameters saved in pickle file, to be read & utilised by main.py (the script for executing a run) →

```
with open(params_folder+"params.pkl",'wb') as parampklfile:
    pickle.dump([   case, imag, pres, wght, lasr    ],  parampklfile)
```

Python for automation: Execution

Time stamped base folder – prevents accidental overwriting →

```
time_stamp = str(datetime.now()).split() ...
case['case_name'] = time_stamp+"_"+case['caselabel']
case['case_folder'] = parent_folder+case['case_name']+"/"
...
```

Folder hierarchy generation (separate folder for multiple file acquisition) →

```
if imag['do']:
    imag['raw_folder'] = case['case_folder']+'imagraw/'
...
```

Functions defined for each 'function' →

```
def acqwght(case, imag, pres, wght, lasr):
def acqimag(case, imag, pres, wght, lasr):
def pltwght(case, imag, pres, wght, lasr):
def pltimag(case, imag, pres, wght, lasr):
...
```

Specifying thread for each function allows separate failure of each function in case of error →

```
# initiating acquisition and plotting
if __name__ =="__main__":
    case['starttime_s'] = time.time()
    thrd_acqwght = threading.Thread(target=acqwght, args=(case, imag, pres, wght, lasr))
    thrd_acqimag = threading.Thread(target=acqimag, args=(case, imag, pres, wght, lasr))
    thrd_pltwght = threading.Thread(target=pltwght, args=(case, imag, pres, wght, lasr))
    thrd_pltimag = threading.Thread(target=pltimag, args=(case, imag, pres, wght, lasr))
```

Nested plotting prevents 'dummy' empty files →

```
    if (wght['do']):
        thrd_acqwght.start()
        if wght['plot_live']:
            thrd_pltwght.start()
...
```

Case-specific parameters dumped in pickle, useful for postprocessing →

```
with open(params_folder+"params_endrun.pkl",'wb') as parampklfile:
    pickle.dump([   case, imag, pres, wght, lasr    ],parampklfile)
```