Example Class 3 — Arrays, spectra, profiles, services and containers

C1 Research Computing — Michaelmas 2026
MPhil in Data Intensive Science and MPhil in Computational Biology

Exercises

Views and copies

`b = np.arange(24.0).reshape(4, 6)` holds 0.0 to 23.0 in row-major order, so `b[i, j]` is $6i + j$. A view reads the buffer of `b` through its own shape and strides; a new array owns its buffer. `np.moveaxis(a, s, d)` moves axis `s` of `a` to position `d`, keeping the order of the others.

(a) For each of `np.moveaxis(b, 0, -1)`, `b.T`, `b[:, ::2]`, `b.reshape(3, 8)`, `b[[0, 2]]` and `b * 1`, give the shape, and say whether it is a view of `b` or a new array.

(b) Each statement below runs on a fresh `b`. Give the element of `b` that it changes, with its old value, or say that `b` is unchanged: (i) `b.reshape(3, 8)[1, 0] = -1.0`; (ii) `b[[0, 2]][0, 0] = -1.0`; (iii) `np.moveaxis(b, 0, -1)[5, 3] = -1.0`.

Half spectra of real signals

The discrete Fourier transform of $x_0, \ldots, x_{N-1}$ is

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-2\pi i k n / N}, \qquad k = 0, 1, \ldots, N-1.$$

`np.fft.fft(x)` returns all N values, and `np.fft.rfft(x)` those with $0 \le k \le N/2$.

(a) Show, as one bullet point, that X_{N-k} is the complex conjugate of X_k when `x` is real.

(b) Give the number of values that `np.fft.rfft` returns for $N = 1000$ and for $N = 1001$, and how many of them are real for every real `x`.

Shapes through an FFT smoother

`smooth`, in Listing 1, smooths an array along one axis by a Gaussian of standard deviation $w/4$, as a product in Fourier space; `np` is NumPy and `math` the standard module.

- `np.fft.rfft(y, n=N, axis=a)` pads `y` with zeros to N values along axis `a`, and returns `N//2 + 1` complex values along it.
- `np.fft.irfft(Y, n=N, axis=a)` turns them back into N real values.
- `np.fft.rfftfreq(N)` returns `N//2 + 1` frequencies.
- `np.take(z, range(n), axis=a)` keeps the first n values along axis `a`.

```
1  def smooth(y, w, axis=0):
2      """Smooth y along axis by a Gaussian of standard deviation w/4."""
3      n = y.shape[axis]
```

```
4      N = 64 * math.ceil((n + w) / 64)
5      Y = np.fft.rfft(y, n=N, axis=axis)
6      f = np.fft.rfftfreq(N)
7      shape = [1] * y.ndim
8      shape[axis] = f.size
9      Y = Y * np.exp(-((np.pi * w * f) ** 2) / 8).reshape(shape)
10     z = np.fft.irfft(Y, n=N, axis=axis)
11     return np.take(z, range(n), axis=axis)
```

Listing 1: `smooth.py`: a Gaussian smoother along one axis.

(a) Give `N` for $n = 100$, $w = 20$, and for $n = 1000$, $w = 30$.
(b) Complete the table for `smooth(y, 20)`, where `y` has shape (100, 3).

name	value or shape
`n` after line 3	
`N` after line 4	
`Y` after line 5	
`f` after line 6	
`shape` after line 8	
`Y` after line 9	
`z` after line 10	
value returned at line 11	

A profile of the Gray–Scott model

`step`, in Listing 2, advances the Gray–Scott model of Lecture 18 by one time step on two periodic $n \times n$ grids, `u` and `v`. `run(n, nsteps)`, at line 14 of `grayscott.py`, creates the grids and calls `step` `nsteps` times.

```
1  import numpy as np
2
3  def laplacian(a):
4      return (np.roll(a, 1, axis=0) + np.roll(a, -1, axis=0)
5              + np.roll(a, 1, axis=1) + np.roll(a, -1, axis=1) - 4 * a)
6
7  def step(u, v, F=0.035, k=0.06, Du=0.16, Dv=0.08, dt=1.0):
8      Lu, Lv = laplacian(u), laplacian(v)
9      uvv = u * v * v
10     u += dt * (Du * Lu - uvv + F * (1 - u))
11     v += dt * (Dv * Lv + uvv - (F + k) * v)
12     return u, v
```

Listing 2: `grayscott.py`, lines 1–12.

On one laptop, a new interpreter ran

```
import cProfile, pstats
from grayscott import run
pr = cProfile.Profile(); pr.runcall(run, 256, 2000)
pstats.Stats(pr).strip_dirs().sort_stats("tottime").print_stats(6)
```

and printed

```
         198006 function calls in 1.009 seconds

   Ordered by: internal time
   List reduced from 19 to 6 due to restriction <6>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2000    0.413    0.000    1.006    0.001 grayscott.py:7(step)
     4000    0.324    0.000    0.592    0.000 grayscott.py:3(laplacian)
    16000    0.240    0.000    0.267    0.000 numeric.py:1226(roll)
    16000    0.013    0.000    0.021    0.000 numeric.py:1427(normalize_axis_tuple)
    32000    0.003    0.000    0.005    0.000 numeric.py:1476(<genexpr>)
        1    0.003    0.003    1.009    1.009 grayscott.py:14(run)
```

`ncalls` counts calls; `tottime` is the time in a function's own lines, not in the functions it calls; `cumtime` is the time from call to return, the functions it calls included. Each `percall` divides the column before it by `ncalls`. Times are in seconds.

(a) Give the function whose own lines take the most time, the lines of Listing 2 that take it, and their operators.

(b) Give the rows whose time is in the `cumtime` of `step`, and check the sum.

(c) Give, as one bullet point, why `u * v * v` at line 9 has no row.

(d) Give the line of `step` that takes the most time, the functions it calls included, with its time, and the lines to optimise first.

(e) Line 9 becomes `uvv = u * v ** 3`. Profiled the same way, the first two rows become

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2000    1.325    0.001    1.918    0.001 grayscott.py:7(step)
     4000    0.323    0.000    0.593    0.000 grayscott.py:3(laplacian)
```

Give the operator responsible and, as one bullet point, why it costs more than the multiplications it replaces.

Status codes of a small service

The FastAPI service of Listing 3 returns the dominant period of a series recorded at a station. A client posts the series as a file of numbers, one per line, to a URL such as `/stations/ely/period?taper=flat`. `STATIONS` gives the sampling interval, and `taper` chooses the window. `fine_period`, from `spectra.py`, scans 20 000 trial periods and returns the one with the most power.

```python
from typing import Literal
import numpy as np
from fastapi import FastAPI, HTTPException, UploadFile
from spectra import fine_period

STATIONS = {"cam": 0.5, "ely": 2.0}      # seconds between two samples
app = FastAPI()

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/stations/{station}/period")
async def period(station: str, file: UploadFile,
                 taper: Literal["hann", "flat"] = "hann"):
    if station not in STATIONS:
        raise HTTPException(404, "no such station")
    raw = await file.read()
    try:
        y = np.array(raw.split(), dtype=float)
    except ValueError:
        raise HTTPException(422, "expected one number per line")
    if not np.isfinite(y).all():
        raise HTTPException(422, "nan or inf in the data")
    if y.size * STATIONS[station] < 90:
        raise HTTPException(422, "less than 90 s of data")
    return {"period": fine_period(y, STATIONS[station], taper)}
```

Listing 3: `service.py`.

FastAPI first matches the path to a route, then converts each declared parameter to its type: `station` from the path, `file` from the form, `taper` from the query. A parameter with a default may be left out, and `Literal["hann", "flat"]` accepts only those two strings. If a parameter is missing or invalid, FastAPI answers 422 before the endpoint runs.

`HTTPException(code, detail)` ends the request with that code, and a normal return gives 200. `np.array(strings, dtype=float)` accepts `"nan"` and `"inf"`, and raises `ValueError` for `"1,5"`.

Each request below differs from a valid one, which posts 200 values to `/stations/cam/period`. Give its status code, and the line of Listing 3 that decides it.

	request	status	line
(A)	to `/stations/york/period?taper=blackman` instead		
(B)	to `/stations/ely/period`, with only the first 45 values		
(C)	to `/stations/york/period` instead		
(D)	with the tenth value written as `1,5`		
(E)	with only the first 100 values, the tenth written as `nan`		

A blocked event loop

A client posts 5000 values to `/stations/cam/period` (Listing 3), in ten chunks 0.1 s apart. For this series, `fine_period` at line 27 takes about 1.5 s on one laptop, and never awaits. A second client sends `GET /health` four times. The three sent while the values arrive are answered in 10 to 22 ms; the one sent after they have arrived waits 1.15 s.

FastAPI's single event-loop thread receives every request, reads each upload in full before it calls the endpoint, and runs the `async def` endpoints; it moves to another request only where it awaits, as it does between the chunks of an upload. It hands plain `def` endpoints to a pool of worker threads. `await asyncio.to_thread(f, *args)` runs `f(*args)` in a worker thread.

(a) Give two bullet points: why the GETs sent during the upload are answered at once, and why the last one waits.

(b) Line 14 alone is changed to a plain `def`. What happens when uvicorn imports the service? Give one bullet point on why.

(c) Give a replacement for line 27, with any import it needs, that lets every GET through while `period` stays `async def`.

Layers and the build cache

The service is shipped as an image built from the `Dockerfile` of Listing 4. Its folder holds `service.py`, `spectra.py`, `requirements.txt`, the `Dockerfile`, `.git` and `__pycache__`.

```
1 FROM python:3.13-slim
2 WORKDIR /srv
3 COPY . .
4 RUN pip install --no-cache-dir -r requirements.txt
5 EXPOSE 8080
6 CMD ["uvicorn", "service:app", "--host", "0.0.0.0", "--port", "8080"]
```

Listing 4: `Dockerfile` of the service.

`docker build` sends the folder as the build context, and `COPY . .` copies all of it, the `Dockerfile` included. A layer comes from the cache unless its instruction, or a file it copies, has changed; that instruction and every later one then rerun. `EXPOSE` and `CMD` only record settings.

(a) Give the instructions, among lines 3 and 4, that rerun after an edit to `service.py`, to `requirements.txt`, and to line 6 only.

(b) Give instructions to replace lines 3–4 so that an edit to `service.py` or `spectra.py` reuses the installed dependencies. Which of them does an edit to `requirements.txt` rerun?

(c) The two modules move into a package, `periodsvc`, whose `pyproject.toml` lists the four requirements as dependencies. `pip install .` installs the package of the current folder and any missing dependency; `pip install --no-deps .` installs the package alone. Give the instruction to add after those of (b), so that the image holds the package and a code edit reinstalls no dependency.