Python and AI/ML for Weather, Climate and Environmental Applications

Let us enjoy playing with Python and AI/ML!

Five-Day Schedule Overview

Time	Day 1	Day 2	Day 3	Day 4	Day 5
09:00–10:00	Opening by ECMWF DG, Start: Coding & Science in the Age of AI	Neural Network Architectures	Diffusion and Graph Networks	MLOps Foundations	Model Emulation, AIFS and AICON
10:00–11:00	**Lab:** Python Startup: Basics	**Lab:** Feed-forward and Graph NNs	**Lab:** Graph Learning with PyTorch	**Lab:** Containers and Reproducibility	**Lab:** Emulation Case Studies
11:00–12:00	Python, Jupyter and APIs	Large Language Models	Agents and Coding with LLMs	CI/CD for Machine Learning	AI-based Data Assimilation
12:00–12:45	**Lab:** Work environments, Python everywhere	**Lab:** Simple Transformer and LLM Use	**Lab:** Agent Frameworks	**Lab:** CI/CD Pipelines	**Lab:** Graph-based Assimilation
12:45–13:30			**Lunch Break**		
13:30–14:30	Visualising Fields and Observations	Retrieval-Augmented Generation (RAG)	DAWID System and Feature Detection	Anemoi: AI-based Weather Modelling	AI and Physics
14:30–15:30	**Lab:** GRIB, NetCDF and Obs Visualisation	**Lab:** RAG Pipeline	**Lab:** DAWID Exploration	**Lab:** Anemoi Training Pipeline	**Lab:** Physics-informed Neural Networks
15:30–16:15	Introduction to AI and Machine Learning	Multimodal Large Language Models	MLflow: Managing Experiments	**The AI Transformation**	Learning from Observations Only
16:15–17:00	**Lab:** Torch Tensors and First Neural Net	**Lab:** Radar, SAT and Multimodal Data	**Lab:** MLflow Hands-on	**Lab: How work style could change**	**Lab:** ORIGEN and Open Discussion
17:00–20:00				Joint Dinner	

Working with Meteorological Data for AI and ML

Context

- Python-based workflows
- Meteorological fields for AI and machine learning
- Real operational datasets

Why this lecture?

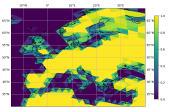
- ML needs structured input data
- Models operate on fields, not files
- Visualization builds physical intuition

Core skills we build

- Access meteorological fields
- Understand grids and metadata
- Convert and preprocess data
- Visualize fields and observations

Outcome

- Data ready for ML pipelines
- Reproducible Python workflows



Installing and Exploring ecCodes

What is ecCodes?

- ECMWF library for GRIB handling
- Standard tool in operational NWP
- Python interface available

Why we need it

- Forecast data comes as GRIB
- We need access to fields and metadata
- Independent of model (IFS, ICON, . . .)

Installation and test

```
# Linux / WSL
sudo apt update
sudo apt install eccodes libeccodes-tools

# macOS (Homebrew)
brew install eccodes

# Python bindings
pip install eccodes

# Test installation
grib_ls -V
python -c "import eccodes; print(eccodes.codes_get_api_version())"
```

Using ecCodes

- See what is inside a GRIB file
- Identify fields
- Get size and structure

Key idea

- A GRIB file contains multiple messages
- Each message holds one field
- ecCodes iterates message by message

Inspecting a GRIB file

```
1  import eccodes
2
3  f = open("icon_eu_t2m_latest.grib2", "rb")
4  while True:
5      gid = eccodes.codes_grib_new_from_file(f)
6      if gid is None:
7          break
8
9      short = eccodes.codes_get(gid, "shortName")
10     level = eccodes.codes_get(gid, "level")
11     size  = eccodes.codes_get_size(gid, "values
       ")
12     print(short, level, size)
13     eccodes.codes_release(gid)
14 f.close()
```

IFS Open Data — Access via Client Library

- IFS data is not exposed as simple directories
- Access is service-based and load-controlled
- Small subsets can be retrieved efficiently

Key idea

- Use ECMWF's official open-data client
- Request fields by parameter, step, level
- Result is a standard GRIB2 file

Downloading IFS fields with ecmwf-opendata

```
from ecmwf.opendata import Client

client = Client(
    source="ecmwf",
    model="ifs",
)

client.retrieve(
    time=0,
    type="fc",
    step=24,
    param=["2t", "msl"],
    target="ifs_2t.grib2")
```

Plotting an IFS Field (Regular Grid)

- IFS open data uses a regular lat–lon grid
- Field values form a 2-D NumPy array
- Plotting is straightforward

Important

- Grid size is fixed by the model
- figsize controls readability only

Plotting IFS 2 m temperature

```
1  import eccodes
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  f = open("ifs_2t.grib2", "rb")
6  gid = eccodes.codes_grib_new_from_file(f)
7
8  nx = eccodes.codes_get(gid, "Ni")
9  ny = eccodes.codes_get(gid, "Nj")
10 values = eccodes.codes_get_array(gid, "values")
11 field = values.reshape(ny, nx)
12
13 plt.figure(figsize=(7, 3.5))
14 plt.imshow(field, origin="lower")
```

Cartopy — Map Projections for Meteorological Fields

- Cartopy provides projection-aware plotting
- Separates data coordinates from map projection
- Built on top of matplotlib

In practice

- Define projection, e.g. `PlateCarree`
- Coastlines, borders, land masks
- Define geographic extent

Cartopy-based map plot

```
1  import cartopy.crs as ccrs
2  import cartopy.feature as cfeature
3
4  fig, ax = plt.subplots( figsize=(10,5),
5      subplot_kw={"projection": ccrs.
       PlateCarree()})
6
7  ax.coastlines()
8  ax.add_feature(cfeature.BORDERS)
9
10 ax.pcolormesh(lon, lat, field,
11     transform=ccrs.PlateCarree(),
12     cmap="jet" )
```

ICON Open Data — File-Based Access

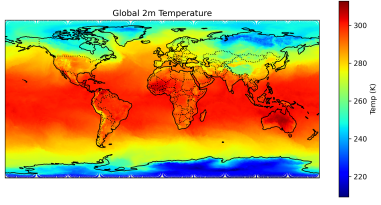
- ICON data is published as static GRIB files
- Organised by model, run, and parameter
- Distributed via DWD Open Data server

Typical workflow

- Query directory listing
- Identify latest available timestamp
- Download and decompress `.grib2.bz2`

Material provided

- Python script: `icon_download_t2m.py`
- Jupyter notebook: ICON download & inspection



See code and notebooks for a robust, reusable implementation.

ICON Fields — Interpolation and Dateline Handling

- ICON uses an unstructured triangular grid
- Direct plotting requires triangulation
- Interpolation needed for:
  - smooth visualization
  - map projections

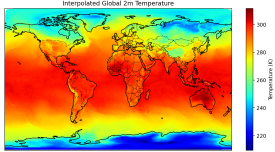
Key challenge

- Triangles crossing the date line
- Incorrect rendering in global projections
- Requires explicit geometric handling

- Use `matplotlib.tri.Triangulation`
- Detect triangles spanning $> 180°$
- Duplicate / shift vertices across $\pm 180°$

Implementation

- See Python script: `fix_dateline_triangles.py` (adapted from ICON tutorial)



Observational Data — Purpose and Characteristics

- Observations anchor models to reality
- Essential for:
  - verification
  - data assimilation
  - ML training and validation

Typical properties

- Irregular spatial distribution
- Sparse and heterogeneous
- Strong metadata dependence

Common observation types

- SYNOP (surface stations)
- TEMP / radiosondes
- AIREP / AMDAR (aircraft)
- Satellite observations

Key difference to models

- No grid
- Point-based measurements
- Representativeness matters

SYNOP Data — Reading Raw Observations

- SYNOP stored in NetCDF
- One record per station
- Variables include:
  - latitude, longitude
  - observed temperature

Minimal requirement

- Read coordinates
- Read physical quantity
- Preserve missing values

Python tools

- `netCDF4.Dataset`
- `numpy`

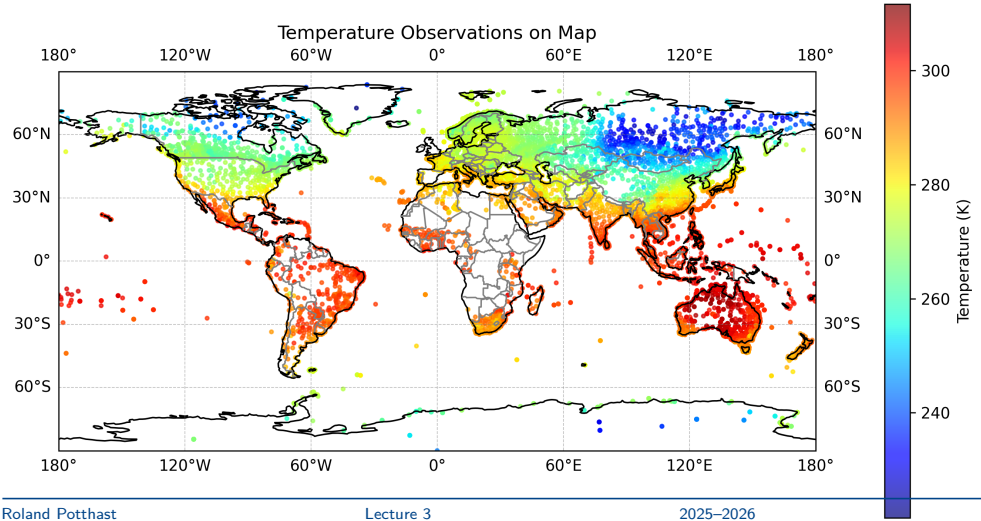
Variables used

- `MLAH` — latitude
- `MLOH` — longitude
- `MTDBT` — temperature

Implementation

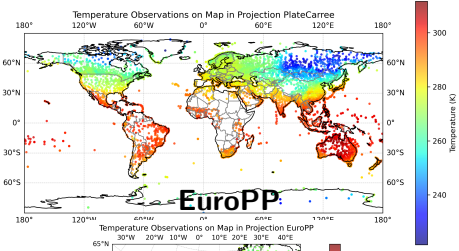
- See function: `read_synop_data()`

SYNOP Observations — Point-Based Visualization

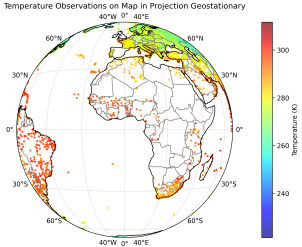


SYNOP Observations — Map Projections

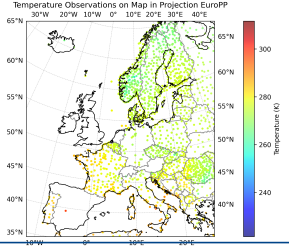
Plate Carree



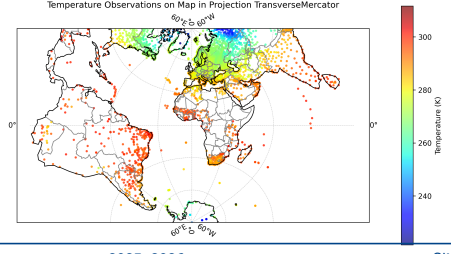
Geostationary



EuroPP



Transverse Mercator



Feedback Files — Linking Models and Observations

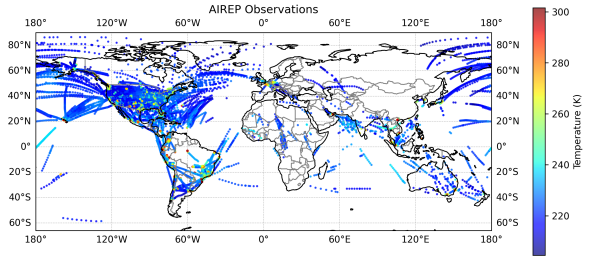
- Feedback files store model–observation pairs
- Generated during data assimilation
- Central diagnostic product

Contained information

- Observation value
- Model equivalent (H(x))
- Quality control flags
- Metadata (time, location, type)

Why they matter

- Quantify model bias and error
- Basis for monitoring systems
- Ideal input for ML analysis



Feedback Files — Structure and Indexing

- Stored as structured NetCDF files
- Separate layers for:
  - report metadata
  - individual observations

Key idea

- One report may contain many observations
- Explicit indexing links both layers

Core variables (excerpt)

```
i_body     (nreport)      start index of report
l_body     (nreport)      number in report

obs        (nobs)         observation
veri_data  (nmodel,nobs)  model equivalent H(x)
```

Interpretation

- `i_body:l_body` maps reports → obs
- `obs` and `veri_data` are aligned
- Enables direct O–B, O–A diagnostics

AIREP Feedback — Statistical Diagnostics

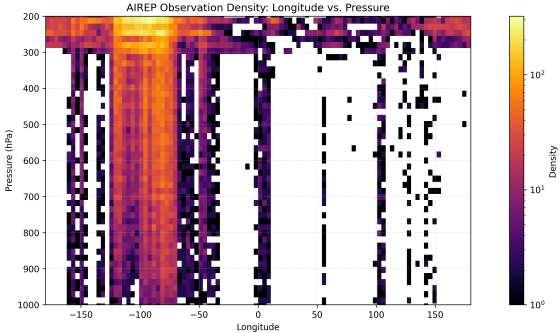
- AIREP / AMDAR provide aircraft observations
- Strong vertical and regional sampling biases
- Feedback files enable systematic evaluation

Typical diagnostics

- Observation density vs height
- Mean and spread of innovations (O–B)
- Model-dependent biases

Why this matters

- Identifies representativeness errors
- Reveals flow- and height-dependent biases



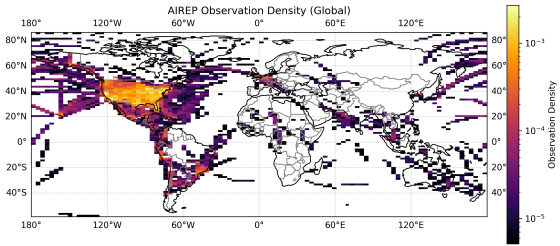
AIREP observation density as a function of height

AIREP Feedback — Horizontal Sampling

- AIREP sampling follows air traffic routes
- Highly inhomogeneous horizontal coverage
- Strong land–sea and regional contrasts

Key implications

- Data density reflects logistics, not physics
- Large unsampled regions persist
- Bias risk if learning ignores sampling



Global horizontal density of AIREP observations

Operational relevance

- Affects verification statistics
- Affects DA weighting
- Critical for ML generalisation

Why GRIB? — Operational Requirements

- GRIB = GRIdded Binary
- Designed for operational NWP
- Optimised for:
  - large data volumes
  - fast I/O
  - network distribution

Core design goals

- Compact encoding
- Self-describing metadata
- Bit-level compression

Why it is still used

- Global models produce terabytes per day
- Multiple grids and vertical coordinates
- Strict real-time constraints

Operational reality

- WMO standard
- Tooling is mature and stable
- Backward compatibility matters

Alternatives to GRIB — Strengths and Limits

NetCDF / CF

- Widely used in research
- Human-readable metadata
- Excellent tool support

Zarr / Cloud-native

- Chunked, parallel access
- Designed for object storage
- Attractive for ML workflows

Why GRIB is not easily replaced

- Operational ecosystems are huge
- Encoding efficiency still unmatched
- Standards evolution is slow by design

Current practice

- GRIB for production and exchange
- NetCDF / Zarr for analysis and ML
- Conversion layers in between

Key message

- Format choice reflects use case

GPU Access in Practice — Why It Depends on the Platform

- Most laptops *do have a GPU*
- But GPU type and backend differ strongly
- Python code must match the backend

Common GPU types

- NVIDIA — typical for HPC clusters
- Apple GPU — Apple Silicon
- AMD / Radeon — some laptops, workstations

Key message

- The software interface matters

GPU backends used in Python

- `CUDA` — NVIDIA GPUs (Linux, HPC)
- `MPS / Metal` — Apple GPUs (macOS)
- `CPU fallback` — always available

Implication for ML workflows

- Development often on laptops
- Training often on HPC systems
- Code must be portable across backends

GPU Tests in Python — Minimal Backend Checks

NVIDIA GPU (CUDA)

CUDA test

```
1 import torch
2 print(torch.cuda.is_available())
```

Apple GPU (Metal / MPS)

Apple GPU test

```
1 import torch
2 print(torch.backends.mps.
      is_available())
```

AMD / Radeon GPUs

- No native PyTorch GPU backend

Interpretation

- `True` → GPU usable
- `False` → CPU fallback
- Backend determines capability

GPU Execution in Python — Laptop vs. HPC

Apple GPU (Metal / MPS)

Apple GPU test

```python
import torch, time

d = torch.device("mps")
x = torch.rand((4000,4000),
    device=d)

t0 = time.time()
y = torch.matmul(x, x)
torch.mps.synchronize()

print("Apple GPU time:",
      round(time.time()-t0,3))
```

NVIDIA A100 (CUDA)

HPC GPU test

```python
import torch, time

d = torch.device("cuda")
x = torch.rand((8000,8000),
    device=d)

t0 = time.time()
y = torch.matmul(x, x)
torch.cuda.synchronize()

print("A100 time:",
      round(time.time()-t0,3))
```

Interactive GPU Access and Verification Workflow

Step 1: Interactive GPU login

Shell alias (interactive GPU)

```
alias gpu1='qlogin -q gp_inter_dgx \
 --gpunum-lhost=1 \
 --cpunum-lhost=16 \
 -l elapstim_req=6:00:00 \
 -l memsz_job=240gb'
```

Step 2: Select GPU explicitly

Shell

```
export CUDA_VISIBLE_DEVICES=7
echo $CUDA_VISIBLE_DEVICES
```

Step 3: Minimal verification

Mini GPU check in Python

```
import torch, os
print("CUDA_VISIBLE_DEVICES ="
    , os.getenv("
    CUDA_VISIBLE_DEVICES"))
print("Visible GPUs =", torch.
    cuda.device_count())
print("GPU name =", torch.cuda
    .get_device_name(0))
```

- Interactive login allocates resources
- GPU visibility must be set explicitly
- Always verify from inside Python

CPU vs GPU — Execution Model and Performance: 30sec –> 3sec

CPU example

```
1 import torch, time
2 n = 30000
3 x0 = torch.rand((n,n))
4 x1 = torch.rand((n,n))
5 t0 = time.time()
6 y0 = torch.matmul(x0,x0)
7 y1 = torch.matmul(x1,x1)
8 print(time.time()-t0)
```

Observed

- Operations are blocking
- Second call starts after first ends
- Parallelism only inside BLAS

Two-GPU example

```
1 import torch, time
2 d0 = torch.device("cuda:0")
3 d1 = torch.device("cuda:1")
4 n. = 30000
5 x0 = torch.rand((n,n),device=d0)
6 x1 = torch.rand((n,n),device=d1)
7 t0 = time.time()
8 torch.matmul(x0,x0)
9 torch.matmul(x1,x1)
10 torch.cuda.synchronize()
11 print(time.time()-t0)
```

- Operations run concurrently

Parallel Matrix Multiplication — What Actually Works

Key idea

- Naive multi-GPU often shows no speed-up
- Reason: data transfers dominate computation
- Solution: true model parallelism

What works

- Build matrices directly on each GPU
- Split the computation (block-wise)
- Avoid CPU–GPU transfers
- Collect results only at the end

True model-parallel matrix multiplication

```
import torch, time
n = 30000
A0 = torch.rand((n//2,n), device="cuda:0")
A1 = torch.rand((n//2,n), device="cuda:1")
B  = torch.rand((n,n),      device="cuda:0")
t0 = time.time()
C0 = A0 @ B
C1 = A1 @ B.to("cuda:1")
torch.cuda.synchronize()
print("Time:", round(time.time()-t0,3))
```

Mixed Precision Computing — Why FP16 Matters

Why reduced precision?

- Modern GPUs are optimized for FP16 / BF16
- Higher throughput, lower memory traffic
- Essential for large ML models

Typical ML pattern

- Dense linear layers
- Nonlinear activation
- Large batch sizes

Transformers, Diffusion, Weather

FP16 neural-network-style workload

```
import torch, time
torch.set_default_dtype(torch.float16)
d = torch.device("cuda")
x = torch.randn((20000,1024), device=d)
W1 = torch.randn((1024,4096), device=d)
W2 = torch.randn((4096,1024), device=d)
t0 = time.time()
y = torch.nn.functional.gelu(x @ W1)
z = y @ W2
torch.cuda.synchronize()
print("FP16 time:", round(time.time()-
    t0,3))
```