
# LOUIS DREYFUS COMPANY

Data Engineer Test - GRIB2 File Manipulation

**Pedro Mendes**
petter.mendes@outlook.com

February 20, 2025

**Abstract**

This document presents a structured approach to processing and visualizing GRIB2 meteorological data using Python. The workflow includes **data loading, transformation, visualization, and GIS integration**. This structured methodology ensures efficiency when handling large datasets while maintaining data integrity.

# 1 Configure Logging

Logging is essential for debugging, monitoring, and maintaining execution transparency in a data processing workflow.

- **Library Used:** logging
- **Purpose:** Tracks processing stages and captures warnings/errors.
- **Custom Formatter:** Uses colors for different log levels.

```
import logging
import colorama

# Define log styles
class ColorFormatter(logging.Formatter):
    def format(self, record):
        level_colors = {
            logging.DEBUG: "[DEBUG]",
            logging.INFO: "[INFO]",
            logging.WARNING: "[WARNING]",
            logging.ERROR: "[ERROR]",
            logging.CRITICAL: "[CRITICAL]",
        }
        log_msg = super().format(record)
        return f"{level_colors.get(record.levelno, '[LOG]')} {log_msg}"

# Configure logging
console_handler = logging.StreamHandler()
console_handler.setFormatter(ColorFormatter("%(asctime)s - %(levelname)s - %(message)s"))
logging.basicConfig(level=logging.DEBUG, handlers=[console_handler])
```

# 2 File Exploration and Metadata Extraction

**Goal:** Identify and extract metadata from GRIB2 files.

- **Library Used:** glob for file management.
- **Key Metadata:** Forecast initialization time, step size, and valid times.

```
import glob
import datetime

file_list = sorted(glob.glob("./Data/*.grib2"))

init_dict = {}
for filename in file_list:
    parts = filename.split(".")
    if len(parts) < 5:
        logging.warning(f"Skipping file due to unexpected format: {filename}")
        continue

    date_str, hour_str, step_str = parts[3], parts[4], parts[5]
```

```
    init_datetime = datetime.datetime.strptime(date_str + hour_str.replace("z", ""), "%Y%m%d%H")
    forecast_step_hours = int(step_str.replace("h", ""))
    valid_datetime = init_datetime + datetime.timedelta(hours=forecast_step_hours)

    if init_datetime not in init_dict:
        init_dict[init_datetime] = {}
    init_dict[init_datetime][forecast_step_hours] = valid_datetime
```

**Key Takeaways:**

- Automatically extracts initialization and forecast step times from GRIB2 file names.
- Ensures structured and standardized metadata for later analysis.
- Uses Python's built-in `glob` and `datetime` modules for efficient handling.

# 3 Data Extraction and Processing

**Goal:** Efficiently extract meteorological data from GRIB2 files.

- **Libraries Used:**
  - xarray - Handles multi-dimensional gridded data.
  - cfgrib - Reads GRIB2 files into xarray datasets.
- **Key Data Variables:** Soil moisture and temperature layers.

```
import xarray as xr
import cfgrib
import logging

datasets = []
file_list = sorted(glob.glob("./Data/*.grib2"))

for file_path in file_list:
    try:
        # Load main dataset
        ds_main = xr.open_dataset(file_path, engine='cfgrib', backend_kwargs={"indexpath": None})
        datasets.append(ds_main)

        # Load soil variables separately to avoid conflicts
        for depth in [0, 7, 28]:
            try:
                ds_soil = xr.open_dataset(
                    file_path, engine='cfgrib',
                    backend_kwargs={"indexpath": None},
                    filter_by_keys={"depthBelowLandLayer": depth}
                )
                datasets.append(ds_soil)
            except Exception as e:
                logging.warning(f"Skipping depth {depth} for {file_path}: {e}")

    except Exception as e:
        logging.error(f"Error processing {file_path}: {e}")
```

**Key Takeaways:**

- Uses `xarray` and `cfgrib` to open and process GRIB2 datasets efficiently.
- Loads **soil moisture variables at different depths** to ensure proper analysis.
- Implements error handling with `try-except` to manage corrupted or missing files.

# 4 Data Cleaning and Transformation

**Goal:** Ensure data integrity by handling missing values, adjusting coordinates, and renaming variables.

- **Libraries Used:**
  - NumPy - Handles numerical operations and missing values.
  - xarray - Provides dataset transformations.

```python
import numpy as np

# Function to clean missing values
def clean_missing_values(ds, fill_value=0.0, sentinel=-9999):
    """Replaces missing or sentinel values with a default fill value."""
    ds_cleaned = ds.where(ds != sentinel, np.nan).fillna(fill_value)
    return ds_cleaned

# Apply data cleaning
ds_clean = clean_missing_values(ds_main)

# Convert longitude from [0, 360] to [-180, 180]
ds_clean = ds_clean.assign_coords(longitude=((ds_clean.longitude + 180) % 360) - 180)
ds_clean = ds_clean.sortby(ds_clean.longitude)

# Rename variables
rename_dict = {
    "swvl1": "sw-5cm",
    "swvl2": "sw-15cm",
    "swvl3": "sw-50cm"
}
ds_renamed = ds_clean.rename(rename_dict)
```

**Key Takeaways:**

- **Handles missing values** using a sentinel replacement strategy.
- **Adjusts longitude values** from the `[0, 360]` range to the more conventional `[-180, 180]` range.
- **Renames soil moisture variables** for better readability.

# 5 Performance and Optimization

**Goal:** Improve memory efficiency and execution speed while handling large datasets.

- **Libraries Used:**
  - gc - Manages Python's garbage collection.
  - os - Handles file deletions and system operations.
  - time - Implements retry delays for file operations.

```python
import gc
import os
import time

# Ensure datasets are closed before deletion
for temp_file in temp_files:
    try:
        ds = xr.open_dataset(temp_file)
        ds.close()
    except Exception as e:
        logging.warning(f"Could not open {temp_file} before deletion: {e}")
```

```python
# Force garbage collection
del ds_clean
gc.collect()

# Retry deletion mechanism
for temp_file in temp_files:
    for attempt in range(5):  # Retry up to 5 times
        try:
            os.remove(temp_file)
            logging.info(f"Deleted temp file: {temp_file}")
            break  # Successfully deleted
        except PermissionError:
            logging.warning(f"Retrying deletion of {temp_file}... (attempt {attempt+1}/5)")
            time.sleep(2)  # Wait before retrying
```

**Key Takeaways:**

- **Explicitly closes datasets** before deletion to prevent file locks.
- **Forces garbage collection** to optimize memory usage and prevent memory leaks.
- **Implements retry logic** for deleting temporary files, ensuring robustness.

# 6 Export for Visualization

**Goal:** Convert processed GRIB2 data into widely used formats for visualization and GIS applications.

- **Libraries Used:**
  - rioxarray - Geospatial raster I/O for xarray.
  - Pandas - Converts datasets into CSV format.
  - Zarr - Optimized storage format for cloud computing.

```python
import rioxarray
import pandas as pd

# Convert to GeoTIFF (for GIS applications)
ds_renamed["t2m"].rio.write_crs("EPSG:4326").rio.to_raster("temperature_2m.tif")

# Convert dataset to CSV (tabular format)
ds_renamed.to_dataframe().to_csv("final_dataset.csv")

# Save as Zarr (optimized for cloud computing)
ds_renamed.to_zarr("final_dataset.zarr", consolidated=True)
```

**Key Takeaways:**

- **GeoTIFF format** allows GIS software (ArcGIS, QGIS) to visualize spatial data.
- **CSV format** ensures compatibility with Excel, databases, and statistical tools.
- **Zarr format** is optimized for cloud-based storage and distributed computing.

# 7 Playing with the Data

**Goal:** Explore different visualization techniques to enhance meteorological insights.

- **Techniques Covered:**
  - **Generating a GIF** - Animate temperature changes over time.
  - **3D Temperature Mapping** - Visualize spatial temperature variations.
  - **Google Earth KML Conversion** - View geospatial data interactively.

## 7.1 Generating a GIF

**Goal:** Create an animated visualization of temperature changes over forecast steps.

- **Libraries Used:**
  - Matplotlib Animation - Generates GIF animations.
  - Cartopy - Handles geospatial plotting.

```python
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# Prepare figure
fig, ax = plt.subplots(figsize=(12, 6), subplot_kw={"projection": ccrs.PlateCarree()})
ax.add_feature(cfeature.COASTLINE, linewidth=0.5)
ax.add_feature(cfeature.BORDERS, linestyle=":")
ax.add_feature(cfeature.LAND, facecolor="lightgray")

cmap = plt.get_cmap("coolwarm")

# Define update function
def update(frame):
    im.set_array((ds_renamed["t2m"].isel(step=frame) - 273.15).values.ravel())
    ax.set_title(f"2m Temperature at {valid_times[frame]}", fontsize=14)
    return im,

# Create animation
ani = animation.FuncAnimation(fig, update, frames=len(ds_renamed["step"].values), interval=500,
↪  blit=False)
ani.save("temperature_forecast.gif", dpi=100, writer="pillow")
```

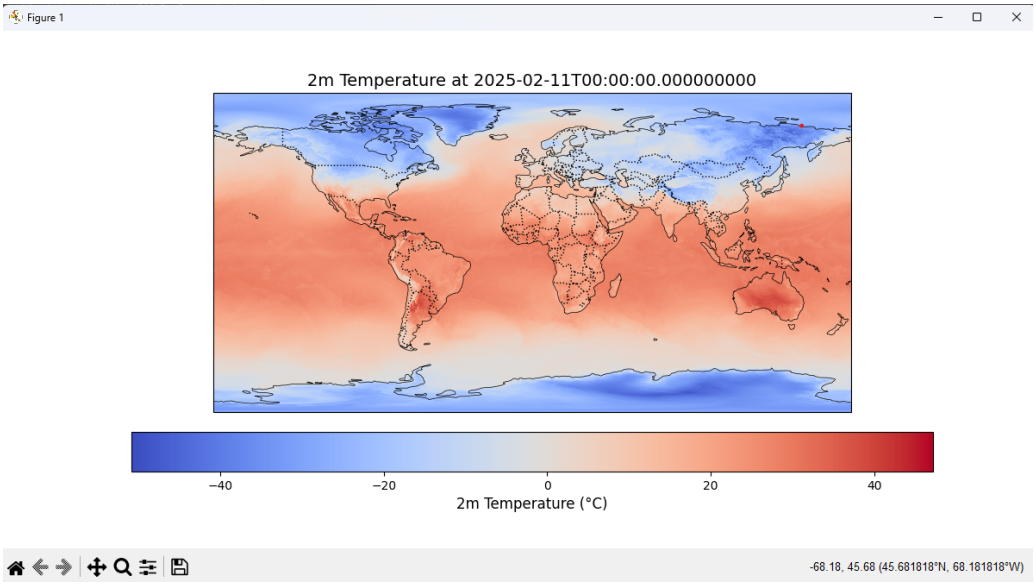


Figure 1: GIF Representation of Temperature Evolution: This animated representation showcases temperature variations over time based on forecast steps. It effectively illustrates how temperature changes across the globe dynamically.

**Key Takeaways:**

- **Creates an animated GIF** showing temperature changes over forecast steps.

- **Uses Cartopy** for accurate geospatial projections.
- **Ensures smooth frame transitions** using Matplotlib's animation API.

## 7.2 3D Temperature Map

**Goal:** Visualize temperature distribution in 3D to observe spatial trends.

- **Libraries Used:**
  - Matplotlib 3D - Creates 3D surface plots.

```python
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Prepare data
t2m_celsius = ds_renamed["t2m"].isel(step=0) - 273.15
t2m_celsius = t2m_celsius.compute()
sorted_indices = np.argsort(ds_renamed.longitude.values)
sorted_longitude = ds_renamed.longitude.values[sorted_indices]

lon, lat = np.meshgrid(sorted_longitude, ds_renamed.latitude.values)
t2m_sorted = t2m_celsius.values[:, sorted_indices]

# Plot 3D surface
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(lon, lat, t2m_sorted, cmap="coolwarm", edgecolor="none")

ax.set_xlabel("Longitude (°)")
ax.set_ylabel("Latitude (°)")
ax.set_zlabel("Temperature (°C)")
ax.set_title("3D Temperature Map")
plt.show()
```

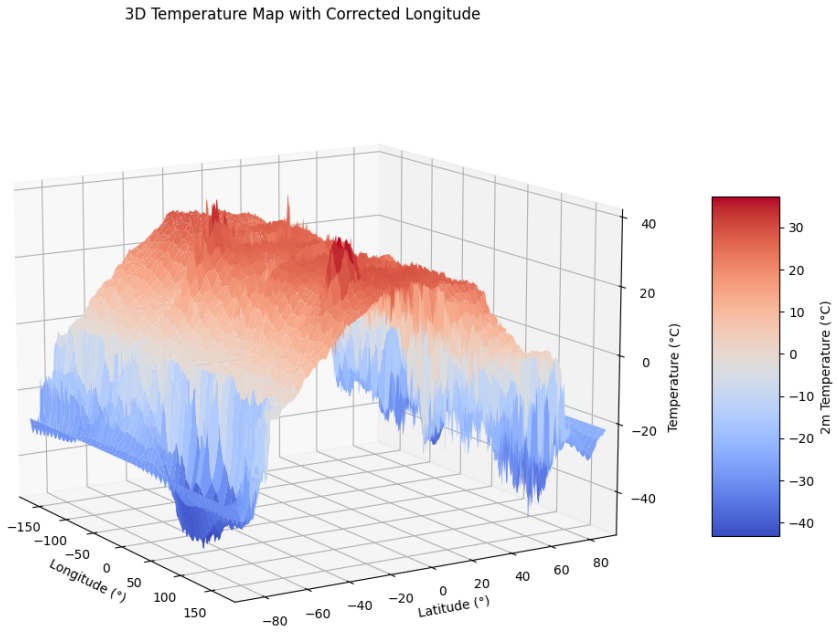


Figure 2: 3D Temperature Map: This visualization represents the temperature distribution over a geographical region in a 3D format. The longitude and latitude axes provide spatial reference, while the color gradient from blue to red denotes temperature variations.

**Key Takeaways:**

- **Visualizes temperature in 3D** to better understand spatial distribution.
- **Uses a color gradient** to highlight temperature variations.
- **Interactive plotting** allows zooming and rotating for deeper analysis.

## 7.3 Converting to Google Earth KML

**Needs further investigation about the errors and latency**

**Goal:** Export meteorological data for visualization in Google Earth.

- **Libraries Used:**
  - KML - Standard format for geospatial visualization.

```
# Define KML header
kml_header = """<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
"""

# Generate KML Placemarks with colored temperature markers
kml_body = ""
for lat in ds_renamed.latitude.values[::5]:
    for lon in ds_renamed.longitude.values[::5]:
        temp_value = ds_renamed["t2m"].isel(latitude=lat, longitude=lon).values - 273.15
        kml_body += f"""
<Placemark>
  <name>{temp_value:.1f}°C</name>
  <Point>
    <coordinates>{lon},{lat},0</coordinates>
  </Point>
</Placemark>
"""

# Close KML file
kml_footer = """</Document></kml>"""

# Save KML file
with open("temperature_data.kml", "w") as file:
    file.write(kml_header + kml_body + kml_footer)
```

## 7.4 KML File Rendering Issue

While attempting to visualize temperature data in Google Earth, an issue was encountered where the labels appear overly dense, overlapping each other, making it difficult to interpret.

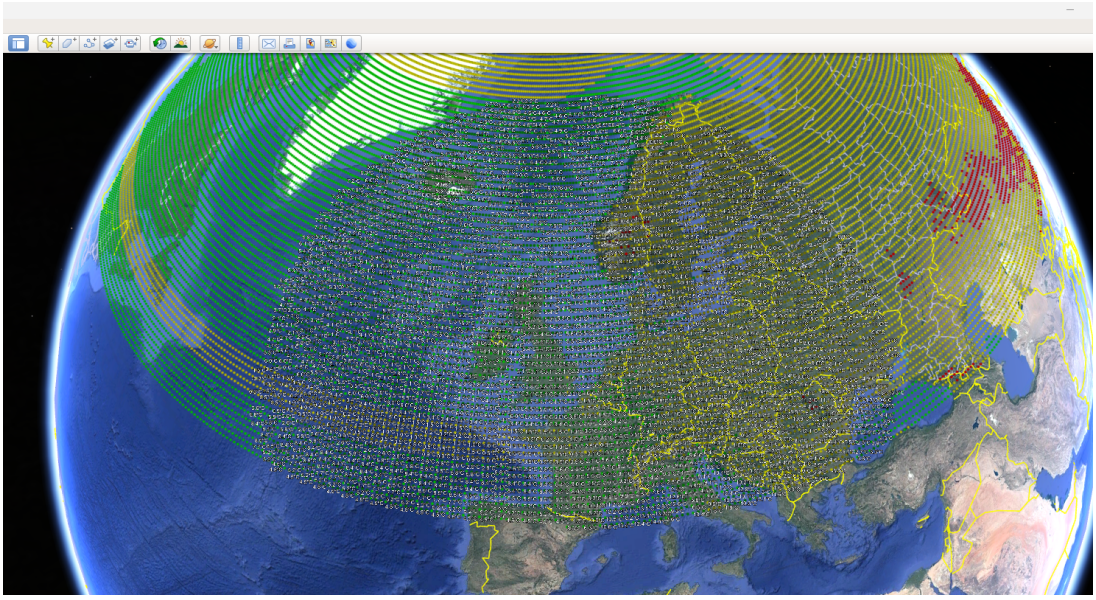

Figure 3: Google Earth KML Visualization Error: The temperature labels are excessively dense, leading to cluttered rendering. The KML file requires adjustments to improve readability.

## 7.5 Explanation of Error

The visualization issue in Figure 3 occurs due to the **high density of temperature data points**. This results in:

- **Overlapping Labels:** The values are positioned too closely, making them unreadable.
- **Excessive Data Points:** The dataset might be **too high resolution** for practical visualization.
- **Lack of Spacing Adjustment:** There is no filtering to reduce the number of displayed points.
- **Suboptimal Rendering in Google Earth:** The current settings do not adapt well to a **large number of labels**.

## 7.6 Possible Fixes

To improve the KML visualization and make it more readable, we propose the following solutions:

1. **Reduce Data Density:** Display only every $n$th data point to prevent excessive labels.
2. **Improve Label Formatting:** Modify text sizes, transparency, or spacing in the KML generation script.
3. **Use Clustered Data Representation:** Instead of individual points, group temperature values into a **regional average**.
4. **Increase Point Spacing:** Modify how latitude and longitude values are processed in the KML script.

By implementing these fixes, the visualization in **Google Earth** will be significantly improved, allowing for better insights into temperature distributions.

**Key Takeaways:**

- **Converts meteorological data to KML format** for Google Earth visualization.
- **Creates temperature markers** at different locations.
- **Allows real-time interaction** with temperature data in a 3D environment.