Reading NetCDF Data with Python

Prepared by
Mohamed Abdelkader[1], Jorge Bravo[1], Marouane Temimi[1], and Jibin Joseph[2]
[1]Civil, Environmental, and Ocean Engineering Department, Stevens Institute of Technology
[2]School of Civil Engineering, Purdue University
mabdelka@stevens.edu

FAIR Science in Climate

Objective

Introduce NetCDF4 and Xarray Libraries: Familiarize students with netCDF4 and Xarray, the two principal Python libraries for reading and handling NetCDF data. Understand their documentation and the unique features each library offers.

Overview of steps

1. Practical Application in Remote Sensing: Apply the netCDF4 and Xarray libraries to read, analyze, and manipulate remote sensing data stored in NetCDF format, with a focus on meteorological datasets.
2. Reading Data with netCDF4: Learn to import and utilize the Dataset class from the netCDF4 library for accessing and exploring NetCDF files, including reading data and metadata exploration.
3. Data Manipulation and Visualization: Understand how to manipulate data within NetCDF files using the Dataset class and how to visualize data using Xarray's plotting capabilities and Matplotlib.
4. Advanced Visualization Techniques: Explore advanced visualization techniques including custom color maps, projections, and adjusting the visual representation of satellite imagery or atmospheric data models.

Resources

GOES ABI (Advanced Baseline Imager) Realtime Imagery
GOES Image Viewer

Instructions

1. Access Lecture_3 folder and initiate “Reading NetCDF Data with Python” notebook.
2. Begin by familiarizing yourself with the netCDF4 and Xarray libraries, which are instrumental in handling NetCDF data. Start by exploring the netCDF4 library to understand how to open, inspect, and modify NetCDF files, focusing on the Dataset class for data access and manipulation. This knowledge is fundamental for working with complex datasets in remote sensing and meteorology.

netCDF4 Library

netCDF4 Documentation

netCDF4-python is a Python interface to the **NetCDF-C library**. It provides a powerful set of tools for working with NetCDF data files.

Xarray Library

Xarray Documentation

Xarray simplifies the process of working with labeled multidimensional arrays in Python. It offers efficiency and ease of use when handling complex datasets.

In this lecture, we will explore how to use these libraries to read and manipulate NetCDF data, enabling us to work effectively with meteorological datasets.

3. Import the `Dataset` class from the `netCDF4` library to begin working with NetCDF files.

```
# import necessary library
from netCDF4 import Dataset
```

4. Define the file path where your ABI data is stored, preparing it for access and analysis.

```
# Define the file path for the ABI data
path2data = "./Input_data/ABI-L2-CMIPC/s20180471917"

# Set the name of the netCDF data file
name2data = "OR_ABI-L2-CMIPC-M3C13_G16_s20180471917196_e20180471919581_c20180471920028.nc"

# Construct the full file path by combining the directory path and file name
full_path = f'{path2data}/{name2data}'
```

5. Open the netCDF file for the C08 channel by using the `Dataset` class, and then display the file content to inspect its structure and attributes.

```
# Open the netCDF file for the C08 channel using the Dataset class
f_C08 = Dataset(f'{path2data}/{name2data}')

f_C08
```

6. Access the Cloud and Moisture Imagery (CMI) variable from the opened netCDF file to work with the specific data it contains.

```
#Access the Cloud and Moisture Imagery (CMI) variable
bC08 = f_C08.variables['CMI']

bC08
```

7. Import xarray library for working with labeled multi-dimensional arrays.

```
# Import xarray library for working with labeled multi-dimensional arrays
import xarray as xr
```

8. Load the dataset into an xarray object using the full file path, and then display the dataset's contents to review its structure and variables.

```
# Define the full file path for the dataset
pth01 = f'{path2data}/{name2data}'

# Load the dataset into an xarray object for easy data manipulation
ncx = xr.open_dataset(pth01)

# Display the contents of the dataset
ncx
```

9. Set up matplotlib to display figures directly in your notebook with `%matplotlib inline`. Then, use xarray's plotting capabilities to visualize the 'CMI' variable from the netCDF dataset.

```
# Configure matplotlib to show figures embedded in the notebook
%matplotlib inline

# Plot the 'CMI' variable from the netCDF dataset using xarray's plotting capabilities
ncx['CMI'].plot()
```

10. Retrieve and display the attributes of the 'CMI' variable in the dataset.

```
# Retrieve and display the attributes of the 'CMI' variable in the dataset
ncx['CMI'].attrs
```

11. Create a figure of specified dimensions and add a subplot. Then, display the 'CMI' variable data as an image, specifying the colormap and origin, to visualize the data in grayscale.

```
# Create a figure with specific dimensions
fig = plt.figure(figsize=(10,10))

# Add a subplot to the figure
ax = fig.add_subplot(1, 1, 1)

# Display the 'CMI' variable data as an image with specified colormap and origin
ir_img = ax.imshow(ncx['CMI'].data, origin='upper', cmap="Greys")
```

12. Import the NumPy library for numerical operations on arrays.

```
# Import the NumPy library for numerical operations on arrays
import numpy as np
```

13. Define a custom color map for infrared imagery, create evenly spaced levels for this color map, and establish a boundary norm to map data values to color intervals. This setup allows for more detailed visualization of infrared imagery data.

```
# Define a custom color map for the infrared imagery
cmap = ['#ffffff', '#ffffff', '#ffffff', '#ffffff', '#ffffff', '#b6ffb6', '#79ff79', '#00ff00', '#ff8e8e', '#ff5151',
        '#0028a2', '#000079', '#fbfb00', '#e7e700', '#d2d200', '#baba00', '#a6a600', '#8e8e00', '#797900', '#656500',
        '#aaaaaa', '#a6a6a6', '#9e9e9e', '#969696', '#8e8e8e', '#868686', '#7d7d7d', '#757575', '#6d6d6d', '#656565',
        '#313131', '#282828', '#202020', '#181818', '#141414', '#000000', '#000000', '#000000', '#000000', '#000000',

# Create evenly spaced levels for the color map ranging from -109 to 56
levels = np.linspace(-109, 56, num=len(cmap))

# Establish a boundary norm which maps values to intervals within the color map
norm = plt.cm.colors.BoundaryNorm(levels, len(levels))

# Create a ListedColormap object using the custom color map
irmap = plt.cm.colors.ListedColormap(cmap)
```

14. Create a figure with a specified size and add a subplot. Then, display the 'CMI' variable data as an image, setting the color range to match specific temperature values and using your custom colormap for enhanced visualization.

```
# Create a figure with a specified size
fig = plt.figure(figsize=(12,12))

# Add a subplot to the figure
ax = fig.add_subplot(1, 1, 1)

# Display the 'CMI' variable data as an image, setting the color range and custom colormap
ir_img = ax.imshow(ncx['CMI'].data, origin='upper', vmin=-109 + 273.15, vmax=56 + 273.15, cmap=irmap)
```

15. Import the ticker module from matplotlib to customize axis ticks, and import the coordinate reference system submodule from cartopy for handling map projections. This will enable more precise control over the appearance of plots and maps.

```
# Import the ticker module from matplotlib for customizing axis ticks
import matplotlib.ticker as mticker

# Import cartopy's coordinate reference system submodule for map projections
import cartopy.crs as ccrs
```

16. Access the 'goes_imager_projection' variable from your dataset to retrieve information about the projection used by the GOES imager, which is crucial for accurate mapping and visualization.

```
# Access the 'goes_imager_projection' variable from the dataset to get projection information
ncx['goes_imager_projection']
```

17. Retrieve essential projection details from your dataset, including the satellite's height, the central longitude, and the Earth's ellipsoid semi-major and semi-minor axes. Also, determine the orientation of the satellite by accessing the sweep angle axis.

```
# Retrieve the projection variable from the dataset
proj_var = ncx['goes_imager_projection']

# Extract the satellite height from the projection variable
sat_h = proj_var.perspective_point_height

# Get the longitude of the projection origin (central longitude)
central_lon = proj_var.longitude_of_projection_origin

# Obtain the values for the semi-major and semi-minor axes of the Earth's ellipsoid
semi_major = proj_var.semi_major_axis
semi_minor = proj_var.semi_minor_axis

# Determine the sweep angle axis (orientation of the satellite)
sweep = proj_var.sweep_angle_axis
```

18. Scale the x and y coordinates by the satellite height to convert them to actual distances, and then calculate the number of lines (rows) and columns in the data

using the dimensions of y and x.

```
# Scale the x and y coordinates by the satellite height to get actual distances
x = ncx.variables['x'][:] * sat_h
y = ncx.variables['y'][:] * sat_h

# Determine the number of lines (rows) and columns in the data based on y and x dimensions
nlines = y.shape[0]
ncols = x.shape[0]
```

19. Create a figure of specified size and set up a geostationary projection using the globe model with given semi-major and semi-minor axes values. Initialize this projection with the central longitude and satellite height. Add a subplot to the figure with this projection and display the 'CMI' variable data as an image, setting the custom extent and colormap.

```
# Create a figure with a specified size
fig = plt.figure(figsize=(12,12))

# Set up a globe model using the semi-major and semi-minor axes values
globe = ccrs.Globe(ellipse='sphere', semimajor_axis=semi_major, semiminor_axis=semi_minor)

# Initialize a geostationary projection using the central longitude and satellite height
geos = ccrs.Geostationary(central_longitude=central_lon, satellite_height=sat_h, sweep_axis=sweep, globe=globe)

# Add a subplot to the figure with the specified geostationary projection
ax = fig.add_subplot(1, 1, 1, projection=geos)

# Display the 'CMI' variable data as an image with custom extent and colormap, using the geostationary projection
ir_img = ax.imshow(ncx['CMI'].data, origin='upper', extent=(x.min(),
                            y.min(), x.max(), y.max()), cmap="Greys_r", transform=geos)
```

20. Import the glob module.

```
# Import the glob module to find all the pathnames matching a specified pattern
from glob import glob
```

21. Define the directory path for the input .nc (netCDF) files and use the glob function to compile a list of all .nc files within this directory. Then, display the compiled list of file paths.

```
# Define the directory path containing the .nc (netCDF) files
fllst = 'Input_data\ABI-L2-CMIPC\s20180471917'

# Use glob to retrieve a list of all .nc files in the specified directory
glst = glob(f'{fllst}/*.nc')

# Display the list of .nc files
glst
```

22. Access the third file in the list of .nc files.

```
# Access the third file in the list of .nc files
glst[2]
```

23. Open the 13th .nc file from the previously compiled list using xarray, and then display the contents of this dataset to examine its structure and variables.

```
# Open the 13th .nc file in the list using xarray
ncx2 = xr.open_dataset(glst[12])

# Display the contents of the opened dataset
ncx2
```

24. Create a figure with a specific size and establish a globe model with the given semi-major and semi-minor axes values. Initialize a geostationary projection with specified central longitude, satellite height, and sweep axis. Add a subplot to this figure using the geostationary projection and display the 'CMI' variable data from the second dataset as an image. Set custom extents, colormap, and value range for the visualization.

```
# Create a figure with a specified size
fig = plt.figure(figsize=(12,12))

# Set up a globe model using the semi-major and semi-minor axes values
globe = ccrs.Globe(ellipse='sphere', semimajor_axis=semi_major, semiminor_axis=semi_minor)

# Initialize a geostationary projection using the central longitude and satellite height
geos = ccrs.Geostationary(central_longitude=central_lon, satellite_height=sat_h, sweep_axis=sweep, globe=globe)

# Add a subplot to the figure with the specified geostationary projection
ax = fig.add_subplot(1, 1, 1, projection=geos)

# Display the 'CMI' variable data from the second dataset as an image with custom extents and colormap,
# using the geostationary projection
ir_img = ax.imshow(ncx2['CMI'].data, origin='upper', vmin=-109+273.15, vmax=56+273.15, extent=(x.min(), y.min(),
                   x.max(), y.max()), cmap="jet_r", transform=geos)
```

25. Visualize 'CMI' variable data from the second dataset using a custom colormap in a geostationary projection. Create a figure, define a custom colormap with specific color codes for different temperature ranges, and map these colors to a range of temperatures. Establish a globe model and initialize a geostationary projection with given parameters.

```
# Create a figure with a specified size for visualization
fig = plt.figure(figsize=(12,12))

# Define a custom colormap for the image
 # A series of color codes representing different temperature ranges
cmap = ['#ffffff', '#ffffff', '#ffffff', '#ffffff', '#ffffff', '#b6ffb6', '#79ff79', '#00ff00',
        '#ff8e8e', '#ff5151', '#ff0000', '#aa0000', '#550000', '#00ffff', '#00bef3', '#0079ca',
        '#0028a2', '#000079', '#fbfb00', '#e7e700', '#d2d200', '#baba00', '#a6a600', '#8e8e00',
        '#797900', '#656500', '#dbdbdb', '#d2d2d2', '#cacaca', '#c2c2c2', '#bababa', '#b2b2b2',
        '#aaaaaa', '#a6a6a6', '#9e9e9e', '#969696', '#8e8e8e', '#868686', '#7d7d7d', '#757575',
        '#6d6d6d', '#656565', '#5d5d5d', '#595959', '#515151', '#494949', '#414141', '#393939',
        '#313131', '#282828', '#202020', '#181818', '#141414', '#000000', '#000000', '#000000',
        '#000000', '#000000', '#000000', '#000000', '#000000', '#000000', '#000000', '#000000',]

# Create a range of levels corresponding to the colors in the custom colormap
levels = np.linspace(-109, 56, num=len(cmap))

# Set up a normalization scheme based on the defined levels
norm = plt.cm.colors.BoundaryNorm(levels, len(levels))

# Create a ListedColormap object with the custom colormap
irmap = plt.cm.colors.ListedColormap(cmap)

# Set up a globe model using the semi-major and semi-minor axes values
globe = ccrs.Globe(ellipse='sphere', semimajor_axis=semi_major, semiminor_axis=semi_minor)

# Initialize a geostationary projection using the central longitude and satellite height
geos = ccrs.Geostationary(central_longitude=central_lon, satellite_height=sat_h, sweep_axis=sweep, globe=globe)

# Add a subplot to the figure with the specified geostationary projection
ax = fig.add_subplot(1, 1, 1, projection=geos)

# Display the 'CMI' variable data from the second dataset as an image with custom extents and colormap,
# using the geostationary projection
ir_img = ax.imshow(ncx2['CMI'].data, origin='upper', vmin=-109+273.15, vmax=56+273.15, extent=(x.min(),
                                                y.min(), x.max(), y.max()), cmap=irmap, transform=geos)
```

26. Initialize a figure for visualizing satellite data with a specific size and a custom colormap that reflects various data intensities or ranges. Establish a normalization rule to ensure consistent data representation across the defined color levels. Utilize a geostationary projection with defined Earth and satellite parameters to display the 'CMI' variable from the dataset. Apply the custom colormap for visualizing data within specified value limits and set the map display to cover designated longitude and latitude boundaries.

```
# Initialize a figure with a specified size (12 inches by 12 inches)
fig = plt.figure(figsize=(12,12))

# Custom colormap defined by specific color codes, likely representing different data ranges or intensities
cmap = ['#ffffff', '#ffffff', '#ffffff', '#ffffff', '#ffffff', '#b6ffb6', '#79ff79', '#00ff00',
        '#ff8e8e', '#ff5151', '#ff0000', '#aa0000', '#550000', '#00ffff', '#00bef3', '#0079ca',
        '#0028a2', '#000079', '#fbfb00', '#e7e700', '#d2d200', '#baba00', '#a6a600', '#8e8e00',
        '#797900', '#656500', '#dbdbdb', '#d2d2d2', '#cacaca', '#c2c2c2', '#bababa', '#b2b2b2',
        '#aaaaaa', '#a6a6a6', '#9e9e9e', '#969696', '#8e8e8e', '#868686', '#7d7d7d', '#757575',
        '#6d6d6d', '#656565', '#5d5d5d', '#595959', '#515151', '#494949', '#414141', '#393939',
        '#313131', '#282828', '#202020', '#181818', '#141414', '#000000', '#000000', '#000000',
        '#000000', '#000000', '#000000', '#000000', '#000000', '#000000', '#000000', '#000000',]

# Create an array of levels corresponding to the custom colormap
levels = np.linspace(-109, 56, num=len(cmap))

# Establish a normalization rule using the defined levels for consistent data representation
norm = plt.cm.colors.BoundaryNorm(levels, len(levels))

# Create a colormap object from the custom list of colors
irmap = plt.cm.colors.ListedColormap(cmap)

# Define a globe model, considering Earth's shape as a sphere with given major and minor axes
globe = ccrs.Globe(ellipse='sphere', semimajor_axis=semi_major, semiminor_axis=semi_minor)

# Set up a geostationary projection using satellite parameters
geos = ccrs.Geostationary(central_longitude=central_lon, satellite_height=sat_h, sweep_axis=sweep, globe=globe)

# Add a subplot to the figure using the geostationary projection
ax = fig.add_subplot(1, 1, 1, projection=geos)

# Display the satellite data (CMI variable from ncx2) as an image in the subplot
# The 'origin' at 'upper' signifies the origin point of the coordinate system
# 'vmin' and 'vmax' set the data value limits for color mapping
# The 'extent' specifies the display boundaries in data coordinates
# The colormap 'irmap' is used for coloring, and 'transform' aligns the data with the specified projection
ir_img = ax.imshow(ncx2['CMI'].data, origin='upper', vmin=-109+273.15, vmax=56+273.15, extent=(x.min(),
                                                y.min(), x.max(), y.max()), cmap=irmap, transform=geos)

# Set the extent of the map display to specific longitude and latitude limits
ax.set_extent([-108.0, -114.0, 21.0, 35.0]) # Set the map display boundaries
```

Ok, you have completed the tutorial!