Introduction to xarray

Managing and analyzing multidimensional datasets

Programme: four blocks

Introduction to DataArrays and Datasets, reading/writing

Plotting capacities

First analyses

Scaling analysis with dask

```
import xarray as xr
```

Part I: xarray objects

DataArrays and Datasets

Comparison with pandas

Multivariable objects, aligned on similar axes/indices

- `pd.DataFrame`
- `xr.Dataset`

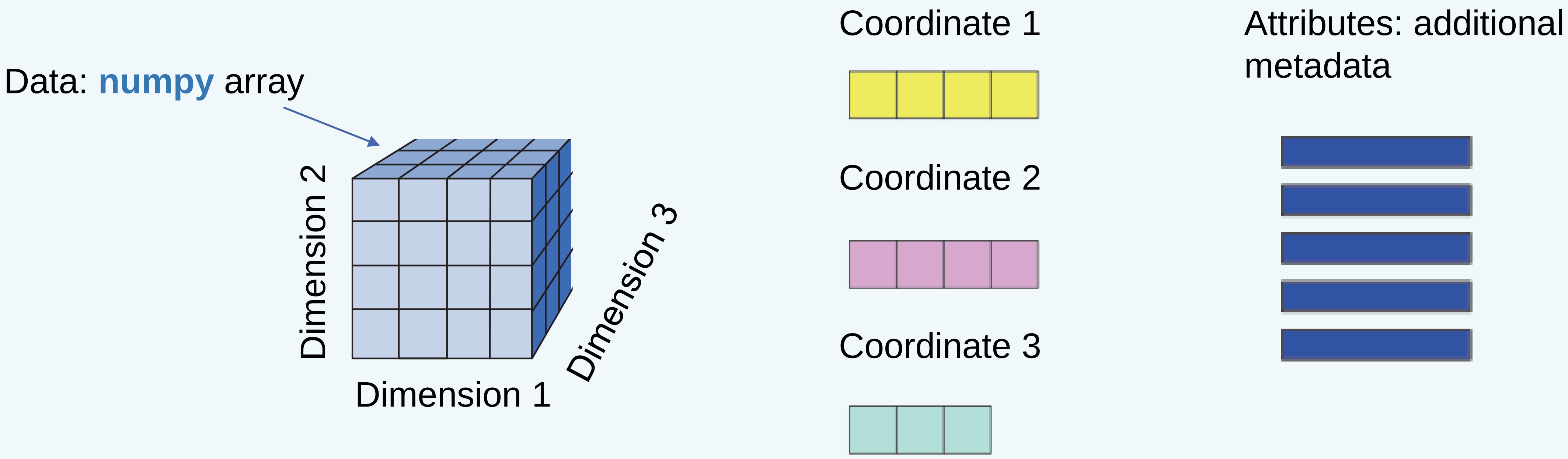
Single variable object:
- `pd.Series`
- `xr.DataArray`

Single variable object :
- `pd.Series`
- `xr.DataArray`

Single variable object :
- `pd.Series`
- `xr.DataArray`

xr.DataArray: presentation

- Array of values representing a **unique** variable:
  - « Wrapping » around a numpy array



xr.DataArray: presentation

- Exemple of a `DataArray`

xarray.DataArray 'thetao' (**time**: 312, **depth**: 13, **latitude**: 157, **longitude**: 265)

```
[168749880 values with dtype=float32]
```

▼ Coordinates:

depth	(depth)	float32	13.47 15.81 18.5 ... 77.85 92.33
latitude	(latitude)	float32	-5.0 -4.917 -4.833 ... 7.917 8.0
time	(time)	datetime64[ns]	1993-01-16T12:00:00 ... 2018-12-...
longitude	(longitude)	float32	-180.0 -179.9 ... -158.1 -158.0

▼ Attributes:

long_name :	Temperature
standard_name :	sea_water_potential_temperature
units :	degrees_C
unit_long :	Degrees Celsius
cell_methods :	area: mean
_ChunkSizes :	[1 7 341 720]

xr.DataArray: creating an object

- Source: an n-dimensional **numpy array** + n lists of **coordinates**

```
data = np.array([[1, 2, 3],
                 [5, 6, 7]])
longitude = [10,15,20]
latitude = [0, 5]
```

- Use `xr.DataArray()`

```
dataarray = xr.DataArray(data,
                         dims = ['latitude','longitude'],
                         coords = {'longitude':longitude,
                                   'latitude':latitude
                                  }
                        )
```

xarray.DataArray (**latitude**: 2, **longitude**: 3)

```
array([[1, 2, 3],
       [5, 6, 7]])
```

▼ Coordinates:

longitude	(longitude)	int32	10 15 20
latitude	(latitude)	int32	0 5

▶ Attributes: (0)

xr.DataArray: adding attributes

- `DataArray.attrs` is a **dictionnary** that can be modified

```python
dataarray.attrs['units'] = '°C'
dataarray.attrs['description'] = "Température de l'OMP"
```

xarray.DataArray (**latitude**: 2, **longitude**: 3)

```
array([[1, 2, 3],
       [5, 6, 7]])
```

▾ Coordinates:

longitude	(longitude)	int32	10 15 20
latitude	(latitude)	int32	0 5

▾ Attributes:

units :	°C
description :	Température de l'OMP

xr.DataArray: accessing the data

Xarray basics !
Access data using coordinates

```
dataarray.sel(longitude = 15)
```

xarray.DataArray (**latitude**: 2)

```
array([2, 6])
```

▾ Coordinates:

longitude	()	int32	15
latitude	(latitude)	int32	0 5

xr.DataArray: accessing the data

- Also works for a list of **multiple** coordinates

```
dataarray.sel(longitude = [15, 10], latitude = 5)
```

- Select using **position** (like in numpy)

```
dataarray.isel(longitude = [0,2])
```

- Access the numpy **array**

```
dataarray.values
```

xr.Dataset: presentation

- Multiple `DataArrays` with **shared dimensions**

xarray.Dataset

▸ Dimensions: (**depth**: 13, **latitude**: 157, **time**: 312, **longitude**: 265)

▾ Coordinates:

depth	(depth)	float32	13.47 15.81 18.5 ... 77.85 9...
latitude	(latitude)	float32	-5.0 -4.917 -4.833 ... 7.917 8.0
time	(time)	datetime64[ns]	1993-01-16T12:00:00 ... 201...
longitude	(longitude)	float32	-180.0 -179.9 ... -158.1 -158.0

▾ Data variables:

vo	(time, depth, latitude, longitude)	float32	...
thetao	(time, depth, latitude, longitude)	float32	29.57 29.59 29.61 ... 20.22 ...
uo	(time, depth, latitude, longitude)	float32	...
thetao_profile	(time, depth)	float32	28.46 28.45 28.45 ... 28.32 ...

▸ Attributes: (17)

xr.DataArray

Global attributes

Dimensions of each DataArray

xr.Dataset: creating an object

- Sources:
  - An ensemble of `DataArrays`

- `xr.Dataset()`

```
dataset = xr.Dataset({"temperature":data_temperature,
                      "salinity":data_salinity,
                      "precipitation":data_precipitation})
```

xr.Dataset: access the data

- **Access one** `DataArray`

```
dataset.temperature
dataset['temperature']
```

- Other access are **similar** to a `DataArray`:
  - `dataset.sel(latitude=…)`
  - `dataset.isel(longitude=[..])`

xr.Dataset: append data

- **Add a `DataArray` as a** new variable
  - Add a variable as if you wanted to access it:

```
dataset['ensoleillement'] = dataarray
```

- This can also be used to **replace** the values of a given `DataArray`

Operations on coordinates

- Simple operations between DataArrays based on **coordinates:**

```
DataArray1**2 + DataArray2/12
```

- Create **new variables** in a dataset from **existing variables**:

```
ds[‘new_variable’] = ds[‘old_v1’]**2 + ds[‘old_v2’]/12
```

Combine dataarrays or datasets

- Concatenate along **existing dimensions**
  - Example: different times but similar geographical grids

```
xr.concat([ds1, ds2], dim = 'latitude')
```

- Concatenate along **new dimensions**

```
xr.concat([ds1, ds2, ds3, ds4], dim = 'modele')
```

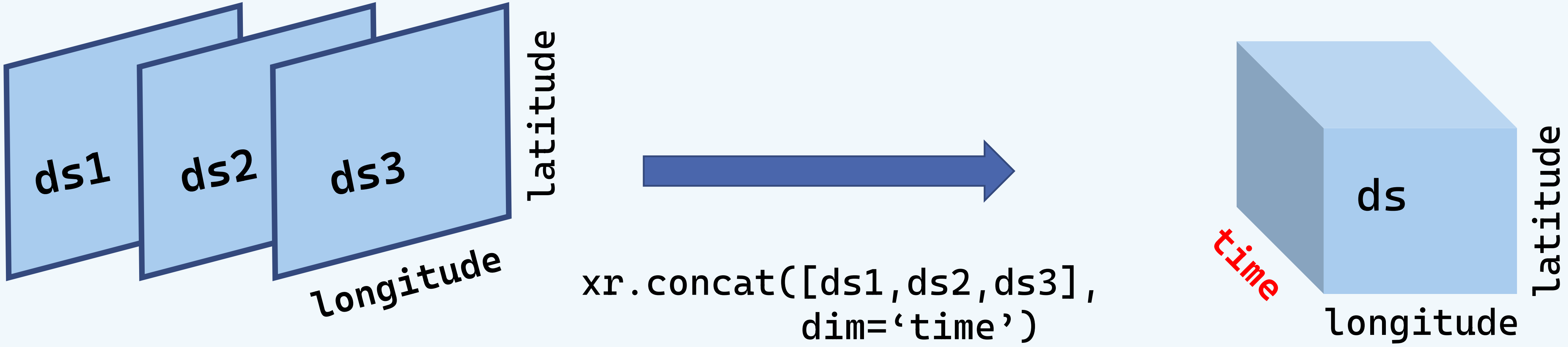
Combine dataarrays or datasets

```
ds1
```

xarray.DataArray 'thetao' (**latitude**: 157, **longitude**: 265)

```
xr.concat([ds1, ds2, ds3], dim='time')
```

xarray.DataArray 'thetao' (time: 3, **latitude**: 157, **longitude**: 265)



Reading/writing files: supported formats

- Xarray supports many different format for reading and writing
  - **netCDF**
  - Other examples
    - GRIB
    - geoTIFF
    - zarr
    - …
- Optimized for **gridded data**
- Can open directly **online** (OpenDAP, cloud buckets, …)

Reading/writing files: reading a file

- To **open** a netcdf file, use `xr.open_dataset()`

```
ds = xr.open_dataset('../data/GLORYS_ocean-temp-currents_1993-2019.nc')
```

- There is also `open_dataarray` if there is only **one** variable

- **Reading multiple files with `xr.open_mfdataset`:**
  - With a pattern

```
dataset = xr.open_mfdataset("../data/GLORYS*.nc")
```

  - With a list of paths

```
dataset = xr.open_mfdataset(["/path/to/file1.nc", "/path/to/file2.nc"])
```

Reading/writing files: writing a file

- To **save** a file, use `Dataset.to_format()`

```
ds.to_netcdf('path_to_file.nc')
```

Part I: Summary of xarray objects

Wrap numpy arrays with:

- **Dimensions** that have names (lon, lat, time, depth, altitude, …)
- **Coordinates** that have values
- **Attributes**

`DataArray`: for a unique variable (equivalent to `pd.Series`)

`Dataset`: for multiple variables on the same « grid » (equivalent to `pd.DataFrame`)

Access data using variable **names** and **coordinates**

```
ds.temperature.sel(lon=54, lat=[12,13,14])
```

Part I: Practicals

Go to the jupyter notebook

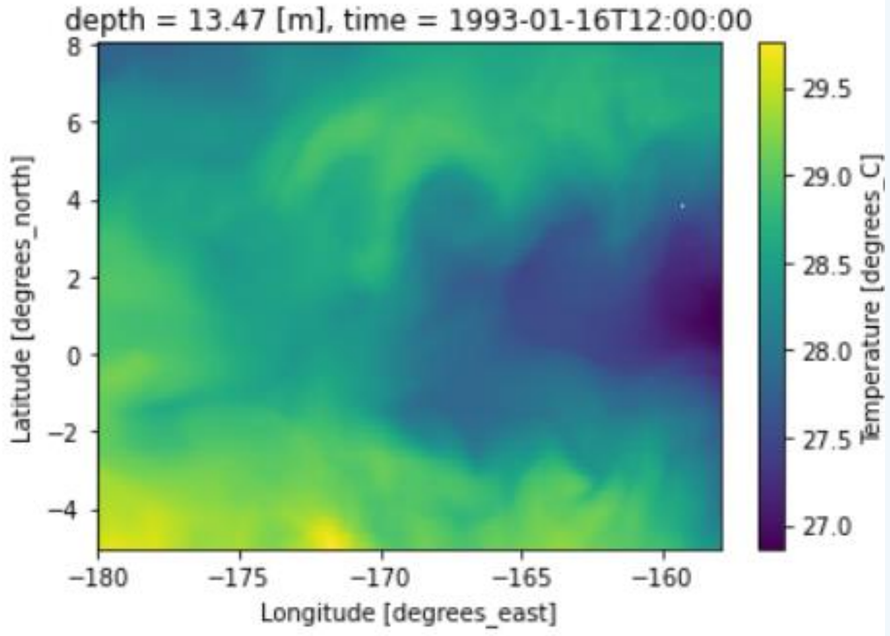
Part II: plotting capacities

Plotting `DataArrays`

- As pandas, xarray integrates **matplotlib** (+ other backends)

```
ds.thetao.isel(time=0, depth=0).plot()
```

```
<matplotlib.collections.QuadMesh at 0x183d4e58430>
```



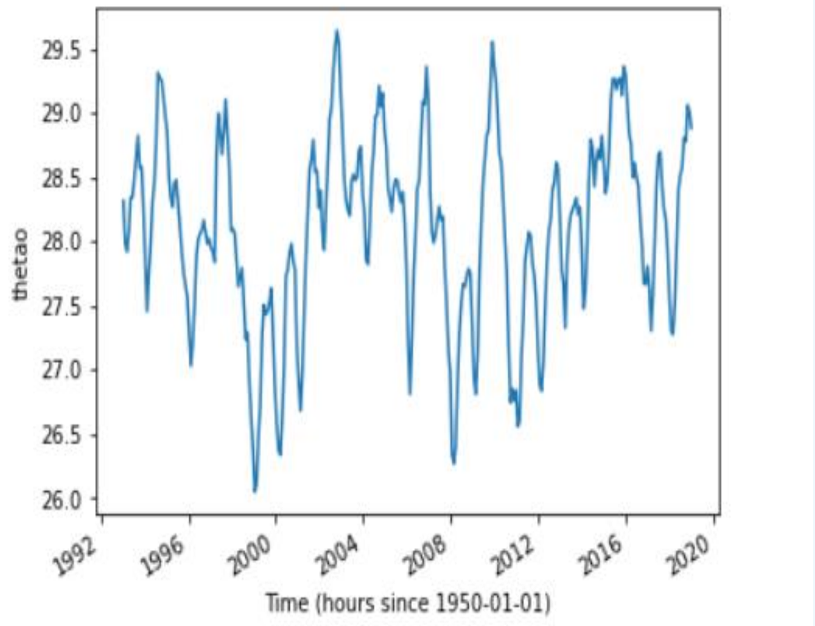
- Plot shows **coordinates**, **dimensions** and **attributes**

Plotting `DataArrays`: default behaviour

- **Line plot** for 1d data

```
ds.thetao.mean(['longitude','latitude','depth']).plot()
```

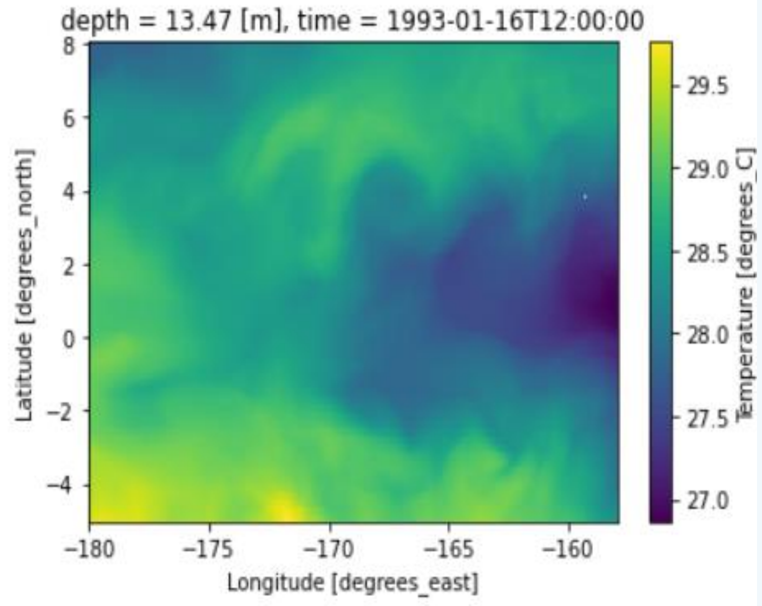
```
[<matplotlib.lines.Line2D at 0x183d4da0fa0>]
```



- **Quadmesh** for 2D data

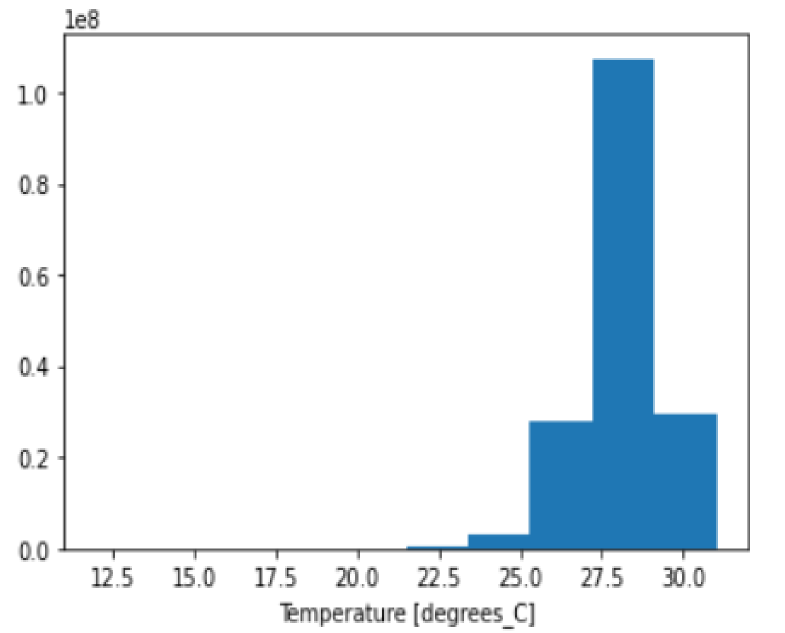
```
ds.thetao.isel(time=0, depth=0).plot()
```

```
<matplotlib.collections.QuadMesh at 0x183d4e58430>
```



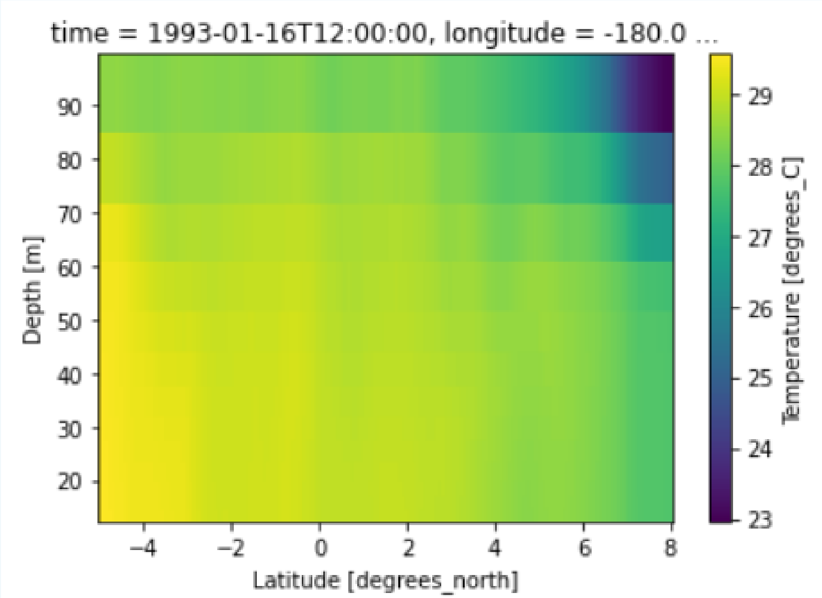
- **Histogram** if more than 2D

```
ds.thetao.plot();
```

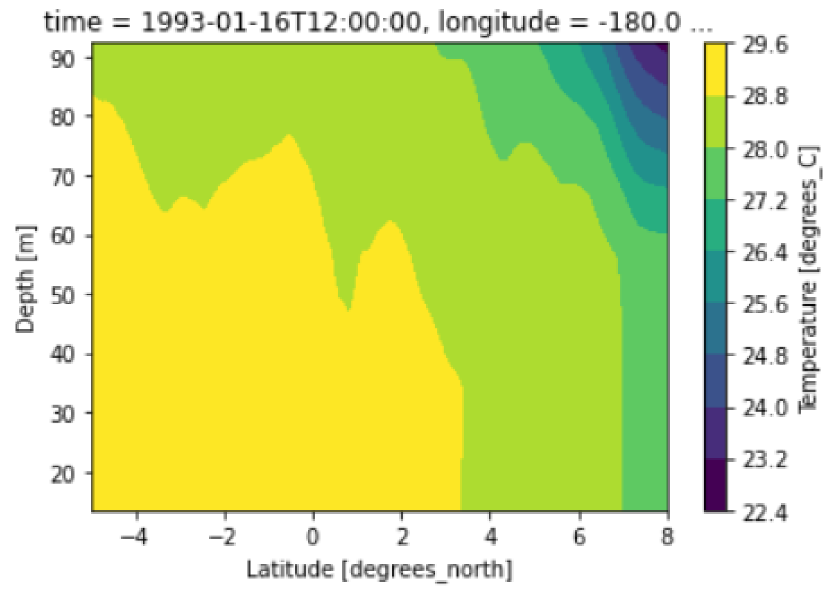


Plotting `DataArrays`: possible 2D plots

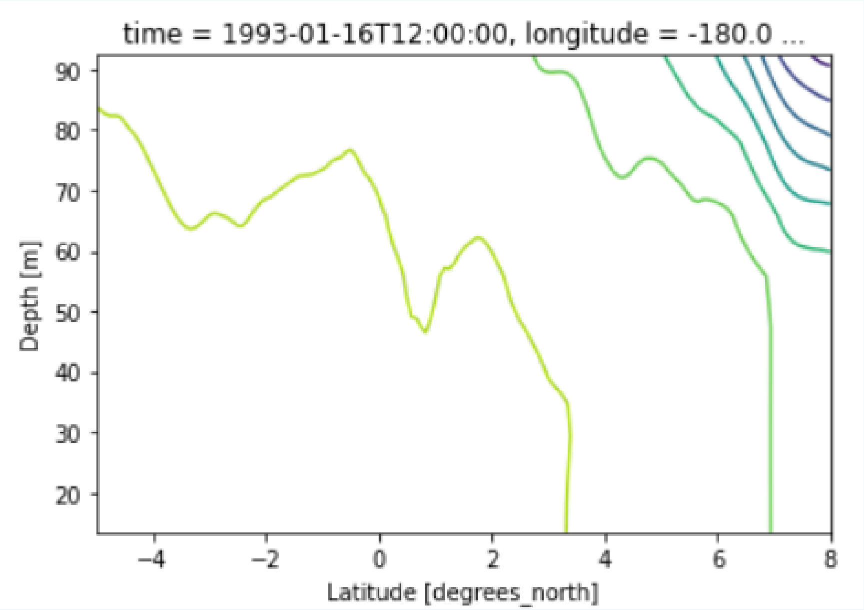
`DataArray.plot.quadmesh()`



`DataArray.plot.contourf()`



`DataArray.plot.contour()`



Important key words:

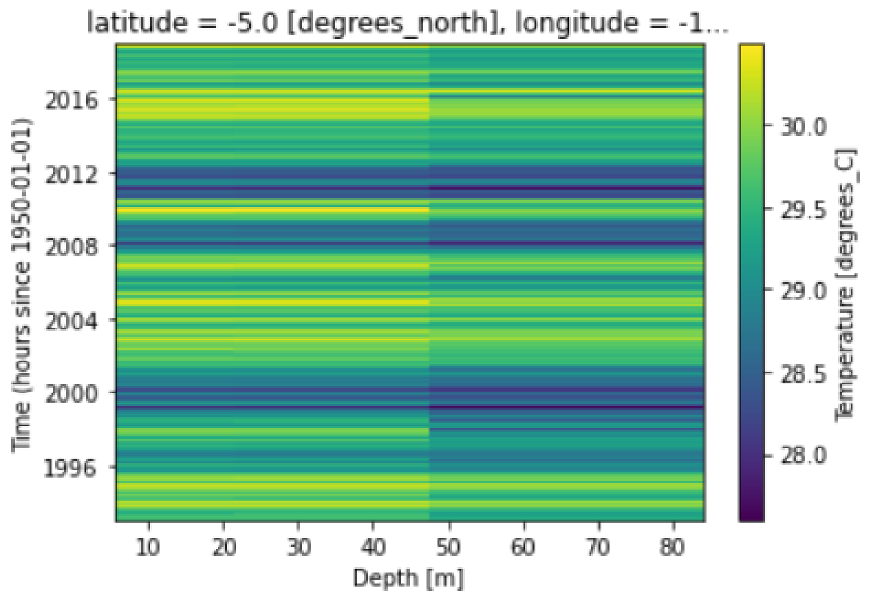
`x=‘longitude’`	Variable in x	`levels=10`	Number of levels or list
`y=‘depth’`	Variable in y	`cmap=‘magma’`	Changes colormap
`yincrease=True`	Reverses y values	`robust=True`	Truncates extreme values

Plotting `DataArrays`: reduce plot dimension

- Plot 2D data as **multiple line plots** with the keword **hue**

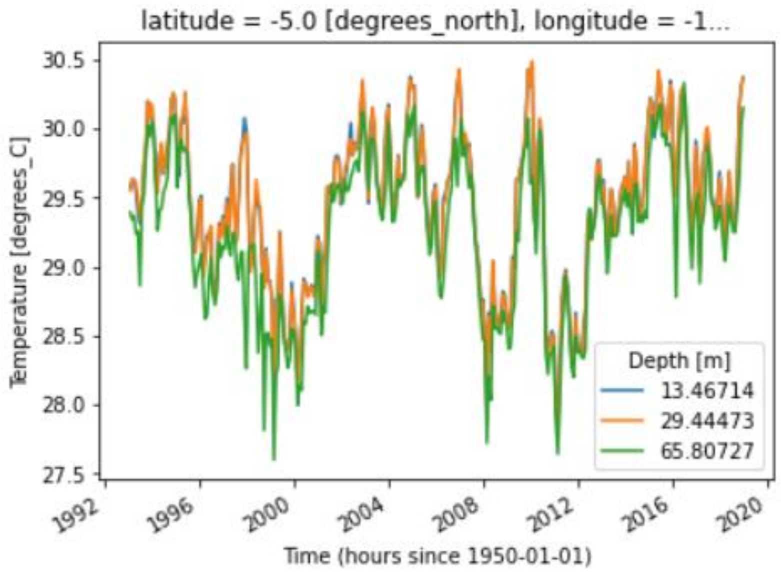
```
ds.isel(longitude=0, latitude=0, depth=[0,5,10]).thetao.plot()
```

```
<matplotlib.collections.QuadMesh at 0x1d29d264d30>
```




```
ds.isel(longitude=0, latitude=0, depth=[0,5,10]).thetao.plot(hue='depth')
```

```
[<matplotlib.lines.Line2D at 0x1d29d31cfd0>,
 <matplotlib.lines.Line2D at 0x1d29d31cd90>,
 <matplotlib.lines.Line2D at 0x1d29d2cc670>]
```

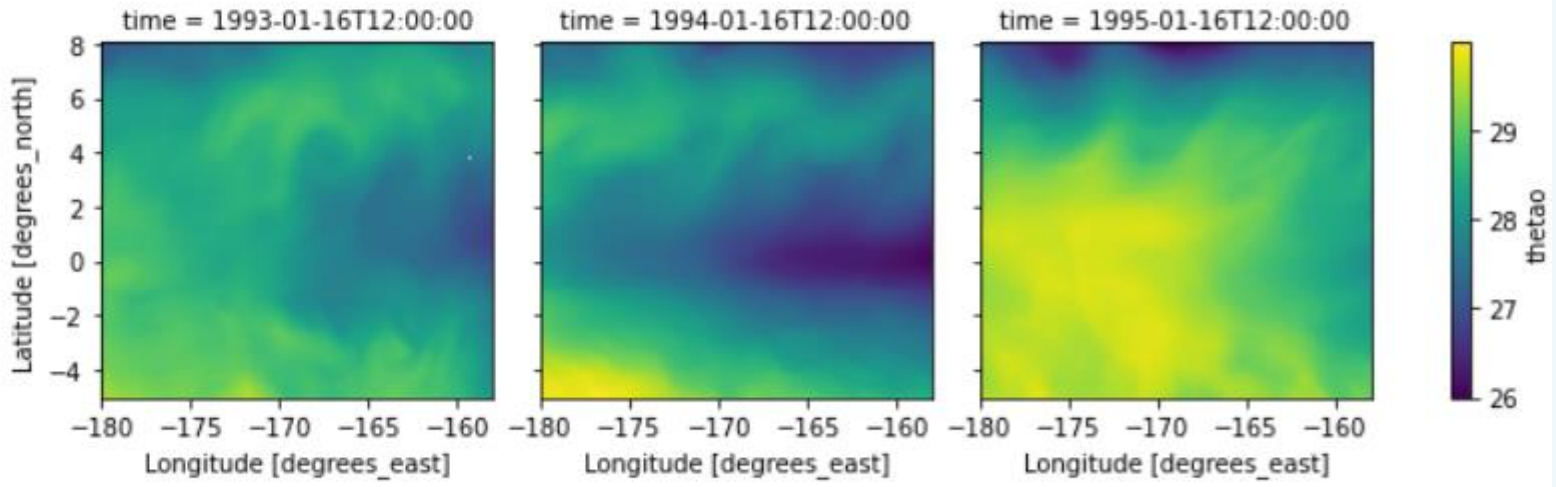


Plotting `DataArrays`: reduce plot dimension

- Plot 3D data as **multiple 2D plots** with the kewords `col/row`

```
ds.mean('depth').isel(time=[0,12,24]).thetao.plot(col='time')
```

```
<xarray.plot.facetgrid.FacetGrid at 0x1d29d307df0>
```



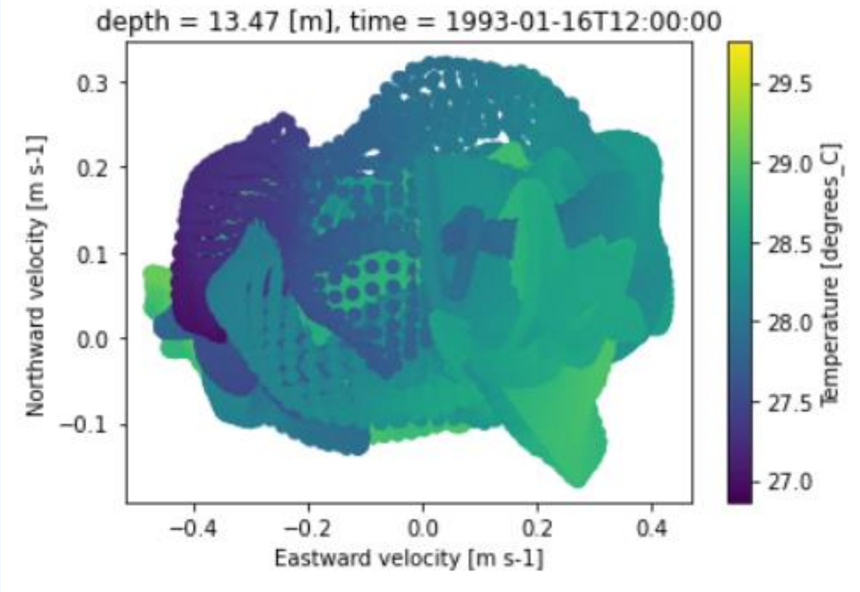
- Use `col_wrap=5` if there are **two many columns**

Plotting datasets

You can plot relationship between **different variables** of the **same dataset**

`Dataset.plot.scatter(x,y)`

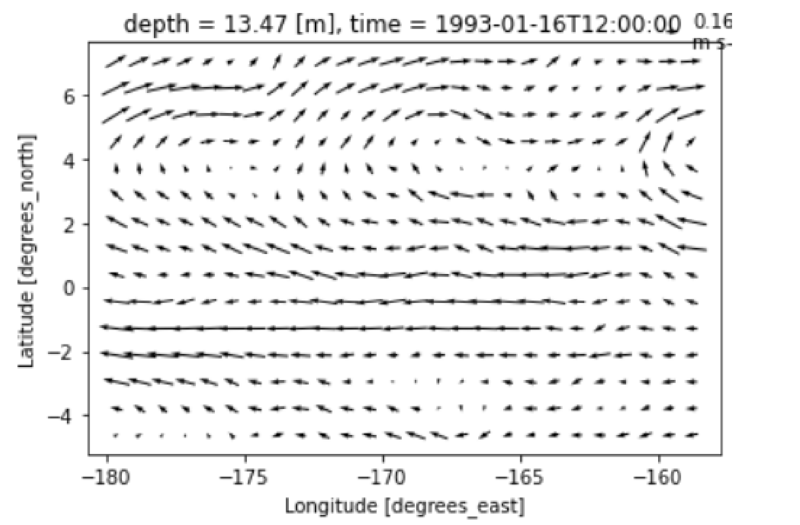
```
ds.isel(time=0, depth=0).plot.scatter(x='uo',
                                      y='vo',
                                      hue='thetao');
```



`Dataset.plot.quiver(x,y,u,v)`

```
ds.isel(time=0, depth=0).plot.quiver(x='longitude',
                                     y='latitude',
                                     u='uo',
                                     v='vo')
```

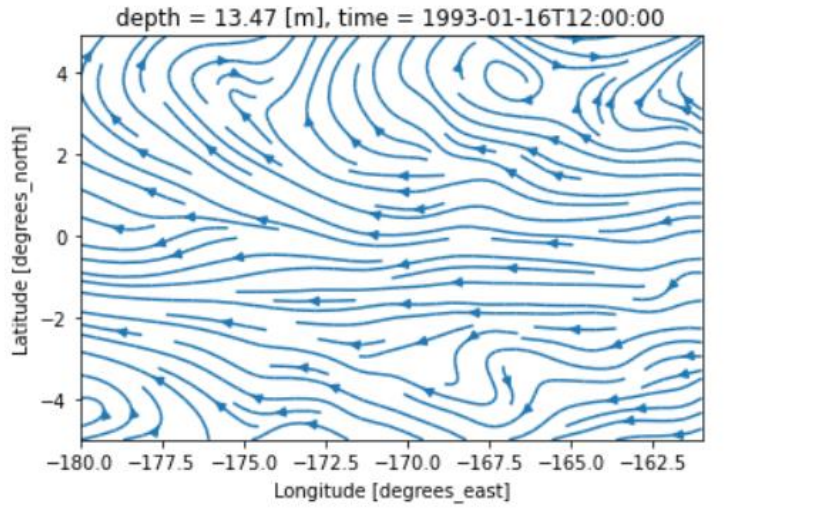
```
<matplotlib.quiver.Quiver at 0x1d2a0932400>
```



`Dataset.plot.streamplot(x,y,u,v)`

```
ds.isel(time=0, depth=0).plot.streamplot(x='longitude',
                                         y='latitude',
                                         u='uo',
                                         v='vo')
```

```
<matplotlib.collections.LineCollection at 0x1d29f5e8670>
```



Bonus: integration with `cartopy`

Import cartopy

```
import cartopy.crs as ccrs
```

Prepare figure

```
fig, ax = plt.subplots(
    figsize=(6,2),
    subplot_kw = {'projection':ccrs.Robinson(200)}
                        )
```

Declare projection

Plot data

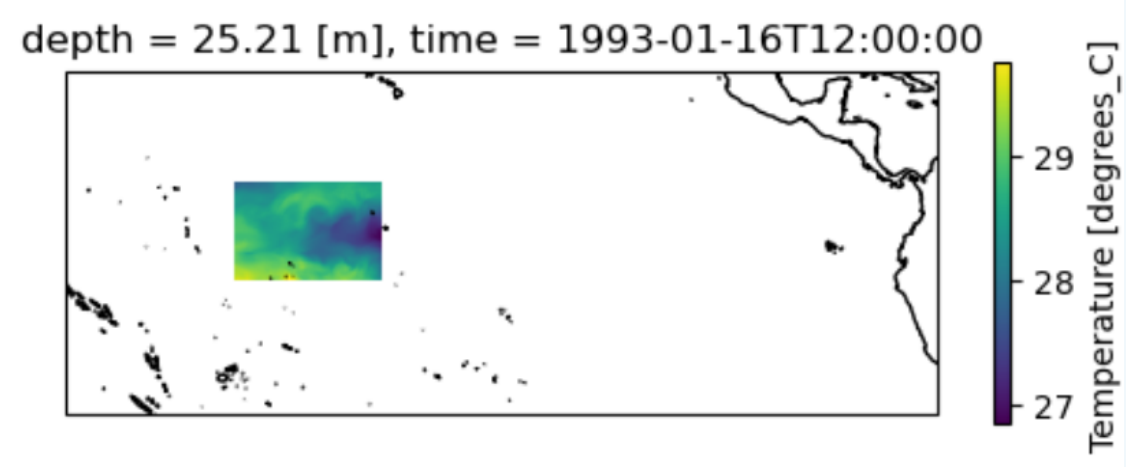
```
data = ds.thetao.isel(time=0, depth=4)

data.plot(ax=ax,
          transform= ccrs.PlateCarree()
         )
```

Declare coordinate system

Change limits/add coastline

```
ax.set_extent((155,285,-10,10))
ax.coastlines()
```



Part II: Practicals

Go to the jupyter notebook

Part III: First analyses

- Data selection
- Statistical operations
- Aggregations, …

Part III: First analyses

- **Data selection**
- Statistical operations
- Aggregations, …

Selecting data

- Select by **coordinates**

```
ds.sel(latitude=-4.75)
```

- Select by **position**

```
ds.isel(depth=5)
```

- **Unique** selection

```
ds.sel(latitude=-4.75)
```

- Selection **from list**

```
ds.isel(depth=[5,8])
```

```
ds.isel(longitude = np.arange(3,9))
```

- **Multicriteria**

```
ds.isel(depth=[5,8], longitude = 12)
```

- **Nearest neighbour**

```
ds.sel(latitude=50, method='nearest')
```

Selecting data

- **Range** of coordinates: `slice(beginning, end)`
  - Select **all values** between beginning and end

```
ds.sel(latitude=slice(-3,3))
```

- **Temporal** selection (similar to pandas):
  - **Exact** selection `ds.sel(time='1993-01-16')`
  - Selection of a **period** `ds.sel(time='2015')`
  - Selection of a **time span** `ds.sel(time=slice('2001', '2015-04'))`

Selecting data

- **Masking** data with `Dataset.where`
  - Selects the data that fulfill a certain **condition**, the rest will be NaN.
  - The condition is **a boolean dataarray** (`True/False`) on the same coordinates

```
ds.where(ds.uo>0.1)
```

```
ds.where(ds.uo>0.1, other=999)
```

  - When the condition is on the **coordinates**, **extract** the data with `drop=True`

```
ds.where(ds.latitude<-2, drop=True)
```

Part III: First analyses

- Data selection
- **Statistical operations**
- Aggregations, …

Numpy-like operations

- Use `DataArray.operation(‘dimension’)` for one object

```
ds.thetao.mean('depth')
```

```
ds.thetao.mean(['longitude', 'latitude'])
```

- Same for `Datasets`

```
ds.mean(['longitude', 'latitude'])
```

- Generally the same operations as in pandas
  - `mean(),sum(), min(), max(), median()`
  - `idxmax(), idxmin(), argmin(), argmax()`
  - `quantile([q1,q2, … ])`
  - `count()`

Some advanced methods

- **Difference** from one step to another

```
ds.diff('time')
```

- **Cumulative** sum

```
ds.cumsum('time')
```

- **Gradient**, **integral**

```
ds.differentiate('time')
ds.integrate('time')
```

Weighted operations

- **Weighted** operations are possible !

- The weights are a `DataArray` with **similar dimensions/coordinates**
  - Example: Global average weighted by cell area of a climate model

```
ds.weighted(cell_size).mean(['longitude','latitude'])
```

Fitting a DataArray or a dataset

- **Polynomial fits** or general **curve fits** as with scipy/numpy

```python
fit = ds.isel(depth=0).polyfit("time", deg=1)
```

xarray.Dataset

▸ Dimensions: (**degree**: 2, **latitude**: 157, **longitude**: 265)

▾ Coordinates:

degree	(degree)	int32	1 0
latitude	(latitude)	float64	-5.0 -4.917 -4.833 ... 7.917 8.0
longitude	(longitude)	float64	-180.0 -179.9 ... -158.1 -158.0

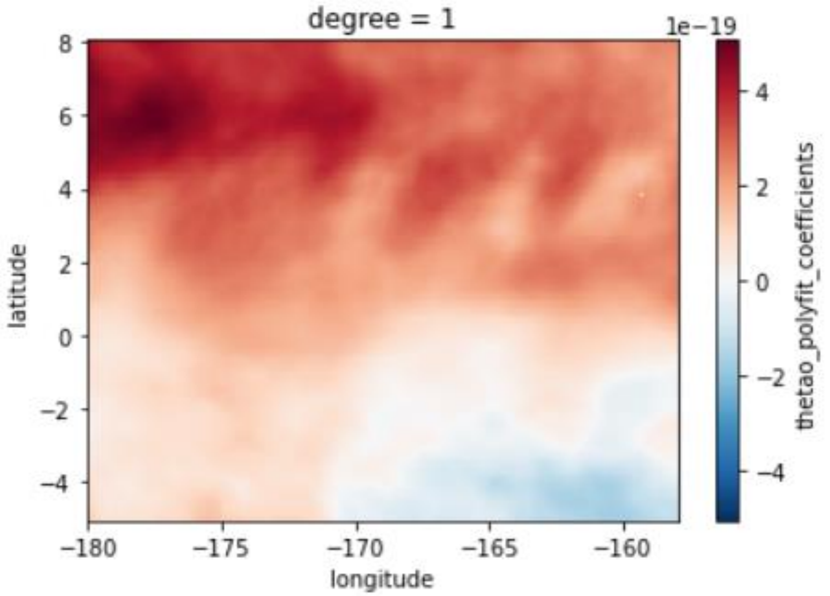
▾ Data variables:

vo_polyfit_coeffi...	(degree, latitude, longitude)	float64	-3.185e-20 -3.72e-20 ... 0.05072
thetao_polyfit_c...	(degree, latitude, longitude)	float64	1.37e-19 1.262e-19 ... 27.78 27.78
uo_polyfit_coeffi...	(degree, latitude, longitude)	float64	-5.6e-20 -5.629e-20 ... 0.105 0.106

▸ Attributes: (17)

```python
fit.thetao_polyfit_coefficients.sel(degree=1).plot()
```

```
<matplotlib.collections.QuadMesh at 0x1d29a8f54c0>
```




```python
def f(x, a,b):
    return a*x+b
fit = ds.isel(depth=0, longitude=0).curvefit("time",f)
```

Big warning: no loops !

There is (almost) always a way to replace a slow loop by a fast xarray function

Example : create **new dimension** instead of loop

BONUS: Applying custom function to some dimension

Define **custom function:**

```
def get_second_highest(data):
    sorted_data = np.sort(data)
    return sorted_data[-2]
```

Apply it to certain dimensions with `xr.apply_ufunc`:

```
xr.apply_ufunc(get_second_highest,
                ds.uo,
                input_core_dims=[['time']],
                output_core_dims=[[]],
                vectorize=True
               )
```

Part III: First analyses

- Data selection
- Statistical operations
- **Aggregations**, …

Resampling and rolling windows: in time

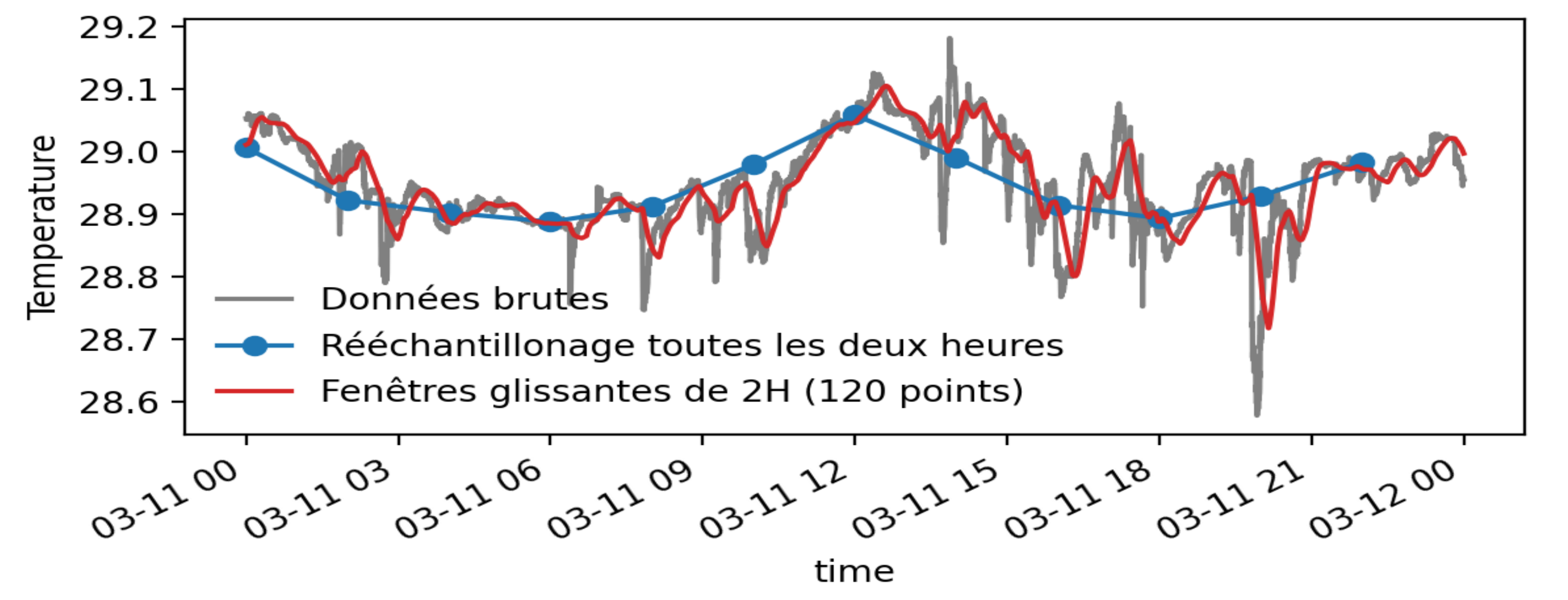
In time: similar to **pandas**

Resampling:

```
ds.resample(time="2H").mean()
```

Rolling windows:

```
ds.rolling(time=120).median()
```

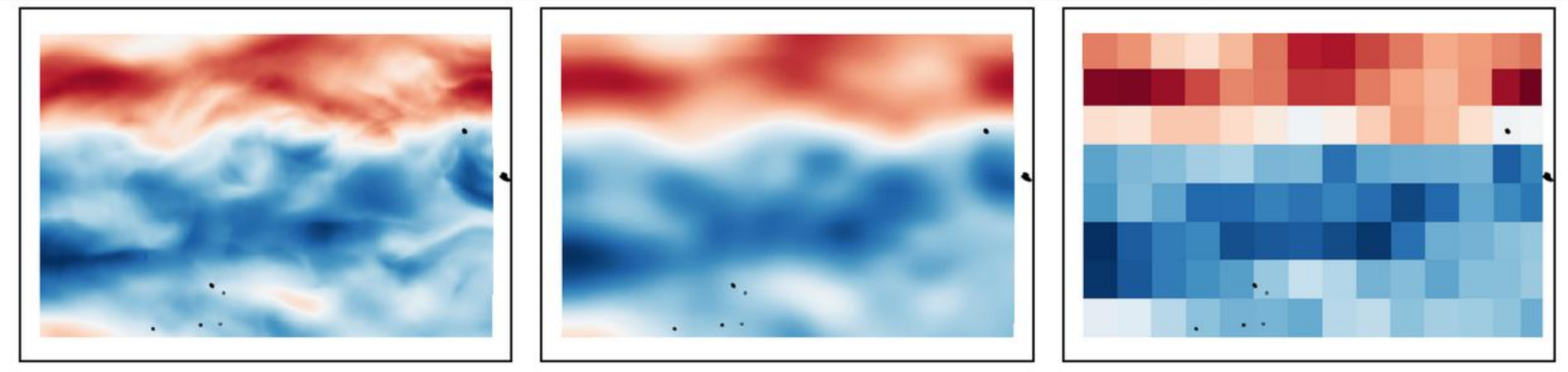


Resampling and rolling windows: in other dimensions

Resampling: `ds.coarsen(longitude=20, latitude=20, boundary='pad’).mean()`

Rolling windows: `ds.rolling(longitude=20, latitude=20, center=True).max()`

Raw `DataArray` | `DataArray.rolling(lon=20, lat=20)` | `DataArray.coarsen(lon=20, lat=20)`

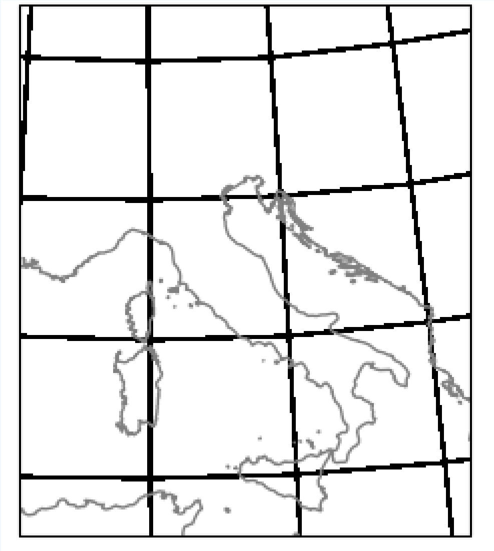


Bonus: regridding with `xesmf`

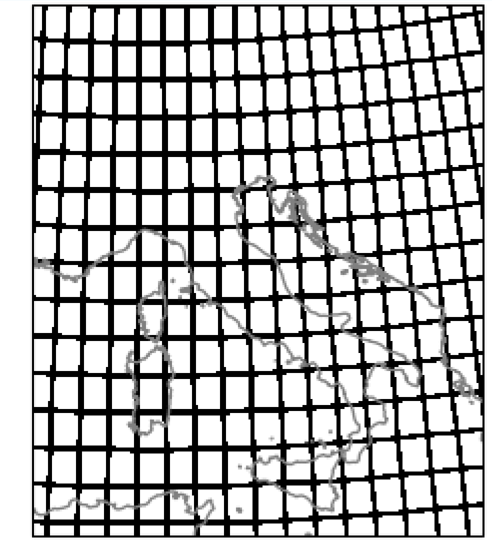
xESMF: Universal Regridder for Geospatial Data

xESMF is a Python package for regridding. It is

Grid 1



Grid 2



- **Curvilinear grid** (e.g. NEMO, global ocean model)
- **Rectilinear grid** (regular lon/lat grid)

Interpolation

- Interpolate on **new coordinate values**

```
ds.interp(latitude=np.arange(-5, 5, 0.1), method='linear')
```

- Fill **missing values**:

```
ds.interpolate_na('latitude', method='cubic')
```

Operations on groups

- Use `Dataset.groupby()` to compute on **separate groups** of data (as in pandas)
  - Apply operation on each group

```
small_ds.groupby(small_ds.uo//0.1).mean()
```

- **Temporal** `groupby` using `Dataset.groupby('time.XXX')`:

```
ds.groupby('time.month')
```

Bonus: faster groupby with `flox`

flox: Faster GroupBy reductions with Xarray

Tuesday, July 18th, 2023 (10 months ago)

Deepak Cherian

- Better and faster algorithms
- Optimized for parallel computing
- Works with dask (see end of the day…)

Summary: first analyses

Select data with coordinates	`ds.sel(longitude=12, latitude=slice(0,40), time="2012")`
Mask data with conditions	`ds.where(ds.temperature > 18)`
Operation along dimension	`ds.max("depth")`
Weighted operations	`ds.weighted(cell_size).mean(["longitude", "latitude"])`
Resampling in time	`ds.resample(time="2H").min()`
Other resampling	`ds.coarsen(longitude=10, latitude=5, boundary="trim").max()`
Rolling windows	`ds.rolling(depth=5, center=True)`
Interpolating on new coordinates	`ds.interp(longitude = [10, 20, 30], latitude=18)`

One-liners

- As in **pandas**, methods **return** an xarray object (`Dataset`, `DataArray`)
- Methods can be **chained** into one line:

```
ds.thetao.where(ds.uo>0.1)\
            .resample(time='Y').mean()\
            .sel(latitude=slice(-2,2))\
            .mean('longitude')\
            .integrate('depth')
```

- **ATTENTION: it can be better to split for development…**

Easy conversion to pandas objects

- xarray objects can be **directly converted** to **pandas object**

```
ds.thetao.to_series()
```

```
ds.to_dataframe()
```

Part III: Practicals

Go to the jupyter notebook

Part IV: Scaling with dask

Handling out of memory datasets & parallel computing

Scaling issues

xarray.DataArray 'thetao' (model: 100, **time**: 312, **depth**: 13, **latitude**: 157, **longitude**: 265)

- How to handle **large amount** of data (Here 63 GB)?
  - Need **large computers ?**

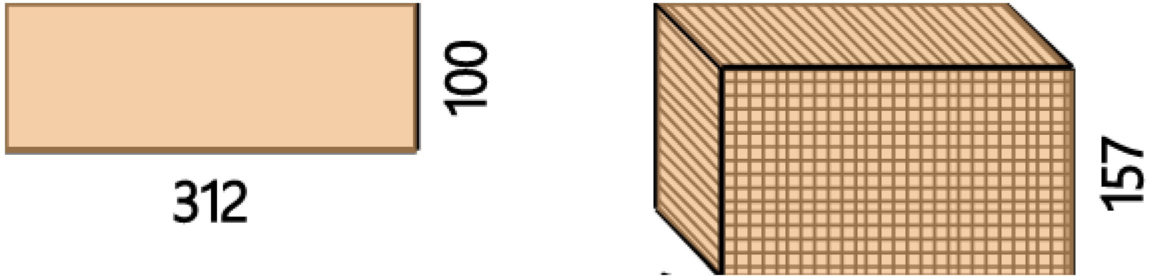
- Limited by **memory size** : can’t load **more** than memory size

Introducing chunks

- Split data into **unit boxes**

xarray.DataArray 'thetao' (model: 100, **time**: 312, **depth**: 13, **latitude**: 157, **longitude**: 265)

	Array	Chunk
Bytes	62.86 GiB	154.72 MiB
Shape	(100, 312, 13, 157, 265)	(100, 312, 13, 10, 10)
Count	865 Tasks	432 Chunks
Type	float32	numpy.ndarray

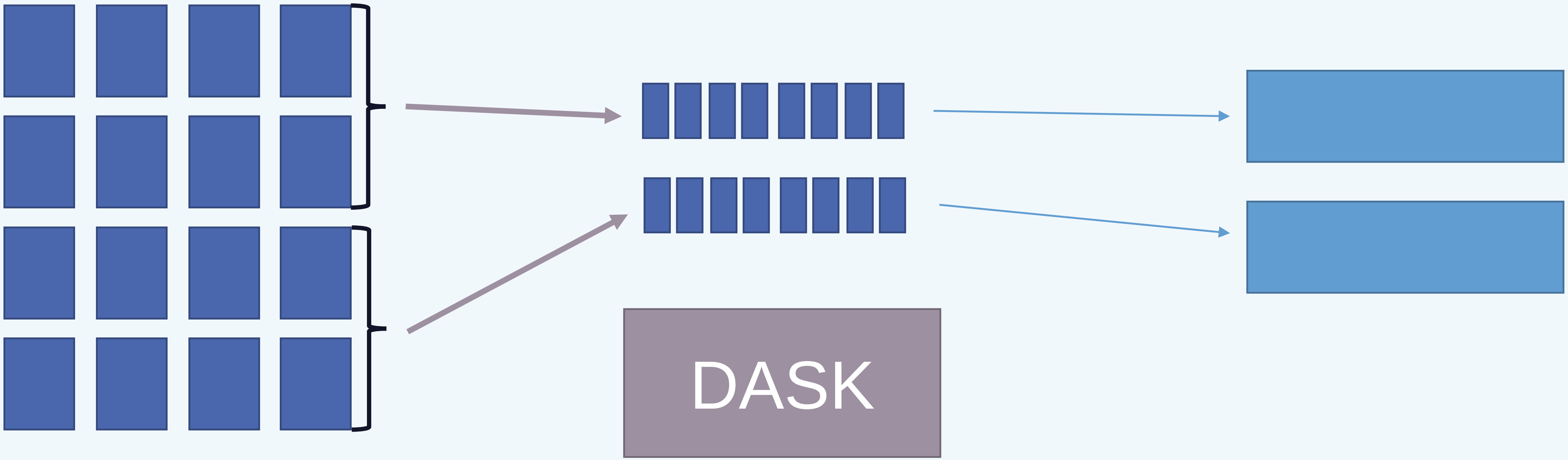


Lazy computing

- When opening a file, only **metadata**, **coordinates** and **dimensions** are loaded
- Compute stuff ...
- The data are loaded into memory only when they are **really needed** (plotting, saving results, printing values)

Parallel computing

- Operations on **individual chunks**
- Sort operations
- Computing them individually



Quick setup

Providing **resource** information:

```
from dask.distributed import Client

client = Client(n_workers=6)
```

Specify explicit **chunks**:

```
ds = xr.open_dataset("../data/GLORYS_ocean-temp-currents_1993-2019.nc",
                     chunks = {'longitude':10, 'latitude':10})
```

Open **multiple files**: one chunk per file

```
ds = xr.open_mfdataset("../data/GLORYS_*.nc")
```

Summary: why use dask?

- **Transparently** integrated with xarray
- Prevents **memory** issues
- Speeds up operations with **parallel computing**
- **Optimized for HPC and cloud**

Live example ...