BEYOND ARRAYS

A brief review of challenges in working with multidimensional downscaled climate data

Guilherme Baggio

7 May 2022

Contents

Summary

Much of human society and infrastructure has been built on the assumption that future climate conditions, including average temperature, precipitation, and the frequency and intensity of extreme events, would follow a similar baseline path experienced in the past. Infrastructure design and maintenance, emergency response management, and long-term investment and planning are all affected by this assumption. Human activities, however, are increasing greenhouse gas emissions in the atmosphere. As a result, human-caused climate change is interacting and altering both new and old patterns of climate variability globally. In this context, climate change impact assessments have become a tool to describe these changes and their risks to society. While these assessments frequently deal with the impacts of complex, interconnected systems on local contexts, climate scientists seem to be on the edge of many shifts. For instance, decision-makers are increasingly interested in climate data services that address their local, specific needs. These services are often the result of partnerships between climate scientists and decision-makers, such as city planners, resource managers, health officials, and farmers, involving a range of interactions across multiple fields of study. Climate scientists also have to deal with progressively larger datasets. As high-resolution climate data become available, there has been a proliferation of new approaches to access and manipulate these data. *Pandas*, *xarray*, and *dask*, for instance, are libraries developed to handle complex data structures in the Python ecosystem. Yet, these libraries are considerably recent, and many developers, scientists, and students rely on online forums as a recourse to the frustrating errors caused by the calculations that far exceed the available space in their servers. As a brief contribution to this discussion, this report shares some learnings from working with downscaled climate data. To achieve this goal, Chapter 1 introduces some of the challenges experienced by climate scientists in their attempt to produce fit-for-purpose high-resolution climate data and concisely describes the importance of array programming. Chapter 2 provides a step-by-step guide to building a xarray function that enables a parallel execution of calculations. In the context of this study, this function calculates the autocorrelation lag-1 and e-folding time of daily temperature. Finally, the key takeaways summarize some of the development aspects of this study.

Chapter 1

Introduction

1.1 Dowscalling climate data

One of the central issues in the global climate change discussion is the need for regional and local climate data that capture these changes. Such data is required to assess the effects of climate change on human and natural systems, as well as to develop appropriate adaptation and mitigation strategies. Particularly, end-users and decisionmakers have been looking for accurate and reliable climate data to guide response options at the regional and local scales (**??**). This includes high-resolution climate data, reanalysis data, and other monitoring products. For instance, ecosystem and agricultural impact assessments often require high-resolution climate data to assess climate change impacts due to the large spatial variations in natural and agricultural landscapes (**??**), while complex topographies and human-made infrastructures such as mountains, coastlines, lakes, irrigation, and urban heat islands can substantially influence a region's climate and its response to climate forcing (**??**).

To bridge this spatial scale gap, various downscaling techniques have been developed to improve low-resolution data from Global Climate Models (GCMs). For instance, *dynamical downscaling* makes use of physically based models, such as GCMs run in limited areas, as boundary conditions to reproduce local climate under different climate change scenarios. These Regional Climate Models (RCMs) are usually computationally intensive, and limited data are available for some regions (**?**). Alternatively, *statistical downscaling* relies on a mathematical relationship developed between historic observed climate data and the output of GCMs for the same historical period. This relationship is used to obtain the future downscaled climate data. Statistical downscaling can be also combined with different bias correction methods (**??**).

Nonetheless, many challenges have been identified in the long chain that ties climate data from GCMs (and the work of climate scientists) to user-friendly high-resolution climate data that can be used by decisionmakers. The UK Climate Change Act, for example, requires the Environment Agency to report the risks caused by climate change and possible actions to address them. However, identifying and providing climate products for decisionmakers remains a challenge (**?**). By partnering with decisionmakers from different organizations, climate scientists from the Great Lakes Integrated Sciences and Assessments program also observed unrealistic expectations on the development of climate information products (**?**). The challenge of moving from the production of climate information to its use in actual decision-making processes has also been described to as a matter of credibility and legitimacy (**?**), and as a challenge of using uncertain science that has also become highly politicized (**?**). At the same time, potentially useful climate information often goes unused due to the disparity between what scientists understand as useful information and what end-users recognize as usable in their decision-making processes (**?**).

1.2 Local data means more data

Climate scientists face another challenge. As downscaled climate data services become available, such as the *Copernicus Climate Change Service*, operated by the European Centre for Medium-Range Weather Forecasts, and *Climate-Data.ca*, a consortium of Canadian governmental and research organizations, the interest towards climate data for specific, local decision-making processes will likely increase (**??**). From a computational perspective, this challenge translates to an "explosion in data from numerical climate model simulations, which have increased greatly in complexity and size" (**?**).

To analyse, process, and present climate data, *array programming* offers a powerful and compact syntax for accessing, manipulating, and operating data. For instance, _NumPy, the most common array programming library for the Python language, has played a key role in the development of science over the last decades that (**?**). This library developed in the mid-1990s provided a structure that efficiently stores and accesses data in vectors, matrices, and higher-dimensional arrays and enables a wide variety of scientific computations (**?**). It also operates on in-memory arrays using the central processing unit (CPU), which means that the library runs on simple embedded devices as well as the world's largest computers. In this context, the Python ecosystem has proved extremely effective as a platform for climate scientists, and several packages have been developed over time to address specific needs, such as *ClimLab*, *CliMetLab*, *PyGeode*, and *MetPy*.

However, downscaled climate data, as any large dataset, can easily exceed the memory capacity of a single machine. This issue has been accompanied by the emergence of new approaches to handle large amounts of data, including distributed data and parallel execution of graphics processing units (GPUs) and tensor processing units (TPUs). Whereas NumPy's inability to utilize such approaches due to its in-memory data functioning led to a "gap between available modern hardware architectures and the tools necessary to leverage their computational power" (**?**), the scientific community's efforts to fill this gap contributed to a proliferation of new array implementations, including *pandas* in 2008, *xarray* in 2014, and *dask* in 2018.

1.3 Working with *xarray*

Chapter 2

Building a function in *xarray*

2.1 Working with *xarray*

Multi-dimensional arrays are an essential part of computational science. In Python, NumPy provides the fundamental syntax to work with this type of array. However, real-world datasets are usually more than just numbers. They often encode information about the location in space and time across multiple variables and provide qualitative information about of these data. Moreover, large datasets frequently require a faster computational execution of calculations not directly implemented in the NumPy library. In this context, the xarray library provides some of the key functionalities to work with large multi-dimensional arrays.

Firstly, the xarray library introduces the concept of labels in the form of dimensions, coordinates, variables, and attributes assembled on top of NumPy-like arrays. The dimension label specifies the axes of the data, while coordinates are arrays along a dimension. Examples of coordinates include latitude, longitude, days, and months. The variables contain the actual data of the dataset, such as temperature, precipitation, and atmospheric pressure. They are assigned for all the dimensions in a way that associates coordinates with variable values. And the attributes provide the metadata. This structure provides a more intuitive, concise workflow while keeping the well-known NumPy syntax.

Secondly, the xarray library provides tools for working with NetCDF files commonly used in climate sciences. These files allow for storing multi-dimensional data such as temperature and precipitation. Each of these variables can be displayed spatially (by having one value per coordinate of latitude and longitude, for example) and throughout time (by having one value per day, month, or year, for example). The multi-dimensional nature of xarray structures makes them particularly suitable for dealing with labelled scientific data stored in netCDF files. The use of dimension names in xarray instead of NumPy-like axis labels creates a more instinctive coding practice.

To organize the dimensions, coordinates, variables, and attributes of data, the xarray library has two fundamental data structures built upon the strengths of the NumPy and pandas libraries. A *DataArray* is the first structure defined by a multi-dimensional array with an underlying variable. Each DataArray contains dimensions, coordinates, and attributes of one variable only. *Datasets* are the second structure defined by a collection of DataArray objects aligned along with any number of shared dimensions. Since this structure is borrowed from the NetCDF file format, Datasets provide a more natural way to work with data stored in that type of file. Datasets also work as dictionaries, and most operations performed on the dimensions of a DataArray can be performed on Datasets as well.

2.2 Creating a simple function in *xarray*

First, let's import the key libraries for creating a function that calculates the average temperature from daily data for each point in space defined by its coordinates `latitude` and `longitude`. This example shows how to use well-known Numpy functions, such as `mean()` an `std()`, in a xarray Dataset.

```
import numpy as np
import xarray as xr
```

The following line uses the command `xr.open_dataset` to open the NetCDF file containing daily temperature for one year. This command will be important later when load a large file in chunks. For now, a simple line is sufficient.

```
t_daily = xr.open_dataset('anusplin_tasmax_1year.nc')
```

Once we load NetCDF file, we can view all the labels in the Dataset.

```
t_daily
```

xarray.Dataset

Dimensions:

lat: 510

lon: 1068

time: 365

```
Coordinates: (3)
lat
(lat)
float64
41.04 41.12 41.21 … 83.37 83.46
long_name :
lat
standard_name :
latitude
NAME :
lat
units :
degrees_north
_Netcdf4Dimid :
1
CLASS :
DIMENSION_SCALE
axis :
Y
lon
(lon)
float64
-141.0 -140.9 … -52.13 -52.04
long_name :
lon
standard_name :
longitude
NAME :
lon
units :
degrees_east
```

```
_Netcdf4Dimid :
0
CLASS :
DIMENSION_SCALE
axis :
X
time
(time)
datetime64[ns]
1950-01-01 ... 1950-12-31
standard_name :
time
NAME :
time
_Netcdf4Dimid :
2
CLASS :
DIMENSION_SCALE
Data variables: (1)
tasmax
(time, lat, lon)
float32
...
units :
degC
Attributes: (14)
grid_resolution :
Square grid, 0.08333333 degrees_north by 0.08333333 degrees_east
_nc3_strict :
1
contact1 :
```

Alex Cannon

contact2 :

acannon@uvic.ca

title :

ANUSPLIN interpolated daily minimum temperature, maximum temperature, and precipitation 300 arc second grids

CDO :

Climate Data Operators version 1.6.9 (http://mpimet.mpg.de/cdo)

CDI :

Climate Data Interface version 1.6.9 (http://mpimet.mpg.de/cdi)

Conventions :

CF-1.4

version_comment :

141.00 to 52.00 W, 41.00 to 84.00 N

version :

ANUSPLIN data obtained 2 April 2012, 14 June 2012, and 30 January 2013 from Pia Papadopol (pia.papadopol@nrcan-rncan.gc.ca)

NCO :

netCDF Operators version 4.7.5 (Homepage = http://nco.sf.net, Code = http://github.com/nco/nco)

input_data :

[pcp,min,max]YYY_DOY.asc ASCII grid files

reference :

McKenney, D.W., et al., 2011. Customized Spatial Climate Models for North America. Bulletin of the American Meteorological Society, 92(12):1611-1622.

history :

Thu Jan 27 15:50:07 2022: ncks -d time,0,364 anusplin_tasmax_final.nc test.nc Fri Aug 05 10:39:54 2016: cdo -b f32 copy anusplin_tasmax.nc anusplin_tasmax_unpack.nc Fri Aug 5 10:03:40 2016: ncks -v tasmax anusplin_300_canada_daily_standard_grids_1950-2010.nc anusplin_tasmax.nc

The xarray library allows for using most of the functions available in the pandas library, such as the command `groupby()`, with the advantage of relying on the dimension labels to execute the NumPy function. For example, it is possible to easily calculate the weekly average temperature.

```
t_week = t_daily.groupby('time.week').mean('time').rename({"tasmax": "t_mean"})
```

xarray.Dataset

Dimensions:

lat: 510

lon: 1068

week: 52

Coordinates: (3)

lat

(lat)

float64

41.04 41.12 41.21 ... 83.37 83.46

long_name :

lat

standard_name :

latitude

NAME :

lat

units :

degrees_north

_Netcdf4Dimid :

1

CLASS :

DIMENSION_SCALE

axis :

Y

lon

(lon)

float64

-141.0 -140.9 ... -52.13 -52.04

long_name :

lon

standard_name :

longitude

NAME :

lon

units :

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

week

(week)

int64

1 2 3 4 5 6 7 ... 47 48 49 50 51 52

Data variables: (1)

t_mean

(week, lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (0)

As we can see, this code automatically adds a new dimension `week` to the Dataset. Similarly, we can calculate the monthly average temperature.

```
t_month = t_daily.groupby('time.month').mean('time').rename({"tasmax": "t_mean"})
```

xarray.Dataset

Dimensions:

lat: 510

lon: 1068

```
month: 12
Coordinates: (3)
lat
(lat)
float64
41.04 41.12 41.21 … 83.37 83.46
long_name :
lat
standard_name :
latitude
NAME :
lat
units :
degrees_north
_Netcdf4Dimid :
1
CLASS :
DIMENSION_SCALE
axis :
Y
lon
(lon)
float64
-141.0 -140.9 … -52.13 -52.04
long_name :
lon
standard_name :
longitude
NAME :
lon
units :
```

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

month

(month)

int64

1 2 3 4 5 6 7 8 9 10 11 12

Data variables: (1)

t_mean

(month, lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (0)

As seen in the codes above, the command `mean('time')` is executing the NumPy function `mean` on the the time dimension of the Dataset and automatically creating a new dimension `month` with 12 elements. We can now plot the monthly mean temperature.

```
fig, axs = plt.subplots(4,3, figsize=(12, 16), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace = .0, wspace= .2)

month = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']

axs = axs.ravel()

for i in range(len(month)):
    pc = axs[i].pcolormesh(t_month.lon,t_month.lat,t_month['t_mean'][i,:,:],cmap="Reds",vmin=-40,
    axs[i].set_title(month[i])
    axs[i].set_ylim([39,81])
    axs[i].set_xlim([-141,-59])

    if i in [0,3,6,9]:
        axs[i].set_yticks([40,50,60,70,80])
    else:
        axs[i].set_yticks([])

    if i in [9,10,11]:
        axs[i].set_xticks([-140,-120,-100,-80,-60])
    else:
        axs[i].set_xticks([])

fig.suptitle('Monthly Mean Temperature ($^{\circ}$C)',
             x=0.5,y=1.0,fontsize=18,fontweight="bold")

cax,kw = mpl.colorbar.make_axes(axs,location='bottom',pad=-0.48,shrink=0.4)
out=fig.colorbar(pc,cax=cax,extend='both',**kw)

out.set_label("Temperature ($^{\circ}$C)", fontweight='bold',fontsize=14)

fig.tight_layout()
plt.show()
```

We can use the command `std('time')` to calculate the standard deviation of the temperature over the time dimension.

```
t_std = t_daily.groupby('time.month').std('time').rename({"tasmax": "t_std"})
```

Xarray also borrows some features from the pandas library to work on the time dimension. The code bellow calculates the mean temperature for each season.

```
t_season = t_daily.groupby('time.season').mean('time').rename({"tasmax": "t_mean"})
t_season
```

xarray.Dataset

Dimensions:

lat: 510

lon: 1068

season: 4

Coordinates: (3)

lat

(lat)

float64

41.04 41.12 41.21 ... 83.37 83.46

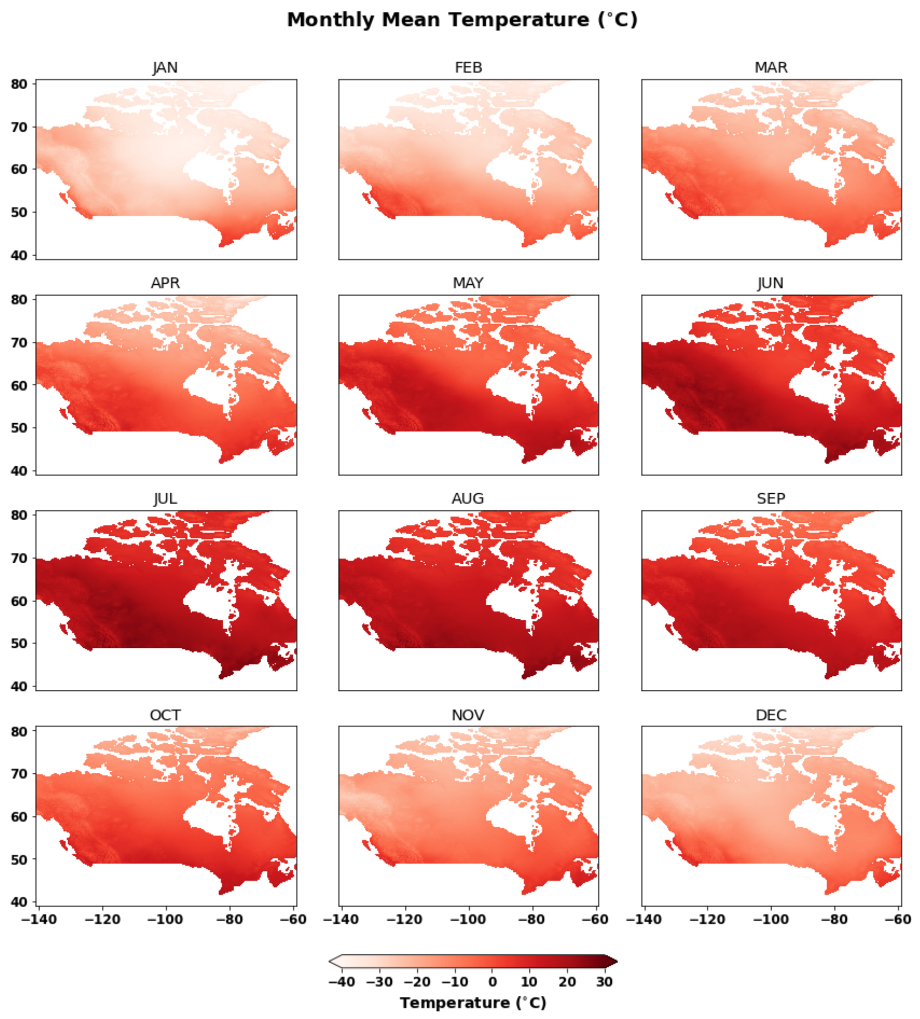

Figure 2.1: png

long_name :

lat

standard_name :

latitude

NAME :

lat

units :

degrees_north

_Netcdf4Dimid :

1

CLASS :

DIMENSION_SCALE

axis :

Y

lon

(lon)

float64

-141.0 -140.9 ... -52.13 -52.04

long_name :

lon

standard_name :

longitude

NAME :

lon

units :

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

season

(season)

object

'DJF' 'JJA' 'MAM' 'SON'

Data variables: (1)

t_mean

(season, lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (0)

This Dataset now has a new dimension `season` with 4 elements. We can now plot the seasonal mean temperature.

```
fig, axs = plt.subplots(2,2, figsize=(12, 14), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace = .0, wspace= .2)

season = ['DFJ','JJA','MAM','SON']

axs = axs.ravel()

for i in range(len(season)):
    pc = axs[i].pcolormesh(t_season.lon,t_season.lat,t_season['t_mean'][i,:,:],cmap="Reds",vmin=-
    axs[i].set_title(season[i])
    axs[i].set_ylim([39,81])
    axs[i].set_xlim([-141,-59])

    if i in [0,2]:
        axs[i].set_yticks([40,50,60,70,80])
    else:
        axs[i].set_yticks([])

    if i in [2,3]:
        axs[i].set_xticks([-140,-120,-100,-80,-60])
    else:
        axs[i].set_xticks([])

fig.suptitle('Seasonal Mean Temperature ($^{\circ}$C)',
             x=0.5,y=1.0,fontsize=18,fontweight="bold")
```

```
cax,kw = mpl.colorbar.make_axes(axs,location='bottom',pad=-0.48,shrink=0.4)
out=fig.colorbar(pc,cax=cax,extend='both',**kw)

out.set_label("Temperature ($^{\circ}$C)", fontweight='bold',fontsize=14)

fig.tight_layout()
plt.show()
```

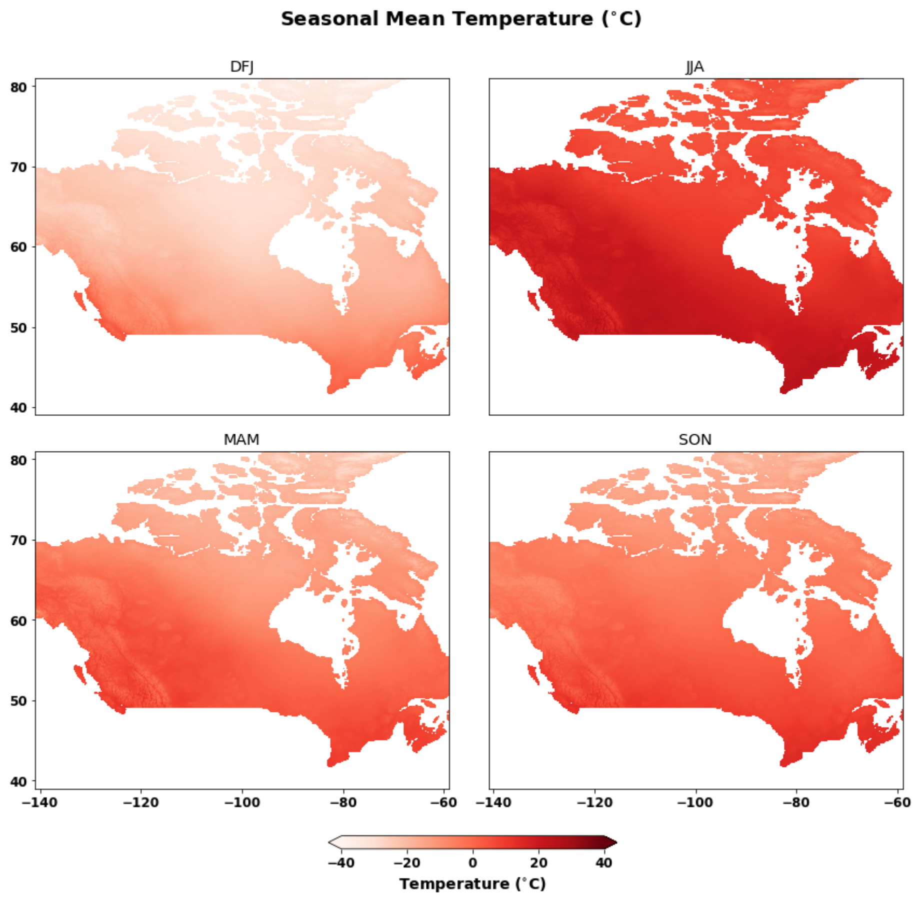

Figure 2.2: png

2.3 Multi-model mean in *xarray*

Now, let's say we have daily temperature data for 24 GCMs. How do we calculate the the mean over the time dimension for each model? We can do that by using a `for` loop as defined in the NumPy library.

For the sake of simplicity, the following code only enters the names of each one of the GCMs. Ultimately, you can use the command `xr.open_dataset` to open the NetCDF files containing the daily data for each one of the models.

```
model_name = ["ACCESS1-0","bcc-csm1-1","bcc-csm1-1-m","BNU-ESM","CanESM2","CCSM4","CESM1-CAM5",
              "CNRM-CM5","CSIRO-Mk3-6-0","GFDL-CM3","GFDL-ESM2G","GFDL-ESM2M","HadGEM2-AO",
              "HadGEM2-CC","inmcm4","IPSL-CM5A-LR","IPSL-CM5A-MR","MIROC5","MIROC-ESM-CHEM",
              "MIROC-ESM","MPI-ESM-LR","MPI-ESM-MR","MRI-CGCM3","NorESM1-ME","NorESM1-M"]
```

Before running the code, we need to import the pandas library. As discussed previously, the xarray library relies on several NumPy and pandas commands.

```
import pandas as pd
```

In the code below, we use the function `append()` to create a DataArray with average monthly temperature values for each GCM. In some way, we are creating a long array that contains the monthly average temperature for all 25 GCMs.

```
t_model_monthly=[]

for file_name in multimodel:
        t_daily = xr.open_mfdataset(file_name)
        t_mean = t_daily.mean('time')
        t_model_monthly.append(t_mean)
```

However, the new variable `t_mean_monthly` is not very practical because xarray does not label the 25 GCMs under a new dimension `model` automatically. The Dataset created from the loop is a collection of 25 xarray objects. To create a new dimension, we can use the xarray function `xr.concat` to concatenate the xarray objects along a new or existing dimension. In this specific case, we want a new dimension.

```
t_model_monthly = xr.concat(t_model_monthly,pd.Index(range(25),name='model')).rename({"tasmax": '
```

In this code, `pd.Index(range(25)` represents the new dimension added along axis=0.

The resulting Dataset is, thus, a new Dataset with a new dimension `model` wtih 25 elements on it.

```
t_model_monthly
```

```
xarray.Dataset
Dimensions:
lat: 510
lon: 1068
model: 25
Coordinates: (3)
lat
(lat)
float64
41.04 41.12 41.21 … 83.37 83.46
long_name :
latitude
standard_name :
latitude
NAME :
lat
units :
degrees_north
_Netcdf4Dimid :
1
CLASS :
DIMENSION_SCALE
axis :
Y
lon
(lon)
float64
-141.0 -140.9 … -52.13 -52.04
long_name :
longitude
standard_name :
```

```
longitude
NAME :
lon
units :
degrees_east
_Netcdf4Dimid :
0
CLASS :
DIMENSION_SCALE
axis :
X
model
(model)
int64
0 1 2 3 4 5 6 ... 19 20 21 22 23 24
Data variables: (1)
t_mean
(model, lat, lon)
float32
dask.array<chunksize=(1, 510, 1068), meta=np.ndarray>
Array
Chunk
Bytes
51.94 MiB
2.08 MiB
Shape
(25, 510, 1068)
(1, 510, 1068)
Count
150 Tasks
25 Chunks
```

Type

float32

numpy.ndarray

1068 510 25

Attributes: (0)

Now, we plot the average temperature for each model.

```
fig, axs = plt.subplots(5,5, figsize=(20, 25), facecolor='w', edgecolor='k')
fig.subplots_adjust(hspace = .2, wspace= .2)

axs = axs.ravel()

for i in range(len(t_model_monthly['model'])):

    pc = axs[i].pcolormesh(t_model_monthly.lon,t_model_monthly.lat,t_model_monthly['t_
    axs[i].set_title(multimodel_name[i])
    axs[i].set_ylim([39,81])
    axs[i].set_xlim([-141,-59])
    axs[i].set_yticks([40,50,60,70,80])
    axs[i].set_xticks([-140,-120,-100,-80,-60])
    cax,kw = mpl.colorbar.make_axes(axs[i],location='bottom')#,pad=0.05,shrink=0.7)
    out=fig.colorbar(pc,cax=cax,extend='both',**kw)    #axs[i].axis("tight")

fig.suptitle('Mean Temperature ($^{\circ}$C) for each GCM',
             x=0.5,y=0.91,fontsize=18,fontweight="bold")

#fig.tight_layout()
plt.show()
```

The beauty of creating a new dimension `model` to store the values for each model is that we can easily calculate the multi-model ensemble mean temperature by simply executing the operation `mean()` over the model dimension.

```
t_ensemble = t_model_monthly.mean('model')
```

xarray.Dataset

Dimensions:

lat: 510

lon: 1068

Coordinates: (2)

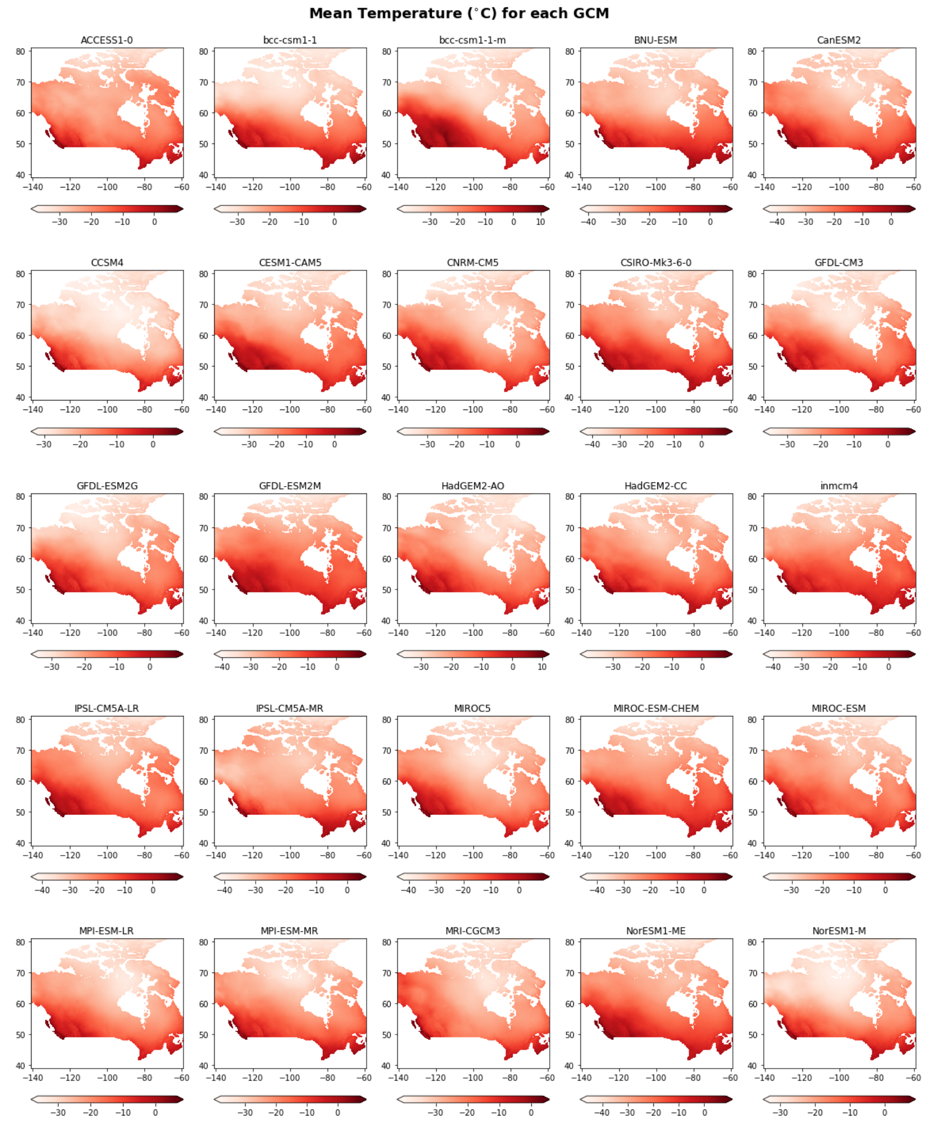

Figure 2.3: Figure 1 - Mean temperature calculated for each GCM

lat

(lat)

float64

41.04 41.12 41.21 ... 83.37 83.46

long_name :

latitude

standard_name :

latitude

NAME :

lat

units :

degrees_north

_Netcdf4Dimid :

1

CLASS :

DIMENSION_SCALE

axis :

Y

lon

(lon)

float64

-141.0 -140.9 ... -52.13 -52.04

long_name :

longitude

standard_name :

longitude

NAME :

lon

units :

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

Data variables: (1)

t_mean

(lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (0)

Finally, we can plot the multi-model ensemble mean.

```
fig = plt.figure(figsize=(16,12))
ax = plt.axes(projection=ccrs.PlateCarree())
#ax.set_global()
ax.coastlines()
ax.gridlines(linewidth=1)
gl = ax.gridlines(crs=ccrs.PlateCarree(), draw_labels=True,
                 linewidth=1, color='darkgrey')
gl.xlabels_top = False
gl.ylabels_left = False
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER
gl.xlabel_style = {'size': 15, 'color': 'gray'}
gl.xlabel_style = {'color': 'black', 'weight': 'bold'}

# uncomment and complete the line below (see the NAO notebook for a reminder)
pc = ax.pcolormesh(t_ensemble.lon,t_ensemble.lat,t_ensemble['t_mean'],cmap="Reds")
cax,kw = mpl.colorbar.make_axes(ax,location='bottom',pad=0.05,shrink=0.7)
out=fig.colorbar(pc,cax=cax,extend='both',**kw)
out.set_label('Temperature ($^{\circ}$C)',size=14)
ax.set_title('Multi-Model Ensemble Mean Temperature ($^{\circ}$C)',pad = 15,fontweight='bold',siz
```

2.4 Calculating autocorrelation lag-1

In the last two section, we saw how to use the well-defined NumPy operations `mean()` and `std()` in a Dataset over the dimension `time`. Now, how would we

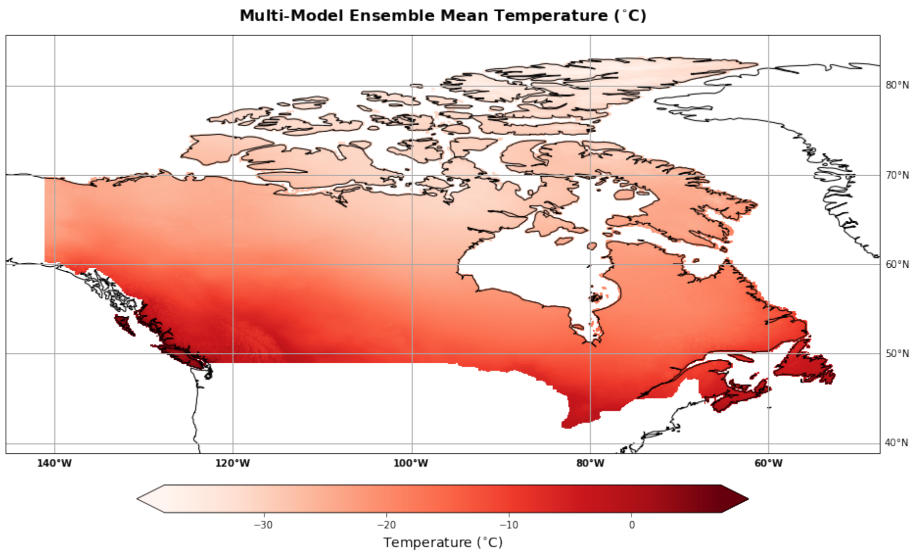

Figure 2.4: Figure 2 - Multi-model mean temperature

create new a function that is not defined in the NumPy library? Let's try first with the lag-1 autocorrelation.

First, let's define create a new function `lag1_numpy` by using the NumPy syntax.

```
def lag1_numpy(array):
    ecorr = np.correlate((array-np.mean(array))/np.std(array),(array-np.mean(array))
                         /np.std(array),'same')/len(array)
    lag1_numpy = ecorr[int(len(ecorr)/2)+1]
    return lag1_numpy
```

This function is taking an `array` and applying the NumPy operation `ecorr`.

Now, we define an xarray function by using the command `xr.apply_ufunc`. Fundamentally, this command allows for executing a NumPy function over a Dataset. It is a powerful tool because it fills a gap whenever the xarray library lacks a built-in function. However, it comes with several trade-offs that will be discussed over this section and the next.

```
def lag1_xarrray(array,dim="time"):
    lag1_xarrray = xr.apply_ufunc(lag1_numpy,array,
                     input_core_dims=[[dim]],
                     vectorize=True)
    return lag1_xarrray
```

In the code above, we defined a function `lag1_xarrray` that executes `lag1_numpy` over the time dimension of a Dataset.

Now, let's consider calculate the lag-1 autocorrelation of daily temperature of 25 GCMs. As described

```
lag1_model=[]

for file_name in multimodel:
        t_daily = xr.open_dataset(file_name)
        lag1 = lag1_xarrray(t_daily['tasmax'])
        lag1_model.append(lag1)
```

As discussed in the previous section, xarray does not automatically a new dimension `model` when we simply loop over 25 NetCDF files. At this moment, our Dataset `lag1_model` is a collection of 25 DataArrays that is not very useful for other purposes. In this case, we cannot simply create a new dimension with `xr.concat` because we lost some information about our variable along the way when using the function `lag1_xarrray`.

In this case, we can use the function `xr.Dataset` to manually create a new variable `lag1` over a new dimension `model`.

```
lag1_model = xr.Dataset(
    data_vars={
        "lag1": (("model","lat", "lon"), lag1_model.data),
    },
    coords={
        "model": range(25),
        "lat": lag1_model.lat,
        "lon": lag1_model.lon,
    },
    attrs = dict(
        variable="Autocorrelation lag-1",
        description="CMIP5 Models Data",
        units="-"))
```

Now we have a Dataset with with a new dimension `model` and variable `lag1`.

```
lag1_model
```

xarray.Dataset

Dimensions:

model: 25

```
lat: 510
lon: 1068
Coordinates: (3)
model
(model)
int64
0 1 2 3 4 5 6 ... 19 20 21 22 23 24
lat
(lat)
float64
41.04 41.12 41.21 ... 83.37 83.46
long_name :
latitude
standard_name :
latitude
NAME :
lat
units :
degrees_north
_Netcdf4Dimid :
1
CLASS :
DIMENSION_SCALE
axis :
Y
lon
(lon)
float64
-141.0 -140.9 ... -52.13 -52.04
long_name :
longitude
```

standard_name :

longitude

NAME :

lon

units :

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

Data variables: (1)

lag1

(model, lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (3)

variable :

Daily Near-Surface Maximum Air Temperature

description :

CMIP5 Models Data

units :

degC

As you probably noticed, it took a long time to run the loop with the function `lag1_xarrray`. This is a problem that can be potentially mitigated by using some features of the Dask library that are integrated into the xarray library. For now, we are just going to plot the multi-model ensemble lag-1 autocorrelation and discuss these features in the next section.

```
lag1_multimodel = lag1_model.mean('model')
```

```
xarray.Dataset
Dimensions:
lat: 510
lon: 1068
Coordinates: (2)
lat
(lat)
float64
41.04 41.12 41.21 ... 83.37 83.46
long_name :
latitude
standard_name :
latitude
NAME :
lat
units :
degrees_north
_Netcdf4Dimid :
1
CLASS :
DIMENSION_SCALE
axis :
Y
lon
(lon)
float64
-141.0 -140.9 ... -52.13 -52.04
long_name :
longitude
standard_name :
longitude
```

NAME :

lon

units :

degrees_east

_Netcdf4Dimid :

0

CLASS :

DIMENSION_SCALE

axis :

X

Data variables: (1)

lag1

(lat, lon)

float32

nan nan nan nan ... nan nan nan nan

Attributes: (0)

Finally, we can plot the multi-model ensemble lag-1 autocorrelation.

```
fig = plt.figure(figsize=(16,12))
ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines()
ax.gridlines(linewidth=1)
gl = ax.gridlines(crs=ccrs.PlateCarree(), draw_labels=True,
                 linewidth=1, color='darkgrey')
gl.xlabels_top = False
gl.ylabels_left = False
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER
gl.xlabel_style = {'size': 15, 'color': 'gray'}
gl.xlabel_style = {'color': 'black', 'weight': 'bold'}

pc = ax.pcolormesh(lag1_multimodel.lon,lag1_multimodel.lat,lag1_multimodel['lag1'],cmap="Reds")
cax,kw = mpl.colorbar.make_axes(ax,location='bottom',pad=0.05,shrink=0.7)
out=fig.colorbar(pc,cax=cax,extend='both',**kw)
out.set_label('Lag 1',size=14)
ax.set_title('Multi-Model Ensemble Lag-1 Autocorrelation',
             pad = 15,fontweight='bold',size=16)
```

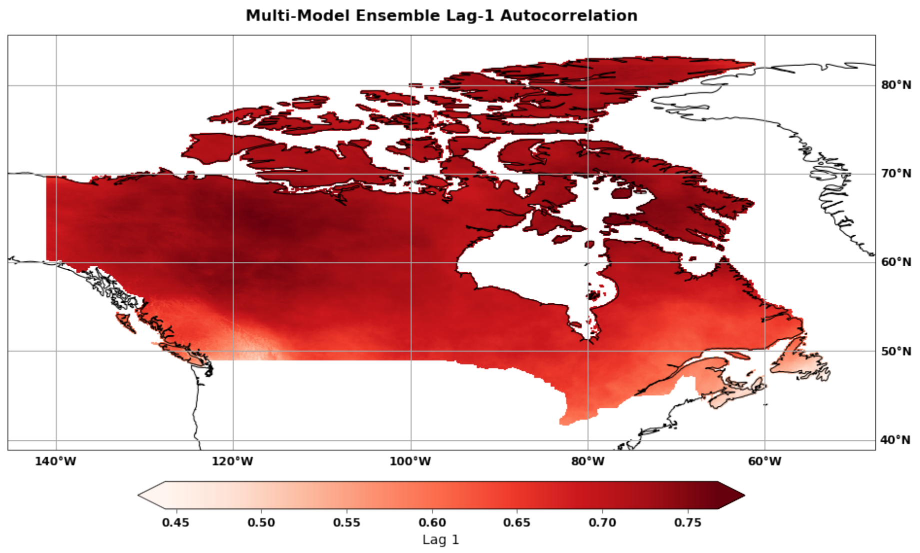

Figure 2.5: png

2.5 *Dask* arrays and parallel calculations

Now, let's create a function that calculates the e-folding time. First, we define this function with NumPy syntax.

```python
def efold_numpy(array):
    ecorr = np.correlate((array-np.mean(array))/np.std(array),(array-np.mean(array))
                        /np.std(array),'same')/len(array)
    lag1 = ecorr[int(len(ecorr)/2)+1]
    efold_numpy = -1/np.log(lag1)
    return efold_numpy
```

Now, we are going to use the function `xr.apply_ufunc` and there are two important features to consider when using Dask. Firstly, we have to execute parallel calculation by including `dask="parallelized"` and `allow_rechunk = True` in the function.

```python
def efold_xarray(array):
    efold_xarray = xr.apply_ufunc(
            efold_numpy,
            t_daily.groupby('time.month'),
            input_core_dims=[["time"]],
            exclude_dims=set(("time",)),
            vectorize=True,
            dask="parallelized",
            dask_gufunc_kwargs = dict(allow_rechunk = True),
            output_dtypes=[array['tasmax'].dtype]).rename({"tasmax": "efold"})
    return efold_xarray
```

Secondly, we have to define the size of our chunks when opening the NetCDF files. By specifying the chunk shape, xarray will automatically create Dask arrays for each data variable in the Dataset.

```
efold_model = []

for file_name in multimodel:
        t_daily = xr.open_dataset(file_name,chunks={"lat": 150, "lon": 200})
        efold = efold_xarray(t_daily)
        efold_model.append(efold)

efold_model = xr.concat(efold_model,pd.Index(range(25),name='model'))
```

```
xarray.Dataset

Dimensions:

month: 2

model: 25

lat: 510

lon: 1068

Coordinates: (4)

month

(month)

int64

1 2

lat

(lat)

float64

41.04 41.12 41.21 ... 83.37 83.46

lon

(lon)

float64
```

-141.0 -140.9 ... -52.13 -52.04

model

(model)

int64

0 1 2 3 4 5 6 ... 19 20 21 22 23 24

Data variables: (1)

efold

(model, month, lat, lon)

float32

dask.array<chunksize=(1, 1, 150, 200), meta=np.ndarray>

Array

Chunk

Bytes

103.89 MiB

117.19 kiB

Shape

(25, 2, 510, 1068)

(1, 1, 150, 200)

Count

9291 Tasks

1200 Chunks

Type

float32

numpy.ndarray

25 1

1068 510 2

Attributes: (0)

As you probably noticed, it took a few seconds (or less) to execute the `efold_xarray` and create a new Dataset with the variable `efold` and a new dimension `model`. From a user standpoint, the key concept of a `dask.array` is the notion of chunk. A chunk is the user-defined shape of the subdataset on which the unitary tasks will be applied.

Building upon dask has several advantages. For instance, the only modification of your code that is needed is your defining the chunks on which the computation should be performed. The Dask library also allows to easily leverage the resources of shared memory architectures (multi-core laptop or work-station) but also the resources of distributed memory architectures (clusters of cpu).

Yet, parallelization comes at a cost. Loading the data with dask.arrays is only done at the execution time if needed. This means that we have access to this Dataset without having to worry about time time it will take to load the data. The chunks also have implications on the performance of the code. If the chunks are too small, queueing up operations will be extremely slow, because Dask will translate each operation into a huge number of operations mapped across chunks. If the chunks are too big, some of the computation power may be wasted, because Dask only computes results one chunk at a time.

Chapter 3

Key takeaways