GCPy_demo Documentation

Release 0.1

Jiawei Zhuang

May 11, 2018

Contents:

Author: Jiawei Zhuang (jiaweizhuang@g.harvard.edu)

Important Note: We recommend the 2018 version of tutorial https://github.com/JiaweiZhuang/GEOSChem-python-tutorial. This old one (written in 2017) is just kept for record.

CHAPTER 1

Python/xarray for GEOS-Chem data analysis

Author: Jiawei Zhuang (jiaweizhuang@g.harvard.edu)

Last updated: 10/16/2017

1.1 Motivation

I have 3 years of experience with IDL and 7 years with MATLAB, but I now use Python for almost all research work because I feel it 10 times more efficient (in terms of human productivity, not just computational performance). Jupyter/IPython Notebook provides a great scientific research environment and avoids slow x11 forwarding when working with a remote server. Not to mention that Python is free and has become the NO.1 programming language in 2017!

This tutorial focuses on xarray, a popular, powerful and elegant Python package for analyzing earth science data. Compared to IDL or MATLAB, Python/xarray allows you to write much less boilerplate codes and focus on real research.

For example, say we want to read a variable *O3* from a NetCDF file *data.nc* and compute its zonal average. The code using xarray would be just 2 lines:

```
ds = xr.open_dataset("data.nc") # all information is read into the object ds
ds['O3'].mean(dim='lon') # the code is self-descriptive
```

The same operation using IDL would be something like (based on IDL documentation)

```
;Ignore those codes if you have never used IDL before (you are lucky).

fid = NCDF_OPEN('data.nc') ; Open the NetCDF file:
var_id = NCDF_VARID(fid, 'O3') ; Get the variable ID
NCDF_VARGET, fid, var_id, data  ; Get the variable data
NCDF_CLOSE, fid ; close the NetCDF file

; then write some code to check data dimension, for example
```

(continues on next page)

(continued from previous page)

```
size(data)

; OK, say we find that the data is a 4D array of shape [lon,lat,lev,time]
; the quickest way to average over the longitude is
mean(data, 1)
; The code would be much longer if you write "for" loops

; What's worse, IDL documentation on this mean() function is wrong!
; If you follow http://www.harrisgeospatial.com/docs/MEAN.html ,
; you would write
mean(data, DIMENSION=1)

; You want to take average over the first dimension "lon",
; But this DIMENSION keyword actually has no effect
; You will just get a global mean from the above expression.
; This bug is hard to find and could mess up your data analysis.
```

I hope this example has convinced you to switch from IDL to Python

1.2 Preparation

Basic knowledge of Python is assumed. For Python beginners, I recommend *Python Data Science Handbook* by *Jake VanderPlas*, which is free available online. Skim through Chapter 1, 2 and 4, then you will be good! (Chapter 3 and 5 are important for general data science but not particularly necessary for working with gridded model data.)

It is crucial to pick up the correct tutorial when learning Python, because Python has a wide range of applications other than just science. For scientific research, you should only read **data science** tutorials. Other famous books such as Fluent Python and The Hitchhiker's Guide to Python are great for general software development, but not quite useful for scientific research.

Once you manage to use Python in Jupyter Notebook, follow this GCPy page to set up Python environment for Geosciences. Now you should have everything necessary for working with GEOS-Chem data and most of earth science data in general.

1.3 Reading and exploring NetCDF file

1.3.1 Opening file

Here we use a GEOS-Chem restart file as an example. You can use your own file or download one at

```
ftp://ftp.as.harvard.edu/gcgrid/data/ExtData/NC_RESTARTS/initial_GEOSChem_rst.4x5_
↪benchmark.nc
```

```
In [1]: # those modules are almost always imported when working with model data
        %matplotlib inline
        import numpy as np
        import xarray as xr
        import matplotlib.pyplot as plt # general plotting
        import cartopy.crs as ccrs # plot on maps, better than the Basemap module
```

`xr.open_dataset()` reads the entire NetCDF file into this `ds` object. You can view all variables, coordinates, dimensions and metadata by just printing this object.

```
In [2]: ds = xr.open_dataset("./initial_GEOSChem_rst.4x5_benchmark.nc")
        ds # same as print(ds) in IPython/Jupyter environment

Out[2]: <xarray.Dataset>
        Dimensions:      (lat: 46, lev: 72, lon: 72, time: 1)
        Coordinates:
          * time         (time) datetime64[ns] 2013-07-01
          * lev          (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 ...
          * lat          (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 ...
          * lon          (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 ...
        Data variables:
            TRC_NO       (time, lev, lat, lon) float64 2.561e-17 2.561e-17 2.561e-17 ...
            TRC_O3       (time, lev, lat, lon) float64 2.615e-08 2.615e-08 2.615e-08 ...
            TRC_PAN      (time, lev, lat, lon) float64 8.709e-12 8.709e-12 8.709e-12 ...
            TRC_CO       (time, lev, lat, lon) float64 6.75e-08 6.75e-08 6.75e-08 ...
            TRC_ALK4     (time, lev, lat, lon) float64 1.753e-10 1.753e-10 1.753e-10 ...
            TRC_ISOP     (time, lev, lat, lon) float64 5.959e-14 5.959e-14 5.959e-14 ...
            TRC_HNO3     (time, lev, lat, lon) float64 5.854e-15 5.854e-15 5.854e-15 ...
            TRC_H2O2     (time, lev, lat, lon) float64 4.73e-11 4.73e-11 4.73e-11 ...
            TRC_ACET     (time, lev, lat, lon) float64 8.546e-10 8.546e-10 8.546e-10 ...
            TRC_MEK      (time, lev, lat, lon) float64 4.22e-11 4.22e-11 4.22e-11 ...
            TRC_ALD2     (time, lev, lat, lon) float64 2.119e-11 2.119e-11 2.119e-11 ...
            TRC_RCHO     (time, lev, lat, lon) float64 1.152e-12 1.152e-12 1.152e-12 ...
            TRC_MVK      (time, lev, lat, lon) float64 3.882e-13 3.882e-13 3.882e-13 ...
            TRC_MACR     (time, lev, lat, lon) float64 1.161e-12 1.161e-12 1.161e-12 ...
            TRC_PMN      (time, lev, lat, lon) float64 1.593e-15 1.593e-15 1.593e-15 ...
            TRC_PPN      (time, lev, lat, lon) float64 6.739e-13 6.739e-13 6.739e-13 ...
            TRC_R4N2     (time, lev, lat, lon) float64 4.331e-12 4.331e-12 4.331e-12 ...
            TRC_PRPE     (time, lev, lat, lon) float64 2.102e-13 2.102e-13 2.102e-13 ...
            TRC_C3H8     (time, lev, lat, lon) float64 4.072e-11 4.072e-11 4.072e-11 ...
            TRC_CH2O     (time, lev, lat, lon) float64 4.576e-11 4.576e-11 4.576e-11 ...
            TRC_C2H6     (time, lev, lat, lon) float64 5.118e-10 5.118e-10 5.118e-10 ...
            TRC_N2O5     (time, lev, lat, lon) float64 1.116e-17 1.116e-17 1.116e-17 ...
            TRC_HNO4     (time, lev, lat, lon) float64 1.659e-15 1.659e-15 1.659e-15 ...
            TRC_MP       (time, lev, lat, lon) float64 2.642e-10 2.642e-10 2.642e-10 ...
            TRC_DMS      (time, lev, lat, lon) float64 3.006e-10 3.006e-10 3.006e-10 ...
            TRC_SO2      (time, lev, lat, lon) float64 5.637e-12 5.637e-12 5.637e-12 ...
            TRC_SO4      (time, lev, lat, lon) float64 7.844e-12 7.844e-12 7.844e-12 ...
            TRC_SO4s     (time, lev, lat, lon) float64 3.705e-16 3.705e-16 3.705e-16 ...
            TRC_MSA      (time, lev, lat, lon) float64 2.933e-12 2.933e-12 2.933e-12 ...
            TRC_NH3      (time, lev, lat, lon) float64 4.384e-13 4.384e-13 4.384e-13 ...
            TRC_NH4      (time, lev, lat, lon) float64 1.183e-11 1.183e-11 1.183e-11 ...
            TRC_NIT      (time, lev, lat, lon) float64 5.173e-12 5.173e-12 5.173e-12 ...
            TRC_NITs     (time, lev, lat, lon) float64 3.166e-17 3.166e-17 3.166e-17 ...
            TRC_BCPI     (time, lev, lat, lon) float64 9.654e-13 9.654e-13 9.654e-13 ...
            TRC_OCPI     (time, lev, lat, lon) float64 6.117e-12 6.117e-12 6.117e-12 ...
            TRC_BCPO     (time, lev, lat, lon) float64 4.49e-16 4.49e-16 4.49e-16 ...
            TRC_OCPO     (time, lev, lat, lon) float64 1.845e-16 1.845e-16 1.845e-16 ...
            TRC_DST1     (time, lev, lat, lon) float64 3.653e-13 3.653e-13 3.653e-13 ...
            TRC_DST2     (time, lev, lat, lon) float64 6.175e-13 6.175e-13 6.175e-13 ...
            TRC_DST3     (time, lev, lat, lon) float64 3.633e-13 3.633e-13 3.633e-13 ...
            TRC_DST4     (time, lev, lat, lon) float64 9.996e-31 9.996e-31 9.996e-31 ...
            TRC_SALA     (time, lev, lat, lon) float64 5.433e-11 5.433e-11 5.433e-11 ...
            TRC_SALC     (time, lev, lat, lon) float64 1.487e-10 1.487e-10 1.487e-10 ...
            TRC_Br2      (time, lev, lat, lon) float64 1.019e-12 1.019e-12 1.019e-12 ...
            TRC_Br       (time, lev, lat, lon) float64 7.141e-19 7.141e-19 7.141e-19 ...
            TRC_BrO      (time, lev, lat, lon) float64 3.617e-14 3.617e-14 3.617e-14 ...
            TRC_HOBr     (time, lev, lat, lon) float64 2.352e-12 2.352e-12 2.352e-12 ...
            TRC_HBr      (time, lev, lat, lon) float64 9.093e-18 9.093e-18 9.093e-18 ...
```

```
TRC_BrNO2    (time, lev, lat, lon) float64 5.092e-16 5.092e-16 5.092e-16 ...
TRC_BrNO3    (time, lev, lat, lon) float64 2.606e-16 2.606e-16 2.606e-16 ...
TRC_CHBr3    (time, lev, lat, lon) float64 1.165e-12 1.165e-12 1.165e-12 ...
TRC_CH2Br2   (time, lev, lat, lon) float64 8.938e-13 8.938e-13 8.938e-13 ...
TRC_CH3Br    (time, lev, lat, lon) float64 6.523e-12 6.523e-12 6.523e-12 ...
TRC_MPN      (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_ISOPN    (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_MOBA     (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_PROPNN   (time, lev, lat, lon) float64 3.486e-14 3.486e-14 3.486e-14 ...
TRC_HAC      (time, lev, lat, lon) float64 3.487e-13 3.487e-13 3.487e-13 ...
TRC_GLYC     (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_MMN      (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_RIP      (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_IEPOX    (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
TRC_MAP      (time, lev, lat, lon) float64 2.064e-11 2.064e-11 2.064e-11 ...
TRC_NO2      (time, lev, lat, lon) float64 2.662e-13 2.662e-13 2.662e-13 ...
TRC_NO3      (time, lev, lat, lon) float64 3.155e-17 3.155e-17 3.155e-17 ...
TRC_HNO2     (time, lev, lat, lon) float64 1.209e-14 1.209e-14 1.209e-14 ...
TRC_N2O      (time, lev, lat, lon) float64 3.228e-07 3.228e-07 3.228e-07 ...
TRC_OCS      (time, lev, lat, lon) float64 5e-10 5e-10 5e-10 5e-10 5e-10 ...
TRC_CH4      (time, lev, lat, lon) float64 1.743e-06 1.743e-06 1.743e-06 ...
TRC_BrCl     (time, lev, lat, lon) float64 7.006e-20 7.006e-20 7.006e-20 ...
TRC_HCl      (time, lev, lat, lon) float64 9.395e-12 9.395e-12 9.395e-12 ...
TRC_CCl4     (time, lev, lat, lon) float64 8.38e-11 8.38e-11 8.38e-11 ...
TRC_CH3Cl    (time, lev, lat, lon) float64 5.506e-10 5.506e-10 5.506e-10 ...
TRC_CH3CCl3  (time, lev, lat, lon) float64 1.68e-11 1.68e-11 1.68e-11 ...
TRC_CFCX     (time, lev, lat, lon) float64 1.006e-10 1.006e-10 1.006e-10 ...
TRC_HCFCX    (time, lev, lat, lon) float64 5.37e-11 5.37e-11 5.37e-11 ...
TRC_CFC11    (time, lev, lat, lon) float64 2.406e-10 2.406e-10 2.406e-10 ...
TRC_CFC12    (time, lev, lat, lon) float64 5.282e-10 5.282e-10 5.282e-10 ...
TRC_HCFC22   (time, lev, lat, lon) float64 1.802e-10 1.802e-10 1.802e-10 ...
TRC_H1211    (time, lev, lat, lon) float64 3.8e-12 3.8e-12 3.8e-12 ...
TRC_H1301    (time, lev, lat, lon) float64 3.3e-12 3.3e-12 3.3e-12 ...
TRC_H2402    (time, lev, lat, lon) float64 3.4e-13 3.4e-13 3.4e-13 ...
TRC_Cl       (time, lev, lat, lon) float64 3.877e-18 3.877e-18 3.877e-18 ...
TRC_ClO      (time, lev, lat, lon) float64 4.297e-15 4.297e-15 4.297e-15 ...
TRC_HOCl     (time, lev, lat, lon) float64 1.078e-14 1.078e-14 1.078e-14 ...
TRC_ClNO3    (time, lev, lat, lon) float64 1.742e-15 1.742e-15 1.742e-15 ...
TRC_ClNO2    (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_ClOO     (time, lev, lat, lon) float64 2.223e-19 2.223e-19 2.223e-19 ...
TRC_OClO     (time, lev, lat, lon) float64 1.361e-19 1.361e-19 1.361e-19 ...
TRC_Cl2      (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_Cl2O2    (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_H2O      (time, lev, lat, lon) float64 0.001177 0.001177 0.001177 ...
TRC_MTPA     (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_LIMO     (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_MTPO     (time, lev, lat, lon) float64 1e-30 1e-30 1e-30 1e-30 1e-30 ...
TRC_TSOG1    (time, lev, lat, lon) float64 5.603e-14 5.603e-14 5.603e-14 ...
TRC_TSOG2    (time, lev, lat, lon) float64 9.888e-13 9.888e-13 9.888e-13 ...
TRC_TSOG3    (time, lev, lat, lon) float64 2.44e-12 2.44e-12 2.44e-12 ...
TRC_TSOG0    (time, lev, lat, lon) float64 1.264e-14 1.264e-14 1.264e-14 ...
TRC_TSOA1    (time, lev, lat, lon) float64 9.28e-14 9.28e-14 9.28e-14 ...
TRC_TSOA2    (time, lev, lat, lon) float64 2.652e-13 2.652e-13 2.652e-13 ...
TRC_TSOA3    (time, lev, lat, lon) float64 7.673e-14 7.673e-14 7.673e-14 ...
TRC_TSOA0    (time, lev, lat, lon) float64 1.336e-13 1.336e-13 1.336e-13 ...
TRC_ISOG1    (time, lev, lat, lon) float64 2.549e-13 2.549e-13 2.549e-13 ...
TRC_ISOG2    (time, lev, lat, lon) float64 1.821e-13 1.821e-13 1.821e-13 ...
TRC_ISOG3    (time, lev, lat, lon) float64 2.367e-12 2.367e-12 2.367e-12 ...
TRC_ISOA1    (time, lev, lat, lon) float64 4.221e-13 4.221e-13 4.221e-13 ...
```

```
    TRC_ISOA2    (time, lev, lat, lon) float64 4.881e-14 4.881e-14 4.881e-14 ...
    TRC_ISOA3    (time, lev, lat, lon) float64 7.445e-14 7.445e-14 7.445e-14 ...
    TRC_BENZ     (time, lev, lat, lon) float64 2.746e-11 2.746e-11 2.746e-11 ...
    TRC_TOLU     (time, lev, lat, lon) float64 8.516e-12 8.516e-12 8.516e-12 ...
    TRC_XYLE     (time, lev, lat, lon) float64 1.201e-12 1.201e-12 1.201e-12 ...
    TRC_ASOG1    (time, lev, lat, lon) float64 5.583e-15 5.583e-15 5.583e-15 ...
    TRC_ASOG2    (time, lev, lat, lon) float64 1.036e-14 1.036e-14 1.036e-14 ...
    TRC_ASOG3    (time, lev, lat, lon) float64 1.367e-13 1.367e-13 1.367e-13 ...
    TRC_ASOAN    (time, lev, lat, lon) float64 5.542e-14 5.542e-14 5.542e-14 ...
    TRC_ASOA1    (time, lev, lat, lon) float64 9.237e-15 9.237e-15 9.237e-15 ...
    TRC_ASOA2    (time, lev, lat, lon) float64 2.774e-15 2.774e-15 2.774e-15 ...
    TRC_ASOA3    (time, lev, lat, lon) float64 4.3e-15 4.3e-15 4.3e-15 ...
Attributes:
    Title:         COARDS/netCDF file created by BPCH2COARDS (GAMAP v2-17+)
    Conventions:   COARDS
    Format:        NetCDF-3
    Model:         GEOS5
    Delta_Lon:     5.0
    Delta_Lat:     4.0
    NLayers:       72
    Start_Date:    20130701
    Start_Time:    0
    End_Date:      20130701
    End_Time:      0
    Delta_Time:    0
    history:       Tue Feb 23 13:37:28 2016: ncatted -a units,TRC_ASOA3,o,c,mo...
```

`ds` is an xarray `Dataset`, which is like an in-memory representation of the entire NetCDF file. Just think this data type as a much more powerful version of MATLAB/IDL *structure* type, if you are unfamiliar with Python's object-oriented programming.

```
In [3]: type(ds)
```

```
Out[3]: xarray.core.dataset.Dataset
```

1.3.2 Extracting variable

A `Dataset` typical contains many variables, just like a NetCDF file. To extract a single variable, simply use `ds['varname']`

```
In [4]: dr = ds['TRC_O3']
        dr
```

```
Out[4]: <xarray.DataArray 'TRC_O3' (time: 1, lev: 72, lat: 46, lon: 72)>
        [238464 values with dtype=float64]
        Coordinates:
          * time     (time) datetime64[ns] 2013-07-01
          * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
          * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
          * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
        Attributes:
            long_name:  O3 tracer
            units:      mol/mol
```

The returning `dr` is a `DataArray`, containing all information for a single variable, including the numerical data itself, the coodinate information and additional attributes like *long_name* and *units* here.

```
In [5]: type(dr)
```

```
Out[5]: xarray.core.dataarray.DataArray
```

`DataArray` (a single variable) and `Dataset` (containing multiple `DataArray`) are the only two data types you need to know in xarray.

1.3.3 Syntax explanation

Recall that you can select a variable just by its name (`ds['varname']`), not by its underlying ID. This syntax is nothing special – it just follows the notation of the `dictionary` type, a basic data type in Python for storing key-value pairs.

```
In [6]: # make a simple dictionary
        # 'var1' is the key, and 1 is the value
        my_dict = dict(var1=1, var2=2)
        my_dict
```

```
Out[6]: {'var1': 1, 'var2': 2}
```

```
In [7]: my_dict['var1'] # retrieve the value by the key
```

```
Out[7]: 1
```

1.3.4 Metadata

`DataArray` has some additional attributes to describe the variable. They can be accessed by `.attrs`.

```
In [8]: dr.attrs
```

```
Out[8]: OrderedDict([('long_name', 'O3 tracer'), ('units', 'mol/mol')])
```

It is again a dictionary type (but with fixed order). Again, retrieve the value by the key:

```
In [9]: dr.attrs['units']
```

```
Out[9]: 'mol/mol'
```

Short cut:

```
In [10]: dr.units
```

```
Out[10]: 'mol/mol'
```

`DataSet` contains global attributes describing that NetCDF file

```
In [11]: ds.attrs['Title']
```

```
Out[11]: 'COARDS/netCDF file created by BPCH2COARDS (GAMAP v2-17+)'
```

All available keys are:

```
In [12]: ds.attrs.keys()
```

```
Out[12]: odict_keys(['Title', 'Conventions', 'Format', 'Model', 'Delta_Lon', 'Delta_Lat', 'NLayers',
```

1.3.5 Convert to numpy array

If you don't need additional information like coordinates and units, you can always convert a `DataArray` to a pure numpy array by `.values`.

```
In [13]: rawdata = dr.values # get pure numpy array
```

It is a 4D numpy array with the shape (time: 1, lev: 72, lat: 46, lon: 72)

```
In [14]: type(rawdata), rawdata.shape
```

```
Out[14]: (numpy.ndarray, (1, 72, 46, 72))
```

1.3.6 Modifying data

Most of the time you don't need to convert `DataArray` to numpy array, because arithmetic operations and numpy functions can directly work on `DataArray`. Here we multiply the data by 10^9 and change the unit to ppbv.

```
In [15]: dr *= 1e9 # v/v -> ppbv
         dr.attrs['units'] = 'ppbv'
         dr
```

```
Out[15]: <xarray.DataArray 'TRC_O3' (time: 1, lev: 72, lat: 46, lon: 72)>
         array([[[[ 26.152373, ...,  26.152373],
                  ...,
                  [  6.460082, ...,   6.460082]],

                 ...,
                 [[ 59.938827, ...,  59.938827],
                  ...,
                  [ 68.322656, ...,  68.322656]]]])
         Coordinates:
           * time     (time) datetime64[ns] 2013-07-01
           * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
           * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
           * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
         Attributes:
             long_name:  O3 tracer
             units:      ppbv
```

Now let's look at 3 common use cases. They are meant to be read in order.

1.4 Case 1: surface field

1.4.1 Selecting data

Conventionally, you will select data by indexing the pure numerical array `rawdata`, like that:

```
In [16]: data_surf = rawdata[0,0,:,:] # get 1st time slice and 1st level
         data_surf.shape # only contain lat and lon dimensions
```

```
Out[16]: (46, 72)
```

With the `DataArray` type, you can index the data by dimension names, without thinking about which dimension means *time* and which one means *level*.

```
In [17]: dr_surf = dr.isel(time=0, lev=0)
         dr_surf
```

```
Out[17]: <xarray.DataArray 'TRC_O3' (lat: 46, lon: 72)>
         array([[ 26.152373,  26.152373,  26.152373, ...,  26.152373,  26.152373,
                  26.152373],
                [ 26.224114,  26.226319,  26.230103, ...,  26.205269,  26.221985,
                  26.20175 ],
                [ 25.707591,  25.481674,  25.132504, ...,  25.927429,  25.662809,
                  25.705889],
                ...,
                [  8.588493,   9.013182,   8.793466, ...,   8.650769,   8.041175,
                   8.159272],
                [  6.469575,   6.469804,   6.465895, ...,   6.476197,   6.476856,
                   6.473523],
                [  6.460082,   6.460082,   6.460082, ...,   6.460082,   6.460082,
```

```
           6.460082]])
Coordinates:
    time     datetime64[ns] 2013-07-01
    lev      float32 0.9925
  * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
  * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
Attributes:
    long_name:  O3 tracer
    units:      ppbv
```

Verify that both methods give the same result:

```
In [18]: np.allclose(data_surf, dr_surf.values)
```

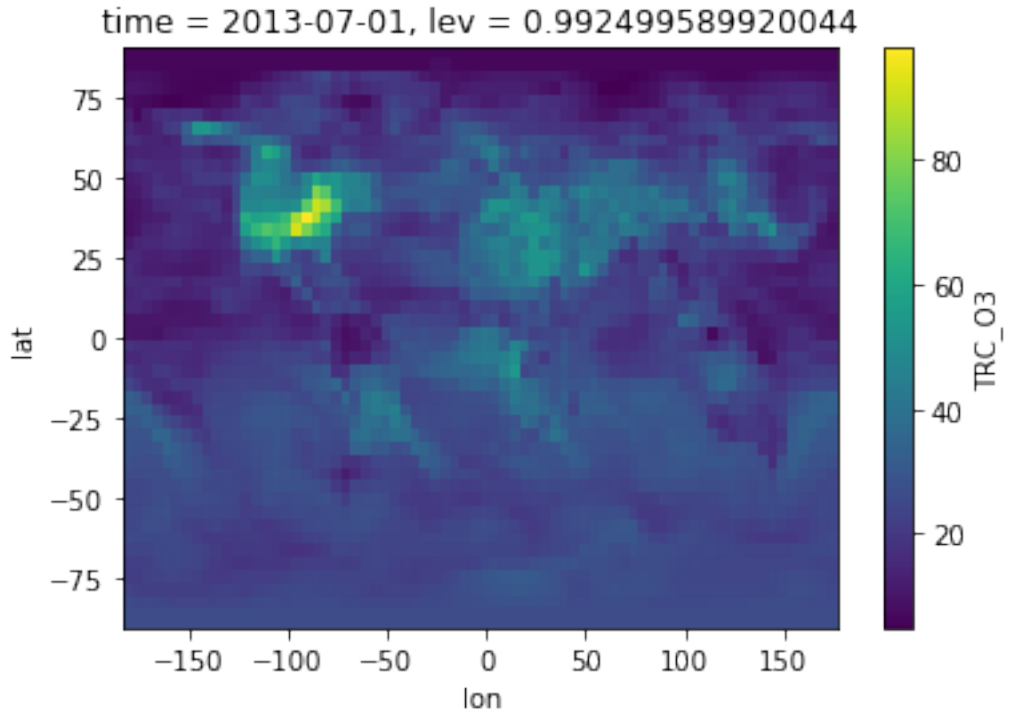
```
Out[18]: True
```

1.4.2 Convenience method for plotting

A 2D `DataArray` has a convenient `plot()` method.

```
In [19]: dr_surf.plot()
```

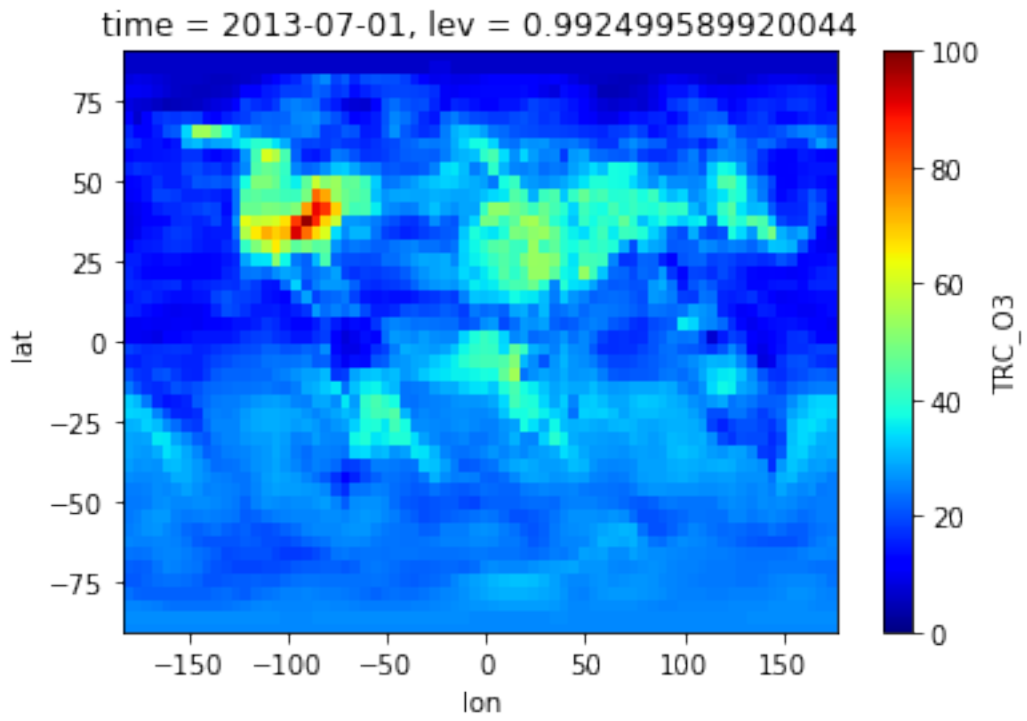
```
Out[19]: <matplotlib.collections.QuadMesh at 0x111eaa550>
```



You can tweak colormap and colorbar range.

```
In [20]: dr_surf.plot(cmap='jet', vmin=0, vmax=100)
```

```
Out[20]: <matplotlib.collections.QuadMesh at 0x112083b00>
```



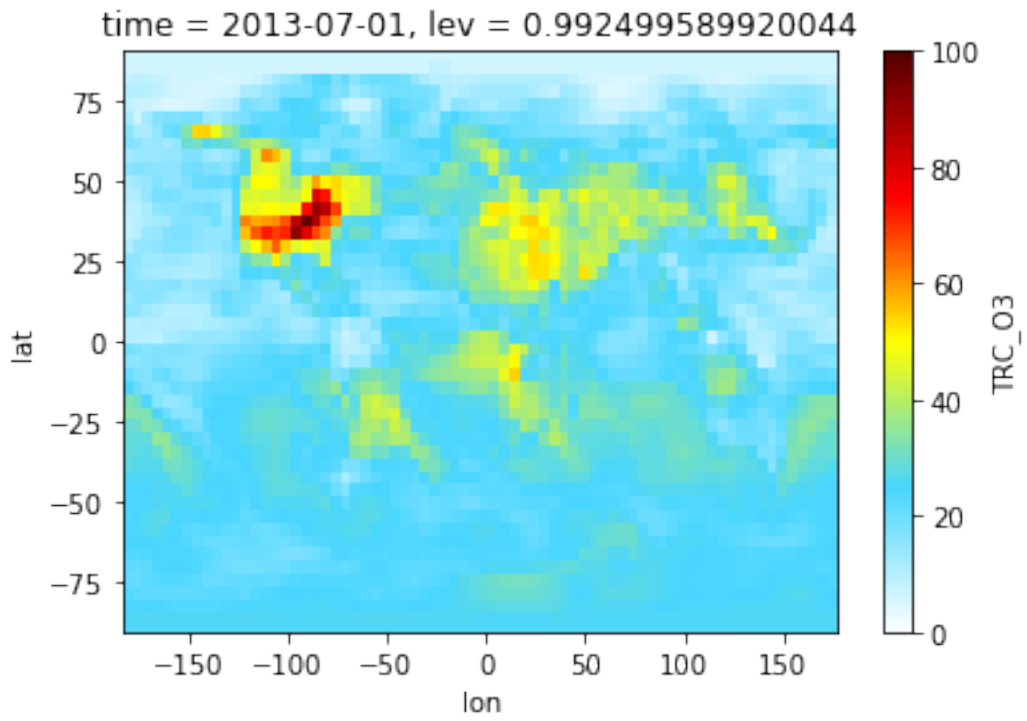
See matplotlib colormap for all available color schemes.

1.4.3 Using gamap's color scheme

We can also use the WhGrYlRd scheme from our good old friend gamap.

Download this WhGrYlRd.txt file and make a custom colormap (That's a quick solution for now. We'll put this into the GCPy package.)

```
In [21]: # get gamap's WhGrYlRd color scheme from file
         from matplotlib.colors import ListedColormap
         WhGrYlRd_scheme = np.genfromtxt('./WhGrYlRd.txt', delimiter=' ')
         WhGrYlRd = ListedColormap(WhGrYlRd_scheme/255.0)

In [22]: dr_surf.plot(cmap=WhGrYlRd, vmin=0, vmax=100)

Out[22]: <matplotlib.collections.QuadMesh at 0x1122156a0>
```



1.5 Extending case 1: visualization details

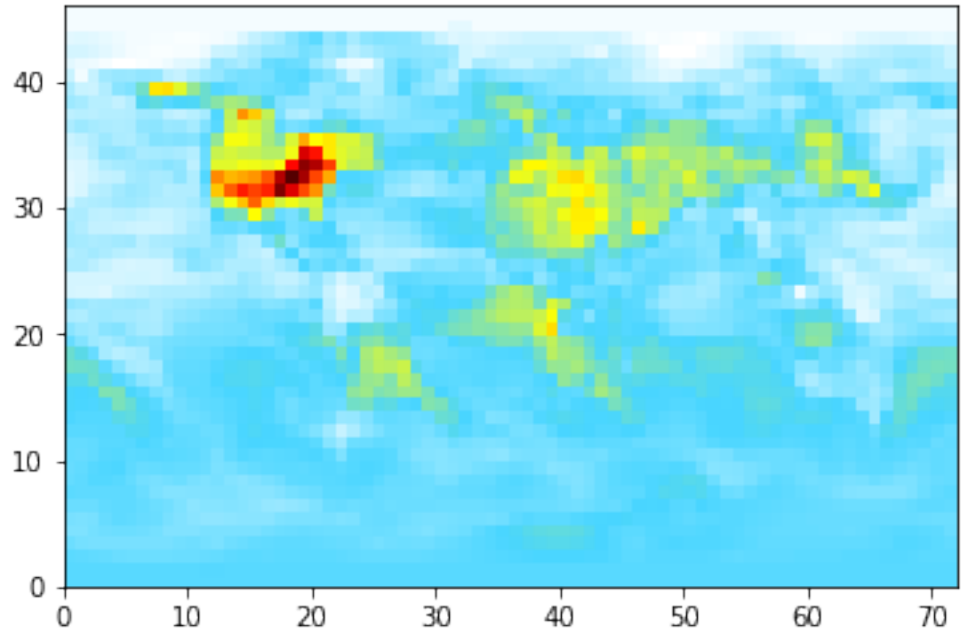
Note: Feel free to jump to Case 2 if you don't care about detailed plotting for now. Here we demonstrate how to make publication-quality plots only using basic, low-level plotting functions. We'll provide high-level wrappers in the GCPy package, but knowing how to write things from scratch is also very useful.

1.5.1 Basic heatmap

To tweak details, we can always fall back to the original matplotlib functions. The default `plt.pcolormesh()` plots 2D heat maps, but it has no idea about the spherical coordinate.

```
In [23]: plt.pcolormesh(dr_surf, cmap=WhGrYlRd)
```

```
Out[23]: <matplotlib.collections.QuadMesh at 0x1123a4780>
```



We can get the coordinate values from the `DataArray`, because the original NetCDF file contains the coordinate information.

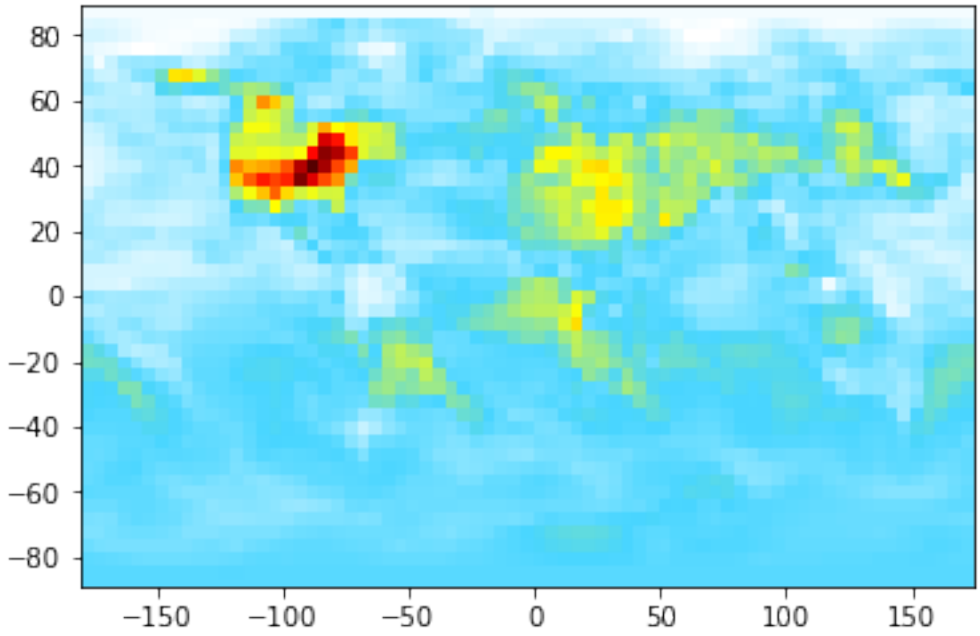
```
In [24]: lat = dr_surf['lat'].values
         lon = dr_surf['lon'].values
         print('lat:\n', lat)
         print('lon:\n', lon)
```

```
lat:
 [-89. -86. -82. -78. -74. -70. -66. -62. -58. -54. -50. -46. -42. -38. -34.
 -30. -26. -22. -18. -14. -10.  -6.  -2.   2.   6.  10.  14.  18.  22.  26.
  30.  34.  38.  42.  46.  50.  54.  58.  62.  66.  70.  74.  78.  82.  86.
  89.]
lon:
 [-180. -175. -170. -165. -160. -155. -150. -145. -140. -135. -130. -125.
 -120. -115. -110. -105. -100.  -95.  -90.  -85.  -80.  -75.  -70.  -65.
  -60.  -55.  -50.  -45.  -40.  -35.  -30.  -25.  -20.  -15.  -10.   -5.
    0.    5.   10.   15.   20.   25.   30.   35.   40.   45.   50.   55.
   60.   65.   70.   75.   80.   85.   90.   95.  100.  105.  110.  115.
  120.  125.  130.  135.  140.  145.  150.  155.  160.  165.  170.  175.]
```

Calling `plt.pcolormesh(X, Y, data)` instead of `plt.pcolormesh(data)` will set correct coordinate values.

```
In [25]: plt.pcolormesh(lon, lat, dr_surf, cmap=WhGrYlRd)
```

```
Out[25]: <matplotlib.collections.QuadMesh at 0x112473cf8>
```

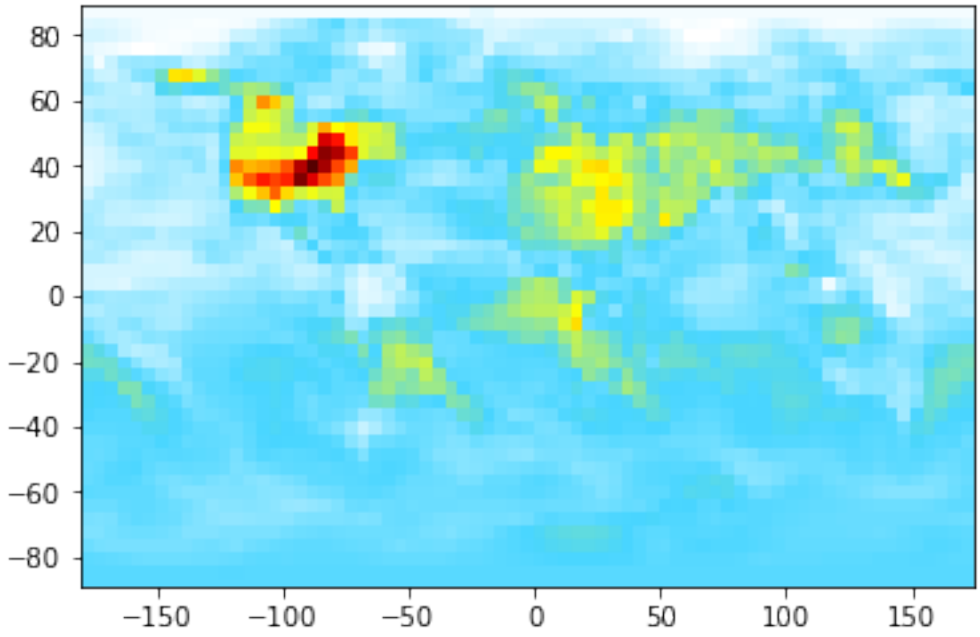


To gain full control on the figure, we can create an `axis` object and call the plotting method on it.

(This is not too useful here but is particularly useful for multi-panel plots, where each `axis` means one subplot so you can fine-tune each panel. See here for a subplot example.)

```
In [26]: # has the same effect as the previous code cell
         ax = plt.axes()
         ax.pcolormesh(lon, lat, dr_surf, cmap=WhGrYlRd)
```

```
Out[26]: <matplotlib.collections.QuadMesh at 0x1125730f0>
```



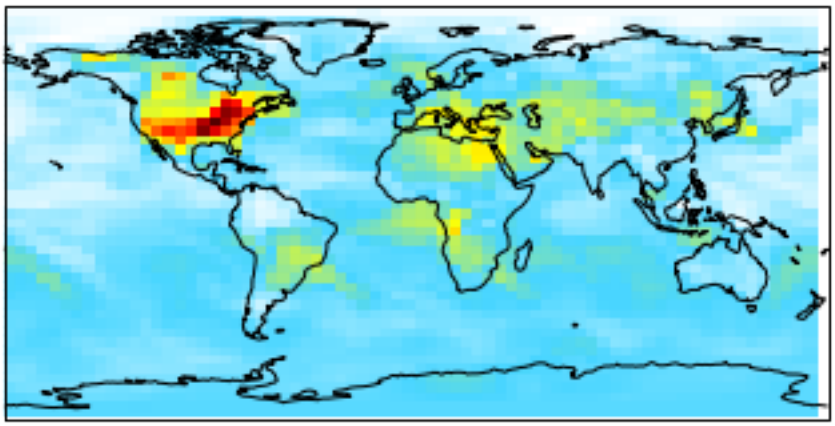
1.5.2 Geographic maps

To plot on geographic maps, the only change is adding the `projection` keyword to `plt.axes()`

(Recall that `ccrs` is from the cartopy the package, which is much easier to use than the old Basemap package).

```
In [27]: ax = plt.axes(projection=ccrs.PlateCarree())
         ax.coastlines()
         ax.pcolormesh(lon, lat, dr_surf, cmap=WhGrYlRd)
```

```
Out[27]: <matplotlib.collections.QuadMesh at 0x113dd19e8>
```

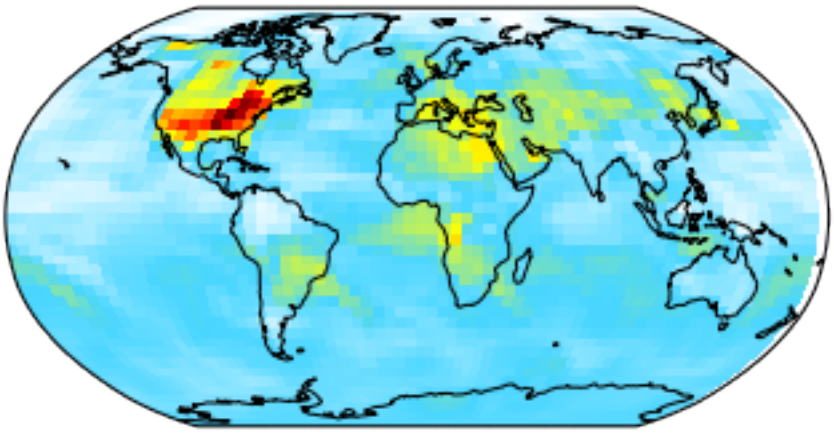


`PlateCarree` is the most commonly used projection but there are tons of projections available. See cartopy documentation for all options.

Let's try the `Robinson` projection. Note that the `transform` keyword still takes `PlateCarree`, for some mathematical reasons.

```
In [28]: ax = plt.axes(projection=ccrs.Robinson())
         ax.coastlines()
         ax.pcolormesh(lon, lat, dr_surf, cmap=WhGrYlRd, transform=ccrs.PlateCarree())
```

```
Out[28]: <matplotlib.collections.QuadMesh at 0x113eeccc0>
```



1.5.3 Correcting map boundaries

You might notice a blank stripe on the right edge ($180°E/W$). That's because `pcolormesh(X, Y, data)` expects `X` and `Y` to be cell bounds, not cell centers. Fortunately, creating cell bound coordinates is trivial:

```
In [29]: lon_b = np.linspace(-182.5, 177.5, 73)
         print(lon_b)
```

```
[-182.5 -177.5 -172.5 -167.5 -162.5 -157.5 -152.5 -147.5 -142.5 -137.5
 -132.5 -127.5 -122.5 -117.5 -112.5 -107.5 -102.5  -97.5  -92.5  -87.5
  -82.5  -77.5  -72.5  -67.5  -62.5  -57.5  -52.5  -47.5  -42.5  -37.5
  -32.5  -27.5  -22.5  -17.5  -12.5   -7.5   -2.5    2.5    7.5   12.5
   17.5   22.5   27.5   32.5   37.5   42.5   47.5   52.5   57.5   62.5
   67.5   72.5   77.5   82.5   87.5   92.5   97.5  102.5  107.5  112.5
  117.5  122.5  127.5  132.5  137.5  142.5  147.5  152.5  157.5  162.5
  167.5  172.5  177.5]
```

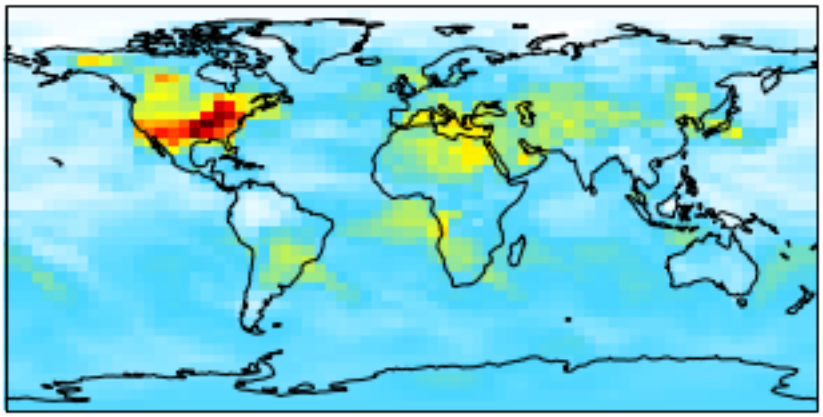
```
In [30]: lat_b = np.linspace(-92, 92, 47)
         lat_b[0] = -90 # -92 => -90
         lat_b[-1] = 90 #  92 =>  90
         print(lat_b)
```

```
[-90. -88. -84. -80. -76. -72. -68. -64. -60. -56. -52. -48. -44. -40. -36.
 -32. -28. -24. -20. -16. -12.  -8.  -4.   0.   4.   8.  12.  16.  20.  24.
  28.  32.  36.  40.  44.  48.  52.  56.  60.  64.  68.  72.  76.  80.  84.
  88.  90.]
```

Now there's no blank stripe in the figure.

```
In [31]: ax = plt.axes(projection=ccrs.PlateCarree())
         ax.coastlines()
         plt.pcolormesh(lon_b, lat_b, dr_surf, cmap=WhGrYlRd)
```

```
Out[31]: <matplotlib.collections.QuadMesh at 0x113fc8978>
```

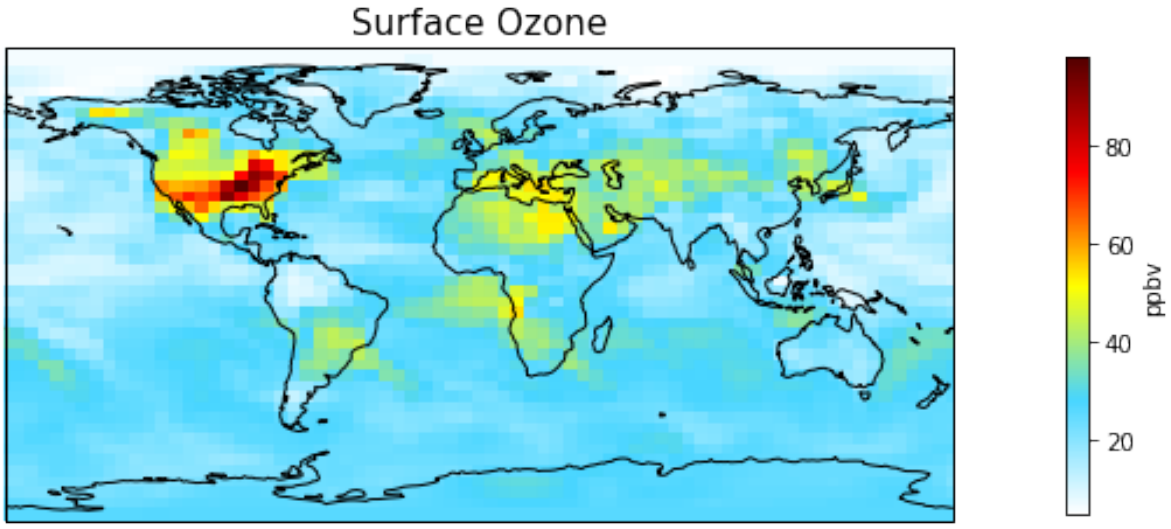


1.5.4 Adding details

Add a few more codes to tweak figure size, title, and colorbar.

```
In [32]: plt.figure(figsize=[10,6]) # make figure bigger

         ax = plt.axes(projection=ccrs.PlateCarree())
         ax.coastlines()
         plt.pcolormesh(lon_b, lat_b, dr_surf, cmap=WhGrYlRd)
```

```
plt.title('Surface Ozone', fontsize=15)
cb = plt.colorbar(shrink=0.6) # use shrink to make colorbar smaller
cb.set_label("ppbv")
```



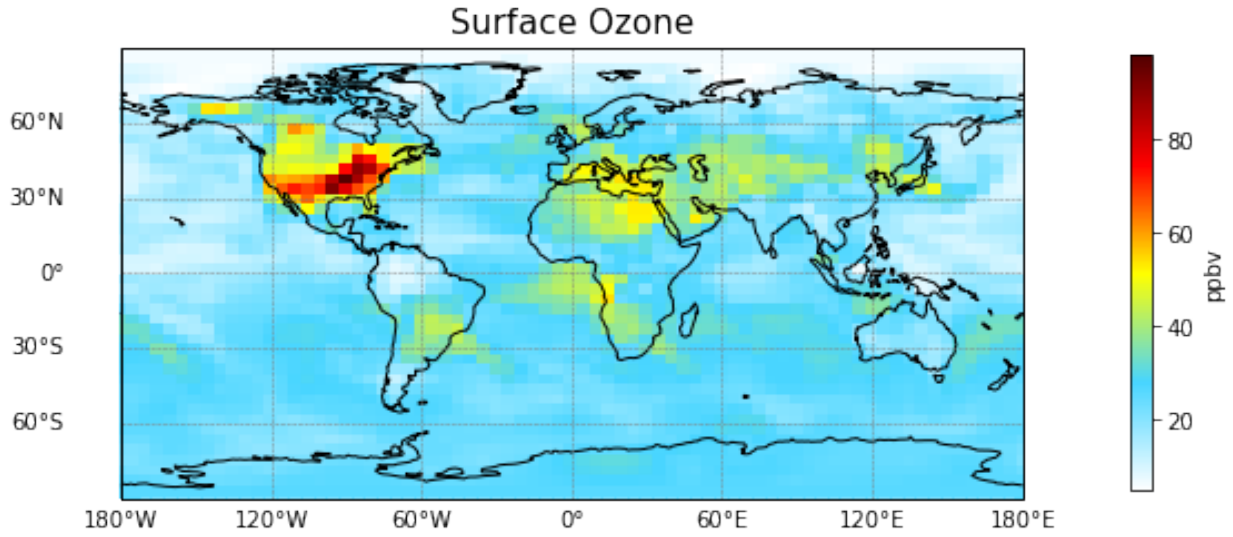
Gridlines and latlon ticks need more codes. But we can create a function and don't have to repeat those those codes for every plot.

```
In [33]: from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER
         import matplotlib.ticker as mticker

         def add_latlon_ticks(ax):
             '''Add latlon label ticks and gridlines to ax

             Adapted from
             http://scitools.org.uk/cartopy/docs/v0.13/matplotlib/gridliner.html
             '''
             gl = ax.gridlines(crs=ccrs.PlateCarree(), draw_labels=True,
                               linewidth=0.5, color='gray', linestyle='--')
             gl.xlabels_top = False
             gl.ylabels_right = False
             gl.xformatter = LONGITUDE_FORMATTER
             gl.yformatter = LATITUDE_FORMATTER
             gl.ylocator = mticker.FixedLocator(np.arange(-90,91,30))
```

Apply the above function to the figure, and finally save the figure to a file.

```
In [34]: fig = plt.figure(figsize=[10,6])

         ax = plt.axes(projection=ccrs.PlateCarree())
         ax.coastlines()
         plt.pcolormesh(lon_b, lat_b, dr_surf, cmap=WhGrYlRd)
         plt.title('Surface Ozone', fontsize=15)

         cb = plt.colorbar(shrink=0.6)
         cb.set_label("ppbv")

         # above codes are exactly the same as the previous example
         # only the two lines below are new
```

```
add_latlon_ticks(ax) # add ticks and gridlines
fig.savefig('Surface_Ozone.png', dpi=200) # save figure to a file
```



`dpi=200` leads to a pretty high-quality plot. You can view it here.

1.6 Case 2: zonal mean

1.6.1 Dropping unnecessary dimension

Recall that our `DataArray` has 4 dimensions, although the time dimension is redundant.

```
In [35]: dr
```

```
Out[35]: <xarray.DataArray 'TRC_O3' (time: 1, lev: 72, lat: 46, lon: 72)>
         array([[[[ 26.152373, ...,  26.152373],
                  ...,
                  [  6.460082, ...,   6.460082]],

                 ...,
                 [[ 59.938827, ...,  59.938827],
                  ...,
                  [ 68.322656, ...,  68.322656]]]])
         Coordinates:
           * time     (time) datetime64[ns] 2013-07-01
           * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
           * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
           * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
         Attributes:
             long_name:  O3 tracer
             units:      ppbv
```

Get rid of this redundant dimension for convenience.

```
In [36]: dr = dr.isel(time=0)  # equivalent to dr.squeeze() in this case
         dr
```

```
Out[36]: <xarray.DataArray 'TRC_O3' (lev: 72, lat: 46, lon: 72)>
         array([[[ 26.152373,  26.152373, ...,  26.152373,  26.152373],
                 [ 26.224114,  26.226319, ...,  26.221985,  26.20175 ],
```

```
                ...,
                [  6.469575,   6.469804, ...,   6.476856,   6.473523],
                [  6.460082,   6.460082, ...,   6.460082,   6.460082]],

               [[ 26.328511,  26.328511, ...,  26.328511,  26.328511],
                [ 26.338292,  26.334288, ...,  26.326118,  26.327644],
                ...,
                [ 10.846659,  10.846609, ...,  10.846533,  10.846559],
                [ 10.847146,  10.847146, ...,  10.847146,  10.847146]],

               ...,
               [[ 59.93882 ,  59.93882 , ...,  59.93882 ,  59.93882 ],
                [ 59.93882 ,  59.93882 , ...,  59.93882 ,  59.93882 ],
                ...,
                [ 68.32267 ,  68.32267 , ...,  68.32267 ,  68.32267 ],
                [ 68.322663,  68.322663, ...,  68.322663,  68.322663]],

               [[ 59.938827,  59.938827, ...,  59.938827,  59.938827],
                [ 59.938813,  59.938813, ...,  59.938813,  59.938813],
                ...,
                [ 68.322663,  68.322663, ...,  68.322663,  68.322663],
                [ 68.322656,  68.322656, ...,  68.322656,  68.322656]]])
        Coordinates:
            time     datetime64[ns] 2013-07-01
          * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
          * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
          * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
        Attributes:
            long_name:  O3 tracer
            units:      ppbv
```

Of course, you can do the same thing with pure numpy array. Recall that `rawdata` is 4D.

```
In [37]: rawdata.shape
```

```
Out[37]: (1, 72, 46, 72)
```

```
In [38]: data_sqz = rawdata[0,...] # equivalent to rawdata.squeeze() in this case
         data_sqz.shape
```

```
Out[38]: (72, 46, 72)
```

1.6.2 Averaging

Here comes the example at the beginning! Take zonal average without thinking about dimension order.

```
In [39]: dr_zmean = dr.mean(dim='lon')
         dr_zmean
```

```
Out[39]: <xarray.DataArray 'TRC_O3' (lev: 72, lat: 46)>
         array([[ 26.152373,  26.194946,  25.725238, ...,  10.479322,   6.539299,
                   6.460082],
                [ 26.328511,  26.341357,  26.298183, ...,  13.434381,  10.794238,
                  10.847146],
                [ 26.498927,  26.484916,  26.580563, ...,  17.146443,  18.342079,
                  18.30816 ],
                ...,
                [ 59.938841,  59.938838,  60.476615, ...,  68.69855 ,  68.322649,
                  68.322649],
                [ 59.93882 ,  59.938823,  60.476616, ...,  68.698553,  68.32267 ,
                  68.322663],
```

```
              [ 59.938827,  59.938816,  60.476615, ...,  68.698551,  68.322663,
                68.322656]])
        Coordinates:
            time     datetime64[ns] 2013-07-01
          * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
          * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
```

You can do the same thing with pure numpy array, as long as you know the longitude axis is the last dimension.

```
In [40]: data_zmean = data_sqz.mean(axis=2)
         data_zmean.shape
```

```
Out[40]: (72, 46)
```

But only having pure numerical data makes me feel unsafe. I often need to double check the `72` above means the number of vertical levels, not the size of longitude dimension.

Check if two approaches give the same result:

```
In [41]: np.allclose(dr_zmean.values, data_zmean)
```

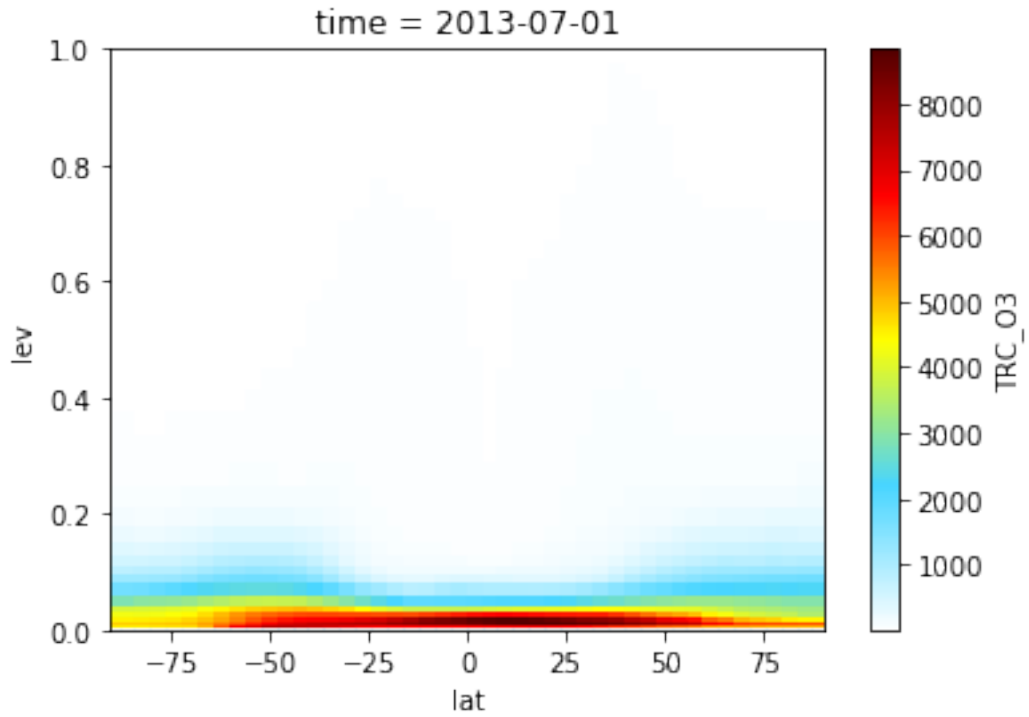
```
Out[41]: True
```

1.6.3 Plotting

`dr_zmean` is a 2D `DataArray`, so it has a convenient plotting method.

```
In [42]: dr_zmean.plot(cmap=WhGrYlRd)
```

```
Out[42]: <matplotlib.collections.QuadMesh at 0x114e3d278>
```



It looks weird, because the `lev` coordinate is the sigma value, not pressure or height.

```
In [43]: dr_zmean['lev']
```

```
Out[43]: <xarray.DataArray 'lev' (lev: 72)>
         array([  9.924996e-01,   9.774562e-01,   9.623704e-01,   9.472854e-01,
                  9.322005e-01,   9.171155e-01,   9.020311e-01,   8.869475e-01,
                  8.718643e-01,   8.567811e-01,   8.416982e-01,   8.266159e-01,
                  8.090211e-01,   7.863999e-01,   7.612653e-01,   7.361340e-01,
                  7.110056e-01,   6.858778e-01,   6.544709e-01,   6.167904e-01,
                  5.791153e-01,   5.414491e-01,   5.037952e-01,   4.661533e-01,
                  4.285283e-01,   3.909265e-01,   3.533493e-01,   3.098539e-01,
                  2.635869e-01,   2.237725e-01,   1.900607e-01,   1.615131e-01,
                  1.372873e-01,   1.166950e-01,   9.919107e-02,   8.431271e-02,
                  7.159988e-02,   6.068223e-02,   5.132635e-02,   4.332603e-02,
                  3.649946e-02,   3.067280e-02,   2.569886e-02,   2.146679e-02,
                  1.787765e-02,   1.484372e-02,   1.228746e-02,   1.014066e-02,
                  8.336023e-03,   6.818051e-03,   5.548341e-03,   4.492209e-03,
                  3.618591e-03,   2.899926e-03,   2.311980e-03,   1.833603e-03,
                  1.446498e-03,   1.134948e-03,   8.855623e-04,   6.868625e-04,
                  5.292849e-04,   4.050606e-04,   3.077095e-04,   2.318676e-04,
                  1.731298e-04,   1.279055e-04,   9.328887e-05,   6.678824e-05,
                  4.618107e-05,   2.974863e-05,   1.613635e-05,   4.934665e-06], dtype=float32)
         Coordinates:
             time     datetime64[ns] 2013-07-01
           * lev      (lev) float32 0.9925 0.977456 0.96237 0.947285 0.9322 0.917116 ...
         Attributes:
             long_name:  Eta Centers
             units:      sigma_level
             positive:   up
             axis:       Z
```

We will provide functions to convert it to height or pressure, but for now let's just use level index.

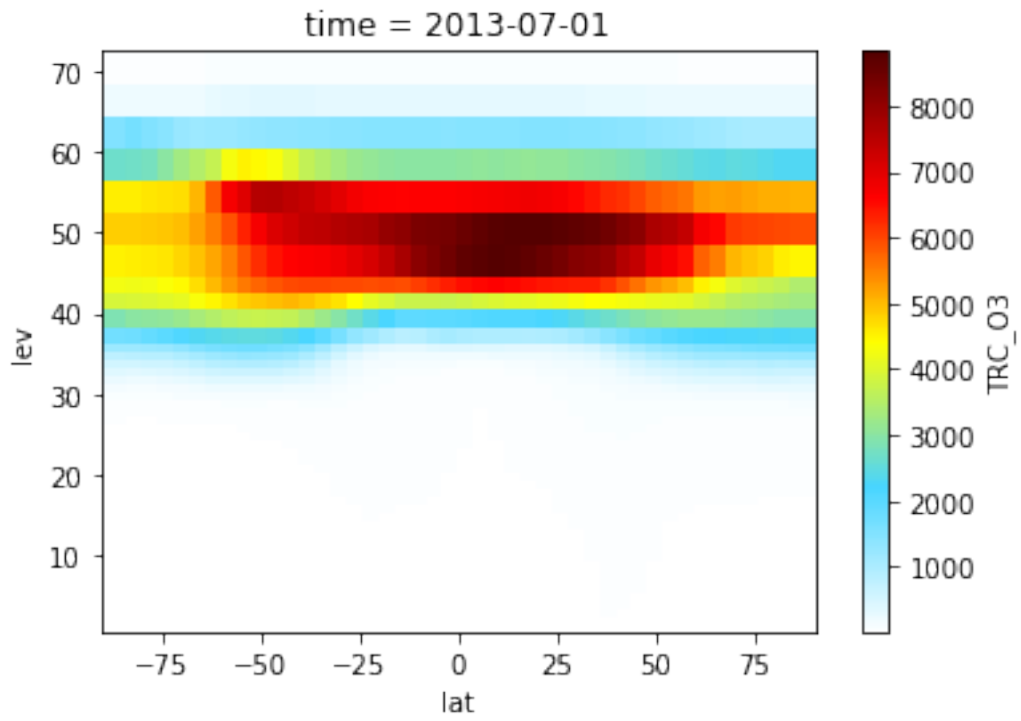
```
In [44]: dr_zmean['lev'].values = np.arange(1,73)
         dr_zmean['lev'].attrs['units'] = 'unitless'
         dr_zmean['lev'].attrs['long_name'] = 'level index'
         dr_zmean['lev']
```

```
Out[44]: <xarray.DataArray 'lev' (lev: 72)>
         array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
                19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
                37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54,
                55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72])
         Coordinates:
             time     datetime64[ns] 2013-07-01
           * lev      (lev) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
         Attributes:
             long_name:  level index
             units:      unitless
             positive:   up
             axis:       Z
```

Now the plot looks normal.

```
In [45]: dr_zmean.plot(cmap=WhGrYlRd)
```

```
Out[45]: <matplotlib.collections.QuadMesh at 0x114e9d4a8>
```



1.6.4 Slicing

Say we only want to plot the troposphere, we can use a `slice` to select level 1~30.

```
In [46]: dr_zmean.isel(lev=slice(0,30))
```

```
Out[46]: <xarray.DataArray 'TRC_O3' (lev: 30, lat: 46)>
         array([[  26.152373,   26.194946,   25.725238, ...,   10.479322,    6.539299,
                    6.460082],
                [  26.328511,   26.341357,   26.298183, ...,   13.434381,   10.794238,
                   10.847146],
                [  26.498927,   26.484916,   26.580563, ...,   17.146443,   18.342079,
                   18.30816 ],
                ...,
                [  53.418344,   53.418347,   45.959175, ...,   80.124412,   84.74668 ,
                   84.746681],
                [  71.646951,   71.646957,   58.809819, ...,  102.935995,  110.957415,
                  110.957423],
                [ 104.243057,  104.243062,   81.957629, ...,  176.496959,  179.441246,
                  179.441244]])
         Coordinates:
             time     datetime64[ns] 2013-07-01
           * lev      (lev) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
           * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
```

The equivalent way in numpy would be:

```
In [47]: data_zmean[0:30,:]
```

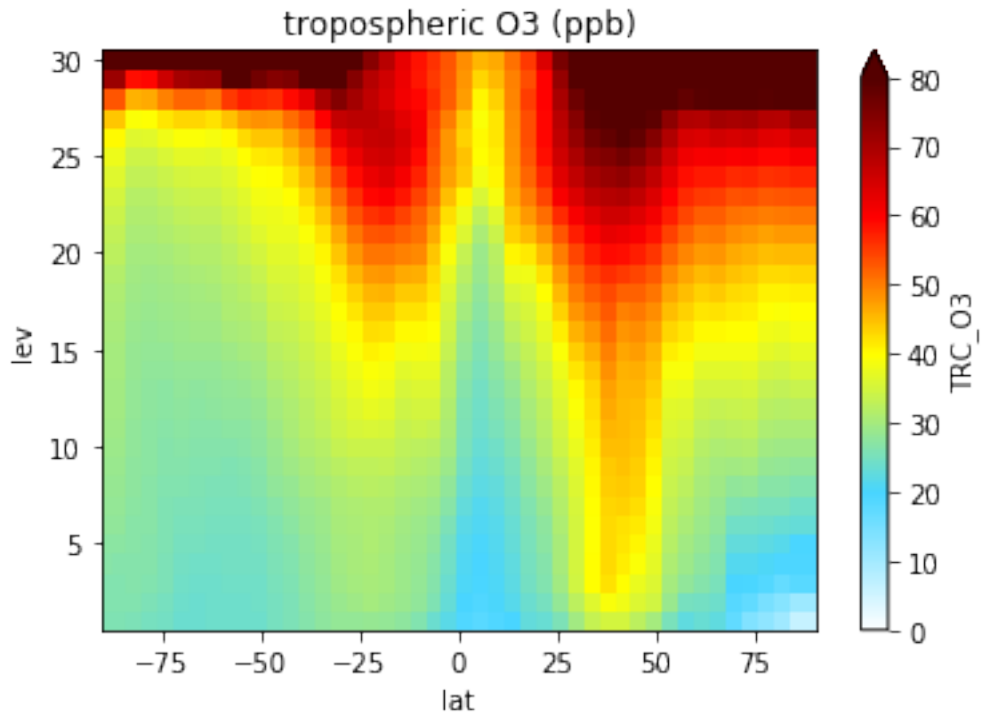
```
Out[47]: array([[  26.15237271,   26.19494634,   25.72523762, ...,   10.47932228,
                    6.53929865,    6.46008225],
                [  26.32851093,   26.34135744,   26.29818292, ...,   13.43438139,
                   10.79423836,   10.84714629],
```

```
       [  26.4989275 ,   26.48491587,   26.58056279, ...,   17.14644329,
          18.34207919,   18.30816032],
       ...,
       [  53.41834353,   53.41834703,   45.95917533, ...,   80.12441219,
          84.74668009,   84.74668078],
       [  71.64695148,   71.64695711,   58.80981943, ...,  102.93599454,
         110.95741481,  110.95742281],
       [ 104.24305685,  104.24306179,   81.95762888, ...,  176.49695927,
         179.44124566,  179.44124409]])
```

Only plot the tropospheric region:

```
In [48]: dr_zmean.isel(lev=slice(0,30)).plot(cmap=WhGrYlRd, vmax=80, vmin=0)
         plt.title('tropospheric O3 (ppb)') # overwrite the default title
```

```
Out[48]: <matplotlib.text.Text at 0x1150f8ac8>
```



1.7 Case 3: vertical profile

1.7.1 Selecting data

Say we want to plot the O3 profile at a specific location. Let's see what locations are available.

```
In [49]: print('lat:\n', dr['lat'].values)
         print('lon:\n', dr['lon'].values)
```

```
lat:
 [-89. -86. -82. -78. -74. -70. -66. -62. -58. -54. -50. -46. -42. -38. -34.
 -30. -26. -22. -18. -14. -10.  -6.  -2.   2.   6.  10.  14.  18.  22.  26.
  30.  34.  38.  42.  46.  50.  54.  58.  62.  66.  70.  74.  78.  82.  86.
  89.]
lon:
```

```
 [-180. -175. -170. -165. -160. -155. -150. -145. -140. -135. -130. -125.
 -120. -115. -110. -105. -100.  -95.  -90.  -85.  -80.  -75.  -70.  -65.
  -60.  -55.  -50.  -45.  -40.  -35.  -30.  -25.  -20.  -15.  -10.   -5.
    0.    5.   10.   15.   20.   25.   30.   35.   40.   45.   50.   55.
   60.   65.   70.   75.   80.   85.   90.   95.  100.  105.  110.  115.
  120.  125.  130.  135.  140.  145.  150.  155.  160.  165.  170.  175.]
```

Say we want to select $(30°N, 60°E)$. Hey, don't count which element in `lat` array is 30! `sel` (instead of `isel`) can select data by coordinate **values**, not by coordinate **index**. This feature allows you to use almost the same code for data at different resolutions.

```
In [50]: profile = dr.sel(lat=30, lon=60)
         profile
```

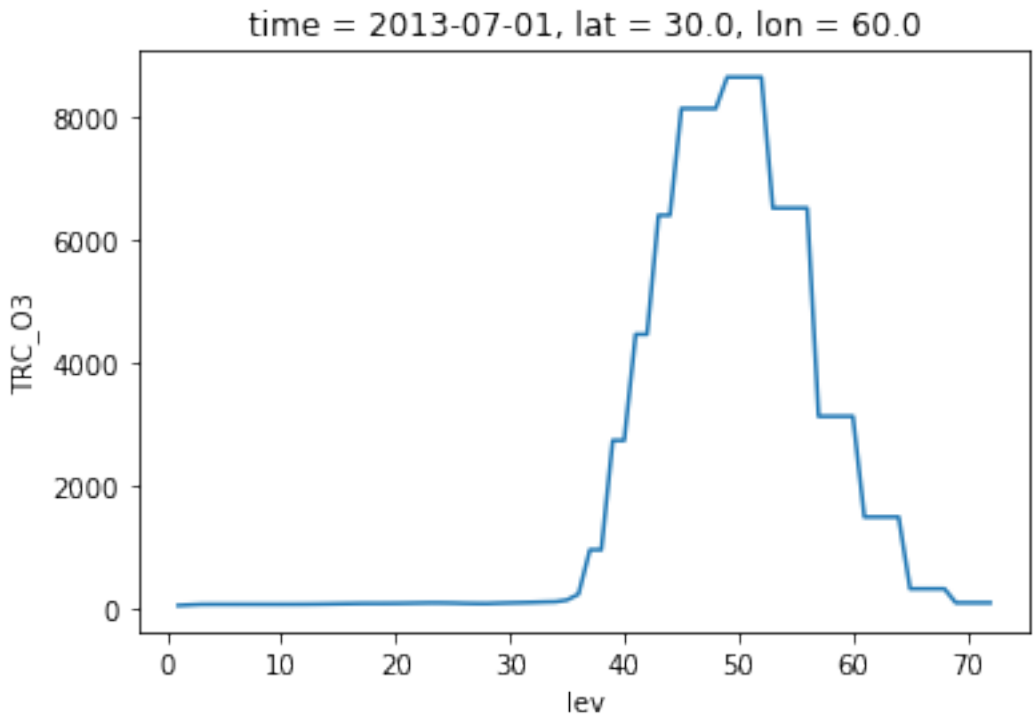
```
Out[50]: <xarray.DataArray 'TRC_O3' (lev: 72)>
         array([   40.683279,    50.502422,    58.76155 ,    59.553575,    59.688794,
                   59.776063,    59.863531,    59.921533,    59.98514 ,    60.082513,
                   60.335651,    61.095989,    62.673543,    65.194925,    67.979464,
                   70.64056 ,    73.474794,    74.231899,    74.188087,    75.221195,
                   77.541642,    80.683797,    83.345057,    82.974218,    80.626258,
                   75.736345,    73.050543,    72.00984 ,    78.042632,    82.909999,
                   86.870436,    91.910749,    97.782276,   102.39502 ,   129.772232,
                  225.23146 ,   950.023207,   950.023036,  2735.300086,  2735.299631,
                 4467.107829,  4467.108738,  6410.347396,  6410.347851,  8152.221199,
                 8152.217561,  8152.21847 ,  8152.222108,  8664.22306 ,  8664.223969,
                 8664.226698,  8664.225788,  6528.357062,  6528.355243,  6528.353879,
                 6528.356153,  3128.377784,  3128.377784,  3128.377784,  3128.37733 ,
                 1485.034545,  1485.03409 ,  1485.034431,  1485.034431,   313.26357 ,
                  313.263598,   313.263513,   313.263541,    85.170825,    85.170811,
                   85.170818,    85.170839])
         Coordinates:
             time     datetime64[ns] 2013-07-01
           * lev      (lev) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
             lat      float32 30.0
             lon      float32 60.0
         Attributes:
             long_name:  O3 tracer
             units:      ppbv
```

1.7.2 Plotting

`profile` is a 1D `DataArray` and it also has a convenience method for plotting.

```
In [51]: profile.plot()
```

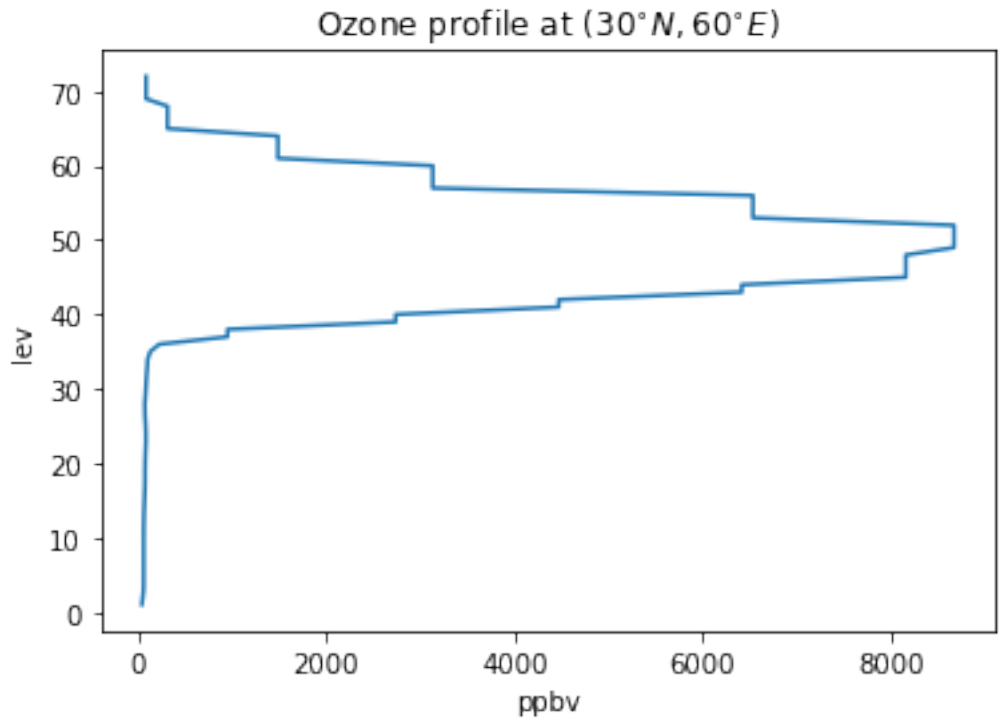
```
Out[51]: [<matplotlib.lines.Line2D at 0x1152b4da0>]
```



This is particularly useful for time-series, but for profile we want `lev` to be the y-axis. We can always fall back to basic matplotlib functions.

```
In [52]: plt.plot(profile, profile['lev'])
         plt.ylabel('lev');plt.xlabel('ppbv')
         plt.title('Ozone profile at $(30^{\circ}N, 60^{\circ}E)$')
```

```
Out[52]: <matplotlib.text.Text at 0x115317e80>
```



1.8 Writing NetCDF file

During the previous 3 cases, we've made several changes to our `DataArray` object, including

- scale its value by 10^9
- change its attribute 'unit' from 'mol/mol' to 'ppbv'
- drop the time dimension
- change its vertical coordinate values to integers
- change the vertical coordinate unit to 'unitless'

```
In [53]: dr # check its content
```

```
Out[53]: <xarray.DataArray 'TRC_O3' (lev: 72, lat: 46, lon: 72)>
         array([[[ 26.152373,  26.152373, ...,  26.152373,  26.152373],
                 [ 26.224114,  26.226319, ...,  26.221985,  26.20175 ],
                 ...,
                 [  6.469575,   6.469804, ...,   6.476856,   6.473523],
                 [  6.460082,   6.460082, ...,   6.460082,   6.460082]],

                [[ 26.328511,  26.328511, ...,  26.328511,  26.328511],
                 [ 26.338292,  26.334288, ...,  26.326118,  26.327644],
                 ...,
                 [ 10.846659,  10.846609, ...,  10.846533,  10.846559],
                 [ 10.847146,  10.847146, ...,  10.847146,  10.847146]],

                ...,
                [[ 59.93882 ,  59.93882 , ...,  59.93882 ,  59.93882 ],
                 [ 59.93882 ,  59.93882 , ...,  59.93882 ,  59.93882 ],
                 ...,
```

```
                [ 68.32267 ,  68.32267 , ...,  68.32267 ,  68.32267 ],
                [ 68.322663,  68.322663, ...,  68.322663,  68.322663]],

               [[ 59.938827,  59.938827, ...,  59.938827,  59.938827],
                [ 59.938813,  59.938813, ...,  59.938813,  59.938813],
                ...,
                [ 68.322663,  68.322663, ...,  68.322663,  68.322663],
                [ 68.322656,  68.322656, ...,  68.322656,  68.322656]]])
        Coordinates:
            time     datetime64[ns] 2013-07-01
          * lev      (lev) int64 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
          * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
          * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
        Attributes:
            long_name:  O3 tracer
            units:      ppbv
```

We can save this modified `DataArray` to a file by just one line of code. This simplicity is just amazing, compare to the terribly complicated procedure in IDL.

```
In [54]: dr.to_netcdf('O3_restart.nc')
```

Use `ncdump` in the Linux shell to check its content:

```
$ncdump -h O3_restart.nc

netcdf O3_restart {
dimensions:
    lev = 72 ;
    lat = 46 ;
    lon = 72 ;
variables:
    float time ;
        time:_FillValue = NaNf ;
        time:long_name = "Time" ;
        time:axis = "T" ;
        time:delta_t = "0000-00-00 00:00:00" ;
        time:units = "hours since 1985-01-01" ;
        time:calendar = "gregorian" ;
    float lev(lev) ;
        lev:_FillValue = NaNf ;
        lev:long_name = "level index" ;
        lev:units = "unitless" ;
        lev:positive = "up" ;
        lev:axis = "Z" ;
    float lat(lat) ;
        lat:_FillValue = NaNf ;
        lat:long_name = "Latitude" ;
        lat:units = "degrees_north" ;
        lat:axis = "Y" ;
    float lon(lon) ;
        lon:_FillValue = NaNf ;
        lon:long_name = "Longitude" ;
        lon:units = "degrees_east" ;
        lon:axis = "X" ;
    float TRC_O3(lev, lat, lon) ;
        TRC_O3:_FillValue = 1.e+30f ;
        TRC_O3:long_name = "O3 tracer" ;
        TRC_O3:units = "ppbv" ;
```

(continues on next page)

(continued from previous page)

```
        TRC_O3:add_offset = 0.f ;
        TRC_O3:scale_factor = 1.f ;

// global attributes:
        :coordinates = "time" ;
}
```

We can see that the file has pretty complete information – dimensions, coordinates, units, everything looks fine.

Finally, open this new file to check everything is correct:

```
In [55]: xr.open_dataarray('O3_restart.nc')
```

```
Out[55]: <xarray.DataArray 'TRC_O3' (lev: 72, lat: 46, lon: 72)>
         [238464 values with dtype=float64]
         Coordinates:
             time     datetime64[ns] 2013-07-01
           * lev      (lev) float32 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 ...
           * lat      (lat) float32 -89.0 -86.0 -82.0 -78.0 -74.0 -70.0 -66.0 -62.0 ...
           * lon      (lon) float32 -180.0 -175.0 -170.0 -165.0 -160.0 -155.0 -150.0 ...
         Attributes:
             long_name:  O3 tracer
             units:      ppbv
```

1.9 Further reading

Check out xarray documentation, especially the examples!

CHAPTER 2

Working with global high-resolution data

A unique feature of xarray is it works well with very large data. This is crucial for global high-resolution models like GCHP, which could produce GBs of data even for just one variable. The data often exceed computer's memory, so IDL/MATLAB programs will just die if you try to read such large data.

2.1 Lazy evaluation

Here we use a native resolution GEOS-FP metfield as an example. You can download it at:

```
ftp://ftp.as.harvard.edu/gcgrid/GEOS_0.25x0.3125/GEOS_0.25x0.3125.d/GEOS_FP/2015/07/
↪GEOSFP.20150701.A3cld.Native.nc
```

```
In [1]: %%bash
        du -h ./GEOSFP.20150701.A3cld.Native.nc
```

```
1.4G    ./GEOSFP.20150701.A3cld.Native.nc
```

The file size is 1.4 GB – not extremely large. But that's already after NetCDF compression! The raw data size (after reading into memory) would be more than 20 GB, which exceeds most computers' memory.

Let's see how long it takes to read such a file with xarray

```
In [2]: import xarray as xr
        %time ds = xr.open_dataset("./GEOSFP.20150701.A3cld.Native.nc")
```

```
CPU times: user 19.8 ms, sys: 4.65 ms, total: 24.4 ms
Wall time: 25.9 ms
```

Wait, just milliseconds?

It looks like we do get the entire file:

```
In [3]: ds
```

```
Out[3]: <xarray.Dataset>
        Dimensions:   (lat: 721, lev: 72, lon: 1152, time: 8)
        Coordinates:
```

```
  * time      (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
  * lev       (lev) float32 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 ...
  * lat       (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
  * lon       (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
Data variables:
    CLOUD     (time, lev, lat, lon) float64 0.1229 0.1229 0.1229 0.1229 ...
    OPTDEPTH  (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    QCCU      (time, lev, lat, lon) float64 9.969e+36 9.969e+36 9.969e+36 ...
    QI        (time, lev, lat, lon) float64 8.382e-08 8.382e-08 8.382e-08 ...
    QL        (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TAUCLI    (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TAUCLW    (time, lev, lat, lon) float64 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
Attributes:
    Title:                 GEOS-FP time-averaged 3-hour cloud parameters (A3c...
    Contact:               GEOS-Chem Support Team (geos-chem-support@as.harva...
    References:            www.geos-chem.org; wiki.geos-chem.org
    Filename:              GEOSFP.20150701.A3cld.Native.nc
    History:               File generated on: 2015/11/29 15:02:54 GMT-0400
    ProductionDateTime:    File generated on: 2015/11/29 15:02:54 GMT-0400
    ModificationDateTime:  File generated on: 2015/11/29 15:02:54 GMT-0400
    Format:                NetCDF-4
    SpatialCoverage:       global
    Conventions:           COARDS
    Version:               GEOS-FP
    Model:                 GEOS-5
    Nlayers:               72
    Start_Date:            20150701
    Start_Time:            00:00:00.0
    End_Date:              20150701
    End_Time:              23:59:59.99999
    Delta_Time:            030000
    Delta_Lon:             0.312500
    Delta_Lat:             0.250000
```

But that's deceptive. xarray only reads the metadata such as dimensions, coordinates and attributes, so it is lightning fast. It will read the actual numerical data if you explicitly execute `ds.load()`. **Never** try this command on this kind of large data because your program will just die.

xarray will automatically retrieve the data from disk **only when** they are actually needed. In other words, it tries to keep the memory usage as low as possible. We call this **lazy evaluation**.

Any operation that doesn't require the actual data will be lightning fast. Such as selecting a variable:

```
In [4]: dr = ds['CLOUD'] # super fast
        dr
```

```
Out[4]: <xarray.DataArray 'CLOUD' (time: 8, lev: 72, lat: 721, lon: 1152)>
        [478420992 values with dtype=float64]
        Coordinates:
          * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
          * lev      (lev) float32 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 ...
          * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
          * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
        Attributes:
            long_name:       Total cloud fraction in grid box
            units:           1
            gamap_category:  GMAO-3D$
```

Even for indexing or any other array operations: (as long as your don't explicitly ask for the data values)

```
In [5]: dr_surf = dr.isel(lev=0) # super fast
```

```
        dr_surf
```

```
Out[5]: <xarray.DataArray 'CLOUD' (time: 8, lat: 721, lon: 1152)>
        [6644736 values with dtype=float64]
        Coordinates:
          * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
            lev      float32 1.0
          * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
          * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
        Attributes:
            long_name:       Total cloud fraction in grid box
            units:           1
            gamap_category:  GMAO-3D$
```

This `dr_surf` object points to the surface level of the *CLOUD* data. This subset of data is sufficiently small so we can read it into memory. Even reading a single level takes 2 seconds, so you can imagine how long it will take to read the full data.

(Note that the NetCDF format allows you to only read a subset of data into memory)

```
In [6]: %time dr_surf.load()
```

```
CPU times: user 1.87 s, sys: 132 ms, total: 2.01 s
Wall time: 2.01 s
```

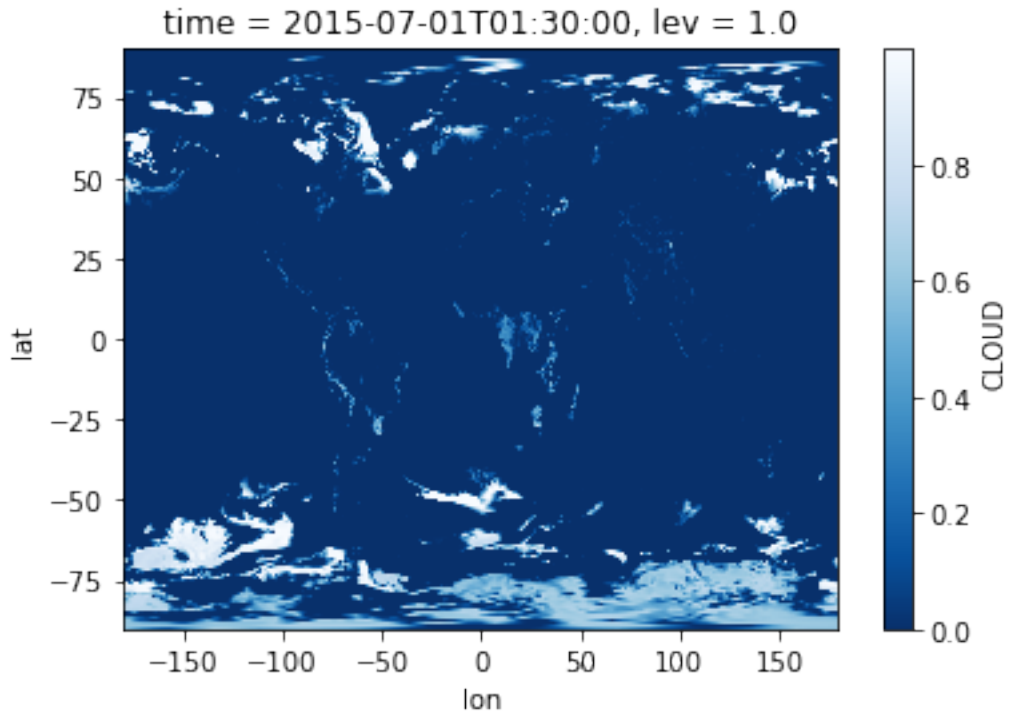
```
Out[6]: <xarray.DataArray 'CLOUD' (time: 8, lat: 721, lon: 1152)>
        array([[[ 0.122925,  0.122925, ...,  0.122925,  0.122925],
                [ 0.289062,  0.290039, ...,  0.287598,  0.288574],
                ...,
                [ 0.      ,  0.      , ...,  0.      ,  0.      ],
                [ 0.      ,  0.      , ...,  0.      ,  0.      ]],

               [[ 0.151123,  0.151123, ...,  0.151123,  0.151123],
                [ 0.27002 ,  0.269043, ...,  0.272461,  0.271484],
                ...,
                [ 0.      ,  0.      , ...,  0.      ,  0.      ],
                [ 0.      ,  0.      , ...,  0.      ,  0.      ]],

               ...,
               [[ 0.211426,  0.211426, ...,  0.211426,  0.211426],
                [ 0.106689,  0.106567, ...,  0.106934,  0.106812],
                ...,
                [ 0.      ,  0.      , ...,  0.      ,  0.      ],
                [ 0.      ,  0.      , ...,  0.      ,  0.      ]],

               [[ 0.548828,  0.548828, ...,  0.548828,  0.548828],
                [ 0.624023,  0.625   , ...,  0.623047,  0.623047],
                ...,
                [ 0.      ,  0.      , ...,  0.      ,  0.      ],
                [ 0.      ,  0.      , ...,  0.      ,  0.      ]]])
        Coordinates:
          * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
            lev      float32 1.0
          * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
          * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
        Attributes:
            long_name:       Total cloud fraction in grid box
            units:           1
            gamap_category:  GMAO-3D$
```

Since the data is now in memory, we can plot it. In fact, you don't need to run `dr_surf.load()` before plotting.

xarray will automatically read the data when you try to plot, because a plotting operation requires the actual data values.

```
In [7]: %matplotlib inline
        dr_surf.isel(time=0).plot(cmap='Blues_r')
```

```
Out[7]: <matplotlib.collections.QuadMesh at 0x1238efe10>
```



2.2 More explanation on data size

So how much data is actually read into memory? Just 50 MB.

```
In [8]: print(dr_surf.nbytes / 1e6, 'MB')
```

```
53.157888 MB
```

An equivalent calculation:

```
In [9]: print(8*721*1152 * 8 / 1e6, 'MB') # ntime*nlon*nlat * 8 byte float
```

```
53.157888 MB
```

What's the size of the entire *CLOUD* data, with all 72 levels?

```
In [10]: print(dr.nbytes / 1e9, 'GB')
```

```
3.827367936 GB
```

An equivalent calculation:

```
In [11]: print(8*721*1152*72 * 8 / 1e9, 'GB') # ntime*nlon*nlat*nlev * 8 byte float
```

```
3.827367936 GB
```

Even if the memory is enough, reading such data will take very long!

Finally, what's the size of the entire NetCDF file?

```
In [12]: print(ds.nbytes / 1e9, 'GB')
```

```
26.791583396 GB
```

Such size will kill almost all laptops. But thanks to lazy evaluation, we have no problem with it.

2.3 Out-of-core data processing

Sometimes you do need to process the entire data (e.g. take global average), not just select a subset. Fortunately, xarray can still deal with it, thanks to the out-of-core computing technique.

We know our NetCDF data has 8 time slices, so let's divide it into 8 pieces so that a single piece would not exceed memory. (see xarray documentation for more about `chunk`)

```
In [13]: ds = ds.chunk({'time': 1})
```

Now our `DataArray` has a new `chunksize` attribute:

```
In [14]: dr = ds['CLOUD']
         dr
```

```
Out[14]: <xarray.DataArray 'CLOUD' (time: 8, lev: 72, lat: 721, lon: 1152)>
         dask.array<xarray-CLOUD, shape=(8, 72, 721, 1152), dtype=float64, chunksize=(1, 72, 721, 115
         Coordinates:
           * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
           * lev      (lev) float32 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 ...
           * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
           * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
         Attributes:
             long_name:       Total cloud fraction in grid box
             units:           1
             gamap_category:  GMAO-3D$
```

We want to take column average over the entire data field. The following command still uses lazy-evaluation without doing actual computation.

```
In [15]: dr.mean(dim='lev')
```

```
Out[15]: <xarray.DataArray 'CLOUD' (time: 8, lat: 721, lon: 1152)>
         dask.array<mean_agg-aggregate, shape=(8, 721, 1152), dtype=float64, chunksize=(1, 721, 1152)
         Coordinates:
           * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
           * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
           * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
```

Use `dr.mean(dim='lev').compute()` when you actually need the results. Then xarray will process each piece ("chunk") one by one, and put the results together.

Because xarray uses dask under the hood, we can use dask ProgressBar to monitor the progress. Otherwise it will look like getting stuck. (The `with` statement below is a context manager. Just think it as activating a temporary funtionality inside the `with` block. For example, a popular visualization package seaborn uses `with` to set temporary style)

```
In [16]: from dask.diagnostics import ProgressBar

         with ProgressBar():
             dr_levmean = dr.mean(dim='lev').compute()
```

```
[########################################] | 100% Completed | 17.9s
```

It takes 20 seconds, kind of time-consuming. But this calculation is not possible at all with traditional tools like IDL/MATLAB.

Now we see the actual numerical data:

```
In [17]: dr_levmean
```

```
Out[17]: <xarray.DataArray 'CLOUD' (time: 8, lat: 721, lon: 1152)>
         array([[[ 0.098929,  0.098929, ...,  0.098929,  0.098929],
                 [ 0.116195,  0.116132, ...,  0.116353,  0.116277],
                 ...,
                 [ 0.027914,  0.027806, ...,  0.028132,  0.028024],
                 [ 0.02229 ,  0.02229 , ...,  0.02229 ,  0.02229 ]],

                [[ 0.103683,  0.103683, ...,  0.103683,  0.103683],
                 [ 0.114203,  0.114143, ...,  0.11426 ,  0.114244],
                 ...,
                 [ 0.026128,  0.026203, ...,  0.026008,  0.026071],
                 [ 0.016974,  0.016974, ...,  0.016974,  0.016974]],

                ...,
                [[ 0.07467 ,  0.07467 , ...,  0.07467 ,  0.07467 ],
                 [ 0.072274,  0.072266, ...,  0.07231 ,  0.072309],
                 ...,
                 [ 0.006045,  0.006052, ...,  0.00604 ,  0.006046],
                 [ 0.008976,  0.008976, ...,  0.008976,  0.008976]],

                [[ 0.069148,  0.069148, ...,  0.069148,  0.069148],
                 [ 0.070722,  0.070705, ...,  0.070723,  0.070725],
                 ...,
                 [ 0.002394,  0.002377, ...,  0.002432,  0.002411],
                 [ 0.003763,  0.003763, ...,  0.003763,  0.003763]]])
         Coordinates:
           * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
           * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
           * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
```

You can use `dr_levmean.to_netcdf("filename.nc")` to save the result to disk.

We can further simplify the above process. Instead of using `.compute()` to get in-memory result, we can simply do a "disk to disk" operation.

```
In [18]: with ProgressBar():
             dr.mean(dim='lev').to_netcdf('column_average.nc')
```

```
[########################################] | 100% Completed | 19.4s
```

xarray automatically reads each piece (chunk) of data into memory, compute the mean, dump the result to disk, and proceed to the next piece.

Open the file to double check:

```
In [19]: xr.open_dataset('column_average.nc')
```

```
Out[19]: <xarray.Dataset>
         Dimensions:  (lat: 721, lon: 1152, time: 8)
         Coordinates:
           * time     (time) datetime64[ns] 2015-07-01T01:30:00 2015-07-01T04:30:00 ...
           * lat      (lat) float32 -89.9375 -89.75 -89.5 -89.25 -89.0 -88.75 -88.5 ...
           * lon      (lon) float32 -180.0 -179.688 -179.375 -179.062 -178.75 ...
         Data variables:
             CLOUD    (time, lat, lon) float64 0.09893 0.09893 0.09893 0.09893 ...
```

2.4 A compact version of previous section

All we've done in the previous section can be summarized by 2 lines of code:

```
In [20]: ds = xr.open_dataset("./GEOSFP.20150701.A3cld.Native.nc", chunks={'time': 1})
         ds['CLOUD'].mean(dim='lev').to_netcdf('column_average_CLOUD.nc')
```

That's it! You've just solved a geoscience big data problem without worrying about any specific "big data" techniques!

CHAPTER 3

Jupyter notebook on remote server

Jupyter notebook works very well with remote servers. That's important for earth science modeling people because we seldom run models and store data on our personal computers!

Traditionally, we display figures on a remote server using x11 forwarding, but that's extremely slow and often annoying. On the contrary, Jupyter simply uses HTTP protocal and runs in a web browser. You user experience will not change at all when using Jupyter on a remote server - still a web browser, still very fast.

3.1 Private server

On your own server or cloud computing platforms, setting up Jupyter is almost trivial. See the following link or other similar tutorials.

- https://techtalktone.wordpress.com/2017/03/28/running-jupyter-notebooks-on-a-remote-server-via-ssh/

3.2 Shared HPC cluster

3.2.1 General reference

The way to set up Jupyter on a shared HPC cluster could be system-dependent. You may ask system architects or refer to:

- https://medium.com/@rabernat/custom-conda-environments-for-data-science-on-hpc-clusters-32d58c63aa95
- http://ipyrad.readthedocs.io/HPC_Tunnel.html

3.2.2 Harvard Odyssey

The following script works on Harvard Odyssey, which uses the slurm system.

```
#!/bin/bash
#SBATCH -p general
#SBATCH -N 1
#SBATCH -c 1
#SBATCH -t 2:00:00
#SBATCH --mem-per-cpu 8000
#SBATCH --job-name notebook
#SBATCH --output jupyter-address-%J.log

## get tunneling info
XDG_RUNTIME_DIR=""
ipnport=$(shuf -i8000-9999 -n1)
ipnip=$(hostname -i)

## print tunneling instructions to the output file
echo -e "
    Copy/Paste this in your local terminal to ssh tunnel with remote
    -----------------------------------------------------------------
    ssh -N -L $ipnport:$ipnip:$ipnport user@host
    -----------------------------------------------------------------

    Then open a browser on your local machine to the following address
    -----------------------------------------------------------------
    localhost:$ipnport  (prefix w/ https:// if using password)
    -----------------------------------------------------------------
    "

## start an ipcluster instance and launch jupyter server
source activate GCPy # change it to your environment name
jupyter-notebook --NotebookApp.token='' --no-browser --port=$ipnport --ip=$ipnip
```