nctoolkit

Robert Wilson

Jul 10, 2020

GETTING STARTED

The goal of nctoolkit is to provide a comprehensive tool in Python for manipulating NetCDF data. The philosophy is to provide sufficient methods to carry out 80-90% of what you want to do with NetCDF files.

nctoolkit is designed with both individual files and ensembles in mind.

Under the hood nctoolkit relies on the command line packages Climate Data Operates (CDO), with NCO as an optional dependency. No prior knowledge of CDO is required to use nctoolkit.

The package is design with two uses in mind: computationally intensive data post-processing and interactive Jupyter notebook type analysis. An auto plotting feature is provided to aid rapid data analysis.

In addition to the guidance given here, tutorials for how to use nctoolkit are available at nctoolkit's GitHub page.

CHAPTER

ONE

DOCUMENTATION

Getting Started

1.1 Installation

1.1.1 Python dependencies

- Python (3.6 or later)
- numpy (1.14 or later)
- pandas (0.24 or later)
- xarray (0.14 or later)
- hvplot (0.5 or later)
- NetCDF4 (1.53 or later)
- panel (0.9.1 or later)

1.1.2 System dependencies

There are two main system dependencies: Climate Data Operators, and NCO. The easiest way to install them is using conda:

```
$ conda install -c conda-forge cdo

$ conda install -c conda-forge nco
```

While CDO is necessary for the package to work, NCO is an optional dependency.

If you want to install CDO from source with NetCDF and HDF5 support, you can use one of the bash scripts available here.

1.1.3 How to install nctoolkit

To install nctoolkit using pip:

```
$ pip install nctoolkit
```

To install the development version of nctoolkit:

```
$ pip install git+https://github.com/r4ecology/nctoolkit.git
```

1.2 Introduction tutorial

The fundamental object of analysis in this package is a nctoolkit dataset. Each object is initialized with a single netcdf file or an ensemble of files, and it will keep track of any manipulations carried out.

Behind the scenes most of the manipulations are done using CDO. Datasets will keep track of all CDO NCO commands used. However, unless you are experienced with CDOyou can ignore all of this.

1.2.1 Opening netcdf data

I will illustrate the basic usage using a climatology of global sea surface temperature from NOAA. We can download this from here. To download using wget:

```
wget ftp://ftp.cdc.noaa.gov/Datasets/COBE/sst.mon.ltm.1981-2010.nc
```

The first step in any analysis will be to import nctoolkit, which I will call nc as shorthand. Please note I am suppressing warnings to make this notebook more readable. I do not recommend suppressing warnings....

[1]:
```
import nctoolkit as nc
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Under the hood nctoolkit will generate temporary netcdf files. The package is designed to remove temp files that are no longer in use, and will automatically clean up any temporary files generated when Python closes. However, this is not 100% guaranteed to work during system crashes etc.

It is therefore recommended to do a deep_clean at the start of any session to remove any leftover netcdf files that might have existed in a previous sessions. Obviously, do not run this if you have multiple instances of nctoolkit running simultaneously.

[2]:
```
nc.deep_clean()
```

We can then set up the dataset, which we will use for manipulating the SST climatology.

[3]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
```

1.2.2 Accessing dataset attributes

At this point there is very little useful information in the dataset. Essentially, all it tell us is the start file. This will always remain the same.

[4]:
```
sst.start
```
[4]:
```
'sst.mon.ltm.1981-2010.nc'
```

The current state of the dataset can be found as follows.

[5]:
```
sst.current
```
[5]:
```
'sst.mon.ltm.1981-2010.nc'
```

We can access the dataset's history as follows, which is initially empty.

[6]:
```
sst.history
```
[6]:
```
[]
```

A simple, but important first task when analyzing netcdf data is knowing the variables in the file. We can do this quickly by accessing the variables attribute

[7]:
```
sst.variables
```
[7]:
```
['sst', 'valid_yr_count']
```

Often, we will want to know the size of a dataset. This is most relevant when we are working with multiple files. We can do this by accessing the size attribute. To speed up computations, variables and size are computed lazily.

[8]:
```
sst.size
```
[8]:
```
'Number of files: 1\nFile size: 4.670688 MB'
```

In this case we can see that the file is 4 MB, and we are also told that there is only one file.

1.2.3 Variable selection and geographic clipping

We can clip netcdf files in space or time using clip. Let's say we only cared about temperature in July for the North Atlantic. This be found very easily using the following.

netcdf files often have variables that we are not interested in. We can therefore easily select or delete variables. If we want to select variables we can use the select_variables method, which requires either a single variable or a list of variables. Here I will select sst.

[9]:
```
sst.select_variables("sst")
```

We can now see that there is only one variable in the sst dataset

[10]:
```
sst.variables
```
[10]:
```
['sst']
```

We can also that a temporary file has been created with only this variable in it

[11]:
```
sst.current
```

[11]:
```
'/tmp/nctoolkitmexmyssrnctoolkittmp1jbp0kvg.nc'
```

If we want to clip the dataset geographically we can use the clip method. All we need is the longitude and latitude range. So if we wanted to clip the SST data to the North Atlantic we would do the following.

[12]:
```
sst.clip(lon = [-80, 20], lat = [30, 80])
```

We have now carried out some manipulations on the dataset. So, the current file has now changed.

Likewise, we now have a history to look at.

[13]:
```
sst.history
```

[13]:
```
['cdo -L -selname,sst sst.mon.ltm.1981-2010.nc /tmp/
↪nctoolkitmexmyssrnctoolkittmp1jbp0kvg.nc',
 'cdo -L  -sellonlatbox,-80,20,30,80 /tmp/nctoolkitmexmyssrnctoolkittmp1jbp0kvg.nc /
↪tmp/nctoolkitmexmyssrnctoolkittmpbp36y2p_.nc']
```

This will give us the list of CDO or NCO commands used under the hood. nctoolkit is designed to be usable without any prior knowledge of CDO or NCO.

1.2.4 Deleting an object

If we want to delete a dataset we simply use the standard python del approach. nctoolkit has been designed so that it is constantly cleaning up the system using a simple rule: only keep temp files created if they are among the current files of datasets in the current session. Right now, we only have one dataset, called "sst". So if we delete "sst" it will also delete the current temp file from that dataset. We can see this by looking at what happens to the temp file related to sst when we delete sst. Right now it exists on the system.

[14]:
```
import os
x = sst.current
os.path.exists(x)
```

[14]:
```
True
```

But if we delete sst, this file will disappear.

[15]:
```
del sst
os.path.exists(x)
```

[15]:
```
False
```

1.2.5 Viewing a dataset using the auto plot feature

nctoolkit has a built in, though slightly experimental, method for quick plotting. This will check the contents of the dataset and plot accordingly. The general approach of autoplot is very similar to ncview on the command line.

[16]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_months(1)
sst.reduce_dims()
sst.plot()
```

Data type cannot be displayed:

[16]:
```
:DynamicMap   [Variable]
   :Image   [lon,lat]   (sst)
```

1.2.6 Statistical operations

nctoolkit has a large number of built in statistical operations, largely built around the methods available in CDO.

1.2.7 Time averaging

Averaging in time is one of the most common operations required on netcdf data. nctoolkit allows users to calculate long-term time averages, monthly climatologies, seasonal summaries and many other common statistics.

In this case we are analyzing a monthly climatology of SST. However, what we really might be interested in is the annual average. This can be calculated using the simple mean method, which will calculate the mean over all time steps.

[17]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.mean()
sst.reduce_dims()
sst.plot()
```

Data type cannot be displayed:

[17]:
```
:Image   [lon,lat]   (sst)
```

Instead of the annual mean, we might be interested in the range of temperatures during the year.

[18]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.range()
sst.reduce_dims()
sst.plot()
```

Data type cannot be displayed:

[18]:
```
:Image   [lon,lat]   (sst)
```

Other operations, such as maximum, minimum, and standard deviation are available.

1.2.8 Spatial statistics

Let's move on to some more advanced methods. I will illustrate these using NOAA's long-term monthly global data set of sea surface temperatures from 1850 to the present day. You can learn more about this data set here. This file is approximately 500 MB.

To download using wget:

```
wget ftp://ftp.cdc.noaa.gov/Datasets/COBE/sst.mon.mean.nc
```

This is a long-term data set of global sea surface temperature. So, let's find out what has happened to average global sea surface temperature since 1850. Unsurprising spoiler: it has been going up. Let's start by setting up the dataset.

[19]:
```
ff =  "sst.mon.mean.nc"
sst = nc.open_data(ff)
```

We now need to calculate the average global SST. We can do this using the spatial_mean method. This will calculate an area weighted mean for each time step.

[20]:
```
sst.spatial_mean()
```

We can now plot the time series of monthly global mean SST since 1850

[21]:
```
sst.plot()
```

Data type cannot be displayed:

[21]:
```
:Curve   [time]   (x)
```

Our time series shows that, as expected SST increased during the 20th Century. However, this figure has too much noise. We do not care about month to month variations.Instead, let's look at the rolling 20 year mean. To do this, we will need to first calculate an annual mean then calculated the rolling mean using a window of 20 years. Alternatively, we can just calculate a rolling mean on the initial data using a rolling mean of 20*12 = 240 months.

[22]:
```
ff =  "sst.mon.mean.nc"
sst = nc.open_data(ff)
sst.spatial_mean()
```

To calculate the annual mean we can simply use the yearly_mean method.

[23]:
```
sst.annual_mean()
```

To calculate the rolling mean, we can use rolling_mean, with window set to 20.

[24]:
```
sst.rolling_mean(window = 20)
```

[25]:
```
sst.plot()
```

Data type cannot be displayed:

[25]:
```
:Curve   [time]   (x)
```

This looks much cleaner. Please note that at present nctoolkit does not adjust the time outputs from CDO. So in this case the rolling mean is centred in the middle of the 20 year period. As nctoolkit evolves more windowing options will be provided to users.

1.3 Ensemble methods

1.3.1 Merging files with different variables

This notebook will outline some general methods for doing comparisons of multiple files. We will work with two different sea surface temperature data sets from NOAA and the Met Office Hadley Centre.

[1]:
```
import nctoolkit as nc
import pandas as pd
import xarray as xr
import numpy as np
```

Let's start by downloading the files using wget. Uncomment the code below to do this (note: you will need to extract the HadISST dataset):

[2]:
```
# ! wget ftp://ftp.cdc.noaa.gov/Datasets/COBE2/sst.mon.mean.nc
# ! wget https://www.metoffice.gov.uk/hadobs/hadisst/data/HadISST_sst.nc.gz
```

The first step is to get the data. We will start by creating two separate datasets for each file.

[3]:
```
sst_noaa = nc.open_data("sst.mon.mean.nc")
sst_hadley = nc.open_data("HadISST_sst.nc")
```

We can see that both variables have sea surface temperature labelled as sst. So we will need to change that.

[4]:
```
sst_noaa.variables
```

[4]:
```
['sst']
```

[5]:
```
sst_hadley.variables
```

[5]:
```
['time_bnds', 'sst']
```

[6]:
```
sst_noaa.rename({"sst":"noaa"})
sst_hadley.rename({"sst":"hadley"})
```

The data sets also cover different time periods, and only have overlapping between 1870 and 2018. so we will need to select those years

[7]:
```
sst_noaa.select_years(range(1870, 2019))
sst_hadley.select_years(range(1870, 2019))
```

We also have a problem in that there are two horizontal grids in the Hadley Centre file. We can solve this by selecting the sst variable only

[8]:
```
sst_hadley.select_variables("hadley")
```

At this point, the datasets have the same number of time steps and months covered. However, the grids are still a bit different. So we want to unify them by regridding one dataset on to the other's grid. This can be done using regrid, or any grid of your choosing.

[9]:
```
sst_noaa.regrid(grid = sst_hadley)
```

We now have two separate datasets. Let's create a new dataset that has both of them, and then merge them. When doing this we need to make sure nas are treated properly. In this case Hadley Centre values not being NAs as they should be, so we need to fix that. The merge method also requires a strict matching criteria for the dates in the merging

files. In this case the Hadley Centre and NOAA data sets both give monthly means, but use a different day of the month. So we will set match to ["year", "month"] this will ensure there are no mis-matches

[10]:
```
all_sst = nc.merge(sst_noaa, sst_hadley, match = ["year", "month"])
all_sst.set_missing([-9000, - 900])
```

Let's work out what the global mean SST was over the time period. Note that this will not be totally accurate as there are some missing values here and there that might bias things.

[11]:
```
all_sst.spatial_mean()
all_sst.annual_mean()
all_sst.rolling_mean(10)
```

[12]:
```
all_sst.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed:

[12]:
```
:DynamicMap   [variable]
   :Curve   [time]   (value)
```

We can also work out the difference between the two. Here we wil work out the monthly bias per cell. Then calculate the mean global difference per year, and then calculate a rolling 10 year mean.

[13]:
```
all_sst = nc.open_data([sst_noaa.current, sst_hadley.current])
all_sst.merge(match = ["year", "month"])
all_sst.transmute({"bias":"hadley-noaa"})
all_sst.set_missing([-9000, - 900])
all_sst.spatial_mean()
all_sst.annual_mean()
all_sst.rolling_mean(10)
all_sst.plot()
```

Data type cannot be displayed:

[13]:
```
:Curve   [time]   (x)
```

You can see that there is a notable difference at the start of the time series.

1.3.2 Merging files with different times

TBC

1.3.3 Ensemble averaging

TBC

1.4 Speeding up code

1.4.1 Lazy evaluation

Under the hood nctoolkit relies mostly on CDO to carry out the specified manipulation of netcdf files. Each time CDO is called a new temporary file is generated. This has the potential to result in slower than necessary processing chains, as IO takes up far too much time.

I will demonstrate this using a netcdf file os sea surface temperature. To download the file we can just use wget:

[1]:
```
import nctoolkit as nc
import warnings
warnings.filterwarnings('ignore')
from IPython.display import clear_output
!wget ftp://ftp.cdc.noaa.gov/Datasets/COBE2/sst.mon.ltm.1981-2010.nc
clear_output()
```

We can then set up the dataset which we will use for manipulating the SST climatology.

[2]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
```

Now, let's select the variable sst, clip the file to the northern hemisphere, calculate the mean value in each grid cell for the first half of the year, and then calculate the spatial mean.

[3]:
```
sst.select_variables("sst")
sst.clip(lat = [0,90])
sst.select_months(list(range(1,7)))
sst.mean()
sst.spatial_mean()
```

The dataset's history is as follows:

[4]:
```
sst.history
```

[4]:
```
['cdo -L -selname,sst sst.mon.ltm.1981-2010.nc /tmp/
↪nctoolkitqhgujflsnctoolkittmpipj7up1l.nc',
 'cdo -L  -sellonlatbox,-180,180,0,90 /tmp/nctoolkitqhgujflsnctoolkittmpipj7up1l.nc /
↪tmp/nctoolkitqhgujflsnctoolkittmp920v1_r7.nc',
 'cdo -L -selmonth,1,2,3,4,5,6 /tmp/nctoolkitqhgujflsnctoolkittmp920v1_r7.nc /tmp/
↪nctoolkitqhgujflsnctoolkittmpbnck_dy2.nc',
 'cdo -L -timmean /tmp/nctoolkitqhgujflsnctoolkittmpbnck_dy2.nc /tmp/
↪nctoolkitqhgujflsnctoolkittmpjmzt1l67.nc',
 'cdo -L -fldmean /tmp/nctoolkitqhgujflsnctoolkittmpjmzt1l67.nc /tmp/
↪nctoolkitqhgujflsnctoolkittmpdus63y8i.nc']
```

In total, there are 5 operations, with temporary files created each time. However, we only want to generate one temporary file. So, can we do that? Yes, thanks to CDO's method chaining ability. If we want to utilize this we need to set the session's evaluation to lazy, using options. Once this is done nctoolkit will only evaluate things either when it needs to, e.g. you call a method that cannot possibly be chained, or if you release it, using release. This works as follows:

[5]:
```
ff =  "sst.mon.ltm.1981-2010.nc"
nc.options(lazy = True)
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.clip(lat = [0,90])
sst.select_months(list(range(1,7)))
sst.mean()
sst.spatial_mean()
sst.release()
```

We can now see that the history is much cleaner, with only one command.

[6]:
```
sst.history
```

[6]:
```
['cdo -L -fldmean -timmean -selmonth,1,2,3,4,5,6  -sellonlatbox,-180,180,0,90 -
↪selname,sst sst.mon.ltm.1981-2010.nc /tmp/nctoolkitqhgujflsnctoolkittmpkdkiwey2.nc']
```

How does this impact run time? Let's time the original, unchained method.

[7]:
```
%%time
nc.options(lazy = False)
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.clip(lat = [0,90])
sst.select_months(list(range(1,7)))
sst.mean()
sst.spatial_mean()
```

```
CPU times: user 37.2 ms, sys: 61.6 ms, total: 98.7 ms
Wall time: 667 ms
```

[8]:
```
%%time
nc.options(lazy = True)
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.clip(lat = [0,90])
sst.select_months(list(range(1,7)))
sst.mean()
sst.spatial_mean()
sst.release()
```

```
CPU times: user 17.3 ms, sys: 4.28 ms, total: 21.6 ms
Wall time: 161 ms
```

This was almost 4 times faster. Exact speed improvements, will of course depend on specific IO requirements, and some times using lazy evaluation will make negligible impact, but in others can make code over 10 times fasteExact speed improvements, will of course depend on specific IO requirements, and some times using lazy evaluation will make negligible impact, but in others can make code over 10 times faster.

1.4.2 Processing files in parallel

When processing a dataset made up of multiple files, it is possible carry out the processing in parallel for more or less all of the methods available in nctoolkit. To carry out processing in parallel with 6 cores, we would use options as follows:

[9]:
```
nc.options(cores = 6)
```

By default the number of cores in use is 1. Of course, this can result in you crashing your computer if the total RAM in use is excessive, so it's best practise to check RAM used with one core first.

1.4.3 Using thread-safe libraries

If the CDO installation being called by nctoolkit is compiled with threadsafe hdf5, then you can achieve potentially significant speed ups with the following command:

[10]:
```
nc.options(thread_safe = True)
```

If you are not sure, if hdf5 has been built thread safe, a simple way to find this out is to run the code below. If it fails, you can be more or less certain it is not threadsafe.

[11]:
```
nc.options(lazy = True)
nc.options(thread_safe = True)
ff =  "sst.mon.ltm.1981-2010.nc"
sst = nc.open_data(ff)
sst.select_variables("sst")
sst.clip(lat = [0,90])
sst.select_months(list(range(1,7)))
sst.mean()
sst.spatial_mean()
sst.release()
```

User Guide

Help & reference

- *An A-Z guide to nctoolkit methods*
- *API Reference*
- *How to guide*

1.5 An A-Z guide to nctoolkit methods

This guide will provide examples of how to use almost every method available in nctoolkit.

1.5.1 add

This method can add to a dataset. You can add a constant, another dataset or a NetCDF file. In the case of datasets or NetCDF files the grids etc. must be of the same structure as the original dataset.

For example, if we had a temperature dataset where temperature was in Celsius, we could convert it to Kelvin by adding 273.15.

```
data = nc.open_data(infile)
data.add(273.15)
```

If we have two sets, we add one to the other as follows:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.add(data2)
```

In the above example, all we are doing is adding infile2 to data2, so instead we could simply do this:

```
data1.add(infile2)
```

1.5.2 annual_anomaly

This method will calculate the annual anomaly for each variable (and in each grid cell) compared with a baseline. This is a standard anomaly calculation where first the mean value is calculated for the baseline period, and the difference between the values is calculated.

For example, if we wanted to calculate the anomalies in a dataset compared with a baseline period of 1900-1919 we would do the following:

```
data = nc.open_data(infile)
data.annual_anomaly(baseline=[1900, 1919])
```

We may be more interested in the rolling anomaly, in particular when there is a lot of annual variation. In the above case, if you wanted a 20 year rolling mean anomaly, you would do the following:

```
data = nc.open_data(infile)
data.annual_anomaly(baseline=[1900, 1919], window=20)
```

By default this method works out the absolute anomaly. However, in some cases the relative anomaly is more interesting. To calculate this we set the metric argument to “relative”:

```
data = nc.open_data(infile)
data.annual_anomaly(baseline=[1900, 1919], metric = "relative")
```

1.5.3 annual_max

This method will calculate the maximum value in each available year and for each grid cell of dataset.

```
data = nc.open_data(infile)
data.annual_max()
```

1.5.4 annual_mean

This method will calculate the maximum value in each available year and for each grid cell of dataset.

```
data = nc.open_data(infile)
data.annual_mean()
```

1.5.5 annual_min

This method will calculate the minimum value in each available year and for each grid cell of dataset.

```
data = nc.open_data(infile)
data.annual_min()
```

1.5.6 annual_range

This method will calculate the range of values in each available year and for each grid cell of dataset.

```
data = nc.open_data(infile)
data.annual_range()
```

1.5.7 bottom

This method will extract the bottom vertical level from a dataset. This is useful for some oceanographic datasets, where the method can let you select the seabed. Note that this method will not work with all data types. For example, in ocean data with fixed depth levels, the bottom cell in the NetCDF data is not the actual seabed. See bottom_mask for these cases.

```
data = nc.open_data(infile)
data.bottom()
```

1.5.8 bottom_mask

This method will identify the bottommost level in each grid with a non-NA value.

```
data = nc.open_data(infile)
data.bottom_mask()
```

1.5.9 cdo_command

This method let's you run a cdo command. CDO commands are generally of the form "cdo {command} infile outfile". cdo_command therefore only requires the command portion of this. If we wanted to run the following CDO command

```
cdo -timmean -selmon,4 infile outfile
```

we would do the following:

```
data = nc.open_data(infile)
data.cdo_command("-timmean -selmon,4")
```

1.5.10 cell_areas

This method either adds the areas of each grid cell to the dataset or converts the dataset to a new dataset showing only the grid cell areas. By default it adds the cell areas (in square metres) to the dataset.

```
data = nc.open_data(infile)
data.cell_areas()
```

If we only want the cell areas we can set join to False:

```
data.cell_areas(join=False)
```

1.5.11 clip

This method will clip a region to a specified longitude and latitude box. For example, if we wanted to clip a dataset to the North Atlantic, we could do this:

```
data = nc.open_data(infile)
data.clip(lon = [-80, 20], lat = [40, 70])
```

1.5.12 compare_all

This method let's us compare all variables in a dataset with a constant. If we wanted to identify the grid cells with values above 20, we could do the following:

```
data = nc.open_data(infile)
data.compare_all(">20")
```

Similarly, if we wanted to identify grid cells with negative values we would do this:

```
data = nc.open_data(infile)
data.compare_all("<0")
```

1.5.13 cor_space

This method calculates the correlation coefficients between two variables in space for each time step. So, if we wanted to work out the correlation between the variables var1 and var2, we would do this:

```
data = nc.open_data(infile)
data.cor_space("var1", "var2")
```

1.5.14 cor_time

This method calculates the correlation coefficients between two variables in time for each grid cell. If we wanted to work out the correlation between two variables var1 and var2 we would do the following:

```
data = nc.open_data(infile)
data.cor_time("var1", "var2")
```

1.5.15 cum_sum

This method will calculate the cumulative sum, over time, for all variables. Usage is simple:

```
data = nc.open_data(infile)
data.cum_sum()
```

1.5.16 daily_max_climatology

This method will calculate the maximum value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the maximum value ever observed on each day.

```
data = nc.open_data(infile)
data.daily_max_climatology()
```

1.5.17 daily_mean_climatology

This method will calculate the mean value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the mean value ever observed on each day.

```
data = nc.open_data(infile)
data.daily_mean_climatology()
```

1.5.18 daily_min_climatology

This method will calculate the minimum value that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the minimum value ever observed on each day.

```
data = nc.open_data(infile)
data.daily_min_climatology()
```

1.5.19 daily_range_climatology

This method will calculate the value range that is observed on each day of the year over time. So, for example, if you had 100 years of daily temperature data, it will calculate the difference between the maximum and minimum observed values each day.

```
data = nc.open_data(infile)
data.daily_range_climatology()
```

1.5.20 divide

This method will divide a dataset by a constant, or the values in another dataset of NetCDF file. If we wanted to divide everything in a dataset by 2, we would do the following:

```
data = nc.open_data(infile)
data.divide(2)
```

If we want to divide a dataset by another, we can do this easily. Note that the datasets must be comparable, i.e. they must have the same grid. The second dataset must have either the same number of variables or only one variable. In the latter case everything is divided by that variable. The same holds for vertical levels.

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.divide(data2)
```

1.5.21 ensemble_max, ensemble_min, ensemble_range and ensemble_mean

These methods will calculate the ensemble statistic, when a dataset is made up of multiple files. Two methods are available. First, the statistic across all available time steps can be calculated. For this ignore_time must be set to False. For example:

```
data = nc.open_data(file_list)
data.ensemble_max(ignore_time = True)
```

The second method is to calculate the maximum value in each given time step. For example, if the ensemble was made up of 100 files where each file contains 12 months of data, ensemble_max will work out the maximum monthly value. By default ignore_time is False.

```
data = nc.open_data(file_list)
data.ensemble_max(ignore_time = False)
```

1.5.22 ensemble_percentile

This method works in the same way as ensemble_mean etc. above. However, it requires an additional term p, which is the percentile. For example, if we had to calculate the 75th ensemble percentile, we would do the following:

```
data = nc.open_data(file_list)
data = nc.ensemble_percentile(75)
```

1.5.23 invert_levels

This method will invert the vertical levels of a dataset.

```
data = nc.open_data(infile)
data.invert_levels()
```

1.5.24 mask_box

This method will set everything outside a specificied longitude/latitude box to NA. The code below illustrates how to mask the North Atlantic in the SST dataset.

```
data = nc.open_data(infile)
data.mask_box(lon = [-80, 20], lat = [40, 70])
```

1.5.25 max

This method will calculate the maximum value of all variables in all grid cells. If we wanted to calculate the maximum observed monthly sea surface temperature in the SST dataset we would do the following:

```
data = nc.open_data(infile)
data.max()
```

1.5.26 mean

This method will calculate the mean value of all variables in all grid cells. If we wanted to calculate the maximum observed monthly sea surface temperature in the SST dataset we would do the following:

```
data = nc.open_data(infile)
data.mean()
```

1.5.27 merge and merge_time

nctoolkit offers two methods for merging the files within a multi-file dataset. These methods operate in a similar way to column based joining and row-based binding in dataframes.

The merge method is suitable for merging files that have different variables, but the same time steps. The merge_time method is suitable for merging files that have the same variables, but have different time steps.

Usage for merge_time is as simple as:

```
data = nc.open_data(file_list)
data.merge_time()
```

Merging NetCDF files with different variables is potentially risky, as it is possible you can merge files that have the same number of time steps but have different times. nctoolkit's merge method therefore offers some security against a major error when merging. It requires a match argument to be supplied. This ensures that the times in each file is comparable to the others. By default match = ["year", "month", "day"], i.e. it checks if the times in each file all have the same year, month and day. The match argument must be some subset of ["year", "month", "day"]. For example, if you wanted to only make sure the files had the same year, you would do the following:

```
data = nc.open_data(file_list)
data.merge(match = ["year", "month", "day"])
```

1.5.28 max

This method will calculate the maximum value of all variables in all grid cells. If we wanted to calculate the maximum observed monthly sea surface temperature in the SST dataset we would do the following:

```
data = nc.open_data(infile)
data.max()
```

1.5.29 mean

This method will calculate the mean value of all variables in all grid cells. If we wanted to calculate the mean observed monthly sea surface temperature in the SST dataset we would do the following:

```
data = nc.open_data(infile)
data.mean()
```

1.5.30 monthly_anomaly

This method will calculate the monthly anomaly compared with the mean value for a baseline period. For example, if we wanted the monthly anomaly compared with the mean for 1990-1999 we would do the below.

```
data = nc.open_data(infile)
data.monthly_anomaly(baseline = [1990, 1999])
```

1.5.31 monthly_max

This method will calculate the maximum value in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the mean value in each month across all available years, use monthly_max_climatology. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_max()
```

1.5.32 monthly_max_climatology

```
This method will calculate, for each month, the maximum value of each variable over␣
↪all time steps.
```

```
data = nc.open_data(infile)
data.monthly_max_climatology()
```

1.5.33 monthly_mean

This method will calculate the mean value of each variable in each month of a dataset. Note that this is calculated for each year. See monthly_mean_climatology if you want to calculate a climatological monthly mean.

```
data = nc.open_data(infile)
data.monthly_mean()
```

1.5.34 monthly_mean_climatology

This method will calculate, for each month, the maximum value of each variable over all time steps. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_mean_climatology()
```

1.5.35 monthly_min

This method will calculate the minimum value in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the mean value in each month across all available years, use monthly_max_climatology. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_min()
```

1.5.36 monthly_min_climatology

This method will calculate, for each month, the minimum value of each variable over all time steps. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_min_climatology()
```

1.5.37 monthly_range

This method will calculate the value range in the month of each year of a dataset. This is useful for daily time series. If you want to calculate the value range in each month across all available years, use monthly_range_climatology. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_range()
```

1.5.38 monthly_range_climatology

This method will calculate, for each month, the value range of each variable over all time steps. Usage is simple:

```
data = nc.open_data(infile)
data.monthly_range_climatology()
```

1.5.39 multiply

This method will multiply a dataset by a constant, another dataset or a NetCDF file. If multiplied by a dataset or NetCDF file, the dataset must have the same grid and can only have one variable.

If you want to multiply a dataset by 2, you can do the following:

```
data = nc.open_data(infile)
data.multiply(2)
```

If you wanted to multiply a dataset data1 by another, data2, you can do the following:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.multiply(data2)
```

1.5.40 mutate

This method can be used to generate new variables using arithmetic expressions. New variables are added to the dataset. The method requires a dictionary, where the key-value pairs are the new variables and expression required to generate it.

For example, if had a temperature dataset, with temperature in Celsius, we might want to convert that to Kelvin. We can do this easily:

```
data = nc.open_data(infile)
data.mutate({"temperature_k":"temperature+273.15"})
```

1.5.41 percentile

This method will calculate a given percentile for each variable and grid cell. This will calculate the percentile using all available timesteps.

We can calculate the 75th percentile of sea surface temperature as follows:

```
data = nc.open_data(infile)
data.percentile(75)
```

1.5.42 phenology

A number of phenological indices can be calculated. These are based on the plankton metrics listed by Ji et al. 2010. These methods require datasets or the files within a dataset to only be made up of individual years, and ideally every day of year is available. At present this method can only calculate the phenology metric for a single variable.

The available metrics are: peak - the time of year when the maximum value of a variable occurs. middle - the time of year when 50% of the annual cumulative sum of a variable is first exceeded start - the time of year when a lower threshold (which must be defined) of the annual cumulative sum of a variable is first exceeded end - the time of year when an upper threshold (which must be defined) of the annual cumulative sum of a variable is first exceeded

For example, if you wanted to calculate timing of the peak, you set metric to "peak", and define the variable to be analyzed:

```
data = nc.open_data(infile)
data.phenology(metric = "peak", var = "var_chosen")
```

1.5.43 plot

This method will plot the contents of a dataset. It will either show a map or a time series, depending on the data type. While it should work on at least 90% of NetCDF data, there are some data types that remain incompatible, but will be added to nctoolkit over time. Usage is simple:

```
data = nc.open_data(infile)
data.plot()
```

1.5.44 range

This method calculates the range for all variables in each grid cell across all steps.

We can calculate the range of sea surface temperatures in the SST dataset as follows:

```
data = nc.open_data(infile)
data.range()
```

1.5.45 regrid

This method will remap a dataset to a new grid. This grid must be either a pandas data frame, a NetCDF file or a single file nctoolkit dataset.

For example, if we wanted to regrid a dataset to a single location, we could do the following:

```
import pandas as pd
data = nc.open_data(infile)
grid = pd.DataFrame({"lon":[-20], "lat":[50]})
data.regrid(grid, method = "nn")
```

If we wanted to regrid one dataset, dataset1, to the grid of another, dataset2, using bilinear interpolation, we would do the following:

```
data1 = nc.open_data(infile1)
data2 = nc.open_data(infile2)
data1.regrid(data2, method = "bil")
```

1.5.46 remove_variables

This method will remove variables from a dataset. Usage is simple, with the method only requiring either a str of a single variable or a list of variables to remove:

```
data = nc.open_data(infile)
data.remove_variables(vars)
```

1.5.47 rename

This method allows you to rename variables. It requires a dictionary, with key-value pairs representing the old variable names and new variables. For example, if we wanted to rename a variable old to new, we would do the following:

```
data = nc.open_data(infile)
data.rename({"old":"new"})
```

1.5.48 rolling_max

This method will calculate the rolling maximum over a specifiied window. For example, if you needed to calculate the rolling maximum with a window of 10, you would do the following:

```
data = nc.open_data(infile)
data.rolling_max(window = 10)
```

1.5.49 rolling_mean

This method will calculate the rolling mean over a specifiied window. For example, if you needed to calculate the rolling mean with a window of 10, you would do the following:

```
data = nc.open_data(infile)
data.rolling_mean(window = 10)
```

1.5.50 rolling_min

This method will calculate the rolling minimum over a specifiied window. For example, if you needed to calculate the rolling minimum with a window of 10, you would do the following:

```
data = nc.open_data(infile)
data.rolling_min(window = 10)
```

1.5.51 rolling_range

This method will calculate the rolling range over a specifiied window. For example, if you needed to calculate the rolling range with a window of 10, you would do the following:

```
data = nc.open_data(infile)
data.rolling_range(window = 10)
```

1.5.52 rolling_sum

This method will calculate the rolling sum over a specifiied window. For example, if you needed to calculate the rolling sum with a window of 10, you would do the following:

```
data = nc.open_data(infile)
data.rolling_sum(window = 10)
```

1.5.53 run

This method will evaluate all of a dataset's unevaluated commands. Usage is simple:

```
nc.options(lazy = True)
data = nc.open_data(infile)
data.select_years(1990)
data.run()
```

1.5.54 seasonal_max

This method will calculate the maximum value observed in each season. Note this is worked out for the seasons of each year. See seasonal_max_climatology for climatological seasonal maximums.

```
data.seasonal_max()
```

1.5.55 seasonal_max_climatology

This method calculates the maximum value observed in each season across all years. Usage is simple:

```
data = nc.open_data(infile)
data.seasonal_max_climatology()
```

1.5.56 seasonal_mean

This method will calculate the mean value observed in each season. Note this is worked out for the seasons of each year. See seasonal_mean_climatology for climatological seasonal means.

```
data = nc.open_data(infile)
data.seasonal_mean()
```

1.5.57 seasonal_mean_climatology

This method calculates the mean value observed in each season across all years. Usage is simple:

```
data = nc.open_data(infile)
data.seasonal_mean_climatology()
```

1.5.58 seasonal_min

This method will calculate the minimum value observed in each season. Note this is worked out for the seasons of each year. See seasonal_min_climatology for climatological seasonal minimums.

```
data = nc.open_data(infile)
data.seasonal_min()
```

1.5.59 seasonal_min_climatology

This method calculates the minimum value observed in each season across all years. Usage is simple:

```
data = nc.open_data(infile)
data.seasonal_min_climatology()
```

1.5.60 seasonal_range

This method will calculate the value range observed in each season. Note this is worked out for the seasons of each year. See seasonal_range_climatology for climatological seasonal ranges.

```
data = nc.open_data(infile)
data.seasonal_range()
```

1.5.61 seasonal_range_climatology

This method calculates the value range observed in each season across all years. Usage is simple:

```
data = nc.open_data(infile)
data.seasonal_range_climatology()
```

1.5.62 select_months

This method allows you to subset a dataset to specific months. This can either be a single month, a list of months or a range. For example, if we wanted the first half of a year, we would do the following:

```
data = nc.open_data(infile)
data.select_months(range(1, 7))
```

1.5.63 select_variables

This method allows you to subset a dataset to specific variables. This either accepts a single variable or a list of variables. For example, if you wanted two variables, var1 and var2, you would do the following:

```
data = nc.open(infile)
data.select_variables(["var1", "var2"])
```

1.5.64 select_years

This method subsets datasets to specified years. It will accept either a single year, a list of years, or a range. For example, if you wanted to subset a dataset the 1990s, you would do the following:

```
data = nc.open_data(infile)
data.select_years(range(1990, 2000))
```

1.5.65 set_missing

This method allows you to set a range to missing values. It either accepts a single variable or two variables, specifying the range to be set to missing values. For example, if you wanted all values between 0 and 10 to be set to missing, you would do the following:

```
data = nc.open_data(infile)
data.set_missing([0, 10])
```

1.5.66 spatial_max

This method will calculate the maximum value observed in space for each variable and time step. Usage is simple:

```
data = nc.open_data(infile)
data.spatial_max()
```

1.5.67 spatial_mean

This method will calculate the spatial mean for each variable and time step. If the grid cell area can be calculated, this will be an area weighted mean. Usage is simple:

```
data = nc.open_data(infile)
data.spatial_mean()
```

1.5.68 spatial_min

This method will calculate the minimum observed in space for each variable and time step. Usage is simple:

```
data = nc.open_data(infile)
data.spatial_min()
```

1.5.69 spatial_percentile

This method will calculate the percentile of variable across space for time step. For example, if you wanted to calculate the 75th percentile, you would do the following:

```
data = nc.open_data(infile)
data.spatial_percentile(p=75)
```

1.5.70 spatial_range

This method will calculate the value range observed in space for each variable and time step. Usage is simple:

```
data = nc.open_data(infile)
data.spatial_range()
```

1.5.71 spatial_sum

This method will calculate the spatial sum for each variable and time step. In some cases, for example when variables are concentrations, it makes more sense to multiply the value in each grid cell by the grid cell area, when doing a spatial sum. This method therefore has an argument by_area which defines whether to multiply the variable value by the area when doing the sum. By default by_area is False.

Usage is simple:

```
data = nc.open_data(infile)
data.spatial_sum()
```

1.5.72 split

Except for methods that begin with merge or ensemble, all nctoolkit methods operate on individual files within a dataset. There are therefore cases when you might want to be able to split a dataset into separate files for analysis. This can be done using split, which let's you split a file into separate years, months or year/month combinations. For example, if you want to split a dataset into files of different years, you can do this:

```
data = nc.open_data(infile)
data.split("year")
```

1.5.73 subtract

This method can subtract from a dataset. You can substract a constant, another dataset or a NetCDF file. In the case of datasets or NetCDF files the grids etc. must be of the same structure as the original dataset.

For example, if we had a temperature dataset where temperature was in Kelvin, we could convert it to Celsiu by subtracting 273.15.

```
data = nc.open_data(infile)
data.substract(273.15)
```

1.5.74 sum

This method will calculate the sum of values of all variables in all grid cells. Usage is simple:

```
data = nc.open_data(infile)
data.sum()
```

1.5.75 surface

This method will extract the surface level from a multi-level dataset. Usage is simple:

```
data = nc.open_data(infile)
data.surface()
```

1.5.76 to_dataframe

This method will return a pandas dataframe with the contents of the dataset. This has a decode_times argument to specify whether you want the times to be decoded. Defaults to True. Usage is simple:

```
data = nc.open_data(infile)
data.to_dataframe()
```

1.5.77 to_latlon

This method will regrid a dataset to a regular latlon grid. The minimum and maximum longitudes and latitudes must be specified, along with the horizontal and vertical resolutions.

```
data = nc.open_data(infile)
data.to_latlon(lon = [-80, 20], lat = [30, 80], res = [1,1])
```

1.5.78 to_xarray

This method will return an xarray datasetwith the contents of the dataset. This has a decode_times argument to specify whether you want the times to be decoded. Defaults to True. Usage is simple:

```
data = nc.open_data(infile)
data.to_xarray()
```

1.5.79 transmute

This method can be used to generate new variables using arithmetic expressions. Existing will be removed from the dataset. See mutate if you want to keep existing variables. The method requires a dictionary, where the key-value pairs are the new variables and expression required to generate it.

For example, if had a temperature dataset, with temperature in Celsius, we might want to convert that to Kelvin. We can do this easily:

```
data = nc.open_data(infile)
data.transmute({"temperature_k":"temperature+273.15"})
```

1.5.80 var

This method calculates the variance of each variable in the dataset. This is calculate across all time steps. Usage is simple:

```
data = nc.open_data(infile)
data.var()
```

1.5.81 vertical_interp

This method interpolates variables vertically. It requires a list of vertical levels, for example depths, you want to interpolate. For example, if you had an ocean dataset and you wanted to interpolate to 10 and 20 metres you would do the following:

```
data = nc.open_data(infile)
data.vertical_interp(levels = [10, 20])
```

1.5.82 vertical_max

This method calculates the maximum value of each variable across all vertical levels. Usage is simple:

```
data = nc.open_data(infile)
data.vertical_max()
```

1.5.83 vertical_mean

This method calculates the mean value of each variable across all vertical levels. Usage is simple:

```
data = nc.open_data(infile)
data.vertical_mean()
```

1.5.84 vertical_min

This method calculates the minimum value of each variable across all vertical levels. Usage is simple:

```
data = nc.open_data(infile)
data.vertical_min()
```

1.5.85 vertical_range

This method calculates the value range of each variable across all vertical levels. Usage is simple:

```
data = nc.open_data(infile)
data.vertical_range()
```

1.5.86 vertical_sum

This method calculates the sum each variable across all vertical levels. Usage is simple:

```
data = nc.open_data(infile)
data.vertical_sum()
```

1.5.87 write_nc

This method allows you to write the contents of a dataset to a NetCDF file. If the target file exists and you want to overwrite it set overwrite to True. Usage is simple:

```
data.write_nc(outfile)
```

1.5.88 zip

This method will zip the contents of a dataset. This is mostly useful for processing chains where you want to minimize disk space usage by the output. Please note this method works lazily. In the code below only one file is generated, a zipped "outfile".

```
nc.options(lazy = True)
data = nc.open_data(infile)
data.select_years(1990)
data.zip()
data.write_nc(outfile)
```

1.6 How to guide

This guide will show how to carry out key nctoolkit operations. We will use a sea surface temperature data set and a depth-resolved ocean temperature data set. The data set can be downloaded from here.

[1]:
```
import nctoolkit as nc
import os
import pandas as pd
import xarray as xr
```

1.6.1 How to select years and months

If we want to select specific years and months we can use the select_years and select_months method

[2]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_years(1960)
sst.select_months(1)
sst.times()
```

[2]:
```
['1960-01-01T00:00:00']
```

1.6.2 How to copy a data set

If you want to make a deep copy of a data set, use the built in copy method. This method will return a new data set. This method should be used because of nctoolkit's built in methods to automatically delete temporary files that are no longer required. Behind the scenes, using copy will result in nctoolkit registering that it needs the NetCDF file for both the original dataset and the new copied one. So if you copy a dataset, and then delete the original, nctoolkit knows to not remove any NetCDF files related to the dataset.

[3]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_years(1960)
sst.select_months(1)
sst1 = sst.copy()
del sst
os.path.exists(sst1.current)
```

[3]:
```
True
```

1.6.3 How to clip to a region

If you want to clip the data to a specific longitude and latitude box, we can use clip, with the longitude and latitude range given by lon and lat.

[4]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_months(1)
sst.select_years(1980)
sst.clip(lon = [-80, 20], lat = [40, 70])
sst.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed:

[4]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

1.6.4 How to rename a variable

If we want to rename a variable we use the rename method, and supply a dictionary where the key-value pairs are the original and new names

[5]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.variables
```

[5]:
```
['sst']
```

The original dataset had only one variable called sst. We can now rename it, and display the new variables.

[6]:
```
sst.rename({"sst": "temperature"})
sst.variables
```

[6]:
```
['temperature']
```

1.6.5 How to create new variables

New variables can be created using arithmetic operations using either mutate or transmute. The mutate method will maintain the original variables, whereas transmute will not. This method requires a dictionary, where the key, values pairs are the names of the new variables and the arithemtic operations to perform. The example below shows how to create a new variable with

[7]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.mutate({"sst_k": "sst+273.15"})
sst.variables
```

[7]:
```
['sst', 'sst_k']
```

1.6.6 How to calculate a spatial average

You can calculate a spatial average using the spatial_mean method. There are additional methods for maximums etc.

[8]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.plot()
```

Data type cannot be displayed:

[8]:
```
:Curve   [time]   (x)
```

1.6.7 How to calculate an annual mean

You can calculate an annual mean using the annual_mean method.

[9]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_mean()
sst.plot()
```

Data type cannot be displayed:

[9]:
```
:Curve   [time]   (x)
```

1.6.8 How to calculate a rolling average

You can calculate a rolling mean using the rolling_mean method, with the window argument providing the number of time steps to average over. There are additional methods for rolling sums etc. The code below will calculate a rolling mean of global SST using a 20 year window.

[10]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_mean()
sst.rolling_mean(20)
sst.plot()
```

Data type cannot be displayed:

[10]:
```
:Curve   [time]   (x)
```

1.6.9 How to calculate temporal anomalies

You can calculate annual temporal anomalies using the anomaly_annual method. This requires a baseline period.

[11]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.spatial_mean()
sst.annual_anomaly(baseline = [1960, 1979])
sst.plot()
```

Data type cannot be displayed:

[11]:
```
:Curve   [time]   (x)
```

1.6.10 How to split data by year etc

Files within a dataset can be split by year, day, year and month or season using the split method. If we wanted to split by year, we do the following:

[12]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
sst.size
```

[12]:
```
'Number of files in ensemble: 169\nEnsemble size: 530.445201 MB\nSmallest file: /tmp/
↪nctoolkitayrhmwtbnctoolkittmp72u9tn3o.1898.nc has size 3.1387289999999997 MB\
↪nLargest file: /tmp/nctoolkitayrhmwtbnctoolkittmp72u9tn3o.1898.nc has size 3.
↪1387289999999997 MB'
```

1.6.11 How to merge files in time

We can merge files based on time using merge_time. We can do this by merging the dataset that results from splitting the original sst dataset. If we split the dataset by year, we see that there are 169 files, one for each year.

[13]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.split("year")
sst.size
```

[13]:
```
'Number of files in ensemble: 169\nEnsemble size: 530.445201 MB\nSmallest file: /tmp/
↪nctoolkitayrhmwtbnctoolkittmp58y4ytyj.1998.nc has size 3.1387289999999997 MB\
↪nLargest file: /tmp/nctoolkitayrhmwtbnctoolkittmp58y4ytyj.1998.nc has size 3.
↪1387289999999997 MB'
```

We can then merge them together to get a single file dataset:

[14]:
```
sst.merge_time()
sst.size
```

[14]:
```
'Number of files: 1\nFile size: 525.828237 MB'
```

1.6.12 How to do variables based merging

If we have two more more files that have the same time steps, but different variables, we can merge them using merge. The code below will first create a dataset with a netcdf file with sst in K, and it will then create a new dataset with this netcd file and the original, and then merge them.

[15]:
```
sst1 = nc.open_data("sst.mon.mean.nc")
sst2 = nc.open_data("sst.mon.mean.nc")
sst2.transmute({"sst_k": "sst+273.15"})
new_sst = nc.open_data([sst1.current, sst2.current])
new_sst.current
new_sst.merge()
new_sst.variables
```

[15]:
```
['sst.mon.mean.nc', '/tmp/nctoolkitayrhmwtbnctoolkittmp8vgtaa28.nc']
```

[15]:
```
['sst', 'sst_k']
```

In some cases we will have two or more datasets we want to merge. In this case we can use the merge function as follows:

[16]:
```
sst1 = nc.open_data("sst.mon.mean.nc")
sst2 = nc.open_data("sst.mon.mean.nc")
sst2.transmute({"sst_k": "sst+273.15"})
new_sst = nc.merge(sst1, sst2)
new_sst.variables
```

[16]:
```
['sst', 'sst_k']
```

1.6.13 How to horizontally regrid data

Variables can be regridded horizontally using regrid. This method requires the new grid to be defined. This can either be a pandas data frame, with lon/lat as columns, an xarray object, a netcdfile or a dataset. I will demonstrate all three methods by regridding SST to the North Atlantic. Let's begin by getting a grid for the North Atlantic.

[17]:
```
new_grid = nc.open_data("sst.mon.mean.nc")
new_grid.clip(lon = [-80, 20], lat = [30, 70])
new_grid.select_months(1)
new_grid.select_years( 2000)
```

First, we will use the new dataset itself to do the regridding. I will calculate mean SST using the original data, and then regrid to the North Atlantic.

[18]:
```
%%time
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = new_grid)
sst.plot()
```

```
CPU times: user 56.4 ms, sys: 94.2 ms, total: 151 ms
Wall time: 1.38 s
```

Data type cannot be displayed:

[18]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

We can also do this using the netcdf, which is new_grid.current

[19]:
```
%%time
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = new_grid.current)
sst.plot()
```

```
CPU times: user 60.1 ms, sys: 38.8 ms, total: 99 ms
Wall time: 1.48 s
```

Data type cannot be displayed:

[19]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

In a similar way we can read the new_grid in as an xarray data set.

[20]:
```
%%time
na_grid = xr.open_dataset(new_grid.current)
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = na_grid)
sst.plot()
```

```
CPU times: user 72.7 ms, sys: 44.4 ms, total: 117 ms
Wall time: 1.49 s
```

Data type cannot be displayed:

[20]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

or we can use a pandas data frame. In this case I will convert the xarray data set to a data frame.

[21]:
```
%%time
na_grid = xr.open_dataset(new_grid.current)
na_grid = na_grid.to_dataframe().reset_index().loc[:,["lon", "lat"]]
sst = nc.open_data("sst.mon.mean.nc")
sst.mean()
sst.regrid(grid = na_grid)
sst.plot()
```

```
CPU times: user 72.6 ms, sys: 39 ms, total: 112 ms
Wall time: 1.46 s
```

Data type cannot be displayed:

[21]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

1.6.14 How to temporally interpolate

Temporal interpolation can be carried out using time_interp. This method requires a start date (start) of the format YYYY/MM/DD and an end date (end), and a temporal resolution (resolution), which is either 1 day ("daily"), 1 week ("weekly"), 1 month ("monthly"), or 1 year ("yearly").

[22]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.time_interp(start = "1990/01/01", end = "1990/12/31", resolution = "daily")
```

1.6.15 How to calculate a monthly average from daily data

If you have daily data, you can calculate a month average using monthly_mean. There are also methods for maximums etc.

[23]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.time_interp(start = "1990/01/01", end = "1990/12/31", resolution = "daily")
sst.monthly_mean()
```

1.6.16 How to calculate a monthly climatology

CDO outputs the date of the final month.

[24]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.select_years(list(range(1990, 2000)))
sst.monthly_mean_climatology()
sst.select_months(1)
sst.plot()
```

Data type cannot be displayed:

[24]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

1.6.17 How to calculate a seasonal climatology

[25]:
```
sst = nc.open_data("sst.mon.mean.nc")
sst.seasonal_mean_climatology()
sst.select_timestep(0)
sst.plot()
```

Data type cannot be displayed:

[25]:
```
:DynamicMap   [time]
   :Image   [lon,lat]   (sst)
```

1.7 API Reference

1.7.1 Session options

options(**kwargs)	Define session options.

nctoolkit.options

nctoolkit.**options**(***kwargs*)

Define session options. Set the options in the session. Available options are thread_safe and lazy. Set thread_safe = True if hdf5 was built to be thread safe. Set lazy = True if you want methods to evaluate lazy by default.

Parameters ****kwargs** – Define options using key, value pairs.

1.7.2 Reading/copying data

open_data([x, suppress_messages, checks])	Read netcdf data as a DataSet object
DataSet.copy(self)	Make a deep copy of an DataSet object

nctoolkit.open_data

nctoolkit.**open_data**(*x=None*, *suppress_messages=False*, *checks=False*)

Read netcdf data as a DataSet object

Parameters

- **x** (*str or list*) – A string or list of netcdf files. The function will check the files exist. If x is not a list, but an iterable it will be converted to a list
- **checks** (*boolean*) – Do you want basic checks to ensure cdo can read files?

nctoolkit.DataSet.copy

DataSet.**copy**(*self*)

Make a deep copy of an DataSet object

1.7.3 Merging or analyzing multiple datasets

merge(*datasets[, match])	Merge datasets
cor_time([x, y])	Calculate the temporal correlation coefficient between two datasets This will calculate the temporal correlation coefficient, in each grid cell, between two datasets
cor_space([x, y])	Calculate the spatial correlation coefficient between two datasets This will calculate the spatial correlation coefficient, for each time step, between two datasets

nctoolkit.merge

nctoolkit.**merge**(**datasets, match=['day', 'year', 'month']*)

Merge datasets

Parameters

- **datasets** (*kwargs*) – Datasets to merge.
- **match** (*list*) – Temporal matching criteria. This is a list which must be made up of a subset of day, year, month. This checks that the datasets have compatible times. For example, if you want to ensure the datasets have the same years, then use match = ["year"].

nctoolkit.cor_time

nctoolkit.**cor_time**(*x=None*, *y=None*)

Calculate the temporal correlation coefficient between two datasets This will calculate the temporal correlation coefficient, in each grid cell, between two datasets

Parameters

- **x** (*dataset*) – First dataset to use
- **y** (*dataset*) – Second dataset to use

nctoolkit.cor_space

nctoolkit.**cor_space**(*x=None*, *y=None*)

Calculate the spatial correlation coefficient between two datasets This will calculate the spatial correlation coefficient, for each time step, between two datasets

Parameters

- **x** (*dataset*) – First dataset to use
- **y** (*dataset*) – Second dataset to use

1.7.4 Accessing attributes

DataSet.variables	List variables contained in a dataset
DataSet.years	List years contained in a dataset
DataSet.months	List months contained in a dataset
DataSet.times	List times contained in a dataset
DataSet.levels	List levels contained in a dataset
DataSet.size	The size of an object This will print the number of files, total size, and smallest and largest files in an DataSet object.
DataSet.current	The current file or files in the DataSet object
DataSet.history	The history of operations on the DataSet
DataSet.start	The starting file or files of the DataSet object

nctoolkit.DataSet.variables

property DataSet.**variables**

List variables contained in a dataset

nctoolkit.DataSet.years

property DataSet.**years**
List years contained in a dataset

nctoolkit.DataSet.months

property DataSet.**months**
List months contained in a dataset

nctoolkit.DataSet.times

property DataSet.**times**
List times contained in a dataset

nctoolkit.DataSet.levels

property DataSet.**levels**
List levels contained in a dataset

nctoolkit.DataSet.size

property DataSet.**size**
The size of an object This will print the number of files, total size, and smallest and largest files in an DataSet object.

nctoolkit.DataSet.current

property DataSet.**current**
The current file or files in the DataSet object

nctoolkit.DataSet.history

property DataSet.**history**
The history of operations on the DataSet

nctoolkit.DataSet.start

property DataSet.**start**
The starting file or files of the DataSet object

1.7.5 Plotting

DataSet.plot(self[, log, vars, panel])	Autoplotting method.
DataSet.view(self)	Open the current dataset's file in ncview

nctoolkit.DataSet.plot

DataSet.**plot**(*self*, *log=False*, *vars=None*, *panel=False*)

Autoplotting method. Automatically plot a dataset.

Parameters

- **log** (*boolean*) – Do you want a plotted data to be logged?
- **vars** (*str or list*) – A string or list of the variables to plot
- **panel** (*boolean*) – Do you want a panel plot, if avaiable?

nctoolkit.DataSet.view

DataSet.**view**(*self*)

Open the current dataset's file in ncview

1.7.6 Variable modification

DataSet.mutate(self[, operations])	Create new variables using mathematical expressions, and keep original variables
DataSet.transmute(self[, operations])	Create new variables using mathematical expressions, and drop original variables
DataSet.rename(self, newnames)	Rename variables in a dataset
DataSet.set_missing(self[, value])	Set the missing value for a single number or a range
DataSet.sum_all(self[, drop])	Calculate the sum of all variables for each time step

nctoolkit.DataSet.mutate

DataSet.**mutate**(*self*, *operations=None*)

Create new variables using mathematical expressions, and keep original variables

Parameters **operations** (*dict*) – operations to apply. The keys are the new variables to generate. The values are the mathematical operations to carry out. Both must be strings.

nctoolkit.DataSet.transmute

DataSet.**transmute**(*self*, *operations=None*)

Create new variables using mathematical expressions, and drop original variables

Parameters **operations** (*dict*) – operations to apply. The keys are the new variables to generate. The values are the mathematical operations to carry out. Both must be strings.

nctoolkit.DataSet.rename

DataSet.**rename**(*self*, *newnames*)

Rename variables in a dataset

Parameters **newnames** (*dict*) – Dictionary with key-value pairs being original and new variable names

nctoolkit.DataSet.set_missing

DataSet.**set_missing**(*self*, *value=None*)

Set the missing value for a single number or a range

Parameters **value** (*2 variable list or int/float*) – If int/float is provided, the missing value will be set to that. If a list is provided, values between the two values (inclusive) of the list are set to missing.

nctoolkit.DataSet.sum_all

DataSet.**sum_all**(*self*, *drop=True*)

Calculate the sum of all variables for each time step

Parameters **drop** (*boolean*) – Do you want to keep variables?

1.7.7 NetCDF file attribute modification

DataSet.set_longnames(self[, name_dict])	Set the long names of variables
DataSet.set_units(self[, unit_dict])	Set the units for variables

nctoolkit.DataSet.set_longnames

DataSet.**set_longnames**(*self*, *name_dict=None*)

Set the long names of variables

Parameters **name_dict** (*dict*) – Dictionary with key, value pairs representing the variable names and their long names

nctoolkit.DataSet.set_units

DataSet.**set_units**(*self*, *unit_dict=None*)
Set the units for variables

Parameters **unit_dict** (*dict*) – A dictionary where the key-value pairs are the variables and new units respectively.

1.7.8 Vertical/level methods

DataSet.surface(self)	Extract the top/surface level from a dataset This extracts the first vertical level from each file in a dataset.
DataSet.bottom(self)	Extract the bottom level from a dataset This extracts the bottom level from each NetCDF file.
DataSet.vertical_interp(self[, levels])	Verticaly interpolate a dataset based on given vertical levels This is calculated for each time step and grid cell
DataSet.vertical_mean(self)	Calculate the depth-averaged mean for each variable This is calculated for each time step and grid cell
DataSet.vertical_min(self)	Calculate the vertical minimum of variable values This is calculated for each time step and grid cell
DataSet.vertical_max(self)	Calculate the vertical maximum of variable values This is calculated for each time step and grid cell
DataSet.vertical_range(self)	Calculate the vertical range of variable values This is calculated for each time step and grid cell
DataSet.vertical_sum(self)	Calculate the vertical sum of variable values This is calculated for each time step and grid cell
DataSet.vertical_cum_sum(self)	Calculate the vertical sum of variable values This is calculated for each time step and grid cell
DataSet.invert_levels(self)	Invert the levels of 3D variables This is calculated for each time step and grid cell
DataSet.bottom_mask(self)	Create a mask identifying the deepest cell without missing values.

nctoolkit.DataSet.surface

DataSet.**surface**(*self*)
Extract the top/surface level from a dataset This extracts the first vertical level from each file in a dataset.

nctoolkit.DataSet.bottom

DataSet.**bottom**(*self*)

Extract the bottom level from a dataset This extracts the bottom level from each NetCDF file. Please note that for ensembles, it uses the first file to derive the index of the bottom level. Use bottom_mask for files when the bottom cell in NetCDF files do not represent the actual bottom.

nctoolkit.DataSet.vertical_interp

DataSet.**vertical_interp**(*self*, *levels=None*)

Verticaly interpolate a dataset based on given vertical levels This is calculated for each time step and grid cell

Parameters **levels** (*list, int or str*) – list of vertical levels, for example depths for an ocean model, to vertically interpolate to. These must be floats or ints.

nctoolkit.DataSet.vertical_mean

DataSet.**vertical_mean**(*self*)

Calculate the depth-averaged mean for each variable This is calculated for each time step and grid cell

nctoolkit.DataSet.vertical_min

DataSet.**vertical_min**(*self*)

Calculate the vertical minimum of variable values This is calculated for each time step and grid cell

nctoolkit.DataSet.vertical_max

DataSet.**vertical_max**(*self*)

Calculate the vertical maximum of variable values This is calculated for each time step and grid cell

nctoolkit.DataSet.vertical_range

DataSet.**vertical_range**(*self*)

Calculate the vertical range of variable values This is calculated for each time step and grid cell

nctoolkit.DataSet.vertical_sum

DataSet.**vertical_sum**(*self*)

Calculate the vertical sum of variable values This is calculated for each time step and grid cell

nctoolkit.DataSet.vertical_cum_sum

DataSet.**vertical_cum_sum**(*self*)
Calculate the vertical sum of variable values This is calculated for each time step and grid cell

nctoolkit.DataSet.invert_levels

DataSet.**invert_levels**(*self*)
Invert the levels of 3D variables This is calculated for each time step and grid cell

nctoolkit.DataSet.bottom_mask

DataSet.**bottom_mask**(*self*)
Create a mask identifying the deepest cell without missing values. This converts a dataset to a mask identifying which cell represents the bottom, for example the seabed. 1 identifies the deepest cell with non-missing values. Everything else is 0, or missing. At present this method only uses the first available variable from netcdf files, so it may not be suitable for all data

1.7.9 Rolling methods

DataSet.rolling_mean(self[, window])	Calculate a rolling mean based on a window
DataSet.rolling_min(self[, window])	Calculate a rolling minimum based on a window
DataSet.rolling_max(self[, window])	Calculate a rolling maximum based on a window
DataSet.rolling_sum(self[, window])	Calculate a rolling sum based on a window
DataSet.rolling_range(self[, window])	Calculate a rolling range based on a window

nctoolkit.DataSet.rolling_mean

DataSet.**rolling_mean**(*self*, *window=None*)
Calculate a rolling mean based on a window

Parameters = int (*window*) – The size of the window for the calculation of the rolling mean

nctoolkit.DataSet.rolling_min

DataSet.**rolling_min**(*self*, *window=None*)
Calculate a rolling minimum based on a window

Parameters = int (*window*) – The size of the window for the calculation of the rolling minimum

nctoolkit.DataSet.rolling_max

DataSet.**rolling_max**(*self*, *window=None*)
Calculate a rolling maximum based on a window

Parameters = int (*window*) – The size of the window for the calculation of the rolling maximum

nctoolkit.DataSet.rolling_sum

DataSet.**rolling_sum**(*self*, *window=None*)
Calculate a rolling sum based on a window

Parameters = int (*window*) – The size of the window for the calculation of the rolling sum

nctoolkit.DataSet.rolling_range

DataSet.**rolling_range**(*self*, *window=None*)
Calculate a rolling range based on a window

Parameters = int (*window*) – The size of the window for the calculation of the rolling range

1.7.10 Evaluation setting

DataSet.run(self)	Run all stored commands in a dataset

nctoolkit.DataSet.run

DataSet.**run**(*self*)
Run all stored commands in a dataset

1.7.11 Cleaning functions

—

1.7.12 Ensemble creation

create_ensemble([path, var, recursive])	Generate an ensemble

nctoolkit.create_ensemble

nctoolkit.**create_ensemble**(*path=''*, *var=None*, *recursive=True*)

Generate an ensemble

Parameters

- **path** (*str*) – The system to search for netcdf files
- **recursive** (*boolean*) – True/False depending on whether you want to search the path recursively. Defaults to True.

Returns A list of files

Return type list

1.7.13 Arithemetic methods

DataSet.mutate(self[, operations])	Create new variables using mathematical expressions, and keep original variables
DataSet.transmute(self[, operations])	Create new variables using mathematical expressions, and drop original variables
DataSet.add(self[, x, var])	Add to a dataset This will add a constant, another dataset or a NetCDF file to the dataset.
DataSet.subtract(self[, x, var])	Subtract from a dataset This will subtract a constant, another dataset or a NetCDF file from the dataset.
DataSet.multiply(self[, x, var])	Multiply a dataset This will multiply a dataset by a constant, another dataset or a NetCDF file.
DataSet.divide(self[, x, var])	Divide the data This will divide the dataset by a constant, another dataset or a NetCDF file.

nctoolkit.DataSet.add

DataSet.**add**(*self*, *x=None*, *var=None*)

Add to a dataset This will add a constant, another dataset or a NetCDF file to the dataset. :param x: An int, float, single file dataset or netcdf file to add to the dataset. If a dataset or netcdf file is supplied, this must have only one variable, unless var is provided. The grids must be the same. :type x: int, float, DataSet or netcdf file :param var: A variable in the x to use for the operation :type var: str

nctoolkit.DataSet.subtract

DataSet.**subtract**(*self*, *x=None*, *var=None*)

Subtract from a dataset This will subtract a constant, another dataset or a NetCDF file from the dataset. :param x: An int, float, single file dataset or netcdf file to subtract from the dataset. If a dataset or netcdf is supplied this must only have one variable, unless var is provided. The grids must be the same. :type x: int, float, DataSet or netcdf file :param var: A variable in the x to use for the operation :type var: str

nctoolkit.DataSet.multiply

DataSet.**multiply**(*self*, *x=None*, *var=None*)

Multiply a dataset This will multiply a dataset by a constant, another dataset or a NetCDF file. :param x: An int, float, single file dataset or netcdf file to multiply the dataset by. If multiplying by a dataset or single file there must only be a single variable in it, unless var is supplied. The grids must be the same. :type x: int, float, DataSet or netcdf file :param var: A variable in the x to multiply the dataset by :type var: str

nctoolkit.DataSet.divide

DataSet.**divide**(*self*, *x=None*, *var=None*)

Divide the data This will divide the dataset by a constant, another dataset or a NetCDF file. :param x: An int, float, single file dataset or netcdf file to divide the dataset by. If a dataset or netcdf file is supplied, this must have only one variable, unless var is provided. The grids must be the same. :type x: int, float, DataSet or netcdf file :param var: A variable in the x to use for the operation :type var: str

1.7.14 Ensemble statistics

DataSet.ensemble_mean(self[, nco, ignore_time])	Calculate an ensemble mean
DataSet.ensemble_min(self[, nco, ignore_time])	Calculate an ensemble min
DataSet.ensemble_max(self[, nco, ignore_time])	Calculate an ensemble maximum
DataSet.ensemble_percentile(self[, p])	Calculate an ensemble percentile This will calculate the percentles for each time step in the files.
DataSet.ensemble_range(self)	Calculate an ensemble range The range is calculated for each time step; for example, if each file in the ensemble has 12 months of data the statistic will be calculated for each month.

nctoolkit.DataSet.ensemble_mean

DataSet.**ensemble_mean**(*self*, *nco=False*, *ignore_time=False*)

Calculate an ensemble mean

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the mean is calculated over all time steps. If False, the ensemble mean is calculated for each time steps; for example, if the ensemble is made up of monthly files the mean for each month will be calculated.

nctoolkit.DataSet.ensemble_min

DataSet.**ensemble_min**(*self*, *nco=False*, *ignore_time=False*)
Calculate an ensemble min

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the min is calculated over all time steps. If False, the ensemble min is calculated for each time steps; for example, if the ensemble is made up of monthly files the min for each month will be calculated.

nctoolkit.DataSet.ensemble_max

DataSet.**ensemble_max**(*self*, *nco=False*, *ignore_time=False*)
Calculate an ensemble maximum

Parameters

- **nco** (*boolean*) – Do you want to use NCO for the calculation? Default is False, i.e. CDO is used. Modify default if run time is an issue.
- **ignore_time** (*boolean*) – If True the max is calculated over all time steps. If False, the ensemble max is calculated for each time steps; for example, if the ensemble is made up of monthly files the max for each month will be calculated.

nctoolkit.DataSet.ensemble_percentile

DataSet.**ensemble_percentile**(*self*, *p=None*)
Calculate an ensemble percentile This will calculate the percentles for each time step in the files. For example, if you had an ensemble of files where each file included 12 months of data, it would calculate the percentile for each month.

Parameters **p** (*float or int*) – percentile to calculate. 0<=p<=100.

nctoolkit.DataSet.ensemble_range

DataSet.**ensemble_range**(*self*)
Calculate an ensemble range The range is calculated for each time step; for example, if each file in the ensemble has 12 months of data the statistic will be calculated for each month.

1.7.15 Subsetting operations

DataSet.clip(self[, lon, lat, nco])	Clip to a rectangular longitude and latitude box
DataSet.select_variables(self[, vars])	Select variables from a dataset
DataSet.remove_variables(self[, vars])	Remove variables This will remove stated variables from files in the dataset.
DataSet.select_years(self[, years])	Select years from a dataset This method will subset the dataset to only contain years within the list given.
DataSet.select_months(self[, months])	Select months from a dataset This method will subset the dataset to only contain months within the list given.

continues on next page

Table 15 – continued from previous page

DataSet.select_season(self[, season])	Select season from a dataset
DataSet.select_timestep(self[, times])	Select timesteps from a dataset

nctoolkit.DataSet.clip

DataSet.**clip**(*self*, *lon=[- 180, 180]*, *lat=[- 90, 90]*, *nco=False*)

Clip to a rectangular longitude and latitude box

Parameters

- **lon** (*list*) – The longitude range to select. This must be two variables, between -180 and 180 when nco = False.
- **lat** (*list*) – The latitude range to select. This must be two variables, between -90 and 90 when nco = False.
- **nco** (*boolean*) – Do you want this to use NCO for clipping? Defaults to False, and uses CDO. Set to True if you want to call NCO. NCO is typically better at handling very large horizontal grids.

nctoolkit.DataSet.select_variables

DataSet.**select_variables**(*self*, *vars=None*)

Select variables from a dataset

Parameters **vars** (*list or str*) – Variable(s) to select.

nctoolkit.DataSet.remove_variables

DataSet.**remove_variables**(*self*, *vars=None*)

Remove variables This will remove stated variables from files in the dataset.

Parameters **vars** (*str or list*) – Variable or variables to be removed from the dataset. Variables that are listed but not in the dataset will be ignored

nctoolkit.DataSet.select_years

DataSet.**select_years**(*self*, *years=None*)

Select years from a dataset This method will subset the dataset to only contain years within the list given. A warning message will be provided when there are missing years. :param years: Years(s) to select. These should be integers :type years: list,range or int

nctoolkit.DataSet.select_months

DataSet.**select_months**(*self*, *months=None*)

Select months from a dataset This method will subset the dataset to only contain months within the list given. A warning message will be provided when there are missing months.

Parameters **months** (*list, range or int*) – Month(s) to select.

nctoolkit.DataSet.select_season

DataSet.**select_season**(*self*, *season=None*)
Select season from a dataset

Parameters **season** (*str*) – Season to select. One of "DJF", "MAM", "JJA", "SON".

nctoolkit.DataSet.select_timestep

DataSet.**select_timestep**(*self*, *times=None*)
Select timesteps from a dataset

Parameters **times** (*list or int*) – time step(s) to select. For example, if you wanted the first time step set times=0.

1.7.16 Time-based methods

DataSet.set_date(self[, year, month, day, ...])	Set the date in a dataset You should only do this if you have to fix/change a dataset with a single, not multiple dates.
DataSet.select_months(self[, months])	Select months from a dataset This method will subset the dataset to only contain months within the list given.
DataSet.select_season(self[, season])	Select season from a dataset
DataSet.select_years(self[, years])	Select years from a dataset This method will subset the dataset to only contain years within the list given.

nctoolkit.DataSet.set_date

DataSet.**set_date**(*self*, *year=None*, *month=None*, *day=None*, *base_year=1900*)
Set the date in a dataset You should only do this if you have to fix/change a dataset with a single, not multiple dates.

Parameters

- **year** (*int*) – The year
- **month** (*int*) – The month
- **day** (*int*) – The day
- **base_year** (*int*) – The base year for time creation in the netcdf. Defaults to 1900.

1.7.17 Interpolation methods

DataSet.regrid(self[, grid, method])	Regrid a dataset to a target grid
DataSet.to_latlon(self[, lon, lat, res, method])	Regrid a dataset to a regular latlon grid
DataSet.time_interp(self[, start, end, ...])	Temporally interpolate variables based on date range and time resolution

nctoolkit.DataSet.regrid

DataSet.**regrid**(*self*, *grid=None*, *method='bil'*)
Regrid a dataset to a target grid

Parameters

- **grid** (*nctoolkit.DataSet, pandas data frame or netcdf file*) – The grid to remap to
- **method** (*str*) – Remapping method. Defaults to "bil". Methods available are: bilinear - "bil"; nearest neighbour - "nn" - "nearest neighbour"; "bic" - "bicubic interpolation";

nctoolkit.DataSet.to_latlon

DataSet.**to_latlon**(*self*, *lon=None*, *lat=None*, *res=None*, *method='bil'*)
Regrid a dataset to a regular latlon grid

Parameters

- **lon** (*list*) – 2 element list giving minimum and maximum longitude of target grid
- **lat** (*list*) – 2 element list giving minimum and maximum latitude of target grid
- **res** (*float, int or list*) – If float or int given, this will be the horizontal and vertical resolution of the target grid. If 2 element list is given, the first element is the longitudinal resolution and the second is the latitudinal resolution.
- **method** (*str*) – remapping method. Defaults to "bil". Bilinear: "bil"; Nearest neighbour: "nn",. . . .

nctoolkit.DataSet.time_interp

DataSet.**time_interp**(*self*, *start=None*, *end=None*, *resolution='monthly'*)
Temporally interpolate variables based on date range and time resolution

Parameters

- **start** (*str*) – Start date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD.
- **end** (*str*) – End date for interpolation. Needs to be of the form YYYY/MM/DD or YYYY-MM-DD. If end is not given interpolation will be to the final available time in the dataset.
- **resolution** (*str*) – Time steps used for interpolation. Needs to be "daily", "weekly", "monthly" or "yearly". Defaults to monthly.

1.7.18 Masking methods

DataSet.mask_box(self[, lon, lat])	Mask a lon/lat box

nctoolkit.DataSet.mask_box

DataSet.**mask_box**(*self*, *lon=[- 180, 180]*, *lat=[- 90, 90]*)
Mask a lon/lat box

Parameters

- **lon** (*list*) – Longitude range to mask. Must be of the form: [lon_min, lon_max]
- **lat** (*list*) – Latitude range to mask. Must be of the form: [lat_min, lat_max]

1.7.19 Summary methods

DataSet.annual_anomaly(self[, baseline, ...])	Calculate annual anomalies for each variable based on a baseline period The anomaly is derived by first calculating the climatological annual mean for the given baseline period.
DataSet.monthly_anomaly(self[, baseline])	Calculate monthly anomalies based on a baseline period The anomaly is derived by first calculating the climatological monthly mean for the given baseline period.
DataSet.phenology(self[, var, metric, p])	Calculate phenologies from a dataset Each file in an ensemble must only cover a single year, and ideally have all days.

nctoolkit.DataSet.annual_anomaly

DataSet.**annual_anomaly**(*self*, *baseline=None*, *metric='absolute'*, *window=1*)
Calculate annual anomalies for each variable based on a baseline period The anomaly is derived by first calculating the climatological annual mean for the given baseline period. Annual means are then calculated for each year and the anomaly is calculated compared with the baseline mean. This will be calculated on a per-file basis in a multi-file dataset.

Parameters

- **baseline** (*list*) – Baseline years. This needs to be the first and last year of the climatological period. Example: a baseline of [1980,1999] will result in anomalies against the 20 year climatology from 1980 to 1999.
- **metric** (*str*) – Set to "absolute" or "relative", depending on whether you want the absolute or relative anomaly to be calculated.
- **window** (*int*) – A window for the anomaly. By default window = 1, i.e. the annual anomaly is calculated. If, for example, window = 20, the 20 year rolling means will be used to calculate the anomalies.

nctoolkit.DataSet.monthly_anomaly

DataSet.**monthly_anomaly**(*self*, *baseline=None*)

Calculate monthly anomalies based on a baseline period The anomaly is derived by first calculating the climatological monthly mean for the given baseline period. Monthly means are then calculated for each year and the anomaly is calculated compared with the baseline mean. This is calculated separately for each file in a multi-file dataset.

Parameters **baseline** (*list*) – Baseline years. This needs to be the first and last year of the climatological period. Example: a baseline of [1985,2005] will result in anomolies against 20 year climatology from 1986 to 2005.

nctoolkit.DataSet.phenology

DataSet.**phenology**(*self*, *var=None*, *metric=None*, *p=None*)

Calculate phenologies from a dataset Each file in an ensemble must only cover a single year, and ideally have all days. The method assumes datasets have daily resolution.

Parameters

- **var** (*str*) – Variable to analyze.
- **metric** (*str*) – Must be peak, middle, start or end. Peak is defined as the day of the maximum value. Middle is the day when the cumulative total of the variable first exceeds the cumulative total for the entire year. Start or end is defined as the first day when the cumulative total exceeds a percentile p of the maximum cumulative total.
- **p** (*str*) – Percentile to use for start or end.

1.7.20 Statistical methods

DataSet.mean(self)	Calculate the temporal mean of all variables
DataSet.min(self)	Calculate the temporal minimum of all variables
DataSet.percentile(self[, p])	Calculate the temporal percentile of all variables
DataSet.max(self)	Calculate the temporal maximum of all variables
DataSet.sum(self)	Calculate the temporal sum of all variables
DataSet.range(self)	Calculate the temporal range of all variables
DataSet.var(self)	Calculate the temporal variance of all variables
DataSet.cum_sum(self)	Calculate the temporal cumulative sum of all variables
DataSet.cor_space(self[, var1, var2])	Calculate the correlation correct between two variables in space This is calculated for each time step.
DataSet.cor_time(self[, var1, var2])	Calculate the correlation correct in time between two variables The correlation is calculated for each grid cell, ignoring missing values.
DataSet.spatial_mean(self)	Calculate the area weighted spatial mean for all variables This is performed for each time step.
DataSet.spatial_min(self)	Calculate the spatial minimum for all variables This is performed for each time step.
DataSet.spatial_max(self)	Calculate the spatial maximum for all variables This is performed for each time step.
DataSet.spatial_percentile(self[, p])	Calculate the spatial sum for all variables This is performed for each time step.

continues on next page

Table 20 – continued from previous page

DataSet.spatial_range(self)	Calculate the spatial range for all variables This is performed for each time step.
DataSet.spatial_sum(self[, by_area])	Calculate the spatial sum for all variables This is performed for each time step.
DataSet.monthly_mean(self)	Calculate the monthly mean for each year/month combination in files.
DataSet.monthly_min(self)	Calculate the monthly minimum for each year/month combination in files.
DataSet.monthly_max(self)	Calculate the monthly maximum for each year/month combination in files.
DataSet.monthly_range(self)	Calculate the monthly range for each year/month combination in files.
DataSet.daily_mean_climatology(self)	Calculate a daily mean climatology
DataSet.daily_min_climatology(self)	Calculate a daily minimum climatology
DataSet.daily_max_climatology(self)	Calculate a daily maximum climatology
DataSet.daily_mean_climatology(self)	Calculate a daily mean climatology
DataSet.daily_range_climatology(self)	Calculate a daily range climatology
DataSet.monthly_mean_climatology(self)	Calculate the monthly mean climatologies Defined as the minimum value in each month across all years.
DataSet.monthly_min_climatology(self)	Calculate the monthly minimum climatologies Defined as the minimum value in each month across all years.
DataSet.monthly_max_climatology(self)	Calculate the monthly maximum climatologies Defined as the maximum value in each month across all years.
DataSet.monthly_range_climatology(self)	Calculate the monthly range climatologies Defined as the range of value in each month across all years.
DataSet.annual_mean(self)	Calculate the annual mean for each variable
DataSet.annual_min(self)	Calculate the annual minimum for each variable
DataSet.annual_max(self)	Calculate the annual maximum for each variable
DataSet.annual_range(self)	Calculate the annual range for each variable
DataSet.seasonal_mean(self)	Calculate the seasonal mean for each year.
DataSet.seasonal_min(self)	Calculate the seasonal minimum for each year.
DataSet.seasonal_max(self)	Calculate the seasonal maximum for each year.
DataSet.seasonal_range(self)	Calculate the seasonal range for each year.
DataSet.seasonal_mean_climatology(self)	Calculate a climatological seasonal mean
DataSet.seasonal_min_climatology(self)	Calculate a climatological seasonal min This is defined as the minimum value in each season across all years.
DataSet.seasonal_max_climatology(self)	Calculate a climatological seasonal max This is defined as the maximum value in each season across all years.
DataSet.seasonal_range_climatology(self)	Calculate a climatological seasonal range This is defined as the range of values in each season across all years.

nctoolkit.DataSet.mean

DataSet.**mean**(*self*)
Calculate the temporal mean of all variables

nctoolkit.DataSet.min

DataSet.**min**(*self*)
Calculate the temporal minimum of all variables

nctoolkit.DataSet.percentile

DataSet.**percentile**(*self*, *p=None*)
Calculate the temporal percentile of all variables

Parameters **p** (*float or int*) – Percentile to calculate

nctoolkit.DataSet.max

DataSet.**max**(*self*)
Calculate the temporal maximum of all variables

nctoolkit.DataSet.sum

DataSet.**sum**(*self*)
Calculate the temporal sum of all variables

nctoolkit.DataSet.range

DataSet.**range**(*self*)
Calculate the temporal range of all variables

nctoolkit.DataSet.var

DataSet.**var**(*self*)
Calculate the temporal variance of all variables

nctoolkit.DataSet.cum_sum

DataSet.**cum_sum**(*self*)
Calculate the temporal cumulative sum of all variables

nctoolkit.DataSet.cor_space

DataSet.**cor_space**(*self*, *var1=None*, *var2=None*)

Calculate the correlation correct between two variables in space This is calculated for each time step. The correlation coefficient coefficient is calculated using values in all grid cells, ignoring missing values.

Parameters

- **var1** (*str*) – The first variable
- **var2** (*str*) – The second variable

nctoolkit.DataSet.cor_time

DataSet.**cor_time**(*self*, *var1=None*, *var2=None*)

Calculate the correlation correct in time between two variables The correlation is calculated for each grid cell, ignoring missing values.

Parameters

- **var1** (*str*) – The first variable
- **var2** (*str*) – The second variable

nctoolkit.DataSet.spatial_mean

DataSet.**spatial_mean**(*self*)

Calculate the area weighted spatial mean for all variables This is performed for each time step.

nctoolkit.DataSet.spatial_min

DataSet.**spatial_min**(*self*)

Calculate the spatial minimum for all variables This is performed for each time step.

nctoolkit.DataSet.spatial_max

DataSet.**spatial_max**(*self*)

Calculate the spatial maximum for all variables This is performed for each time step.

nctoolkit.DataSet.spatial_percentile

DataSet.**spatial_percentile**(*self*, *p=None*)

Calculate the spatial sum for all variables This is performed for each time step. :param p: Percentile to calculate. 0<=p<=100. :type p: int or float

nctoolkit.DataSet.spatial_range

DataSet.**spatial_range**(*self*)
Calculate the spatial range for all variables This is performed for each time step.

nctoolkit.DataSet.spatial_sum

DataSet.**spatial_sum**(*self*, *by_area=False*)
Calculate the spatial sum for all variables This is performed for each time step.

Parameters **by_area** (*boolean*) – Set to True if you want to multiply the values by the grid cell area before summing over space. Default is False.

nctoolkit.DataSet.monthly_mean

DataSet.**monthly_mean**(*self*)
Calculate the monthly mean for each year/month combination in files. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_min

DataSet.**monthly_min**(*self*)
Calculate the monthly minimum for each year/month combination in files. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_max

DataSet.**monthly_max**(*self*)
Calculate the monthly maximum for each year/month combination in files. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_range

DataSet.**monthly_range**(*self*)
Calculate the monthly range for each year/month combination in files. This applies to each file in an ensemble.

nctoolkit.DataSet.daily_mean_climatology

DataSet.**daily_mean_climatology**(*self*)
Calculate a daily mean climatology

nctoolkit.DataSet.daily_min_climatology

DataSet.**daily_min_climatology**(*self*)
Calculate a daily minimum climatology

nctoolkit.DataSet.daily_max_climatology

DataSet.**daily_max_climatology**(*self*)
Calculate a daily maximum climatology

nctoolkit.DataSet.daily_range_climatology

DataSet.**daily_range_climatology**(*self*)
Calculate a daily range climatology

nctoolkit.DataSet.monthly_mean_climatology

DataSet.**monthly_mean_climatology**(*self*)
Calculate the monthly mean climatologies Defined as the minimum value in each month across all years. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_min_climatology

DataSet.**monthly_min_climatology**(*self*)
Calculate the monthly minimum climatologies Defined as the minimum value in each month across all years. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_max_climatology

DataSet.**monthly_max_climatology**(*self*)
Calculate the monthly maximum climatologies Defined as the maximum value in each month across all years. This applies to each file in an ensemble.

nctoolkit.DataSet.monthly_range_climatology

DataSet.**monthly_range_climatology**(*self*)
Calculate the monthly range climatologies Defined as the range of value in each month across all years. This applies to each file in an ensemble.

nctoolkit.DataSet.annual_mean

DataSet.**annual_mean**(*self*)
Calculate the annual mean for each variable

nctoolkit.DataSet.annual_min

DataSet.**annual_min**(*self*)
Calculate the annual minimum for each variable

nctoolkit.DataSet.annual_max

DataSet.**annual_max**(*self*)
Calculate the annual maximum for each variable

nctoolkit.DataSet.annual_range

DataSet.**annual_range**(*self*)
Calculate the annual range for each variable

nctoolkit.DataSet.seasonal_mean

DataSet.**seasonal_mean**(*self*)
Calculate the seasonal mean for each year. Applies at the grid cell level.

nctoolkit.DataSet.seasonal_min

DataSet.**seasonal_min**(*self*)
Calculate the seasonal minimum for each year. Applies at the grid cell level.

nctoolkit.DataSet.seasonal_max

DataSet.**seasonal_max**(*self*)
Calculate the seasonal maximum for each year. Applies at the grid cell level.

nctoolkit.DataSet.seasonal_range

DataSet.**seasonal_range**(*self*)
Calculate the seasonal range for each year. Applies at the grid cell level.

nctoolkit.DataSet.seasonal_mean_climatology

DataSet.**seasonal_mean_climatology**(*self*)
Calculate a climatological seasonal mean

Parameters = int (*window*) – The size of the window for the calculation of the rolling sum

nctoolkit.DataSet.seasonal_min_climatology

DataSet.**seasonal_min_climatology**(*self*)

Calculate a climatological seasonal min This is defined as the minimum value in each season across all years.

Parameters = int (*window*) – The size of the window for the calculation of the rolling sum

nctoolkit.DataSet.seasonal_max_climatology

DataSet.**seasonal_max_climatology**(*self*)

Calculate a climatological seasonal max This is defined as the maximum value in each season across all years.

Parameters = int (*window*) – The size of the window for the calculation of the rolling sum

nctoolkit.DataSet.seasonal_range_climatology

DataSet.**seasonal_range_climatology**(*self*)

Calculate a climatological seasonal range This is defined as the range of values in each season across all years.

Parameters = int (*window*) – The size of the window for the calculation of the rolling sum

1.7.21 Seasonal methods

DataSet.seasonal_mean(self)	Calculate the seasonal mean for each year.
DataSet.seasonal_min(self)	Calculate the seasonal minimum for each year.
DataSet.seasonal_max(self)	Calculate the seasonal maximum for each year.
DataSet.seasonal_range(self)	Calculate the seasonal range for each year.
DataSet.seasonal_mean_climatology(self)	Calculate a climatological seasonal mean
DataSet.seasonal_min_climatology(self)	Calculate a climatological seasonal min This is defined as the minimum value in each season across all years.
DataSet.seasonal_max_climatology(self)	Calculate a climatological seasonal max This is defined as the maximum value in each season across all years.
DataSet.seasonal_range_climatology(self)	Calculate a climatological seasonal range This is defined as the range of values in each season across all years.
DataSet.select_season(self[, season])	Select season from a dataset

1.7.22 Merging methods

DataSet.merge(self[, match])	Merge a multi-file ensemble into a single file Merging will occur based on the time steps in the first file.
DataSet.merge_time(self)	Time-based merging of a multi-file ensemble into a single file This method is ideal if you have the same data split over multiple files covering different data sets.

nctoolkit.DataSet.merge

DataSet.**merge**(*self*, *match=['year', 'month', 'day']*)

Merge a multi-file ensemble into a single file Merging will occur based on the time steps in the first file. This will only be effective if you want to merge files with the same times, but with different variables.

Parameters

- **match** (*list, str*) –
- **list or str stating what must match in the netcdf files. Defaults to year/month/day. This list must be some combination of year/month/day. An error will be thrown if the elements of time in match do not match across all netcdf files. The only exception is if there is a single date file in the ensemble.** (*a*) –

nctoolkit.DataSet.merge_time

DataSet.**merge_time**(*self*)

Time-based merging of a multi-file ensemble into a single file This method is ideal if you have the same data split over multiple files covering different data sets.

1.7.23 Climatology methods

DataSet.daily_mean_climatology(self)	Calculate a daily mean climatology
DataSet.daily_min_climatology(self)	Calculate a daily minimum climatology
DataSet.daily_max_climatology(self)	Calculate a daily maximum climatology
DataSet.daily_mean_climatology(self)	Calculate a daily mean climatology
DataSet.daily_range_climatology(self)	Calculate a daily range climatology
DataSet.monthly_mean_climatology(self)	Calculate the monthly mean climatologies Defined as the minimum value in each month across all years.
DataSet.monthly_min_climatology(self)	Calculate the monthly minimum climatologies Defined as the minimum value in each month across all years.
DataSet.monthly_max_climatology(self)	Calculate the monthly maximum climatologies Defined as the maximum value in each month across all years.
DataSet.monthly_range_climatology(self)	Calculate the monthly range climatologies Defined as the range of value in each month across all years.

1.7.24 Splitting methods

DataSet.split(self[, by])	Split the dataset Each file in the ensemble will be separated into new files based on the splitting argument.

nctoolkit.DataSet.split

DataSet.**split**(*self*, *by=None*)

Split the dataset Each file in the ensemble will be separated into new files based on the splitting argument.

Parameters **by** (*str*) – Available by arguments are 'year', 'month', 'yearmonth', 'season', 'day'. year will split files by year, month will split files by month, yearmonth will split files by year and month; season will split files by year, day will split files by day.

1.7.25 Output methods

DataSet.write_nc(self, out[, zip, overwrite])	Save a dataset to a named file This will only work with single file datasets.
DataSet.to_xarray(self[, decode_times])	Open a dataset as an xarray object
DataSet.to_dataframe(self[, decode_times])	Open a dataset as a pandas data frame
DataSet.zip(self)	Zip the dataset This will compress the files within the dataset.

nctoolkit.DataSet.write_nc

DataSet.**write_nc**(*self*, *out*, *zip=True*, *overwrite=False*)

Save a dataset to a named file This will only work with single file datasets.

Parameters

- **out** (*str*) – Output file name.
- **zip** (*boolean*) – True/False depending on whether you want to zip the file. Default is True.
- **overwrite** (*boolean*) – If out file exists, do you want to overwrite it? Default is False.

nctoolkit.DataSet.to_xarray

DataSet.**to_xarray**(*self*, *decode_times=True*)

Open a dataset as an xarray object

Parameters **decode_times** (*boolean*) – Set to False if you do not want xarray to decode the times. Default is True. If xarray cannot decode times, CDO will be used.

nctoolkit.DataSet.to_dataframe

DataSet.**to_dataframe**(*self*, *decode_times=True*)

Open a dataset as a pandas data frame

Parameters **decode_times** (*boolean*) – Set to False if you do not want xarray to decode the times prior to conversion to data frame. Default is True.

nctoolkit.DataSet.zip

DataSet.**zip**(*self*)
Zip the dataset This will compress the files within the dataset. This works lazily.

1.7.26 Miscellaneous methods

DataSet.cell_areas(self[, join])	Calculate the area of grid cells.
DataSet.cdo_command(self[, command])	Apply a cdo command
DataSet.nco_command(self[, command, ensemble])	Apply an nco command
DataSet.compare_all(self[, expression])	Compare all variables to a constant
DataSet.reduce_dims(self)	Reduce dimensions of data This will remove any dimensions with only one value.
DataSet.reduce_grid(self, mask)	Reduce the dataset to non-zero locations in a mask :param mask: single variable dataset or path to .nc file.

nctoolkit.DataSet.cell_areas

DataSet.**cell_areas**(*self*, *join=True*)
Calculate the area of grid cells. Area of grid cells is given in square meters.

Parameters **join** (*boolean*) – Set to False if you only want the cell areas to be in the output. join=True adds the areas as a variable to the dataset. Defaults to True.

nctoolkit.DataSet.cdo_command

DataSet.**cdo_command**(*self*, *command=None*)
Apply a cdo command

Parameters **command** (*string*) – cdo command to call. This command must be such that "cdo {command} infile outfile" will run.

nctoolkit.DataSet.nco_command

DataSet.**nco_command**(*self*, *command=None*, *ensemble=False*)
Apply an nco command

Parameters

- **command** (*string*) – nco command to call. This must be of a form such that "nco {command} infile outfile" will run.
- **ensemble** (*boolean*) – Set to True if you want the command to take all of the files as input. This is useful for ensemble methods.

nctoolkit.DataSet.compare_all

DataSet.**compare_all**(*self*, *expression=None*)

Compare all variables to a constant

Parameters **expression** (*str*) – This a regular comparison such as "<0", ">0", "==0"

nctoolkit.DataSet.reduce_dims

DataSet.**reduce_dims**(*self*)

Reduce dimensions of data This will remove any dimensions with only one value. For example, if only selecting one vertical level, the vertical dimension will be removed.

nctoolkit.DataSet.reduce_grid

DataSet.**reduce_grid**(*self*, *mask*)

Reduce the dataset to non-zero locations in a mask :param mask: single variable dataset or path to .nc file. The mask must have an identical grid to the dataset. :type mask: str or dataset

INDEX

O

P

R

S

T

V

W

Y

Z