xbpch Documentation

Release 0.2.0

Daniel Rothenberg

Mar 19, 2018

Contents

xpbch is a simple utility for reading the proprietary binary punch format (bpch) outputs used in versions of GEOS-Chem earlier than v11-02. The utility allows a user to load this data into an xarray- and dask-powered workflow without necessarily pre-processing the data using GAMAP or IDL. This opens the door to out-of-core and parallel processing of GEOS-Chem output.

CHAPTER 1

Installation

1.1 Requirements

xbpch is written in pure Python (version >= 3.5), and leans on two important libraries:

1. xarray (version >= 0.9): a pandas-like toolkit for working with labeled, *n*-dimensional data

2. dask (version >= 0.14): a library for performing out-of-core, parallel computations on both tabular and array-like datasets

The easiest way to install these libraries is to use the conda package manager:

```
$ conda install -c conda-forge xarray dask
```

conda can be obtained as part of the Anaconda Python distribution from Continuum IO, although you do not need all of the packages it provides in order to use **xbpch**. Note that we recommend installing the latest versions from community-maintained conda-forge collection, since these usually contain bug-fixes and additional features.

Note: Basic support for Python 2.7 is available in **xbpch** but it has not been tested, since the evolutionary GCPy package will only support Python 3. If, for some reason, you must use Python 2.7 and encounter problems, please reach out to us and we may be able to fix them.

1.2 Installation via conda

The preferred way to install **xbpch** is also via conda:

```
$ conda install -c conda-forge xbpch
```

1.3 Installation via pip

xbpch is available on PyPI, and can be installed using setuptools:

```
$ pip install xbpch
```

1.4 Installation from source

If you're developing or contributing to **xbpch**, you may wish instead to install directly from a local copy of the source code. To do so, you must first clone the the master repository (or a fork) and install locally via pip:

```
$ git clone https://github.com/darothen/xbpch.git
$ cd xbpch
$ python setup.py install
```

You will need to substitute in the path to your preferred repository/mirror of the source code.

Note that you can also install directly from the source using setuptools:

```
$ pip install git+https://github.com/darothen/xbpch.git
```

CHAPTER 2

Quick Start

Assuming you're already familiar with xarray, it's easy to dive right in to begin reading bpch data. If you don't have any GEOS-Chem data handy to test with, I've archived a sample dataset here here consisting of 14 days of hourly, ND49 output - good for diagnosing surface air quality statistics.

Download the data and extract it to some directory:

```
$ wget https://ndownloader.figshare.com/files/8251094
$ tar -xvzf sample_nd49.tar.gz
```

You should now see 14 `bpch` files in your directory, and two `.dat` files.

The whole point of **xbpch** is to read these data files natively into an xarray.Dataset. You can do this with the *`xbpch.open_bpchdataset()`* method:

```
In [1]: import xbpch

In [2]: fn = "ND49_20060102_ref_e2006_m2010.bpch"

In [3]: ds = xbpch.open_bpchdataset(fn)
```

If we print the dataset back out, we'll get a familiar representation:

```
<xarray.Dataset>
Dimensions:          (lat: 91, lev: 47, lon: 144, nv: 2, time: 24)
Coordinates:
  * lev              (lev) float64 0.9925 0.9775 0.9624 0.9473 0.9322 0.9171 ...
  * lon              (lon) float64 -180.0 -177.5 -175.0 -172.5 -170.0 -167.5 ...
  * lat              (lat) float64 -89.5 -88.0 -86.0 -84.0 -82.0 -80.0 -78.0 ...
  * time             (time) datetime64[ns] 2006-01-01T01:00:00 ...
  * nv               (nv) int64 0 1
Data variables:
    IJ_AVG_S_NO      (time, lon, lat) float32 1.16601e-12 1.1599e-12 ...
    time_bnds        (time, nv) datetime64[ns] 2006-01-01T01:00:00 ...
    IJ_AVG_S_O3      (time, lon, lat) float32 9.25816e-09 9.25042e-09 ...
    IJ_AVG_S_SO4     (time, lon, lat) float32 1.41706e-10 1.4142e-10 ...
```

(continues on next page)

(continued from previous page)

```
    IJ_AVG_S_NH4     (time, lon, lat) float32 1.16908e-11 1.16658e-11 ...
    IJ_AVG_S_NIT     (time, lon, lat) float32 9.99837e-31 9.99897e-31 ...
    IJ_AVG_S_BCPI    (time, lon, lat) float32 2.46206e-12 2.45698e-12 ...
    IJ_AVG_S_OCPI    (time, lon, lat) float32 2.65303e-11 2.6476e-11 ...
    IJ_AVG_S_BCPO    (time, lon, lat) float32 4.19881e-19 4.18213e-19 ...
    IJ_AVG_S_OCPO    (time, lon, lat) float32 2.49109e-22 2.53752e-22 ...
    IJ_AVG_S_DST1    (time, lon, lat) float32 7.11484e-12 7.10209e-12 ...
    IJ_AVG_S_DST2    (time, lon, lat) float32 1.55181e-11 1.54779e-11 ...
    IJ_AVG_S_SALA    (time, lon, lat) float32 3.70387e-11 3.69923e-11 ...
    OD_MAP_S_AOD     (time, lon, lat) float32 0.292372 0.325568 0.358368 ...
    OD_MAP_S_DSTAOD  (time, lon, lat) float32 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
Attributes:
    modelname:    GEOS5_47L
    halfpolar:    1
    res:          (2.5, 2.0)
    center180:    1
    tracerinfo:   tracerinfo.dat
    diaginfo:     diaginfo.dat
    filetitle:    b'GEOS-CHEM DIAG49 instantaneous timeseries'
    source:       ND49_20060101_ref_e2006_m2010.bpch
    filetype:     b'CTM bin 02'
    Conventions:  CF1.6
```

You can then proceed to process the data using the conventional routines you'd use in any xarray-powered workflow.

In the sample dataset highlighted here, the 14 days of hourly output are split across 14 files - one for each day's worth of data. **xbpch** provides a second method, *xbpch.open_mfbpchdataset()*, for reading in multiple-file datasets like these, and automatically concatenating them on the *time* record dimension:

```
In [4]: import xbpch

In [5]: from glob import glob

# List all the bpch files in the current directory
In [6]: fns = glob("ND49_*.bpch")

# Helper function to extract spatial mean O3 from each file
In [7]: def _preprocess(ds):
   ...:     return ds[['IJ_AVG_S_O3', ]].mean(['lon', 'lat'])
   ...:

In [8]: ds = xbpch.open_mfbpchdataset(
   ...:     fns, preprocess=_preprocess, dask=True, memmap=True
   ...: )
   ...:
```

Again, printing yields the expected results:

```
<xarray.Dataset>
Dimensions:      (time: 336)
Coordinates:
  * time         (time) datetime64[ns] 2006-01-01T01:00:00 ...
Data variables:
    IJ_AVG_S_O3  (time) float32 2.5524e-08 2.55541e-08 2.55588e-08 ...
```

Finally, if you don't want to drop into a Python interpreter but just want to quickly convert your binary data to NetCDF, you can run the utility script *bpch_to_nc* which is shipped with this library:

```
$ bpch_to_nc /path/to/my/data.bpch /path/to/my/output.nc

Reading in file(s)...

Decoding variables...

Writing to /path/to/my/output.nc ...
syncing
[####################################] | 100% Completed | 52.1s
```

CHAPTER 3

Usage and Examples

3.1 Reading Output

The routines for reading bpch files from disk into `xarray.Datasets` is based mostly on the `xarray.open_dataset` method. However, to handle some of the idiosyncrasies of GEOS-Chem output, our implementation of *`open_bpchdataset()`* has a few additional arguments to know about.

3.1.1 Main `open_bpchdataset()` Arguments

The majority of the time, you'll want to load/read data via xarray, using the method *`open_bpchdataset()`*, as shown in the *Quick Start*. This routine fundamentally requires three arguments:

- `filename`: the full path to the output file you want to load
- `tracerinfo_file`: the full path to the file *tracerinfo.dat*, which contains the names and indices of each tracer output by GEOS-Chem
- `diaginfo_file`: the full path to the file *diaginfo.dat*, which contains the listing of categories and their tracer number offsets in the tracer output index.

If you don't pass a value for either `tracerinfo_file` or `diaginfo_file`, **xbpch** will look for them in the current directory, assuming the Default naming scheme. However, if it *still* can't find a file, it'll raise an error (we do not assume to know what is in your output!)

In many simulations, GEOS-Chem will write multiple timesteps of a large number of fields to a single output file. This can result in outputs on the order of 10's of GB! If you know for certain that you only want a specific tracer or category of tracers, you can supply a list of their names to either `fields` or `categories`.

For instance, using the v11-01 diagnostics for reference, we can load in any tracer with the name "O3" by passing

```
In [1]: import xbpch

In [2]: o3_data = xbpch.open_bpchdataset("my_data.bpch", fields=['O3', ])
```

Alternatively, we can load all the tracers associated with a given category by specifying the `categories` argument. To grab all the saved 2D meteorology fields, this would entail

```
In [3]: met_data = xbpch.open_bpchdataset(
   ...:     "my_data.bpch", categories=["DAO-FLDS", ]
   ...: )
   ...:
```

3.1.2 What Works and Doesn't Work

xbpch should work with most standard GEOS-Chem outputs going back to at least v9-02. It has been tested against some of these standard outputs, but not exhaustively. If you have an idiosyncratic GEOS-Chem output (e.g. from a specialized version of the model with custom tracers or a new grid), please give **xbpch** a try and if it fails, post a an Issue on our GitHub page to let us know.

The following configurations have been tested and vetted:

- Standard output on standard grids
- ND49 output on standard grids
- ND49 output on nested North America grid (should work for all nested grids)

3.1.3 Eager vs Lazy Loading

One of the main advantages to using **xbpch** is that it allows you to access data without immediately need to read it all from disk. On a modern analysis cluster, this isn't a problem, but if you want to process output on your laptop, you can quickly run into situations where all of your data won't fit in memory. In those situations, you have to tediously block your analysis algorithms/pipeline.

Note: Even though you may request lazily-loaded data, **xpbch** still needs to read your input file to parse its contents. This requires iterating line-by-line through the input file, so it may take some time (about ~10 seconds to read a 6GB file on my late-2016 MacBook Pro). Unfortunately, if we don't do this, we can't infer the tracers or their distribution over multiple timesteps containined in the input file.

The keyword arguments `memmap` and `dask` control how data is read from your bpch files.

`memmap` if enabled, the data for each timestep and variable will be accessed through a memory-map into the input file

`dask` if enabled, the function to read each timestep for each variable will be wrapped in a `dask.delayed` object, initiating a task graph for accessing the data

Warning: Opening a dataset using `memmap=True` and `dask=False` *will not work*. Each memory-mapped array counts as an open file, which will quickly add up and hit your operating system's limit on simultaneously open files.

If `dask=True` is used to open a dataset, then all of the data in the bpch file is represented by `dask.arrays`, and all operations are lazy. That is, they are not evaluated until the user explicitly instruct them to be, and instead a graph representing your computation is constructed.

3.1.4 Chunking

When data is loaded with the `dask` flag enabled, all the operations necessary to create contiguous chunks of data are deferred. Because of the way data is written to bpch files by GEOS-Chem, these deferred actions are all based on single timesteps of data for each variable by default. Thus, in the parlance of dask, all the data is implicitly chunked on the **time** dimension.

When dask encounters chunked calculations, it will automatically attempt to parallelize them across all the cores available on your machine, and will attempt to limit the amount of data held in-memory at any give time.

To illustrate this, consider a monthly history dataset `ds` loaded via *`open_bpchdataset()`*. The inital task graph representing this data may look something like:

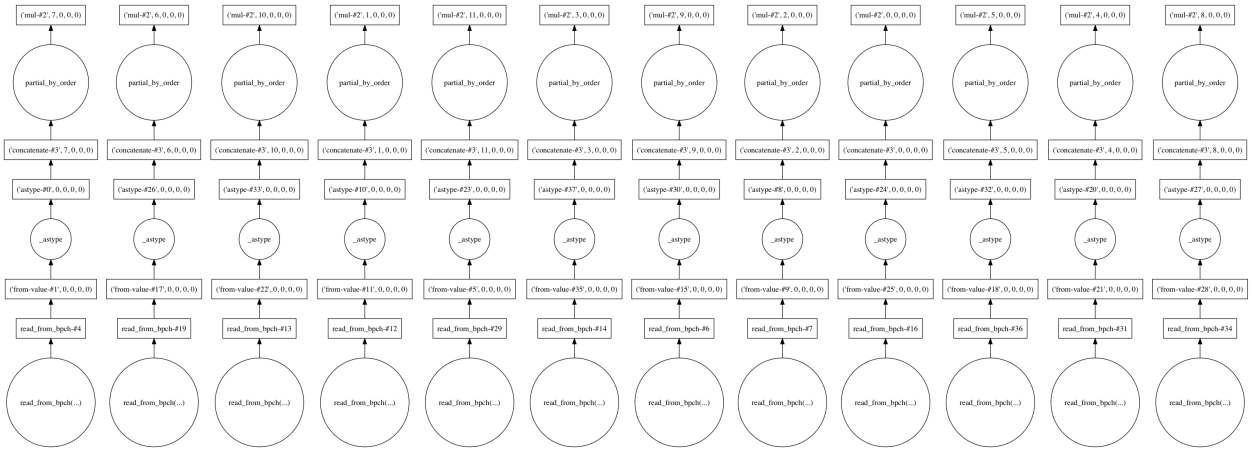

Fig. 1: Tasks for reading and processing monthly output for a single variable in a year-long bpch output file

This graph illustrates that dask is expected to process 12 chunks of data - one for each month (timestep) in the dataset. The graph shows the operations for reading the data, casting it to the correct data type, and re-scaling, which are applied automatically by **xbpch** and xarray.

At this point, the data has only been processed in such a way that it fits the numpy.ndarray memory model, and thus can be used to construct xarray objects. A trivial calculation on this data may be to normalize the timeseries of data in each grid cell to have zero mean and unit variance. For any `xarray.DataArray` we could write this operation as

```
In [4]: da_normal = (da - da.mean('time'))/da.std('time')
```

which produces the computational graph

A second key function of `dask` is to analyze and parse these computational graphs into a simplified form. In practice, the resulting graph will be much simpler, which can dramatically speed up your analysis. For instance, if you sub-sample the variables and timesteps used in your analysis, **xbpch** (through dask) will avoid reading extra, unused data from the input files you passed it.

Note: Sometimes it's advantagous to re-chunk a dataset (see here for a discussion on when this may be the case). This is easily accomplished through xarray, or can be done directly on the `dask.arrays` containing your data if you have a more complex analysis to perform.

Finally, it's important to know that the computational graphs that dask produces are never evaluated until you explicitly call `.load()` on a dask array or xarray Data{Array,set}. Different computations or uses for your data might imply an automatic `load()`; for instance, if you use the plotting wrapper built into xarray, it will (necessarily) eagerly load your data. If you'd like to monitor the progress of a very long analysis built through **xbpch**/xarray/dask, you can use the built-in diagnostic tools from dask:

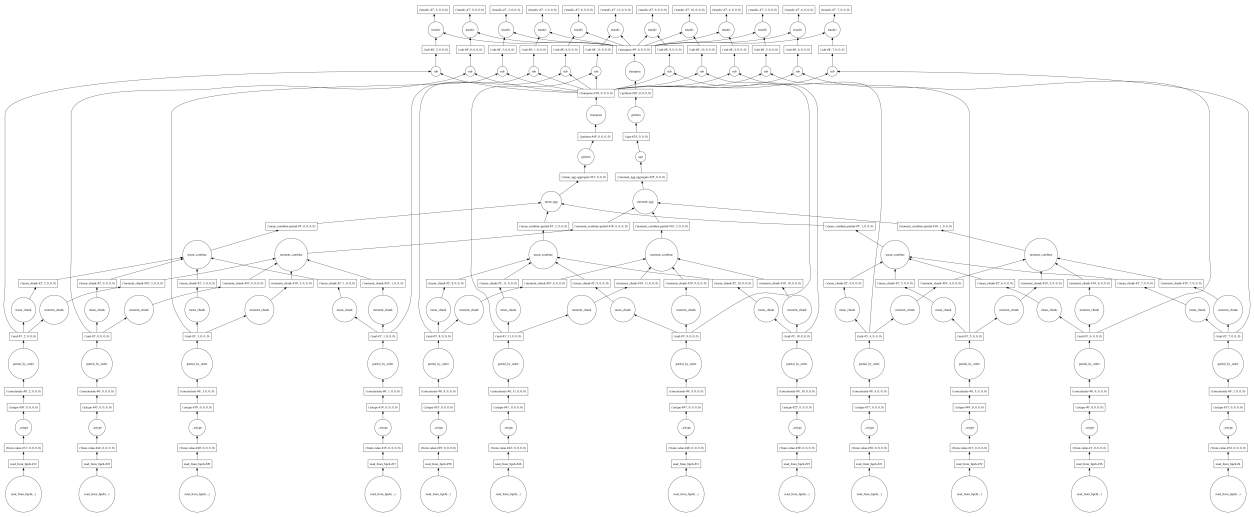

Fig. 2: Computational graph for normalizing monthly data

```
In [5]: from dask.diagnostics import ProgressBar

# Construct some analysis
In [6]: my_ds = ...

# Eagerly compute the results
In [7]: with ProgressBar() as pb:
   ...:     my_ds.load()
   ...:
```

```
[########################################] | 100% Completed | 10.2s
```

3.2 Geographic Visualization

One easy application of **xbpch** is for the visualization of your data. For cartographic or geographic plots, we recommend using the cartopy package maintained by the UK Met Office.

Plotting on a cartopy map is straightforward. Suppose we have a Dataset `ds` read from a bpch file. We can first compute an analysis of interest - say, the difference between mean fields for summer versus winter:

```
In [8]: ds_seas = ds.groupby("time.season").mean('time')

In [9]: diff = ds_seas.sel(season='DJF') - ds_seas.sel(season='JJA')
```

```
<xarray.Dataset>
Dimensions:      (lat: 91, lev: 47, lon: 144, nv: 2)
Coordinates:
  * lev          (lev) float64 0.9925 0.9775 0.9624 0.9473 0.9322 0.9171 ...
  * lon          (lon) float64 -180.0 -177.5 -175.0 -172.5 -170.0 -167.5 ...
  * lat          (lat) float64 -89.5 -88.0 -86.0 -84.0 -82.0 -80.0 -78.0 ...
  * nv           (nv) int64 0 1
Data variables:
```

(continues on next page)

(continued from previous page)

```
    ANTHSRCE_O3  (lon, lat) float32 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    IJ_AVG_S_O3  (lon, lat, lev) float32 -23.1014 -23.2715 -23.4614 -23.5216 ...
```

Plotting a portion of this dataset on a cartopy map is straightforward. First, we create a figure and add an axes with the map projection information encoded:

```
In [10]: import matplotlib.pyplot as plt

In [11]: import cartopy.crs as ccrs

In [12]: fig = plt.figure()

In [13]: ax = fig.add_subplot(111, projection=ccrs.PlateCarree(), aspect='auto')
```

Then, we can plot our data as normal. cartopy has a few helper functions which we can use to add basic geographic elements such as coastlines and borders to the plot.

```
In [14]: import cartopy.feature as cfeature

# Select some data to plot
In [15]: da = diff.isel(lev=0).IJ_AVG_S_O3

In [16]: im = ax.contourf(da.lon.values, da.lat.values, da.values.T)

In [17]: cb = fig.colorbar(im, ax=ax, orientation='horizontal')

In [18]: ax.add_feature(cfeature.COASTLINE)

In [19]: ax.add_feature(cfeature.BORDERS)
```

Alternatively, we can use xarray's matplotlib wrappers to automate some of this plotting for us. For instance, we can quickly make a faceted plot of our seasonal data (including with a cartopy axis) with just a few lines of code:

```
# Select some data to plot
In [20]: da = ds_seas.isel(lev=0).IJ_AVG_S_O3

In [21]: da = da - ds.isel(lev=0).IJ_AVG_S_O3.mean('time')

In [22]: g = da.plot.imshow('lon', 'lat', col='season', col_wrap=2,
   ....:                    subplot_kws=dict(projection=ccrs.Robinson()),␣
↪transform=ccrs.PlateCarree())
   ....:

In [23]: for ax in g.axes.flatten():
   ....:     ax.add_feature(cfeature.COASTLINE)
   ....:
```

There's a lot going on in this code sample:

1. First, we take the seasonal mean data we formerly computed.
2. Subtract out the annual mean from each seasonal mean.
3. Use imshow to plot each grid cell in our dataset.
    - We tell the plotting function to use `"lon"` and `"lat"` as the keys to access the x/y data for the dataset
    - We further instruct xarray to facet over the `"season"` coordinate, and include two columns per row in the resulting facet grid

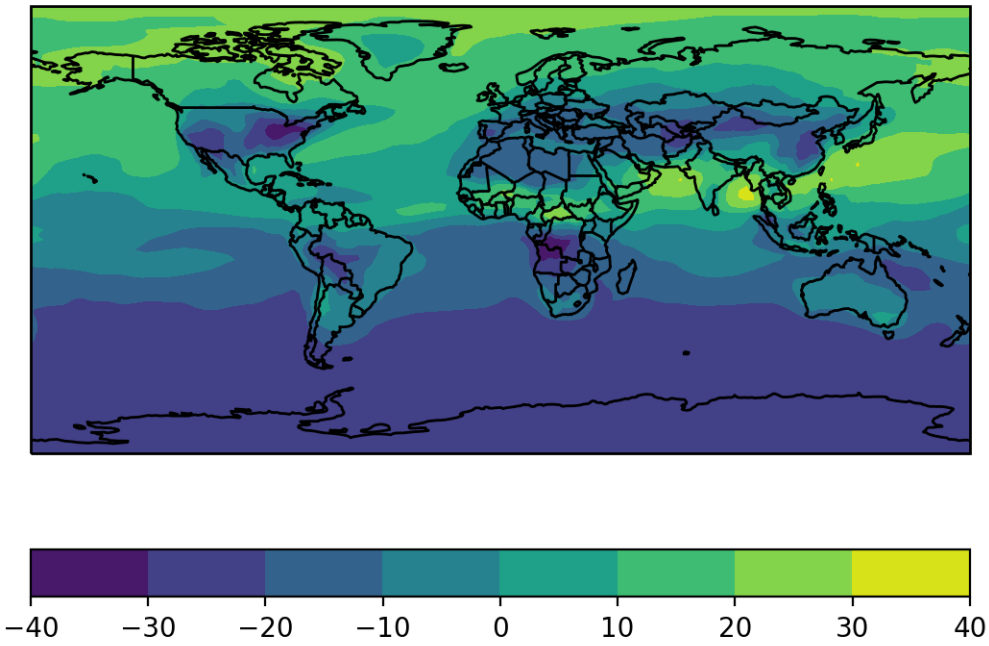

Fig. 3: Example of a simple plot with cartopy

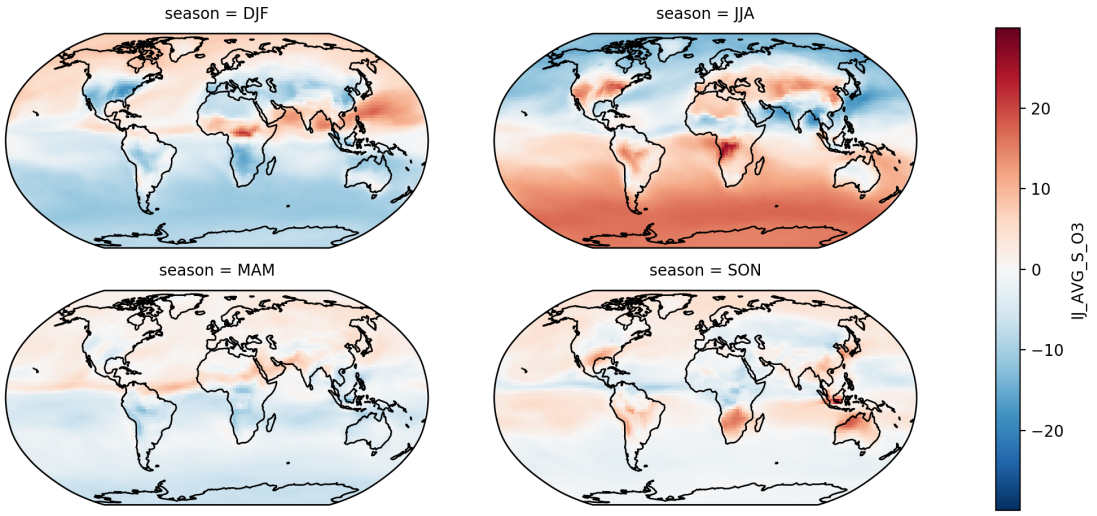

Fig. 4: Faceting over a non-coordinate dimension using xarray's built-in plotting tools.

- We pass a dictionary of keyword arguments to `subplot_kws`, which is used when creating each subplot in our facet grid. In this case, we tell each subplot to use a Robinson map projection
- We pass a final keyword argument, `transform`, which is passed to each invocation of `imshow()` on the facet grid; this tells cartopy how to map from the projection data to our actual data. Here, a `ccrs.PlateCarree()` is a standard, equally-spaced latitude-longitude grid

4. Iterate over each axis in the facet grid, and add our coastlines to it.

3.3 Timeseries Analysis

Another application that **xbpch**/xarray makes easy is timeseries analysis. For example, consider the timesries of ND49 output from the *Quick Start*. A classic timeseries analysis atmospheric chemistry is computing the daily maximum 8-hour average for a given tracer. The core of this computation can be achieved in just a few lines of code via xarray:

```
In [24]: o3 = ds.IJ_AVG_S_O3

In [25]: mda8_o3 = (
   ....:     o3.rolling(time=8, min_periods=6).mean()
   ....:       .resample("D", "time", how='max')
   ....: )
   ....:
```

This code is highly performant; the `.rolling()` operation is farmed out to a high-performance C library (bottleneck) and all operations are applied by broadcasting over the time dimension.

Note: bottleneck does not work with dask arrays, so you will need to eagerly `.load()` the data into memory if it hasn't already been done. Future versions of xarray will wrap functionality in dask to perform these operations in parallel, but this is a work in progress.

3.4 Save to NetCDF

Without any extra work, datsets read in via **xbpch** can easily be serialized back to disk in NetCDF format

```
In [26]: ds.to_netcdf("my_bpch_data.nc")
```

They can then be read back in via xarray

```
In [27]: import xarray as xr

In [28]: ds = xr.open_dataset("my_bpch_data.nc")
```

Note: As of v0.2.0, immediately writing to netcdf may not work due to the way variable units and scaling factors are encoded when they are read into **xbpch**. This will be fixed once some upstream issues with xarray are patched. If you run into the following `ValueError`:

```
ValueError: Failed hard to prevent overwriting key 'scale_factor'
```

then before you save it, process it with the `xbpch.common.fix_attr_encoding()` method

```
In [29]: my_ds = xbpch.common.fix_attr_encoding(my_ds)

In [30]: my_ds.to_netcdf("my_data.nc")
```

CHAPTER 4

Reading BPCH Files

xbpch provides three main utilities for reading bpch files, all of which are provided as top-level package imports. For most purposes, you should use `open_bpchdataset()`, however a lower-level interface, `BPCHFile()` is also provided in case you would prefer manually processing the bpch contents.

See *Usage and Examples* for more details.

`xbpch.`**`open_bpchdataset`**(*filename*, *fields=[]*, *categories=[]*, *tracerinfo_file='tracerinfo.dat'*, *diaginfo_file='diaginfo.dat'*, *endian='>'*, *decode_cf=True*, *memmap=True*, *dask=True*, *return_store=False*)

Open a GEOS-Chem BPCH file output as an xarray Dataset.

Parameters **filename** : string

Path to the output file to read in.

{tracerinfo,diaginfo}_file : string, optional

Path to the metadata "info" .dat files which are used to decipher the metadata corresponding to each variable in the output dataset. If not provided, will look for them in the current directory or fall back on a generic set.

fields : list, optional

List of a subset of variable names to return. This can substantially improve read performance. Note that the field here is just the tracer name - not the category, e.g. 'O3' instead of 'IJ-AVG-$_O3'.

categories : list, optional

List a subset of variable categories to look through. This can substantially improve read performance.

endian : {'=', '>', '<'}, optional

Endianness of file on disk. By default, "big endian" (">") is assumed.

decode_cf : bool

Enforce CF conventions for variable names, units, and other metadata

default_dtype : numpy.dtype, optional

Default datatype for variables encoded in file on disk (single-precision float by default).

memmap : bool

Flag indicating that data should be memory-mapped from disk instead of eagerly loaded into memory

dask : bool

Flag indicating that data reading should be deferred (delayed) to construct a task-graph for later execution

return_store : bool

Also return the underlying DataStore to the user

Returns **ds** : xarray.Dataset

Dataset containing the requested fields (or the entire file), with data contained in proxy containers for access later.

store : xarray.AbstractDataStore

Underlying DataStore which handles the loading and processing of bpch files on disk

xbpch.**open_mfbpchdataset**(*paths*, *concat_dim='time'*, *compat='no_conflicts'*, *preprocess=None*, *lock=None*, ***kwargs*)

Open multiple bpch files as a single dataset.

You must have dask installed for this to work, as this greatly simplifies issues relating to multi-file I/O.

Also, please note that this is not a very performant routine. I/O is still limited by the fact that we need to manually scan/read through each bpch file so that we can figure out what its contents are, since that metadata isn't saved anywhere. So this routine will actually sequentially load Datasets for each bpch file, then concatenate them along the "time" axis. You may wish to simply process each file individually, coerce to NetCDF, and then ingest through xarray as normal.

Parameters **paths** : list of strs

Filenames to load; order doesn't matter as they will be lexicographically sorted before we read in the data

concat_dim : str, default='time'

Dimension to concatenate Datasets over. We default to "time" since this is how GEOS-Chem splits output files

compat : str (optional)

String indicating how to compare variables of the same name for potential conflicts when merging:

- 'broadcast_equals': all values must be equal when variables are broadcast against each other to ensure common dimensions.
- 'equals': all values and dimensions must be the same.
- 'identical': all values, dimensions and attributes must be the same.
- 'no_conflicts': only values which are not null in both datasets must be equal. The returned dataset then contains the combination of all non-null values.

preprocess : callable (optional)

A pre-processing function to apply to each Dataset prior to concatenation

lock : False, True, or threading.Lock (optional)

Passed to `dask.array.from_array()`. By default, xarray employs a per-variable lock when reading data from NetCDF files, but this model has not yet been extended or implemented for bpch files and so this is not actually used. However, it is likely necessary before dask's multi-threaded backend can be used

****kwargs** : optional

Additional arguments to pass to *xbpch.open_bpchdataset()*.

class xbpch.**BPCHFile**(*filename*, *mode='rb'*, *endian='>'*, *diaginfo_file=''*, *tracerinfo_file=''*, *eager=False*, *use_mmap=False*, *dask_delayed=False*)

A file object for representing BPCH data on disk

Attributes

fp	(FortranFile) A pointer to the open unformatted Fortran binary output (the original bpch file)
var_data, var_attrs	(dict) Containers of 'BPCHDataBundle's and dicts, respectively, holding the accessor functions to the raw bpch data and their associated metadata

__init__(*filename*, *mode='rb'*, *endian='>'*, *diaginfo_file=''*, *tracerinfo_file=''*, *eager=False*, *use_mmap=False*, *dask_delayed=False*)

Load a BPCHFile

Parameters **filename** : str

Path to the bpch file on disk

mode : str

Mode string to pass to the file opener; this is currently fixed to "rb" and all other values will be rejected

endian : str {">", "<", ":"}

Endian-ness of the Fortran output file

{tracerinfo, diaginfo}_file : str

Path to the tracerinfo.dat and diaginfo.dat files containing metadata pertaining to the output in the bpch file being read.

eager : bool

Flag to immediately read variable data; if "False", then nothing will be read from the file and you'll need to do so manually

use_mmap : bool

Use memory-mapping to read data from file

dask_delayed : bool

Use dask to create delayed references to the data-reading functions

__weakref__

list of weak references to the object (if defined)

_read()

Parse the entire bpch file on disk and set up easy access to meta- and data blocks.

_read_header()
Process the header information (data model / grid spec)

_read_metadata()
Read the main metadata packaged within a bpch file, indicating the output filetype and its title.

_read_var_data()
Iterate over the block of this bpch file and return handlers in the form of ‘BPCHDataBundle‘s for access to the data contained therein.

close()
Close this bpch file.

CHAPTER 5

Recent Updates

v0.3.3 (March 18, 2018)

- Clean-up for xarray v0.10.2 compatibility
- Tweak to more reliably infer and unpack 3D field shape (from Jenny Fisher)

CHAPTER 6

License

Index

Symbols

B

C

O