Obs_Py documentation

Table of Contents

Document source:

Original Word document

Version Controlled PDF doc

1. What is obs_py?

The Python-3 package *obs_py* is designed for comparing observations with gridded fields, providing support for detailed analysis of observation quality and model performance.

The reason for developing this is that observations from different sources come in different formats and with different names for the same parameter. While observations that have gone through the Bureau's numerical weather prediction assimilation system will be in a standard format (Observation Data Base-2, or ODB2), not all observations do go through the system. Even for observations within ODB2, there are times when it is convenient to have them stored over different time ranges or geographic areas. Furthermore, ODB2 only provides information against the background fields used at the time, and does not directly support comparison with longer forecasts, different ensemble members or different modelling systems

Similarly files of gridded values also come in a variety of directory structures, filenames, variable names and data arrangements.. *obs_py* provides a flexible way of accessing gridded data, and is aware of many existing archive structures. In addition the names for parameters in grid files rarely match the names in the original observation files, and often have different units. *obs_py* provides consistent naming conventions and units.

Finally, *obs_py* also includes the corrections to near surface model fields to compensate for differences between model topography and actual station elevation.

2. How do I use *obs_py*?

There are currently two main steps in *obs_py*.

1. Creation of netCDF files of observations
2. Augmentation of observation netCDF files with collocated gridded data.

Reasonably comprehensive scripts are provided to support both of these steps.

There are also older scripts to support statistical analysis of the interpolated netCDF files, however the plan is to move towards using *MET/METplus* for the statistical analysis and display.

3. Location of source code:

The code is maintained on the Met Office Science Repository Service (MOSRS) under svn:

https://code.metoffice.gov.uk/svn/utils/access/branches/dev/petersteinle/r5885_obspy

A copy of this exists on gadi under

/g/data/dp9/pjs548/mosrs_utils/r5885_obspy

All references to software or configuration files are relative to these root directories.

With a run time environment of:

```
module unload bom_envs
module use -a /g/data/access/varpy/modules
```

```
module load varpy/2021.03.0_patched
module load varpytools/2021.03.0
module use -a /projects/access/modules
module load odbserver/0.16.2-ops-intel1903
module use -a /g/data/hh5/public/modules
module load conda/analysis3
export PYTHONPATH=/g/data/dp9/pjs548/mosrs_utils/r5885_obspy:${PYTHONPATH}
```

will allow *obs_py* to be imported into Python3

4. Creation of netCDF file of observations

Currently *obs_py* can process either ODB2 files or CSV (comma separated values) files. Example scripts are *py_scripts/odb2nc.py* and *py_scripts/cotl_csv_nc.py* . The basic structure of these scripts is

1. Process command line options
2. Define standard names and units etc. for *obs_py*
3. Characterize how observation data is arranged within the input file
4. Read the data into a pandas DataFrame
5. Write the DataFrame to a netCDF file

Standard techniques are used for much of this. Command line options are handed via the Python utility *argparse*. Configurations and descriptions of datasets are managed via Python classes. A particular instance of a class describes a given type of dataset. The configurations are manaed by YAML (Yet Another Markup Language) files, denoted by **.yml** in the same directory as the software containing the Python classes

Each of these steps are described in detail below.

4.1. Command line options

There are currently two types of observation input streams handled by *obs_py*, ODB2 (Observation Data Base-2) and CSV (Comma Separated Variable) files from COtL (Conditions Over the Land) network managers. Each data stream has its own script.

4.1.1. *odb2nc*: Converting ODB2 files to netCDF

Command line arguments are managed via *argparse* utility and are listed in Table 1.

The flow of this script is simple

1. Define and process command line arguments
2. Initialize, *fob_info*, an instance of the class with information on observations (*obs_py* names, units etc.)
3. Initialize *nwk*, an instance of the class of network information for the relevant observation data stream (e.g. odb_surface) that includes mappings of reported parameter names to *obs_py* names, and information on any parameter transformations.
4. Define the latitude and longitude filter for the ODB

5. Create a pandas DataFrame using *obs_py.odb2_df*
6. Write the netCDF file using *obs_py.df_nc*

Note that ODB2 data is assumed to be in standard units. If this does not hold for a parameter, it can be corrected using *obs_py.df_std_units*

Option	Abbreviation	Meaning	Default value
--input	-i	Input file	
--output	-o	Output file	
--obs_type	-O	Observation types to process	surface
--network	-n	Type of ODB2 file	odb_surface
--region	-r	Latitude/ longitude selection	'(lon>110 and lon<160 and lat>50 and lat<-5)'
-verbose	-v	Debugging level (according to *logger* utility)	30 (warning)

Table 1: command line arguments for odbs2nc - the script to convert from ODB2 files to netCDF files of observations

4.1.2. *cotl_csv_nc*: Converting CSV files from COtL to netCDF

Note: How this works for other network managers remains to be seen as COtL provides a unique set of parameters. Others may produce a strict subset or different formats or structures.

Command line arguments are managed via *argparse* utility and are listed in Table 2

The flow of this script is simple

1. Define and process command line arguments
2. Initialize, *fob_info*, an instance of the class with information on observations (*obs_py* names, units etc.)
3. Initialize *nwk*, an instance of the class of network information for the relevant observation data stream (e.g. odb_surface) that includes mappings of reported parameter names to *obs_py* names, and information on any parameter transformations.
4. Extract the station meta data from the *stn_list* file, with *obs_py.obs_csv_stn_info*
5. Create a pandas DataFrame using *obs_py.csv_df*
6. Convert all columns in the DataFrame to standard units with *obs_py.df_std_units*
7. Write the netCDF file using *obs_py.df_nc*

Note that ODB2 data is assumed to be in standard units. If this does not hold for a parameter, it can be corrected using *obs_py.df_std_units*

Option	Abbreviation	Meaning	Default value
--input	-i	Input file	cotl.csv
--output	-o	Output file	obs.nc
--network	-n	Name of network	cotl
--stnlist	-s	Station-based meta data (location, elevation etc.)	stns.csv
-verbose	-v	Debugging level (according to *logger* utility)	30 (warning)

Table 2: command line arguments for cotl_csv_ncc - the script to convert csv files from COtL to netCDF files of observations

Note that that names of columns in the station meta data file (*stnlist* option) must be a subset of those in the input file. This is not always the case, so an example header is included in

py_scripts/cotl_csv_header.txt .

This file can be used by

bash_scripts/process_cotl_csv.sh

to generate consistency between the two input data sets.

4.2. Define standard names and units etc. for *obs_py*

To deal with a wide variety of data streams, of both observations and grids, *obs_py* uses a standard set of names to define parameters. For readability, long versions of the names are used, as in general they only need to be entered into configuration files.

The list of parameter names, and any meta data, such as units, long names, variable types etc. is stored in *obs_py/obfile_info.yml* , and will automatically loaded when an instance of the class *obs_var_info* from *obs_py.obfile_info* is created. The default settings can be augmented or overwritten a local version of *obfile_info.yml* in the working directory.

> **NOTE**: The copy in the working directory only needs to contain changes. This makes it easy to see what the differences are.

The instance of *obs_var_info* will contain the augmented meta-data. This will also include information such as which variable is to be a netCDF axis for the observations.

4.3. Characterizing observation data in "raw" data files

The next step is to map the names of variables in the input data to the standard *obs_py* names. The relevant mappings are defined in an instance of the class from *obs_py.ob_network*. The mapping is defined in *obs_py/ob_network.yml* and this, along with any modification from a local version of *ob_network.yml* in the working directory will be invoked when the instance of *obs_py.ob_network* is created.

> **NOTE**: The copy in the working directory only needs to contain changes. This makes it easy to see what the differences are.

This YAML file also defines which variables are used to index observations in the DataFrame. The default is the MulitIndex: *date, time* and *station_index*.

This file also provides information on columns to skip (*nc_skip* key) and data transformations (*transform* key for a given parameter) and any station meta-data not included in the standard input files.

Details of configuring *obs_py* using these YAML files is given in 5 Configuring *obs_py* via YAML files

4.4. *odb2_df* and *csv_df*: Creation of pandas DataFrame

Using the information on observations from an instance of *obs_py.ob_network* and information on the variables written to the netCDF file from an instance of *obs_py.obfile_info* DataFrames are created by either *obs_py.odb2_df* or *obs_py.csv_df* .

4.4.1. *odb2_df* : Creating DataFrame from ODB2

*odb2_df(odb_file, netwk, fob_info, odb_filter=None, obs_type='*surface*')*

Routine Arguments	
odb_file	name of ODB2 file
netwk	instance of ob_network class
fob_info	instance of obs_var_info (observation file info)
odb_filter	output from make_odb_typefilter
obs_type	type of observations to use

A fairly straightforward routine.

Columns extracted from ODB2 file of surface observations into DataFrame are controlled by the *odb_surface* section in *ob_network.yml*.

df_index lists the *obs_py* observation elements to use as a mulit-index

obs_dict provides the necessary information on how to get *obs_py* observation elements from the ODB2 file, and the units (not stored in ODB2)

tzone is there for completeness as some other networks provide data in local time (empty value implies UTC)

Includes

- data conversions (*ob_convert*)
- copying of flags from one variable to another (e.g. from dew-point to relative humidty). Managed by including *mask_copy* as an key in the dictionary for an observation
    - *mask_copy* is set to the variable corresponding to the flag being copied appears in both the source and destination variables

WILL NEED TO ADD MORE SECTIONS FOR OTHER ODB2 TYPES

4.4.2. *csv_df* : Creating DataFrame from CSV

4.5. *df_nc*: Conversion from DataFrame to netCDF file

5. Configuring *obs_py* via YAML files

5.1. Class *obfile_info*: Defining obs_py variables/elements

One of reasons fort *obs_py* is that each model, set of observations etc. has its own name for the same variable or element of an observation. Alternatively many standards use truncated names (e.g. GRIB-1 4 character names). There are also times when observations may not even be a commonly used variable (e.g. vertical temperature difference between 1.5 and 10m). On top of this units vary.

Since *obs_py* processes determines which variables to process by which variables are in the original observed datafiles, the netCDF variable names are rarely typed in interactive runs. They mainly reside in files, and so the cost of typing out a longer variable name once into a file is not considered a great imposition.[1]

Mapping observed data to these *obs_py* names is done with the class *ob_network* (5.2)

The configuration is fairly standard YAML file, containing

- *file_attr*: a list of general file attributes to be created (used in *df_nc,* 4.5)
- *var_dict*: a dictionary of dictionaries containing information on each variable type, including
    - *obs_type:* which type of ODB2 obs_types the element refers to
    - *type*: numpy type and size
    - *units*: units to be used on netCDF file
    - *long_name*: full description of variable/element

[1] This also means it is very easy to examine information about a particular station, by dumping a netcdf file in full mode (i.e. with array indices included), i.e.

ncdump -f f <obsfile>.nc > nc.lst
Find the index/indices of the station
grep '<station_identifier>' nc.lst
Find all data about that report
grep '(<index>)' nc.lst
and it becomes very obvious what each value is

- *nc_axis*: denotes variable to use as the axis for netCDF file (best left reserved for *obs_index*)

5.2. Class *ob_network*: Data read from original observation files

What data is read from observation files is controlled via *ob_network.yml*. The data class *ob_network* provides the necessary utilities to read the data.

Each "observing network" has its own entry (e.g. *odb_surface, cotl*)

Includes information on

- Time zones (*tzone*)
- Variable transformations (*transform*)
- Units for each observed element (*units*)
  - Conversion occurs in *df_std_units*
- Mapping from reported name of observed elements to the names of *obs_py* elements (*obs_dict*)

  NOTE
  - ***obs_dict* for ODB is indexed by *obs_py* element name**
  - ***obs_dict* for CSV data is indexed by reported element name[2]**

- Elements not to be written (*nc_skip*)
- Which variables to use as an axis
  - *df_index* for a list of *obs_py* element names
  - *csv_axis* for a list of reported element names
- Transformations are described by the *transform* key in *obs_dict[csv_element_name]*
  - This will be another dictionary with keys referring to the *obs_py* element names that the reported data will be transformed into
  - Within this dictionary are items including
    - *func*: Name of the routine to perform the transformation
    - *units*: Units for the transformed variable
    - and any other variables or information used in the transformation
- Any meta data about the station not included in auxiliary files

5.3. Class *archive*: Location of gridded data

The starting point for finding gridded data is the nominal time of the observations. Given a nominal observation time, a nominal forecast length relative to the centre of a specified window about the observation time, and a specific field, the class *archive* provides the name of the relevant (netCDF) grid file and a file handle.

[2] Just noticed this little gotcha/anomaly. Reflects construction of these at different times (also ODB2 doesn't really have useful element names)

The class *archive* is made aware of various conventions around

- Variable names, and how they relate to *obs_py* element names
- File names, and how they relate to *obs_py* element names

via *archive.yml*

The path for all grid files is relative to the environment variable *NC_ARCH* which defaults to current working directory.

Note: Wild cards such as '*' and '?' along with environment variables can be used in filenames. Wild cards can be included in environment variables too, but should be preceded by '\'

Environment variables are resolved by the Python utility *os.path.expandvars*

Wild cards are resolved by *glob.glob*

Datetime formats are resolved by *datetime.datetime.strftime*

For each archiving convention, default values are given for:

- *time_type*: defines how times in the file are defined
  - *date_time*: integers in *YYYYMMDD HHmmss* format
  - *time_stamp*: dates in the form "<period> since <date>"
    - the default format for *<date>* is *YYYY-MM-DD HH:mm:ss*, although this can be specified independently
- *vtime_var*: name of variable containing grid valid time
- *btime_var*: name of variable containing grid forecast base (reference) time
  - if *btime_var* is the same as *vtime_var* then the reference date for *vtime_var* is used the basedate-time
  - otherwise the values of *btime_var* are read and used as increments to the reference datetime given in the *btime_var.units* attribute
- *vtime_ref_format:* format for reference date in *vtime_var* description
- *btime_ref_format:* format for reference date in *btime_var* description
- *fc_type*: the string used to distinguish different forecast type such as
  - *an*: analysis
  - *fc*: standard forecast
  - *fcmm*: 10minue data
  - etc.

  Basically gets edited into the name of the full path of the grid file as needed.

  Can be reset within local copy of *interp_ob.yml* if want to change from *fcmm* to *fc* or *an*

- *lev_type*: the string used to stratify files according to type of vertical information
  - *sfc:* for near surface data
  - *ml:* model level data
  - *pl:* pressure level data
  - etc.
- *pert_id:* any string added to the perturbation index in the path name

- the actual perturbation index is given by the environment variable *ENS_ID*, as this will often be a loop variable, and it is undesirable to update a YAML file for each pass through a loop (e.g. it prevents parallel loops)

These can also be specified for all variables, but also modified for individual netCDF variables.

For each *obs_py* element name, both:

- *fname*: the file name (without .nc suffix)
- *fvar_name*: the netCDF variable name

must be provided. Other exceptions or tailorings to the general values listed earlier can also be included by adding an appropriately named key

NOTE:

1. **Filenames can include wild cards. There just needs to be enough information to make the path and filename unique**
2. ***fc_type*, *lev_type, pert_id*, and *fname* are just strings that are edited into specific parts of the file path and name. They do not really need to refer to anything in particular, and can include "/" as well as wildcards. The names are just there to be reminders for where each string gets put into the full pathname.**
3. **New archive structures can be defined in user *archive.yml* files, BUT**
    a. **Filenames will be resolved to *arch_root/fname* where**
        i. ***arch_root* is defined when archive instance is initiated**
        ii. ***fname* fname needs to be the entire path relative to *arch_root***
    b. **This does allow the introduction of new mappings between variables and filenames**

5.4. Script *interp_ob.py*: Interpolation of model grids, and the creation of matched observation-grid pairs

This is the main script for generating matched pairs, and is rather extensive as it not only does horizontal interpolations but also generates corrections for differences between model and actual elevations at stations and converts between various variables.

The command line options are:

--input (-i)	netCDF file of observations
--output (-o)	netCDF file of matched observations and interpolated grid values
--forecast (-f)	Forecast length (for middle of observation window covered by input file)
--verbose (-v)	Verbosity level (Python logger level, default=20=INFO)

NOTE: If the data assimilation window is wider than the window of observations in an observation netCDF file the script needs to be run with different values of –*forecast* for the different nominal observation times.

This occurs when using grid files from ERA-5 are from a 12 hourly assimilation window, but Bureau ODB2 files, and hence the related netCDF files are for 6 hour windows. In this case 00 (or [21,03Z]) and 12Z (or [09Z,15Z]) observations than for the 06Z (or [03,09Z]) and 18Z (or [15Z,21Z]) observations. So there is a 6hour difference between the forecast length for 00Z/12Z nominal observation times and the 06Z/18Z nominal observation times.

Other than the command line options, and the two environment variables (*NC_ARCH* and possible *ENS_ID*) this script is configured by *interp_ob.yml*. This YAML file contains:

allow_all_sea	Include land-based observations where all surrounding grid points are sea or ice
auxiliary_field	A list of grid fields to be interpolated, even if no observations are directly related to them. • Useful for deriving other fields, and/or exploring the nature of differences between observations and forecasts.
da_window	Time (minutes) between different data assimilation cycles. • A grid_field is matched to observations in [-da_window/2, da_window/2)
date_range	Confine the processing to certain days (*YYYMMDD)*
del_auxiliary_field	List of fields to delete from default *auxiliary_field* list
del_transformed_field	List of fields to delete from default *transformed_field* list
dt_range	Confine the processing to a given datetime range (integer) • Datetime format given by *obs_py.dtf_grid()* • Default is *YYYMMDDhhmmss*
dt_str_range	Confine the processing to a given datetime range (integer) • Datetime format given by *obs_py.dtf_grid()*
fc_frequency	Frequency of update of forecasts • *Possibly to be deprecated?*
fc_step_unit	Units for command line forecast length option
file_dt_check	What sort of datetimes are used to determine if a gridfile contains the relevant fields. Options are • base_dt: checks date in timestamp • valid_dt: checks if obs within first and last valid datetimes
gfile_meta	Can over-ride settings in the class *archive* using these • E.g. some of the difference between *mars_ncdf.0* and *mars_ncdf.1* files could be achieved by just changing the value of *time_type*.
lat_range	Confine the processing to a range of latitudes
lon_range	Confine the processing to a range of longitudes • Should handle straddling Greenwich
max_base_dt	Confine the processing to a limited number of analysis basedate-times. • Mainly for debugging (especially when files have a month worth of observations)
max_check_file_iter	Some grid files may only be available within analysis data (e.g. topography, land-sea mask)

	If a file is missing, obs_py will search nearby assimilation cycles for a suitable file. This limits how far the search goes. Such files could be hard wired, but the search strategy means that if updates are applied, will use the right file (especially if include sea-ice)
stn_id	List of *station_identifier* to use. • Default is all. • Mainly for debugging/testing
time_range	Confine the processing to a certain period of the day • Should handle straddling midnight
transformed_field	List of variables to be derived from other variables • Code checks all variables required are present
use_end_dt	Option to include observations at the end of the da_window in the output file

6. Adding variables to *obs_py*

The steps involved in introducing a new variable (e.g. *new_var*) to *obs_py*

6.1. Define new variable *obfile_info.yml*

Create an entry in *obsfile_info.yml* with the necessary meta-data for the variable. There will generally be a similar variable to use as a template.

Need to add separate entries for observations, background fields and any height corrected fields using *obs_<new_var>*, *bkg_<new_var>* and *adj_<new_var>*

6.2. Map observations to *new_var* using *ob_network.yml*

Add a mapping from the relevant source of observations to *new_var*. Again there should be a similar variable that can provide a template

6.3. Provide information on relevant grids in *archive.yml*

Add mappings for any relevant variables into *archive.yml* .

- If *new_var* is model variable then add an entry for *new_var*
- Otherwise, add entries for any new grids required

6.4. Ensure relevant grids are interpolated with *interp_ob.yml*

If *new_var* has a directly equivalent variable in the archive, it will automatically be interpolated.

If the modelled value needs to be derived:

- Add the input grids into the *auxiliary_field* list in *interp_ob.yml*
    - These may also need new entries in *archive.yml* if they are also new
- Add *new_var* to the *derived field* list in *interp_ob.yml*
- Add the necessary code into derive the fields in *interp_ob.py*
    - Again there will probably be related fields to provide a template

7. Archive structures to *archive.py*

Archives generally have various information such as basedate-times, level type, forecast type, ensemble perturbation etc. encoded within the path, as well as various filenames for different variables. In order to find the appropriate grid data, *obs_py* needs to know all this information. This information is included in *archive.yml*

Existing archive structures are

mars_ncdf.0	File given by: *$NC_ARCH/YYYYMMDD/HH00/<fc_type>/<lev_type>/<fname>.nc* Base date is from a variable given by *bdate_var* Valid date is from a variable given by v*date_var* • Dates are integers, YYYYMMDD Base time is from a variable given by *btime_var* Valid time is from a variable given by *vtime_var* • Times are integers HHMMSS Default • *fc_type*: fcmm • *lev_type* sfc • *pert_id*: None
mars_ncdf.1	As for *mars_ncdf.0*, but Base-datetime is from a time stamp variable *"<period> since <reference_date>"* Valid-datetime is from a time stamp variable *"<period> since <reference_date>"*
iris_ncdf.0	Files generated by IRIS File given by: *$NC_ARCH/yyyymmddThh00Z/nc/<fc_type>/<fname>-<lev_type>- yyyymmddThh00Z.nc* Base-datetimes & valid-datetimes are timestamps
era5_ncdf.0	Files from ERA-5 Base-datetimes & valid-datetimes are timestamps • By default, the variables for base datetime and valid datetime are the same so base datetime is the reference time for valid datetime timestamp. Can be over-ridden Files given by: *$NC_ARCH/yyyy/mm/dd/<fname>.<lev_type>-.yyyymmddhh.nc* File for different ensemble members are given by *$NC_ARCH/yyyy/mm/dd/<fname>.<pert_id>$ENS_ID.<lev_type>.yyyymmddhh.nc*
era5_monthly.0	Files from ERA-5 Base-datetimes & valid-datetimes are timestamps • By default, the variables for base datetime and valid datetime are the same so base datetime is the reference time for valid datetime timestamp. Can be over-ridden Files given by: *$NC_ARCH/single-levels/reanalysis/**V**/%Y/**V**_era5_oper_sfc_%Y%m01-%Y%m??.nc* Where ***V*** is the variable used in filenames
barra2_ncdf.0	Files from BARRA-2

	Base-datetimes & valid-datetimes are timestamps • By default base-datetime comes from a different variable to valid datetime. Can be over-ridden. **Default filenames include wild cards** Files given by: *$NC_ARCH/yyyymmddThh00Z/<lev_type>/<fname>* File for different ensemble members are given by *$NC_ARCH/yyyymmddThh00Z/pert_id$ENSID/<lev_type>/<fname>*