pysatModels Documentation

Release 0.1.0-alpha

Burrell, Angeline G.; Klenzing, Jeff; Stoneback, Russell

May 20, 2022

CONTENTS

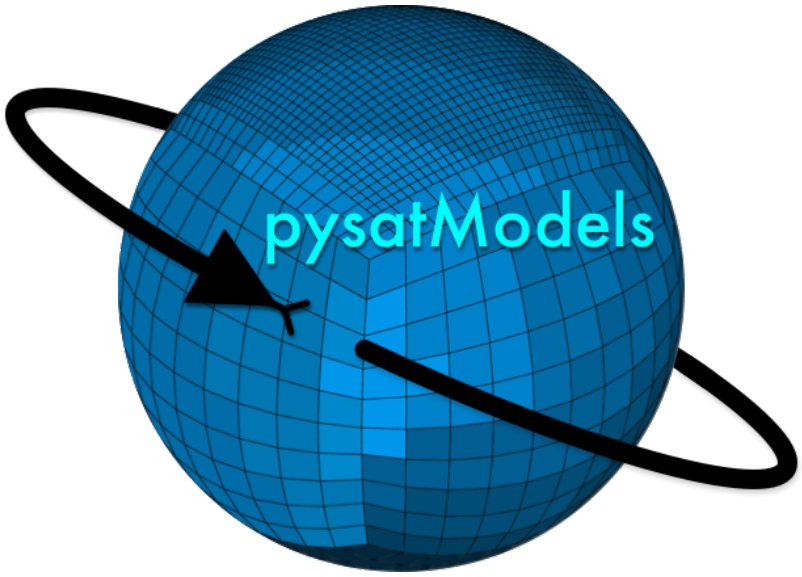
This documentation describes the pysatModels module, which contains routines to load model data as pysat.Instrument objects and utilities to perform typical model-oriented analysis, such as model validation.

CHAPTER

ONE

OVERVIEW

Working together with modelled and observational data sets can be challenging. Modelled data sets can be very large, with each model populating data along a different grid. This package provides a variety of tools to aid model-data analysis. This includes tools for model valiation, modelling instrument output, and incorporating modelled data sets into general data analysis routines written to use data stored in pysat.Instrument objects or xarray.Datasets.



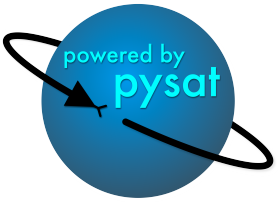
CHAPTER

TWO

INSTALLATION

The following instructions will allow you to install pysatModels.

2.1 Prerequisites



pysatModels uses common Python modules, as well as modules developed by and for the Space Physics community. This module officially supports Python 3.6+.

Common modules	Community modules
numpy	pysat
pandas	pyForecastTools
requests	
scipy	
xarray	

2.2 Installation Options

1. Clone the git repository

```
git clone https://github.com/pysat/pysatModels.git
```

2. Install pysatModels: Change directories into the repository folder and run the setup.py file. There are a few ways you can do this:

   A. Install on the system (root privileges required):

   ```
   sudo python3 setup.py install
   ```

   B. Install at the user level:

   ```
   python3 setup.py install --user
   ```

C. Install with the intent to develop locally:

```
python3 setup.py develop --user
```

2.3 Post Installation

After installation, you may register the `pysatModel` model `Instrument` sub-modules with pysat. If this is your first time using pysat, check out the quickstart guide for pysat. Once pysat is set up, you may choose to register the the `pysatModel` model `Instruments` sub-modules by:

```
import pysat
import pysatModels as pymod

pysat.utils.registry.register_by_module(pymod.models)
```

You may then use the pysat `platform` and `name` keywords to initialize the model `Instrument` instead of the `inst_module` keyword argument.

CHAPTER

THREE

CITATION GUIDELINES

When publishing work that uses pysatModels, please cite the package and any package it depends on that plays an important role in your analysis. Specifying which version of pysatModels used will also improve the reproducibility of your presented results.

3.1 pysatModels

- Burrell, A. G., et al. (2022). pysat/pysatModels: Alpha Release (Version 0.1.0). https://github.com/pysat/pysatModels

```
@Misc{pysatModels,
      author = {Burrell, A. G. and Stoneback, R. and Klenzing, J. H.},
      title  = {pysat/pysatModels: Alpha Release},
      year   = {2020},
      date   = {2020-04-01},
      url    = {https://github.com/pysat/pysatModels},
      }
```

CHAPTER

FOUR

SUPPORTED MODELS

4.1 DINEOF

Supports the pyDINEOF model output. pyDINEOF is a Python implementation of the Data INterpolating Emperical Orthogonal Function method. Information about this package can obtained by contacting Russell Stoneback.

Support exported model data from pyDINEOF.

4.1.1 Properties

platform 'pydineof'

name 'dineof'

tag '', 'test'

inst_id ''

Note: pyDINEOFs is a Python package that interfaces with a version of Data Interpolation Empirical Orthogonal Functions (DINEOFs). This module couples into the systematic export pyDINEOF format and thus should support all exports from the package.

Specific tags are not listed here as this method is intended to support all pyDINEOF export models. Place the desired model (daily files) at '{pysat_data_dir}/pydineof/dineof/{tag}'. Each model series is identified using the *tag* keyword. It is presumed the default naming scheme of 'dineof_{year:04d}-{month:02d}-{day:02d}.nc' has been retained. Use the *file_format* option for custom filenames.

DINEOFs are a purely data based method that can analyze a data-set, with data gaps, and extract a series of basis functions that optimally reproduce the input data. The quality of the reconstruction is primarily determined by the quantity and quality of the input data.

References

J.-M. Beckers and M. Rixen. EOF calculations and data filling from incomplete oceanographic data sets. Journal of Atmospheric and Oceanic Technology, 20(12):1839-1856, 2003.

`pysatModels.models.pydineof_dineof.`**`download`**(*date_array*, *tag*, *inst_id*, *data_path*)

Download pydineof data.

Parameters

- **date_array** (*array-like*) – List of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*str*) – Tag identifier used for particular dataset. This input is provided by pysat.
- **inst_id** (*str*) – Instrument ID string identifier used for particular dataset. This input is provided by pysat.
- **data_path** (*str*) – Path to directory where download data will be stored.

Note: This routine is invoked by pysat and is not intended for direct use by the end user. Currently only retrieves test data from github.

The test object generates the datetime requested by the user, which may not match the date of the model run.

Examples

```
import datetime as dt
import pysat

inst = pysat.Instrument('pydineof', 'dineof', 'test')
inst.download(start=dt.datetime(2009, 1, 1))
```

pysatModels.models.pydineof_dineof.**init**(*self*)

Initialize the Instrument object with instrument specific values.

pysatModels.models.pydineof_dineof.**load**(*fnames*, *tag=''*, *inst_id=''*, ***kwargs*)

Load pydineof data using xarray.

Parameters

- **fnames** (*array-like*) – Iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*str*) – Tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **inst_id** (*str*) – Instrument ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- ****kwargs** (*dict*) – Pass-through for additional keyword arguments specified when instantiating an Instrument object. These additional keywords are passed through to this routine by pysat.

Returns

- **data** (*xarray.Dataset*) – pysat formatted xarray Dataset
- **meta** (*pysat.Meta*) – Model run meta data

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine.

Examples

```
inst = pysat.Instrument(inst_module=pysatModels.models.pydineof_dineof)
inst.load(2019, 1)
```

4.2 SAMI2

Supports the SAMI2 (Sami2 is Another Model of the Ionosphere 2) model through the sami2py interface. Sami2py is a python module that runs the SAMI2 model, as well as archives, loads and plots the resulting modeled values. SAMI2 is a model developed by the Naval Research Laboratory to simulate the motions of plasma in a 2D ionospheric environment along a dipole magnetic field [Huba et al, 2000]. Information about this model can be found at the sami2py github page, along with a list of the SAMI2 principle papers.

Support loading data from files generated using the sami2py model.

sami2py file is a netCDF file with multiple dimensions for some variables. The sami2py project is at https://github.com/sami2py/sami2py

4.2.1 Properties

platform 'sami2py'

name 'sami2'

tag '', 'test'

inst_id ''

pysatModels.models.sami2py_sami2.**download**(*date_array*, *tag*, *inst_id*, *data_path*)

Download sami2py data.

Parameters

- **date_array** (*array-like*) – List of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*str*) – Tag identifier used for particular dataset. This input is provided by pysat.
- **inst_id** (*str*) – Instrument ID string identifier used for particular dataset. This input is provided by pysat.
- **data_path** (*str*) – Path to directory to download data to.

Note: This routine is invoked by pysat and is not intended for direct use by the end user.

The test object generates the datetime requested by the user, which may not match the date of the model run.

Examples

```
import datetime as dt
import pysat

inst = pysat.Instrument('sami2py', 'sami2', 'test')
inst.download(start=dt.datetime(2020, 3, 8))
```

pysatModels.models.sami2py_sami2.**init**(*self*)

Initialize the Instrument object with instrument specific values.

pysatModels.models.sami2py_sami2.**load**(*fnames*, *tag=''*, *inst_id=''*, ***kwargs*)

Load sami2py data using xarray.

This routine is called as needed by pysat. It is not intended for direct user interaction.

Parameters

- **fnames** (*array-like*) – Iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*str*) – Tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **inst_id** (*str*) – Instrument ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- ****kwargs** (*dict*) – Passthrough for additional keyword arguments specified when instantiating an Instrument object. These additional keywords are passed through to this routine by pysat.

Returns

- **data** (*xarray.Dataset*) – pysat formatted xarray Dataset
- **meta** (*pysat.Metadata*) – Model run meta data

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine.

Examples

```
inst = pysat.Instrument('sami2py', 'sami2')
inst.load(2019, 1)
```

4.3 TIE-GCM

Supports the UCAR (University Corporation for Atmospheric Research) model, Thermosphere-Ionosphere-Electrodynamics General Circulation Model (TIE-GCM). Information about this model can be found at the UCAR TIE-GCM website, along with a list of the principle papers and references.

Support loading data from files generated using TIEGCM model.

TIEGCM (Thermosphere Ionosphere Electrodynamics General Circulation Model) file is a netCDF file with multiple dimensions for some variables.

4.3.1 Properties

platform 'ucar'

name 'tiegcm'

tag ''

inst_id ''

pysatModels.models.ucar_tiegcm.**download**(*date_array*, *tag*, *inst_id*, *data_path=None*, ***kwargs*)

Download UCAR TIE-GCM (placeholder). Doesn't do anything.

Parameters

- **date_array** (*array-like*) – List of datetimes to download data for. The sequence of dates need not be contiguous.
- **tag** (*str*) – Tag identifier used for particular dataset. This input is provided by pysat.
- **inst_id** (*str*) – Instrument ID string identifier used for particular dataset. This input is provided by pysat.
- **data_path** (*str or NoneType*) – Path to directory to download data to. (default=None)
- ****kwargs** (*dict*) – Additional keywords supplied by user when invoking the download routine attached to a pysat.Instrument object are passed to this routine via kwargs.

Note: This routine is invoked by pysat and is not intended for direct use by the end user.

pysatModels.models.ucar_tiegcm.**init**(*self*)

Initialize the Instrument object with instrument specific values.

pysatModels.models.ucar_tiegcm.**load**(*fnames*, *tag=''*, *inst_id=''*, ***kwargs*)

Load TIE-GCM data using xarray.

Parameters

- **fnames** (*array-like*) – Iterable of filename strings, full path, to data files to be loaded. This input is nominally provided by pysat itself.
- **tag** (*str*) – Tag name used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')
- **inst_id** (*str*) – Instrument ID used to identify particular data set to be loaded. This input is nominally provided by pysat itself. (default='')

- ****kwargs** (*dict*) – Passthrough for additional keyword arguments specified when instantiating an Instrument object. These additional keywords are passed through to this routine by pysat.

Returns

- **data** (*xarray.Dataset*) – pysat formatted xarray Dataset
- **meta** (*pysat.Metadata*) – Model run meta data

Note: Any additional keyword arguments passed to pysat.Instrument upon instantiation are passed along to this routine.

Examples

```
inst = pysat.Instrument('ucar', 'tiegcm')
inst.load(2019, 1)
```

CHAPTER

FIVE

UTILITIES

5.1 Convert

Routines to support extracting pysat.Instrument data as xarray.Datasets.

pysatModels.utils.convert.**convert_pysat_to_xarray**(*inst*)

Extract data from a model Instrument as a Dataset with metadata.

Parameters **inst** (*pysat.Instrument*) – Model instrument object

Returns **inst_data** – Dataset from pysat Instrument object or None if there is no data

Return type xarray.Dataset or NoneType

pysatModels.utils.convert.**load_model_xarray**(*ftime*, *model_inst=None*, *filename=None*)

Load and extract data from a model Instrument at the specified time.

Parameters

- **ftime** (*dt.datetime*) – Desired time for model Instrument input
- **model_inst** (*pysat.Instrument*) – Model instrument object
- **filename** (*str or NoneType*) – Model filename, if the file is not include in the Instrument filelist. or a filename that requires time specification from ftime (default=None)

Returns **model_xarray** – Dataset from pysat Instrument object or None if there is no data

Return type xarray.Dataset or NoneType

5.2 Match

Routines to match modelled and observational data.

pysatModels.utils.match.**collect_inst_model_pairs**(*start*, *stop*, *tinc*, *inst*, *inst_download_kwargs=None*, *model_load_rout=<function load_model_xarray>*, *model_load_kwargs=None*, *inst_clean_rout=None*, *inst_lon_name=None*, *mod_lon_name=None*, *lon_pos='end'*, *inst_name=None*, *mod_name=None*, *mod_datetime_name=None*, *mod_time_name=None*, *mod_units=None*, *sel_name=None*, *time_method='min'*, *pair_method='closest'*, *method='linear'*, *model_label='model'*, *comp_clean='clean'*)

Pair instrument and model data.

Parameters

- **start** (*dt.datetime*) – Starting datetime
- **stop** (*dt.datetime*) – Ending datetime
- **tinc** (*dt.timedelta*) – Time incriment for model files
- **inst** (*pysat.Instrument*) – Instrument object for which modelled data will be extracted
- **inst_download_kwargs** (*dict or NoneType*) – Optional keyword arguments for downloading instrument data (default=None)
- **model_load_rout** (*func*) – Routine to load model data into an xarray using datetime as argument input input and other necessary data as keyword arguments. If the routine requires a time-dependent filename, ensure that the load routine uses the datetime input to construct the correct filename, as done in load_model_xarray. (default=load_model_xarray)
- **model_load_kwargs** (*dict or NoneType*) – Keyword arguments for the model loading routine. (default=None)
- **inst_clean_rout** (*func*) – Routine to clean the instrument data. (default=None)
- **inst_lon_name** (*str*) – variable name for instrument longitude
- **mod_lon_name** (*str*) – variable name for model longitude
- **lon_pos** (*str or int*) – Accepts zero-offset integer for list order or 'end' (default='end')
- **inst_name** (*list or NoneType*) – List of names of the data series to use for determing instrument location. (default=None)
- **mod_name** (*list or NoneType*) – List of names of the data series to use for determing model locations in the same order as inst_name. These must make up a regular grid. (default=None)
- **mod_datetime_name** (*str*) – Name of the data series in the model Dataset containing datetime info
- **mod_time_name** (*str*) – Name of the time coordinate in the model Dataset
- **mod_units** (*list or NoneType*) – Units for each of the mod_name location attributes. Currently supports: rad/radian(s), deg/degree(s), h/hr(s)/hour(s), m, km, and cm. (default=None)
- **sel_name** (*list or NoneType*) – list of names of modelled data indices to append to instrument object, or None to append all modelled data (default=None)
- **time_method** (*str*) – Pair data using larger (max) or smaller (min) of the smallest instrument/model time increments (default='min')
- **pair_method** (*str*) – Find all relevent pairs ('all') or just the closest pairs ('closest'). (default='closest')
- **method** (*str*) – Interpolation method. Supported are 'linear', 'nearest', and 'splinef2d'. The last is only supported for 2D data and is not recommended here. (default='linear')
- **model_label** (*str*) – name of model, used to identify interpolated data values in instrument (default="model")
- **comp_clean** (*str*) – Clean level for the comparison data ('clean', 'dusty', 'dirty', 'none') (default='clean')

Returns

matched_inst –

Instrument object with observational data from *inst* and paired modelled data.

Return type pysat.Instrument

Raises `ValueError` – If input is incorrect

Note: Perform the data cleaning after finding the times and locations where the observations and model align.

5.3 Extract

Routines to extract observational-style data from model output.

pysatModels.utils.extract.**extract_modelled_observations**(*inst*, *model*, *inst_name*, *mod_name*, *mod_datetime_name*, *mod_time_name*, *mod_units*, *sel_name=None*, *time_method='min'*, *pair_method='closest'*, *method='linear'*, *model_label='model'*, *model_units_attr='units'*)

Extract instrument-aligned data from a modelled data set.

Parameters

- **inst** (*pysat.Instrument*) – Instrument object for which modelled data will be extracted
- **model** (*xarray.Dataset*) – Modelled data set
- **inst_name** (*array-like*) – List of names of the data series to use for determining instrument location
- **mod_name** (*array-like*) – List of names of the data series to use for determining model locations in the same order as inst_name. These must make up a regular grid.
- **mod_datetime_name** (*str*) – Name of the data series in the model Dataset containing datetime info
- **mod_time_name** (*str*) – Name of the time coordinate in the model Dataset
- **mod_units** (*list of strings*) – Units for each of the mod_name location attributes. Currently supports: rad/radian(s), deg/degree(s), h/hr(s)/hour(s), m, km, and cm
- **sel_name** (*array-like or NoneType*) – list of names of modelled data indices to append to instrument object, or None to append all modelled data (default=None)
- **time_method** (*str*) – Pair data using larger (max) or smaller (min) of the smallest instrument/model time increments (default='min')
- **pair_method** (*str*) – Find all relevent pairs ('all') or just the closest pairs ('closest'). (default='closest')
- **method** (*str*) – Interpolation method. Supported are 'linear', 'nearest', and 'splinef2d'. The last is only supported for 2D data and is not recommended here. (default='linear')
- **model_label** (*str*) – name of model, used to identify interpolated data values in instrument (default="model")
- **model_units_attr** (*str*) – Attribute for model xarray values that contains units (default='units')

Returns **interp_data.keys()** – List of keys of modelled data added to the instrument

Return type list

Raises `ValueError` – For incorrect input arguments

Notes

For best results, select clean instrument data after alignment with model

pysatModels.utils.extract.**instrument_altitude_to_model_pressure**(*inst*, *model*, *inst_name*, *mod_name*, *mod_datetime_name*, *mod_time_name*, *mod_units*, *inst_alt*, *mod_alt*, *mod_alt_units*, *scale=100.0*, *inst_out_alt='model_altitude'*, *inst_out_pres='model_pressure'*, *tol=1.0*)

Interpolates altitude values onto model pressure levels.

Parameters

- **inst** (*pysat.Instrument*) – Instrument object with observational data
- **model** (*xarray.Dataset*) – Model data in xarray format
- **inst_name** (*array-like*) – List of variable names containing the observational data coordinates at which the model data will be interpolated. Must be in the same order as mod_name.
- **mod_name** (*array-like*) – list of names of the coordinates to use for determining model locations. Must be in the same order as *mod_alt* is stored within xarray. The coordinates must make up a regular grid.
- **mod_datetime_name** (*str*) – Name of the data series in the model Dataset containing datetime info
- **mod_time_name** (*str*) – Name of the time coordinate in the model Dataset
- **mod_units** (*list*) – units for each of the mod_name location attributes. Currently supports: rad/radian(s), deg/degree(s), h/hr(s)/hour(s), m, km, and cm
- **inst_alt** (*str*) – String identifier used in inst for the altitude variable
- **mod_alt** (*str*) – Variable identifier for altitude data in the model e.g. 'ZG' in standard TIEGCM files.
- **mod_alt_units** (*str*) – units for the altitude variable. Currently supports: m, km, and cm
- **scale** (*float*) – Scalar used to roughly translate a change in altitude with a change in pressure level, the scale height. Same units as used by inst. (default=100.)
- **inst_out_alt** (*str*) – Label assigned to the model altitude data when attached to inst (default='model_altitude').
- **inst_out_pres** (*str*) – Label assigned to the model pressure level when attached to inst (default='model_pressure').
- **tol** (*float*) – Allowed difference between observed and modelled altitudes. Interpreted to have the same units as *inst_alt* (default=1.0).

Returns **[inst_out_alt, inst_out_pres]** – List of keys corresponding to the modelled data that was added to the instrument.

Return type list

Raises `ValueError` – For incorrect input arguments

Notes

Uses an iterative regular grid interpolation to find the appropriate pressure level for the given input locations.

pysatModels.utils.extract.**instrument_view_through_model**(*inst*, *model*, *inst_name*, *mod_name*, *mod_datetime_name*, *mod_time_name*, *mod_units*, *sel_name=None*, *methods=['linear']*, *model_label='model'*)

Interpolates model values onto instrument locations.

Parameters

- **inst** (*pysat.Instrument*) – Instrument object with observational data
- **model** (*xarray.Dataset*) – Modelled data
- **inst_name** (*array-like*) – List of variable names containing the observational data coordinates at which the model data will be interpolated. Do not include 'time', only spatial coordinates.
- **mod_name** (*array-like*) – List of model dimension names used for organizing model data in the same order as *inst_name*. These must make up a regular grid. Do not include 'time', only spatial dimensions.
- **mod_datetime_name** (*str*) – Name of the data series in the model Dataset containing datetime info.
- **mod_time_name** (*str*) – Name of the time coordinate in the model Dataset.
- **mod_units** (*list*) – Units for each of the mod_name location dimensions. Currently supports: rad/radian(s), deg/degree(s), h/hr(s)/hour(s), m, km, and cm
- **sel_name** (*array-like or NoneType*) – List of names of modelled data indices to append to Instrument object, or None to append all modelled data. (default=None)
- **methods** (*str*) – 'linear' interpolation or 'nearest' neighbor options for RegularGrid. Must supply an option for each variable. (default=['linear'])
- **model_label** (*str*) – Name of model, used to identify interpolated data values in instrument (default="model")

Returns **interp_data.keys()** – Keys of modelled data added to the instrument

Return type Keys

Raises `ValueError` – For incorrect input arguments

Note: Updates the inst Instrument with interpolated data from the model Instrument. The interpolation is performed via the RegularGridInterpolator for quick performance.

This method may require the use of a pre-processor on coordinate dimensions to ensure that a regular interpolation may actually be performed.

Models, such as TIEGCM, have a regular grid in pressure, not in altitude. To use this routine for TIEGCM please use instrument_altitude_to_model_pressure first to transform instrument altitudes to pressure levels suitable for this method.

Variables that vary exponentially in height may be approximated by taking a log before interpolating, though this does also yield an exponential variation along the horizontal directions as well.

Expects units strings to have the units as the first word, if a long description is provided (e.g., 'degrees', 'degrees North', or 'deg_N' and not 'geographic North degrees')

See also:

`pysat.utils.scale_units`

pysatModels.utils.extract.**interp_inst_w_irregular_model_coord**(*inst*, *model*, *inst_name*, *mod_name*, *mod_datetime_name*, *mod_units*, *mod_reg_dim*, *mod_irreg_var*, *mod_var_delta*, *sel_name=None*, *model_label='model'*)

Interpolate irregular-coordinate model data onto Instrument path.

Parameters

- **inst** (*pysat.Instrument*) – pysat object that will receive interpolated data based upon position.
- **model** (*pysat.Instrument*) – Xarray pysat Instrument with model data that will be interpolated onto the *inst* locations.
- **inst_name** (*list*) – List of variable names containing the instrument data coordinates at which the model data will be interpolated. Do not include 'time', only spatial coordinates. Same ordering as used by mod_name.
- **mod_name** (*list*) – List of names of the data dimensions used to organize model data, in the same order as *inst_name*. These dimensions must make up a regular grid. Values from *mod_irreg_var* will be used to replace one of these regular dimensions, *mod_reg_dim*, with irregular values.
- **mod_datetime_name** (*str*) – Name of the data series in the model Dataset containing datetime info.
- **mod_units** (*list*) – Units for each of the *mod_name* dimensions. Users must provide units for *mod_irreg_var' instead of the units for `mod_reg_dim*. Currently supports: rad/radian(s), deg/degree(s), h/hr(s)/hour(s), m, km, and cm.
- **mod_reg_dim** (*str*) – Existing regular dimension name (must be in *mod_name*) used to organize model data that will be replaced with values from *mod_irreg_var* before performing interpolation.
- **mod_irreg_var** (*str*) – Variable name in model used to define the irregular grid value locations along *mod_reg_dim*. Must have same dimensions as *mod_name*.
- **mod_var_delta** (*list*) – List of delta values to be used when downselecting model values before interpolation, max(min(inst) - delta, min(model)) <= val <= min(max(inst) + delta, max(model)). Interpreted in the same order as *mod_name*.
- **sel_name** (*list*) – List of strings denoting model variable names that will be interpolated onto inst. The coordinate dimensions for these variables must correspond to those in *mod_irreg_var*.
- **model_label** (*str*) – Name of model, used to identify interpolated data values in instrument (default="model")

Returns **output_names** – Keys of interpolated model data added to the instrument

Return type list

Raises ValueError – For incorrect input arguments

Notes

Expects units strings to have the units as the first word, if a long description is provided (e.g., 'degrees', 'degrees North', or 'deg_N' and not 'geographic North degrees').

See also:

`pysat.utils.scale_units`

5.4 Compare

Routines to align and work with pairs of modelled and observational data.

pysatModels.utils.compare.**compare_model_and_inst**(*pairs*, *inst_name*, *mod_name*, *methods=['all']*, *unit_label='units'*)

Compare modelled and measured data.

Parameters

- **pairs** (*xarray.Dataset*) – Dataset containing only the desired observation-model data pairs
- **inst_name** (*list*) –

  Ordered list of strings indicating whicch instrument measurements to compare to modelled data
- **mod_name** (*list*) – Ordered list of strings indicating which modelled data to compare to instrument measurements
- **methods** (*list*) – Statistics to calculate. See Notes for accecpted inputs. (default=['all'])
- **unit_label** (*str*) – Unit attribute for data in *pairs* (default='units')

Returns

- **stat_dict** (*dict*) – Dict of dicts where the first layer of keys denotes the instrument data name and the second layer provides the desired statistics
- **data_units** (*dict*) – Dict containing the units for the data

Raises ValueError – If input parameters are improperly formatted

See also:

`PyForecastTools`

Notes

Statistics are calculated using PyForecastTools (imported as verify).

1. all: all statistics
2. all_bias: bias, meanPercentageError, medianLogAccuracy, symmetricSignedBias
3. accuracy: returns dict with mean squared error, root mean squared error, mean absolute error, and median absolute error
4. scaledAccuracy: returns dict with normaled root mean squared error, mean absolute scaled error, mean absolute percentage error, median absolute percentage error, median symmetric accuracy
5. bias: scale-dependent bias as measured by the mean error
6. meanPercentageError: mean percentage error
7. medianLogAccuracy: median of the log accuracy ratio
8. symmetricSignedBias: Symmetric signed bias, as a percentage
9. meanSquaredError: mean squared error
10. RMSE: root mean squared error
11. meanAbsError: mean absolute error
12. medAbsError: median absolute error
13. nRMSE: normaized root mean squared error
14. scaledError: scaled error (see PyForecastTools for references)
15. MASE: mean absolute scaled error
16. forecastError: forecast error (see PyForecastTools for references)
17. percError: percentage error
18. absPercError: absolute percentage error
19. logAccuracy: log accuracy ratio
20. medSymAccuracy: Scaled measure of accuracy
21. meanAPE: mean absolute percentage error
22. medAPE: median absolute perceentage error

CHAPTER

SIX

MODEL METHODS

6.1 General

General functions for model instruments.

pysatModels.models.methods.general.**clean**(*inst*)

Raise a low-level log message about lack of cleaning.

pysatModels.models.methods.general.**download_test_data**(*remote_url*, *remote_file*, *data_path*, *test_date=None*, *format_str=None*)

Download test data from an online repository.

Parameters

- **remote_url** (*str*) – URL of the target repository, including the path to the test file
- **remote_file** (*str*) – Remote file name
- **data_path** (*str*) – Path to directory where local file will be stored
- **test_date** (*dt.datetime or NoneType*) – Datetime for which the test file will be assigned, does not need to correspond to the test model run time. Only used if *format_str* is also provided. (default=None)
- **format_str** (*str or NoneType*) – Format string to construct a pysat-compatible filename or None to not change the filename (default=None)

Note: This routine is invoked by pysat and is not intended for direct use by the end user.

The test object generates the datetime requested by the user, which may not match the date of the model run.

CHAPTER

SEVEN

EXAMPLES

Here are some examples that demonstrate how to use various pysatModels tools. These examples use the following python modules:

Standard	Outside	pysat ecosystem
datetime os	matplotlib numpy pandas	pysat pysatMadrigal pysatModels pysatNASA

7.1 Loading Model Data

7.1.1 Load Model Data into a pysat Instrument

pysatModels uses pysat to load modelled data sets. As specified in the pysat tutorial, data may be loaded using the following commands. TIE-GCM is used as an example, and so to execute this code snippet the user will need to obtain a TIE-GCM data file from UCAR.

```
import pysat
import pysatModels as ps_mod

filename = 'tiegcm_filename.nc'
tiegcm = pysat.Instrument(inst_module=ps_mod.models.ucar_tiegcm)
tiegcm.load(fname=filename)
```

7.1.2 Load Model Data into an xarray Dataset

There are situations (such as when developing a new model) when it may be inconvenient to create a pysat Instrument object for a modelled data set. Many of the pysatModels utilities allow `xarray.Dataset` objects as input. For these routines or to retrieve an `xarray.Dataset` for other purposes, you can use the `load_model_xarray()` routine in *Match*.

In this example, the time is irrelevent because a full filename is provided:

```
import datetime as dt
import pysat
import pysatModels as ps_mod

# Data directory definition is needed if you don't save the TIE-GCM file
# to your pysat_data/ucar/tiegcm/ directory.  This definition assumes the
# file specified by `filename` lives in your current working directory.
```

(continues on next page)

(continued from previous page)

```
data_dir = '.'

# Define the file name, time, and initialize the instrument
ftime = dt.datetime(2010, 1, 1)
filename = 'tiegcm_filename.nc'
tiegcm = pysat.Instrument(platform='ucar', name='tiegcm', data_dir=data_dir)

tg_dataset = ps_mod.utils.match.load_model_xarray(ftime, tiegcm, filename)
```

In this example, the filename includes temporal information, which is provided within the loading function by the input time:

```
import datetime as dt
import pysat
import pysatModels as ps_mod

ftime = dt.datetime(2010, 1, 1)
filename = 'tiegcm_%Y%j.nc'
tiegcm = pysat.Instrument(platform='ucar', name='tiegcm')

tg_dataset = ps_mod.utils.match.load_model_xarray(ftime, tiegcm, filename)
```

In this example, the routine takes advantage of the pysat file organization system, and will return a `NoneType` object if no files are found for the specified time:

```
import datetime as dt
import pysat
import pysatModels as ps_mod

ftime = dt.datetime(2010, 1, 1)
tiegcm = pysat.Instrument(platform='ucar', name='tiegcm')

tg_dataset = ps_mod.utils.match.load_model_xarray(ftime, tiegcm)
```

7.2 Pair Modelled and Observed Data

One common analytical need is to obtain a combined data set of modelled and observed observations at the same times and locations. This can be done using the `collect_inst_model_pairs()` function in *Match*. This routine takes a date range as input, and then extracts modelled observations at the specified instrument location. However, it does not interpolate in time. Details about the interpolation of modelled data onto the instrument location can be found in the *Extract* routine, `extract_modelled_observations()`.

7.2.1 Match Data by Location

In the example below, we load a DINEOFs file created for testing purposes and pair it with C/NOFS IVM data. This is a global example for a single time, since in this instance DINEOFs was used to create a day-specific emperical model. Comparisons with output from a global circulation model would look different, as one would be more likely to desire the the closest observations to the model time rather than all observations within the model time.

This example uses the external modules:

1. pysat
2. pysatNASA

Load a test model file from pyDINEOFs. It can be downloaded from the GitHub repository using the standard `pysat.Instrument.download()` method. We will also set the model-data input keyword arguements that are determined by the model being used. The DINEOFs test data applies for an entire day, so the `pair_method` is `'all'` and the `time_method` is `'max'`.

```
import datetime as dt
from os import path
import pysat
import pysatNASA
import pysatModels as ps_mod

# Set the instrument inputs
inst_module = ps_mod.models.pydineof_dineof
inst_id = ''
tag = 'test'

# Initialize the model input information, including the keyword arguements
# needed to load the model Instrument into an xarray Dataset using
# the match.load_model_xarray routine
stime = inst_module._test_dates['']['test']
tinc = dt.timedelta(days=1)  # Required input is not used in this example
dineofs = pysat.Instrument(inst_module=inst_module, tag=tag, inst_id=inst_id)

# If you don't have the test file, download it
try:
    filename = dineofs.files.files[stime]
except KeyError:
    dineofs.download(stime, stime)
    filename = dineofs.files.files[stime]

# Set the input keyword args for the model data
input_kwargs = {"model_load_kwargs": {'model_inst': dineofs,
                                      'filename': filename},
                "mod_lon_name": "lon",
                "mod_name": [None, None, "lon", "lt"],
                "mod_units": ["km", "deg", "deg", "h"],
                "mod_datetime_name": "time",
                "mod_time_name": "time",
                "model_label": dineofs.name,
                "pair_method": "all",
                "time_method": "max",
                "sel_name": "model_equator_model_data"}
```

Next, get observational data to match with the model data. In this example, we will use C/NOFS-CINDI IVM data, since the DINEOFs test file contains meridional **E** x **B** drift values.

```
# Initialize the CINDI instrument, and ensure the best model interpolation
# by extracting the clean data only after matching.
cindi = pysat.Instrument(inst_module=pysatNASA.instruments.cnofs_ivm,
                         clean_level='none')

try:
    cindi.files.files[stime]
except KeyError:
    # Desired date not on filesystem, download missing data.
    cindi.download(stime, stime)

# Set the input keyword args for the CINDI data
input_kwargs["inst_clean_rout"] = pysatNASA.instruments.cnofs_ivm.clean
input_kwargs["inst_download_kwargs"] = {"skip_download": True}
input_kwargs["inst_lon_name"] = "glon"
input_kwargs["inst_name"] = ["altitude", "alat", "glon", "slt"]
```

With all of the data obtained and the inputs set, we can pair the data. It doesn't matter if the longitude range for the model and observational data use different conventions, as the `collect_inst_model_pairs()` function will check for compatibility and adjust the range as needed.

```
matched_inst = ps_mod.utils.match.collect_inst_model_pairs(
    stime, stime + tinc, tinc, cindi, **input_kwargs)
```

The `collect_inst_model_pairs()` function returns a `pysat.Instrument` object with the CINDI and DINEOFs data at the same longitudes and local times, after raising warnings for times and places when the observed data location lies outside of the model interpolation limits. The CINDI data has the same names as the normal `pysat.Instrument`. The DINEOFs data has the same name as the normal `pysat.Instrument`, but with `'dineof_'` as a prefix to prevent confusion. You can change this prefix using the `model_label` keyword argument, allowing multiple models to be matched to the same observational data set.

```
# Using the results from the prior example
print([var for var in matched_inst.variables
       if var.find(input_kwargs['model_label']) == 0])
```

This produces the output line: `['dineof_model_equator_model_data']`.

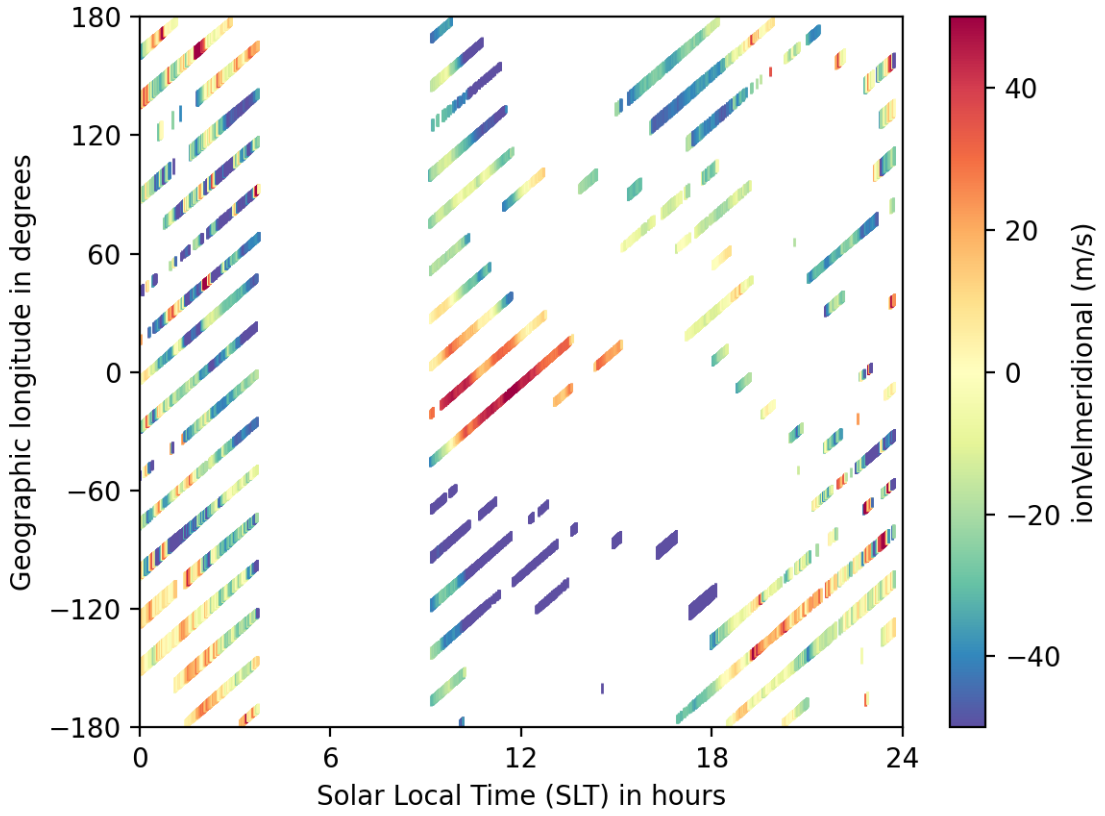
To see what the matched data looks like, let's create a plot that shows the locations and magnitudes of the modelled and measured meridional **E** x **B** drifts. We aren't directly comparing the values, since the test file is filled with randomly generated values that aren't realistic.

```
import matplotlib as mpl
import matplotlib.pyplot as plt

# Initialize the figure
fig = plt.figure()
ax = fig.add_subplot(111)

# Plot the data
ckey = 'ionVelmeridional'
dkey = 'dineof_model_equator_model_data'
vmax = 50
```

(continues on next page)

(continued from previous page)

```
con = ax.scatter(matched_inst['slt'], matched_inst['glon'],
                 c=matched_inst[ckey], s=matched_inst[dkey].abs() * 10 + 10,
                 marker='|', vmin=-vmax, vmax=vmax, lw=1,
                 cmap=mpl.cm.get_cmap('Spectral_r'))
cb = plt.colorbar(con, ax=ax)

# Format the figure
ax.xaxis.set_major_locator(mpl.ticker.MultipleLocator(6))
ax.yaxis.set_major_locator(mpl.ticker.MultipleLocator(60))
ax.set_xlabel(matched_inst.meta['slt', matched_inst.meta.labels.desc])
ax.set_ylabel(matched_inst.meta['glon', matched_inst.meta.labels.desc])
ax.set_xlim(0, 24)
ax.set_ylim(-180, 180)
cb_label = "{:s} ({:s})".format(
    matched_inst.meta[ckey, matched_inst.meta.labels.name],
    matched_inst.meta[ckey, matched_inst.meta.labels.units])
cb.set_label(cb_label)

# If you can display and are not running interactivly:
plt.show()
```



7.2.2 Match Data by Location and Time

For models with Universal Time variations over the desired period, you can also match model and data results by both time and location. This is done by setting the `time_method` keyword arguement to `'min'`.

7.3 Compare Paired Data Sets

Model verification and validation is supported in pysatModels through pyForecastTools. Many of the standard statistics used for these purposes can be run in a single go using the utility *pysatModels.utils.compare.compare_model_and_inst()*. The following example uses the paired data produced in example *Match Data by Location*. The `matched_inst` object at this point should display as shown below.

```
print(matched_inst)

pysat Instrument object
-----------------------
Platform: 'cnofs'
Name: 'ivm'
Tag: ''
Instrument id: ''

Data Processing
---------------
Cleaning Level: 'clean'
Data Padding: None
Keyword Arguments Passed to list_files: {}
Keyword Arguments Passed to load: {}
Keyword Arguments Passed to preprocess: {}
Keyword Arguments Passed to download: {}
Keyword Arguments Passed to list_remote_files: {}
Keyword Arguments Passed to clean: {}
Keyword Arguments Passed to init: {}
Custom Functions: 1 applied
    0: <function update_longitude at 0x12fcb8310>
     : Kwargs={'low': -180.0, 'lon_name': 'glon', 'high': 180.0}

Local File Statistics
---------------------
Number of files: 930
Date Range: 01 January 2009 --- 01 April 2015

Loaded Data Statistics
----------------------
Date: 01 January 2009
DOY: 001
Time range: 01 January 2009 00:00:00 --- 01 January 2009 23:59:59
Number of Times: 168596
Number of variables: 89

Variable Names:
RPAflag                         driftMeterflag                  ionVelocityX
                                    ...
```

(continues on next page)

(continued from previous page)

```
ECISC_index1                    LVLHSC_index1                    dineof_model_equator_
↪model_data

pysat Meta object
-----------------
Tracking 15 metadata values
Metadata for 92 standard variables
Metadata for 0 ND variables
```

Now, we need to convert this `pysat.Instrument` object to an `xarray.Dataset`. This conversion is needed to simplify the comparison analysis, since `pysat.Instrument.data` may be either `xarray.Dataset` or `pandas.DataFrame` objects. pysatModels uses `xarray.Dataset` as the base analysis class, because this class is best suited for modelled output. This can be easily done using *pysatModels.utils.convert.convert_pysat_to_xarray()*. Before we do that, though, we're going to update the units of the modelled data. This is necessary for the comparison, since the *pysatModels.utils.compare.compare_model_and_inst()* tests to make sure the paired data have the same units. It can handle converting between different units of the same type, so we will specify that the modelled data is a velocity in *cm/s*, while the observations are a velocity measured in *m/s*.

```
from pysatModels.utils import convert

inst_data_keys = ['ionVelmeridional']
model_data_keys = ['dineof_model_equator_model_data']
matched_inst.meta[model_data_keys[0]] = {
    matched_inst.meta.labels.units: "cm/s"}
paired_data = convert.convert_pysat_to_xarray(matched_inst)
print(paired_data)

<xarray.Dataset>
Dimensions:                          (index: 168596)
Coordinates:
  * index                            (index) datetime64[ns] 2009-01-01T00:00:...
Data variables: (12/89)
  RPAflag                          (index) int16 4 4 4 4 4 4 4 ... 4 3 4 4 4 3
  driftMeterflag                   (index) int16 0 0 0 0 0 0 0 ... 0 0 0 0 0 0
  ionVelocityX                     (index) float32 nan nan nan ... nan nan nan
  ionVelocityY                     (index) float32 633.8 589.8 ... 234.5 238.1
  ionVelocityZ                     (index) float32 106.3 105.6 ... 119.6 118.5
  vXvariance                       (index) float32 0.0 0.0 0.0 ... 0.0 0.0 0.0
  ...                               ...
  meridionalunitvectorX            (index) float32 -0.04526 ... 0.02975
  meridionalunitvectorY            (index) float32 -0.2083 -0.208 ... -0.392
  meridionalunitvectorZ            (index) float32 -0.977 -0.9771 ... -0.9195
  ECISC_index1                     (index) float64 nan nan nan ... nan nan nan
  LVLHSC_index1                    (index) float64 nan nan nan ... nan nan nan
  dineof_model_equator_model_data  (index) float64 -0.06359 ... -1.612
```

Now we can compare the paired model and observed data. The example below will only calculate the bias-related statistics, but there are several options for the `method` keyword argument that allow single or groups of statistics to be calculated in one call.

Note the statistical output is in the units of the observed data set. The `stat_dict` output is a *dict* with the observed data variable name(s) as the first set of keys and the requested statistics for each data type as a nested *dict*.

Not all of the statistics were appropriate for the data set, as indicated by the `RuntimeWarning` messages seen when

running *compare_model_and_inst()*. The values show that, unsurprisingly, the random data from the test model file does not agree well with the C/NOFS meridional **E** x **B** drifts.

7.4 Extract Observational-Style Data

Comparison of model and Instrument data is supported in pysatModels, in part, by enabling the extraction (or interpolation) of model output onto the same locations as an observation-style data set. One common example is 'flying' a satellite through a model. The satellite locations are used to extract relevant model data enabling direct comparison of observed and modeled values.

7.4.1 Regular Grid Models

pysatModels.utils.extract.extract_modelled_observations() supports extracting values from models on a regular grid onto observed locations. The function can linearly interpolate model values onto instrument locations or use the nearest modeled location. Uses `scipy.interpolate.interpn()` as the underlying interpolation function. This function can handle either `pandas` or `xarray` formatted `pysat.Instrument` observational data. Because multi-dimensional data can be more complicated, let's use Jicamarca ISR drift data as an example.

```
import datetime as dt
import pandas as pds
import pysat
from pysatMadrigal.instruments import jro_isr
import pysatModels

# Initalize the observed data
stime = dt.datetime(2021, 1, 3)
jro = pysat.Instrument(inst_module=jro_isr, tag='drifts', user='Your Name',
                       password='your.email@inst.type')

# Download data if necessary
if stime not in jro.files.files:
    jro.download(start=stime)

# Get fake model data from the pysat model test instrument
mod_drange = pds.date_range(stime, stime + dt.timedelta(days=1), freq='1D')
model = pysat.Instrument('pysat', 'testmodel', tag='',
                         file_date_range=mod_drange)
model.load(date=stime)

# Get the model longitude range, and make sure the loaded data has the
# same range
if model['longitude'].min() >= 0 and model['longitude'].max() > 180:
    min_lon = 0.0
    max_lon = 360.0
else:
    min_lon = -180.0
    max_lon = 180.0

jro.custom_attach(pysat.utils.coords.update_longitude,
                  kwargs={'lon_name': 'gdlonr', 'high': max_lon,
                  'low': min_lon})
```

(continues on next page)

(continued from previous page)

```
jro.load(date=stime)

# Check the loaded variables, you may receive a warning for unknown data
# variables (this is ok).
print(jro.variables, model.variables)
```

This yields:

```
['time', 'gdalt', 'gdlatr', 'gdlonr', 'kindat', 'kinst', 'nwlos', 'range',
 'vipn', 'dvipn', 'vipe', 'dvipe', 'vi7', 'dvi7', 'vi8', 'dvi8', 'paiwl',
 'pacwl', 'pbiwl', 'pbcwl', 'pciel', 'pccel', 'pdiel', 'pdcel', 'jro10',
 'jro11', 'year', 'month', 'day', 'hour', 'min', 'sec', 'spcst', 'pl',
 'cbadn', 'inttms', 'azdir7', 'eldir7', 'azdir8', 'eldir8', 'jro14',
 'jro15', 'jro16', 'ut1_unix', 'ut2_unix', 'recno'] ['uts', 'time',
 'latitude', 'longitude', 'altitude', 'slt', 'mlt', 'dummy1', 'dummy2']
```

To extract the desired data points, you need to specify the model time variable names, the matching observation and model coordinate names and dimensions, the variables you want to select for extraction, and the extraction method. For this example, we'll be matching the vertical drift from JRO (`'vipn'`) to the fake model variable with the appropriate dimensions (`'dummy2'`).

```
# Set the model dummy variable units
model.meta['dummy2'] = {model.meta.labels.units: 'm/s'}

# Get the xarray data from the model instrument, with metadata attached
model_data = pysatModels.utils.convert.convert_pysat_to_xarray(model)

# Set the extract input parameters
input_args = [jro, model_data, ["gdlonr", "gdlatr", "gdalt"],
              ["longitude", "latitude", "altitude"], "time", "time",
              ["deg", "deg", "km"]]
input_kwargs = {'sel_name': ['dummy2']}

# Run the extract function
added_vars = pysatModels.utils.extract.extract_modelled_observations(
    *input_args, **input_kwargs)
```

The output from this function will let you know the variable names that were added to the observational data `Instrument`. If we plot this data, we can visualize how the selection occurred.

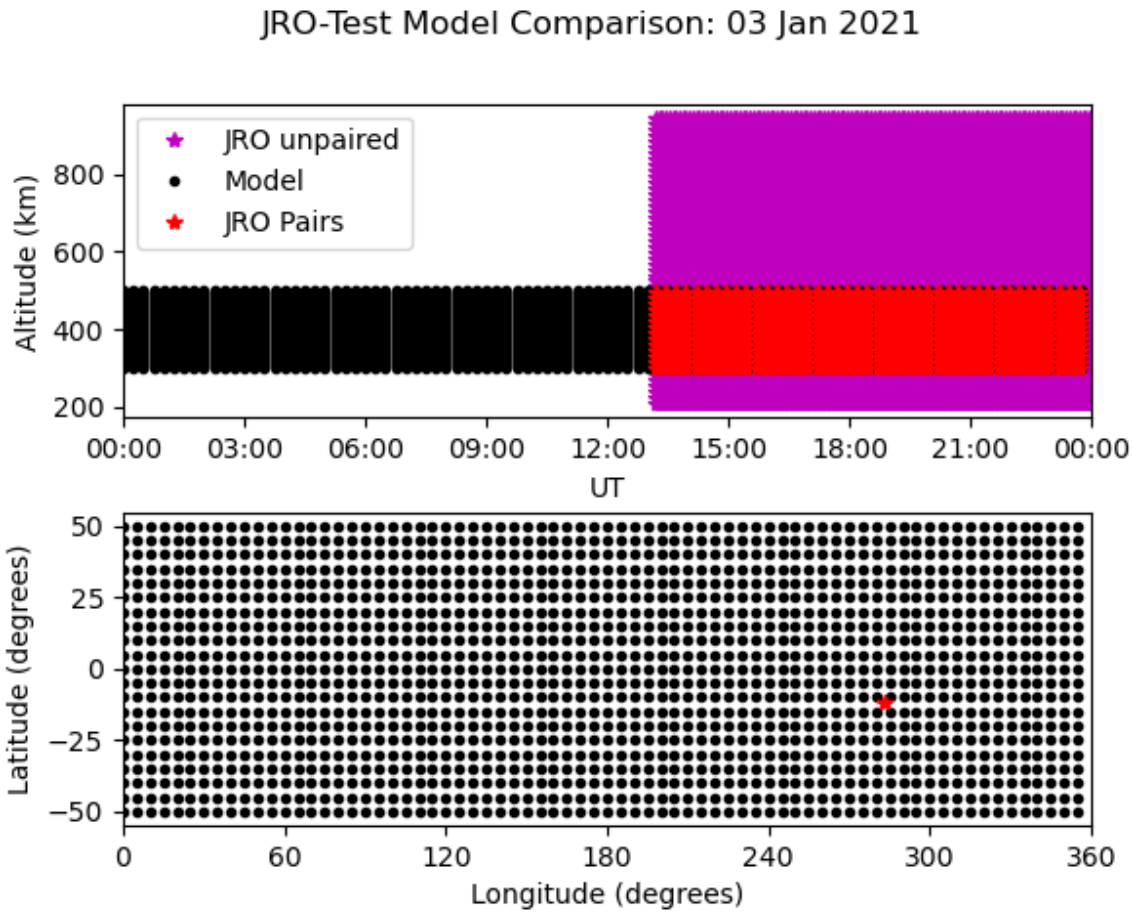
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np

# Initialize a figure with two subplots
fig = plt.figure()
ax_alt = fig.add_subplot(211)
ax_loc = fig.add_subplot(212)

# Create plottable model locations
mlon, mlat = np.meshgrid(model['longitude'], model['latitude'])
mtime, malt = np.meshgrid(model.index, model['altitude'])
```

(continues on next page)

(continued from previous page)

```
# Get the paired and unpaired JRO indices
igood = np.where(~np.isnan(jro[added_vars[0]]))
ibad = np.where(np.isnan(jro[added_vars[0]]))

# Plot the altitude/time data
ax_alt.plot(jro.index[ibad[0]], jro['gdalt'][ibad[1]], 'm*',
            label='JRO unpaired')
ax_alt.plot(mtime, malt, 'k.')
ax_alt.plot(jro.index[igood[0]], jro['gdalt'][igood[1]], 'r*',
            label='JRO Pairs')

# Plot the the lat/lon data
ax_loc.plot(mlon, mlat, 'k.')
ax_loc.plot(jro['gdlonr'], jro['gdlatr'], 'r*')

# Format the figure
ax_loc.set_xlim(0, 360)
ax_loc.xaxis.set_major_locator(mpl.ticker.MultipleLocator(60))
ax_loc.set_xlabel('{:} ({:})'.format(
    model.meta['longitude', model.meta.labels.name],
    model.meta['longitude', model.meta.labels.units]))
ax_loc.set_ylabel('{:} ({:})'.format(
    model.meta['latitude', model.meta.labels.name],
    model.meta['latitude', model.meta.labels.units]))

ax_alt.lines[1].set_label('Model')
ax_alt.legend(loc=2)
ax_alt.set_xlabel('UT')
ax_alt.set_xlim(stime, stime + dt.timedelta(days=1))
ax_alt.xaxis.set_major_formatter(mpl.dates.DateFormatter('%H:%M'))
ax_alt.set_ylabel('{:} ({:})'.format(
    model.meta['altitude', model.meta.labels.name],
    model.meta['altitude', model.meta.labels.units]))

fig.suptitle('JRO-Test Model Comparison: {:}'.format(
    stime.strftime('%d %b %Y')))
fig.subplots_adjust(hspace=.3)

# If not working interactively
plt.show()
```

And this should show the figure below.



pysatModels.utils.extract.instrument_view_through_model() supports interpolating values from regular grid models onto Instrument locations using `scipy.interpolate.RegularGridInterpolator()`. Consider the following example that interpolates model data onto a satellite data set using pysat testing data sets.

```
import datetime as dt

import pysat
import pysatModels

# Load simulated satellite Instrument data set
inst = pysat.Instrument('pysat', 'testing', max_latitude=45.)
inst.load(2009, 1)

# Load simulated regular-grid model Instrument
model = pysat.Instrument('pysat', 'testmodel')
model.load(2009, 1)
```

Looking at the loaded `model.data` we can see that the model is indeed regular.

```
<xarray.Dataset>
Dimensions:    (time: 96, latitude: 21, longitude: 73, altitude: 41)
Coordinates:
  * time       (time) datetime64[ns] 2009-01-01 ... 2009-01-01T23:45:00
  * latitude   (latitude) float64 -50.0 -45.0 -40.0 -35.0 ... 40.0 45.0 50.0
```

(continues on next page)

(continued from previous page)

```
  * longitude  (longitude) float64 0.0 5.0 10.0 15.0 ... 345.0 350.0 355.0 360.0
  * altitude   (altitude) float64 300.0 305.0 310.0 315.0 ... 490.0 495.0 500.0
Data variables:
    uts        (time) float64 0.0 900.0 1.8e+03 ... 8.37e+04 8.46e+04 8.55e+04
    slt        (time, longitude) float64 0.0 0.3333 0.6667 ... 23.08 23.42 23.75
    mlt        (time, longitude) float64 0.2 0.5333 0.8667 ... 23.28 23.62 23.95
    dummy1     (time, latitude, longitude) float64 0.0 0.0 0.0 ... 0.0 3.0 6.0
    dummy2     (time, latitude, longitude, altitude) float64 0.0 0.0 ... 18.0
```

The coordinates are `time`, `latitude`, `longitude`, and `altitude`, and are all one-dimensional and directly relevant to a physical satellite location. The equivalent satellite variables are `latitude`, `longitude`, and `altitude`, with `time` taken from the associated `Instrument` time index (`Instrument.data.index`). The output from `inst.variables` and `inst.data.index` should be

```
Index(['uts', 'mlt', 'slt', 'longitude', 'latitude', 'altitude', 'orbit_num',
       'dummy1', 'dummy2', 'dummy3', 'dummy4', 'string_dummy',
       'unicode_dummy', 'int8_dummy', 'int16_dummy', 'int32_dummy',
       'int64_dummy', 'model_dummy2'], dtype='object')

DatetimeIndex(['2009-01-01 00:00:00', '2009-01-01 00:00:01',
               '2009-01-01 00:00:02', '2009-01-01 00:00:03',
               '2009-01-01 00:00:04', '2009-01-01 00:00:05',
               '2009-01-01 00:00:06', '2009-01-01 00:00:07',
               '2009-01-01 00:00:08', '2009-01-01 00:00:09',
               ...
               '2009-01-01 23:59:50', '2009-01-01 23:59:51',
               '2009-01-01 23:59:52', '2009-01-01 23:59:53',
               '2009-01-01 23:59:54', '2009-01-01 23:59:55',
               '2009-01-01 23:59:56', '2009-01-01 23:59:57',
               '2009-01-01 23:59:58', '2009-01-01 23:59:59'],
              dtype='datetime64[ns]', name='Epoch', length=86400, freq=None)
```

Interpolating `model` data onto `inst` is accomplished via

```
new_data_keys = pysatModels.utils.extract.instrument_view_through_model(inst,
                           model.data, ['longitude'], ['longitude'], 'time',
                           'time', ['deg'], ['mlt'])
```

where `inst` and `model.data` provide the required `pysat.Instrument` object and `xarray.Dataset`. The

```
['longitude']
```

term provides the content and ordering of the coordinates for model variables to be interpolated. The subsequent

```
['longitude']
```

term provides the equivalent content from the satellite's data set, in the same order as the model coordinates. In this case, the same labels are used for both the satellite and modeled data sets. The

```
'time', 'time'
```

terms cover the model labels used for time variable and coordinate (which may be the same, as here, or different). The

```
['deg']
```

term covers the units for the model dimensions (longitude). Units for the corresponding information from `inst` are taken directly from the `pysat.Instrument` object. The final presented input

```
['mlt']
```

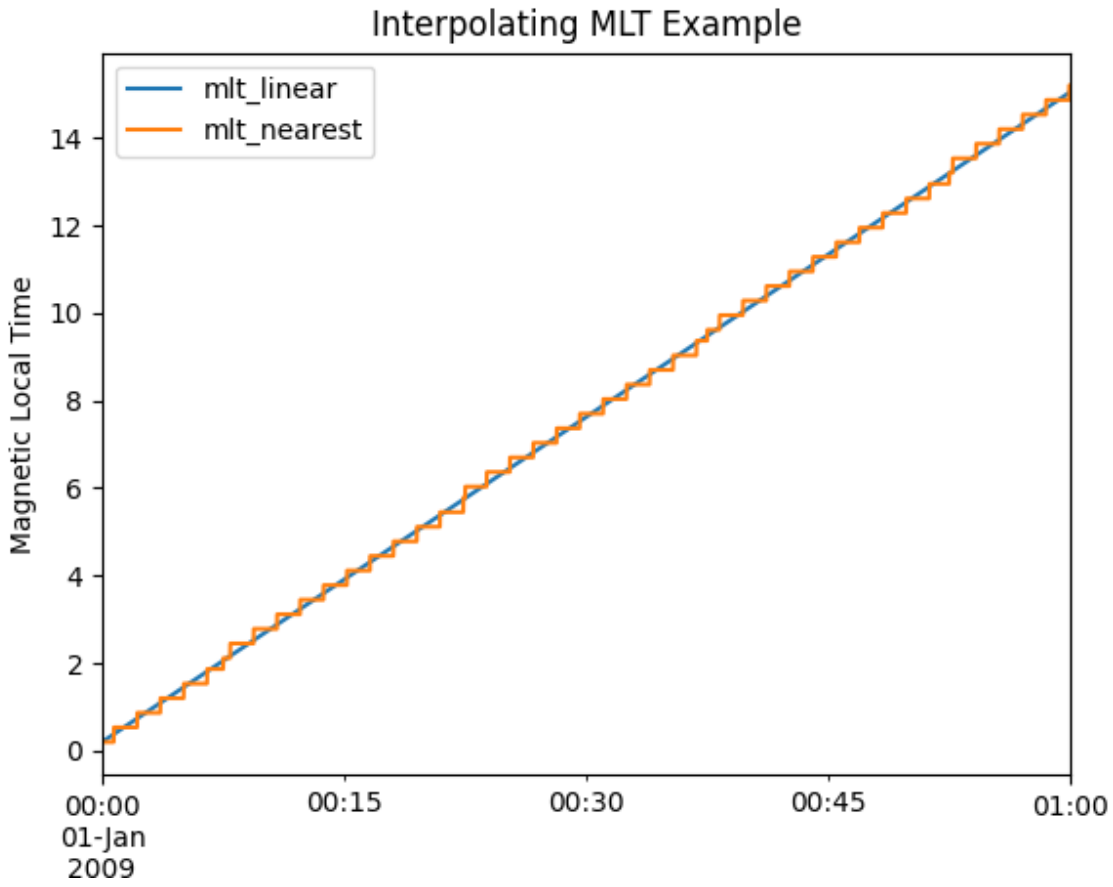
is a list of model variables that will be interpolated onto `inst`. By default a linear interpolation is performed but a nearest neighbor option is also supported.

```
# Store results for linear interpolation
inst.rename({new_data_keys[0]: "mlt_linear"})

# Run interpolation using 'nearest'
new_data_keys = pysatModels.utils.extract.instrument_view_through_model(
    inst, model.data, ['longitude'], ['longitude'], 'time', 'time',
    ['deg'], ['mlt'], ['nearest'])
inst.rename({new_data_keys[0]: "mlt_nearest"})

# Set up time range for plotting results
stime = inst.date
etime = inst.date + dt.timedelta(hours=1)
```

The results of

```
title = 'Interpolating MLT Example'
ylabel = 'Magnetic Local Time'
inst[stime:etime, ['mlt_linear', 'mlt_nearest']].plot(title=title,
                                                      ylabel=ylabel)
```

are shown below.



Multidimensional interpolation is performed in the same manner.

```
new_data_keys = pysatModels.utils.extract.instrument_view_through_model(inst,
                            model.data, ['latitude', 'longitude', 'altitude'],
                            ['latitude', 'longitude', 'altitude'], 'time',
                            'time', ['deg', 'deg', 'km'], ['dummy2'])
```

The

```
['latitude', 'longitude', 'altitude']
```

term provides the content and ordering of the coordinates for model variables to be interpolated. The subsequent

```
['latitude', 'longitude', 'altitude']
```

term provides the equivalent content from the satellite's data set, in the same order as the model coordinates. The

```
'time', 'time'
```

terms cover the model labels used for time variable and coordinate. The

```
['deg', 'deg', 'km']
```

term covers the units for the model dimensions (latitude/longitude/altitude). Units for the corresponding information from `inst` are taken directly from the `pysat.Instrument` object. The final presented input

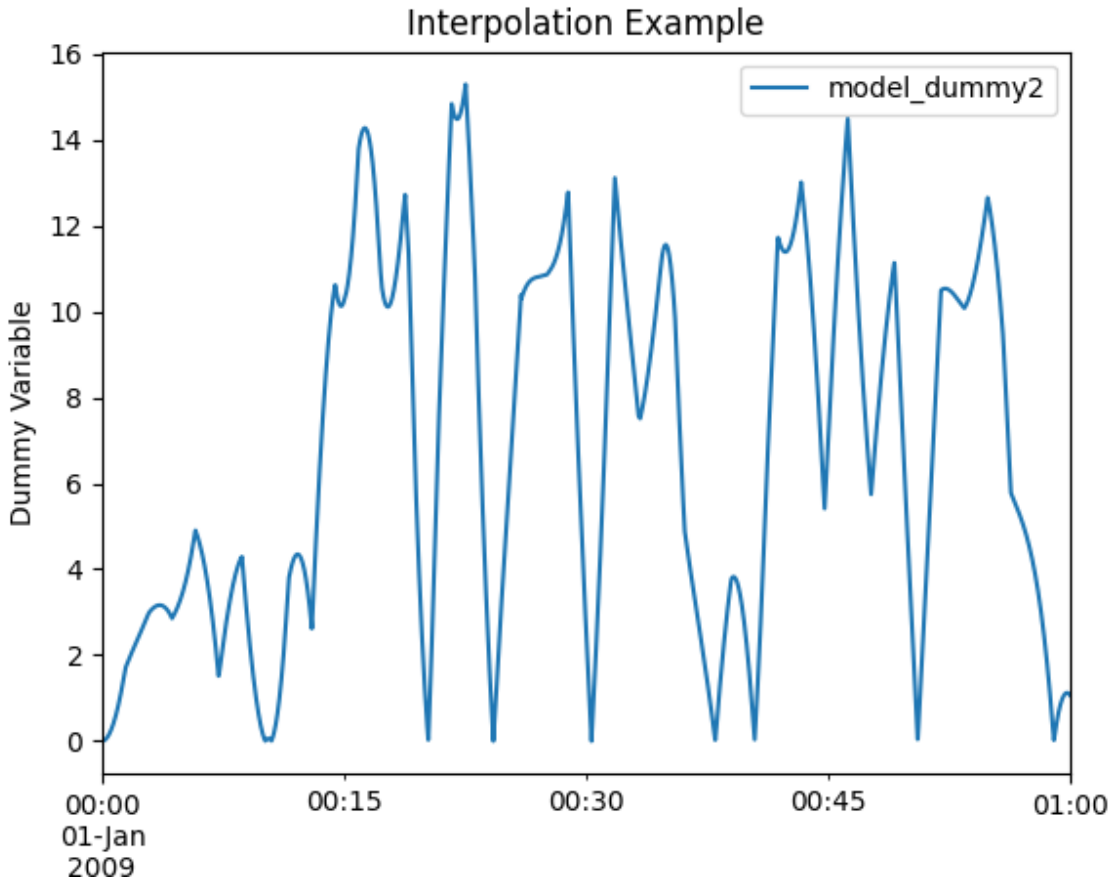
```
['dummy2']
```

is a list of model variables that will be interpolated onto `inst`.

The results of

```
    # Use the same time range as the prior example
ylabel = 'Dummy Variable'
inst[stime:etime, new_data_keys].plot(title='Interpolation Example',
                                      ylabel=ylabel)
```

are shown below.



7.4.2 Irregular Grid Models

Some models aren't on a regular grid, or may not be a regular grid across the coordinates of interest. Consider an alternative model data set,

```
model = pysat.Instrument('pysat', 'testmodel', tag='pressure_levels')
model.load(2009, 1)
model.data

<xarray.Dataset>
```

(continues on next page)

(continued from previous page)

```
Dimensions:       (time: 24, latitude: 72, longitude: 144, lev: 57, ilev: 57)
Coordinates:
  * time          (time) datetime64[ns] 2009-01-01 ... 2009-01-01T23:00:00
  * latitude      (latitude) float64 -88.75 -86.25 -83.75 ... 83.75 86.25 88.75
  * longitude     (longitude) float64 -180.0 -177.5 -175.0 ... 172.5 175.0 177.5
  * lev           (lev) float64 -7.0 -6.75 -6.5 -6.25 -6.0 ... 6.25 6.5 6.75 7.0
  * ilev          (ilev) float64 -6.875 -6.625 -6.375 ... 6.625 6.875 7.125
Data variables:
    uts           (time) float64 0.0 3.6e+03 7.2e+03 ... 7.92e+04 8.28e+04
    altitude      (time, ilev, latitude, longitude) float64 0.0 0.0 ... 5.84e+07
    dummy_drifts  (time, ilev, latitude, longitude) float64 0.0 0.0 ... 83.01
    slt           (time, longitude) float64 12.0 12.17 12.33 ... 10.67 10.83
    mlt           (time, longitude) float64 12.2 12.37 12.53 ... 10.87 11.03
    dummy1        (time, latitude, longitude) float64 0.0 0.0 0.0 ... 0.0 9.0
```

Model variables, such as `dummy_drifts`, are regular over `(time, ilev, latitude, longitude)`, where `ilev` is a constant pressure level. Unfortunately, the observational data in `inst` doesn't contain pressure level as a simulated/measured parameter. However, `altitude` is present in the model data but varies over all four coordinates. Interpolating `dummy_drifts` onto `inst` requires either adding an appropriate value for `ilev` into `inst`, or interpolating model variables using the irregular variable `altitude` instead of `ilev`.

Altitude to Pressure

pysatModels.utils.extract.instrument_altitude_to_model_pressure() will use information in a model to generate appropriate pressure levels for a supplied altitude in an observational-like data set.

```
import pysatModels

keys = pysatModels.utils.extract.instrument_altitude_to_model_pressure(inst,
            model.data, ["altitude", "latitude", "longitude"],
            ["ilev", "latitude", "longitude"],
            "time", "time", ['', "deg", "deg"],
            'altitude', 'altitude', 'cm')
```

The function will guess a pressure level for all locations in `inst` and then use the regular mapping from pressure to altitude to obtain the equivalent altitude from the model. The pressure is adjusted up/down an increment based upon the comparison and the process is repeated until the target tolerance (default is 1 km) is achieved. The keys for the model derived pressure and altitude values added to `inst` are returned from the function.

```
inst['model_pressure']

Epoch
2009-01-01 00:00:00    3.104662
2009-01-01 00:00:01    3.104652
2009-01-01 00:00:02    3.104642
2009-01-01 00:00:03    3.104632
2009-01-01 00:00:04    3.104623
                          ...
2009-01-01 23:59:55    2.494845
2009-01-01 23:59:56    2.494828
2009-01-01 23:59:57    2.494811
2009-01-01 23:59:58    2.494794
```

(continues on next page)

(continued from previous page)

```
2009-01-01 23:59:59    2.494776
Name: model_pressure, Length: 86400, dtype: float64
```

```
# Calculate difference between interpolation techniques
inst['model_altitude'] - inst['altitude']

Epoch
2009-01-01 00:00:00   -0.744426
2009-01-01 00:00:01   -0.744426
2009-01-01 00:00:02   -0.744425
2009-01-01 00:00:03   -0.744424
2009-01-01 00:00:04   -0.744424
                          ...
2009-01-01 23:59:55   -0.610759
2009-01-01 23:59:56   -0.610757
2009-01-01 23:59:57   -0.610754
2009-01-01 23:59:58   -0.610751
2009-01-01 23:59:59   -0.610749
Length: 86400, dtype: float64
```

Using the added `model_pressure` information model values may be interpolated onto `inst` using regular grid methods.

```
new_keys = pysatModels.utils.extract.instrument_view_through_model(inst,
                model.data, ['model_pressure', 'latitude', 'longitude'],
                ['ilev', 'latitude', 'longitude'], 'time', 'time',
                ['', 'deg', 'deg'], ['dummy_drifts'])
```

```
inst['model_dummy_drifts']

Epoch
2009-01-01 00:00:00    30.289891
2009-01-01 00:00:01    30.305303
2009-01-01 00:00:02    30.320704
2009-01-01 00:00:03    30.336092
2009-01-01 00:00:04    30.351469
                          ...
2009-01-01 23:59:55    63.832658
2009-01-01 23:59:56    63.868358
2009-01-01 23:59:57    63.904047
2009-01-01 23:59:58    63.939724
2009-01-01 23:59:59    63.975389
Name: model_dummy_drifts, Length: 86400, dtype: float64
```

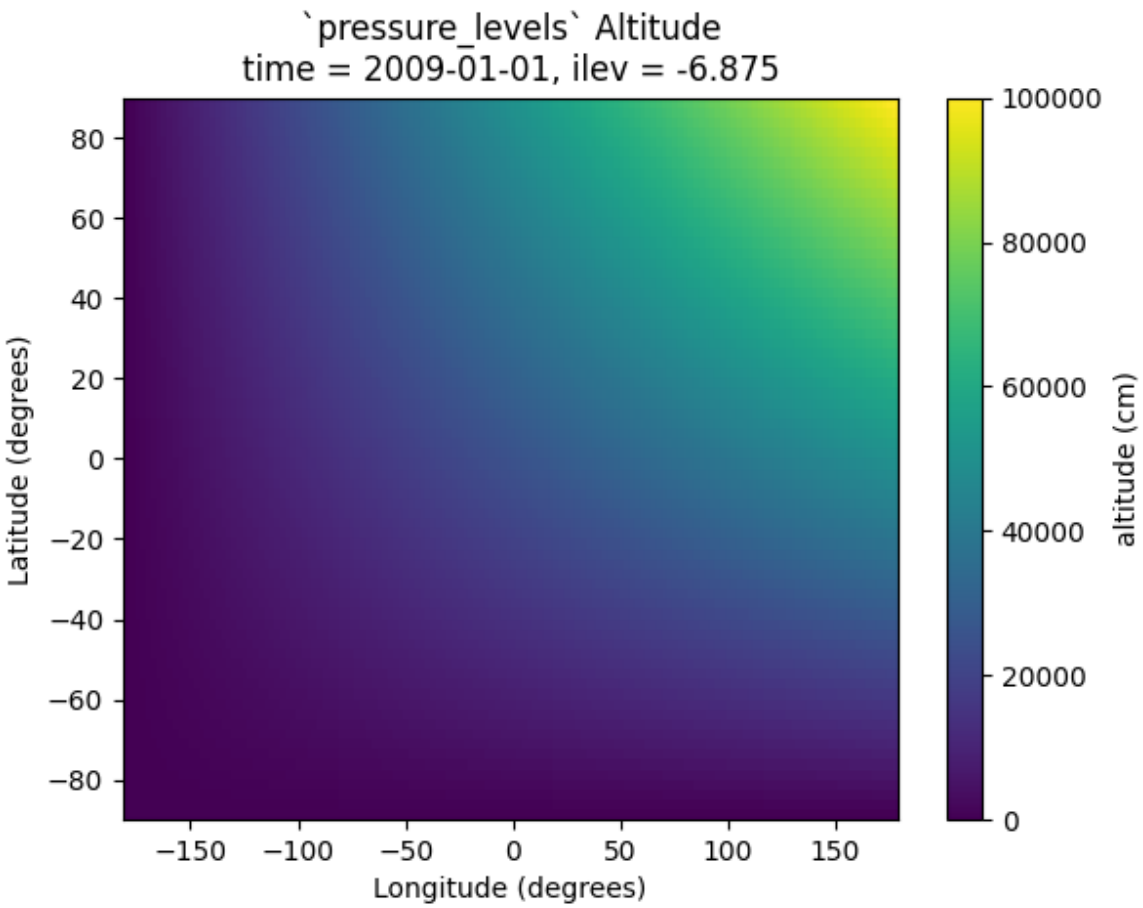
The time to translate altitude to model pressure is ~3 s, and the regular interpolation takes an additional ~300 ms.

Irregular Variable

More generally, *pysatModels.utils.extract.interp_inst_w_irregular_model_coord()* can deal with irregular coordinates when interpolating onto an observational-like data set using `scipy.interpolate.griddata()`. The `model` loaded above is regular against pressure level, latitude, and longitude. However, it is irregular with respect to altitude.

Here is a sample distribution of the `model['altitude']` for `ilev=0` and the first model time.

```
import matplotlib.pyplot as plt

# Make base plot of 'altitude' for ilev=0 and time=0
model[0, 0, :, :, "altitude"].plot()

# Prep labels
xlabel = "".join([model.meta["longitude", model.meta.labels.name], " (",
                  model.meta["longitude", model.meta.labels.units],
                  ")"])
ylabel = "".join([model.meta["latitude", model.meta.labels.name], " (",
                  model.meta["latitude", model.meta.labels.units],
                  ")"])
cblabel = "".join([model.meta["altitude", model.meta.labels.name], " (",
                   model.meta["altitude", model.meta.labels.units],
                   ")"])

# Update labels
plt.xlabel(xlabel)
plt.ylabel(ylabel)

# Update color bar and title
fig = plt.gcf()
fig.axes[1].set_ylabel(cblabel)
fig.axes[0].set_title("".join(["`pressure_levels` Altitude\n",
                              fig.axes[0].title.get_text()]))

plt.show()
```



To interpolate against the irregular variable, the *pysatModels.utils.extract.interp_inst_w_irregular_model_coord()* function should be used. Generalized irregular interpolation can take significant computational resources, so we start this example by loading smaller `pysat.Instrument` objects.

```
inst = pysat.Instrument('pysat', 'testing', max_latitude=10.,
                        num_samples=100)
model = pysat.Instrument('pysat', 'testmodel', tag='pressure_levels',
                         num_samples=5)
inst.load(2009, 1)
model.load(2009, 1)

keys = pysatModels.utils.extract.interp_inst_w_irregular_model_coord(inst,
            model.data, ["altitude", "latitude", "longitude"],
            ["ilev", "latitude", "longitude"],
            "time", ["cm", "deg", "deg"], "ilev",
            "altitude", [50., 2., 5.],
            sel_name=["dummy_drifts", "altitude"])

# CPU times: user 419 ms, sys: 13 ms, total: 432 ms
# Wall time: 431 ms

# Print results from interpolation
inst['model_dummy_drifts']
```

(continues on next page)

(continued from previous page)

```
Epoch
2009-01-01 00:00:00    22.393249
2009-01-01 00:00:01    22.405926
2009-01-01 00:00:02    22.418600
2009-01-01 00:00:03    22.431272
2009-01-01 00:00:04    22.443941
                          ...
2009-01-01 00:01:35    23.592833
2009-01-01 00:01:36    23.605252
2009-01-01 00:01:37    23.617668
2009-01-01 00:01:38    23.630081
2009-01-01 00:01:39    23.642492
Name: model_dummy_drifts, Length: 100, dtype: float64
```

In the interpolation function, `inst` and `model.data` provide the required data through the `pysat.Instrument` and `xarray.Dataset` objects. The

```
["altitude", "latitude", "longitude"]
```

term provides the content and ordering of the spatial locations for `inst`. The subsequent

```
["ilev", "latitude", "longitude"]
```

term provides the equivalent regular dimension labels from `model.data`, in the same order as the underlying model dimensions. While this function does operate on irregular data it also needs information on the underlying regular memory structure of the variables. The

```
"time"
```

terms cover the model label used for the datetime coordinate. The

```
["cm", "deg", "deg"]
```

term covers the units for the model information (altitude/latitude/longitude) that maps to the `inst` information in the coordinate list `["altitude", "latitude", "longitude"]`. Note that the `"cm"` covers units for `'altitude'` in `model.data`, the variable that will replace `'ilev'`, while the second two list elements (both `"deg"`) covers the units for the latitude and longitude dimensions. Units for the corresponding information from `inst` are taken directly from the `pysat.Instrument` object. The

```
"ilev"
```

identifies the regular model dimension that will be replaced with irregular data for interpolation. The

```
"altitude"
```

identifies the irregular model variable that will replace the regular coordinate. The

```
[50., 10., 10.]
```

term is used to define a half-window for each of the `inst` locations, in units from `inst`, used to downselect data from `model.data` to reduce computational requirements. In this case a window of +/-50 km in altitude, +/-10 degrees in latitude, and +/-10 degrees in longitude is used. The keyword argument

```
sel_name = ["dummy_drifts", "altitude"]
```

identifies the `model.data` variables that will be interpolated onto `inst`. If you don't account for the irregularity in the desired model coordinates, the interpolation results are affected.

```
keys = pysatModels.utils.extract.instrument_altitude_to_model_pressure(inst,
             model.data, ["altitude", "latitude", "longitude"],
             ["ilev", "latitude", "longitude"],
             "time", "time", ['', "deg", "deg"],
             'altitude', 'altitude', 'cm')
new_data_keys = pysatModels.utils.extract.instrument_view_through_model(
    inst, model.data, ['model_pressure', 'latitude', 'longitude'],
    ['ilev', 'latitude', 'longitude'], 'time', 'time', ['', 'deg', 'deg'],
    ['dummy_drifts'], model_label='model2')

# CPU times: user 3.11 ms, sys: 388 µs, total: 3.5 ms
# Wall time: 3.14 ms

# Compare interpolated `dummy_drifts` between two techniques
inst['model2_dummy_drifts'] - inst['model_dummy_drifts']

Epoch
2009-01-01 00:00:00   -0.024180
2009-01-01 00:00:01   -0.023968
2009-01-01 00:00:02   -0.023756
2009-01-01 00:00:03   -0.023544
2009-01-01 00:00:04   -0.023332
                          ...
2009-01-01 00:01:35   -0.011532
2009-01-01 00:01:36   -0.011326
2009-01-01 00:01:37   -0.011120
2009-01-01 00:01:38   -0.010914
2009-01-01 00:01:39   -0.010708
Length: 100, dtype: float64
```

CHAPTER

EIGHT

GUIDE FOR DEVELOPERS

8.1 Contributor Covenant Code of Conduct

8.1.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

8.1.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

8.1.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

8.1.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

8.1.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at pysat.developers@gmail.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

8.1.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct/

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq

8.2 Contributing

Bug reports, feature suggestions and other contributions are greatly appreciated! pysatModels is a community-driven project and welcomes both feedback and contributions.

8.3 Short version

- Submit bug reports and feature requests at GitHub Issues
- Make pull requests to the `develop` branch

8.4 Bug reports

When reporting a bug please include:

- Your operating system name and version
- Any details about your local setup that might be helpful in troubleshooting
- Detailed steps to reproduce the bug

8.5 Feature requests and feedback

The best way to send feedback is to file an issue at GitHub Issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

8.6 Development

To set up `pysatModels` for local development:

- Fork pysat on GitHub.
- Clone your fork locally:

```
git clone git@github.com:your_name_here/pysatModels.git
```

- Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally. Tests for new instruments are performed automatically. Tests for custom functions should be added to the appropriately named file in `pysatModels/tests`. For example, the averaging routines in avg.py are tested in `pysatModels/tests/test_avg.py`. If no test file exists, then you should create one. This testing uses pytest, which will run tests on any python file in the test directory that starts with `test_`.

- When you're done making changes, run all the checks from the `pysatModels/tests` directory to ensure that nothing is broken on your local system. You may need to install pytest and pytest-flake8 first.

```
python -m pytest -vs --flake8
```

- Update or add documentation (in `docs`), if relevant. If you have added a new routine, you will need to add an example in the `docs/examples` folder.
- Commit your changes and push your branch to GitHub. Our commit statements follow the basic rules in the Numpy/SciPy workflow:

```
git add .
git commit -m "TYPE: Brief description of your changes"
git push origin name-of-your-bugfix-or-feature
```

- Submit a pull request through the GitHub website. Pull requests should be made to the `develop` branch.

8.6.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code, just make a pull request. Pull requests should be made to the `develop` branch.

For merging, you should:

1. Include an example for use
2. Add a note to `CHANGELOG.md` about the changes
3. Ensure that all checks passed (current checks include GitHub Actions and Coveralls).

If you don't have all the necessary Python versions available locally or have trouble building all the testing environments, you can rely on GitHub to run the tests for each change you add in the pull request. Because testing here will delay tests by other developers, please ensure that the code passes all tests on your local system first.

CHAPTER

NINE

CHANGE LOG

All notable changes to this project will be documented in this file. This project adheres to Semantic Versioning.

9.1 [0.1.0] - 2022-05-20

- Initial release

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX