SpacePy Documentation

Release 0.1.0

The SpacePy Team

August 23, 2011

CONTENTS

SpacePy is a package for Python, targeted at the space sciences, that aims to make basic data analysis, modeling and visualization easier. It builds on the capabilities of the well-known NumPy and MatPlotLib packages. Publication quality output direct from analyses is emphasized among other goals:

- Quickly obtain data
- Create publications quality plots
- Perform complicated analysis easily
- Run common empirical models
- Change coordinates effortlessly
- Harness the power of Python

The SpacePy project seeks to promote accurate and open research standards by providing an open environment for code development. In the space physics community there has long been a significant reliance on proprietary languages that restrict free transfer of data and reproducibility of results. By providing a comprehensive, open-source library of widely-used analysis and visualization tools in a free, modern and intuitive language, we hope that this reliance will be diminished.

When publishing research which used SpacePy, please provide appropriate credit to the SpacePy team via citation or acknowledgment.

To cite SpacePy in publications, use (BibTeX code): @INPROCEEDINGS{spacepy11, author = {{Morley}, S.~K. and {Koller}, J. and {Welling}, D.~T. and {Larsen}, B.~A. and {Henderson}, M.~G. and {Niehof}, J.~T.}, title = "{Spacepy - A Python-based library of tools for the space sciences}", booktitle = "{Proceedings of the 9th Python in science conference (SciPy 2010)}", year = 2011, address = {Austin, TX} }

Certain modules may provide additional citations in the `__citation__` attribute. Contact a module's author before publication or public presentation of analysis performed by that module. This allows the author to validate the analysis and receive appropriate credit for his or her work.

CHAPTER

ONE

SPACEPY DOCUMENTS

1.1 SpacePy - A Quick Start Documentation

The SpacePy Team (Steve Morley, Josef Koller, Dan Welling, Brian Larsen, Jon Niehof, Mike Henderson)

1.1.1 Installation

Option 1) to install it in a standard location (depending on your system):

```
python setup.py install
```

or:

```
sudo python setup.py install
```

or:

```
python setup.py install --user
```

If you do not have administrative privileges, or you will be developing for SpacePy, the latter is recommended.

Option 2) to install in custom location, e.g.:

```
python setup.py install --home=/n/packages/lib/python
```

It is also possible to select a specific compiler for installing the IRBEM-LIB library as part of SpacePy. Currently the following flags are supported: gnu95, gnu, pg. You can invoke these by using one of the following commands below but not all of them are supported on all platforms:

- `python setup.py install --fcompiler=pg` #(will use pgi compiler suite)
- `python setup.py install --fcompiler=gnu` #(will use g77)
- `python setup.py install --fcompiler=gnu95` #(default option for using gfortran)

The installer will create a `.spacepy` directory in your `$HOME` folder. If you prefer a different location for this directory, set the environment variable `$SPACEPY` to a location of your choice. For example, with a `csh`, or `tcsh` you would:

```
setenv SPACEPY /a/different/dir
```

for the `bash` shell you would:

```
export SPACEPY=/a/different/dir
```

Make sure you add the environment variable `$SPACEPY` to your `.cshrc`, `.tcshrc`, or `.bashrc` script.

1.1.2 Toolbox - A Box Full of Tools

Contains tools that don't fit anywhere else but are, in general, quite useful. The following functions are a selection of those implemented:

- `toolbox.windowMean()`: windowing mean with variable window size and overlap
- `toolbox.dictree()`: pretty prints the contents of dictionaries (recursively)
- `toolbox.loadpickle()`: single line convenience routine for loading Python pickles
- `toolbox.savepickle()`: same as loadpickle, but for saving
- `toolbox.update()`: updates the OMNI database and the leap seconds database (internet connection required)
- `toolbox.tOverlap()`: find interval of overlap between two time series
- `toolbox.tCommon()`: find times common to two time series
- `toolbox.binHisto()`: calculate number of bins for a histogram
- `toolbox.medAbsDev()`: find the median absolute deviation of a data series
- `toolbox.normalize()`: normalize a data series
- `toolbox.listUniq()`: returns the uniq items in a list (in order)
- `toolbox.leapyear()`: ultra fast leap year query function
- `toolbox.applySmartTimeTicks()`: smartens up the time ticks on a plot
- `toolbox.feq()`: floating point equals

Import this module as:

```
>>> import spacepy.toolbox as tb
```

Examples:

```
>>> import spacepy.toolbox as tb
>>> a = {'entry1':'val1', 'entry2':2, 'recurse1':{'one':1, 'two':2}}
>>> tb.dictree(a)
+
|____entry1
|____entry2
|____recurse1
     |____one
     |____two
>>> import numpy as np
>>> dat = np.random.random_sample(100)
>>> tb.binHisto(dat)
(0.19151723370512266, 5.0)
```

1.1.3 Time and Coordinate Transformations

Import the modules as:

```
>>> import spacepy.time as spt
>>> import spacepy.coords as spc
```

Ticktock Class

The Ticktock class provides a number of time conversion routines and is implemented as a container class built on the functionality of the Python datetime module. The following time coordinates are provided

- UTC: Coordinated Universal Time implemented as a `datetime.datetime` class
- ISO: standard ISO 8601 format like `2002-10-25T14:33:59`
- TAI: International Atomic Time in units of seconds since Jan 1, 1958 (midnight) and includes leap seconds, i.e. every second has the same length
- JD: Julian Day
- MJD: Modified Julian Day
- UNX: UNIX time in seconds since Jan 1, 1970
- RDT: Rata Die Time (Gregorian Ordinal Time) in days since Jan 1, 1 AD midnight
- CDF: CDF Epoch time in milliseconds since Jan 1, year 0
- DOY: Day of Year including fractions
- leaps: Leap seconds according to ftp://maia.usno.navy.mil/ser7/tai-utc.dat

To access these time coordinates, you'll create an instance of a Ticktock class, e.g.:

```
>>> t = spt.Ticktock('2002-10-25T12:30:00', 'ISO')
```

Instead of ISO you may use any of the formats listed above. You can also use numpy arrays or lists of time points. `t` has now the class attributes:

```
>>> t.dtype = 'ISO'
>>> t.data = '2002-10-25T12:30:00'
```

FYI `t.UTC` is added automatically.

If you want to convert/add a class attribute from the list above, simply type e.g.:

```
>>> t.RTD
```

You can replace RTD with any from the list above.

You can find out how many leap seconds were used by issuing the command:

```
>>> t.getleapsecs()
```

Tickdelta Class

You can add/substract time from a Ticktock class instance by creating a Tickdelta instance first.:

```
>>> dt = spt.Tickdelta(days=2.3)
```

Then you can add by e.g.:

```
>>> t+dt
```

Coords Class

The spatial coordinate class includes the following coordinate systems in Cartesian and sphericals.

- GZD: (altitude, latitude, longitude in km, deg, deg

- GEO: cartesian, Re
- GSM: cartesian, Re
- GSE: cartesian, Re
- SM: cartesian, Re
- GEI: cartesian, Re
- MAG: cartesian, Re
- SPH: same as GEO but in spherical
- RLL: radial distance, latitude, longitude, Re, deg, deg.

Create a Coords instance with spherical='sph' or cartesian='car' coordinates:

```
>>> spaco = spc.Coords([[1,2,4],[1,2,2]], 'GEO', 'car')
```

This will let you request for example all y-coordinates by `spaco.y` or if given in spherical coordinates by `spaco.lati`. One can transform the coordinates by `newcoord = spaco.convert('GSM', 'sph')`. This will return GSM coordinates in a spherical system. Since GSM coordinates depend on time, you'll have to add first a Ticktock vector with the name `ticks` like `spaco.ticks = spt.Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:00:00'], 'ISO')`

Unit conversion will be implemented in the future.

1.1.4 The radbelt Module

The radiation belt module currently includes a simple radial diffusion code as a class. Import the module and create a class instance:

```
>>> import spacepy.radbelt as sprb
>>> rb = sprb.RBmodel()
```

Add a time grid for a particular period that you are interested in:

```
>>> rb.setup_ticks('2002-02-01T00:00:00', '2002-02-10T00:00:00', 0.25)
```

This will automatically lookup required geomagnetic/solar wind conditions for that period. Run the diffusion solver for that setup and plot the results:

```
>>> rb.evolve()
>>> rb.plot()
```

1.1.5 The Data Assimilation Module

This module includes data assimilation capabilities, through the assimilation class. The class assimilates data for the radiation belt model using the Ensemble Kalman Filter. The algorithm used is the SVD method presented by Evensen in 2003 (Evensen, G., Ocean dynamics, 53, pp.343–367, 2003). To compensate for model errors, three inflation algorithms are implemented. The inflation methodology is specified by the inflation argument, where the options are the following:

- inflation == 0: Add model error (perturbation for the ensemble) around model state values only where observations are available (DEFAULT).
- inflation == 1: Add model error (perturbation for the ensemble) around observation values only where observations are available.

- inflation == 2: Inflate around ensemble average for EnKF.

Prior to assimilation, a set of data values has to be specified by setting the start and end dates, and time step, using the `setup_ticks` function of the radiation belt model:

```
>> import spacepy
>> import datetime
>> from spacepy import radbelt

>> start = datetime.datetime(2002,10,23)
>> end = datetime.datetime(2002,11,4)
>> delta = datetime.timedelta(hours=0.5)
>> rmod.setup_ticks(start, end, delta, dtype='UTC')
```

Once the dates and time step are specified, the data is added using the `add_PSD` function:

```
>> rmod.add_PSD()
```

The observations are averaged over the time windows, whose interval is give by the time step. Once the dates and data are set, the assimilation is performed using the `assimilate` function:

```
>> rmod.assimilate(inflation=1)
```

This function will add the PSDa values, which are the analysis state of the radiation belt using the observations within the dates. To plot the analysis simply use the `plot` function:

```
>> rmod.plot(values=rmod.PSDa,clims=[-10,-6],Lmax=False,Kp=False,Dst=False)
```

Additionally, to create a summary plot of the observations use the `plot_obs` function within the radbelt module. For reference, the last closed drift shell, Dst, and Kp are all included. These can be disabled individually using the corresponding Boolean kwargs.

The clims kwarg can be used to manually set the color bar range. To use, set it equal to a two-element list containing minimum and maximum Log_10 value to plot. Default action is to use [0,10] as the log_10 of the color range. This is good enough for most applications. The title of the top most plot defaults to 'Summary Plot' but can be customized using the title kwarg.

The figure object and all three axis objects (PSD axis, Dst axis, and Kp axis) are all returned to allow the user to further customize the plots as necessary. If any of the plots are excluded, None is returned in their stead.

Example:

```
>>> rmod.plot_obs(clims=[-10,-6],Lmax=False,Kp=False,Dst=False,title='Observations Plot')
```

This command would create the summary plot with a color bar range of 10^(-10) to 10^(-6). The Lmax line, Kp and Dst values would be excluded. The title of the topmost plot (phase space density) would be set to 'Observations Plot'.

1.1.6 OMNI Module

The OMNI database is an hourly resolution, multi-source data set with coverage from November 1963; higher temporal resolution versions of the OMNI database exist, but with coverage from 1995. The primary data are near-Earth solar wind, magnetic field and plasma parameters. However, a number of modern magnetic field models require derived input parameters, and Qin and Denton (2007) have used the publicly-available OMNI database to provide a modified version of this database containing all parameters necessary for these magnetic field models. These data are available through ViRBO - the Virtual Radiation Belt Observatory.

In SpacePy this data is made available on request on install; if not downloaded when SpacePy is installed and attempt to import the omni module will ask the user whether they wish to download the data. Should the user require the latest data, the toolbox.update function can be used to fetch the latest files from ViRBO.

The following example fetches the OMNI data for the storms of October and November, 2003.:

```
>>> import spacepy.time as spt
>>> import spacepy.omni as om
>>> import datetime as dt
>>> st = dt.datetime(2003,10,20)
>>> en = dt.datetime(2003,12,5)
>>> delta = dt.timedelta(days=1)
>>> ticks = spt.tickrange(st, en, delta, 'UTC')
>>> data = om.get_omni(ticks)
```

data is a dictionary containing all the OMNI data, by variable, for the timestamps contained within the `Ticktock` object *ticks*. Now it is simple to plot Dst values for instance:

```
>>> import pyplot as p
>>> p.plot(ticks.eDOY, data['Dst'])
```

1.1.7 The irbempy Module

ONERA (Office National d'Etudes et Recherches Aerospatiales) initiated a well-known FORTRAN library that provides routines to compute magnetic coordinates for any location in the Earth's magnetic field, to perform coordinate conversions, to compute magnetic field vectors in geospace for a number of external field models, and to propagate satellite orbits in time. Older versions of this library were called ONERA-DESP-LIB. Recently the library has changed its name to IRBEM-LIB and is maintained by a number of different institutions.

A number of key routines in IRBEM-LIB have been made available through the module *irbempy*. Current functionality includes calls to calculate the local magnetic field vectors at any point in geospace, calculation of the magnetic mirror point for a particle of a given pitch angle (the angle between a particle's velocity vector and the magnetic field line that it immediately orbits such that a pitch angle of 90 degrees signifies gyration perpendicular to the local field) anywhere in geospace, and calculation of electron drift shells in the inner magnetosphere.:

```
>>> import spacepy.time as spt
>>> import spacepy.coordinates as spc
>>> import spacepy.irbempy as ib
>>> t = spt.Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> y = spc.Coords([[3,0,0],[2,0,0]], 'GEO', 'car')
>>> ib.get_Bfield(t,y)
{'Blocal': array([  976.42565251,  3396.25991675]),
   'Bvec': array([[ -5.01738885e-01,  -1.65104338e+02,   9.62365503e+02],
   [  3.33497974e+02,  -5.42111173e+02,   3.33608693e+03]])}
```

One can also calculate the drift shell L* for a 90 degree pitch angle value by using:

```
>>> ib.get_Lstar(t,y, [90])
{'Bmin': array([  975.59122652,  3388.2476667 ]),
 'Bmirr': array([[  976.42565251],
   [ 3396.25991675]]),
 'Lm': array([[ 3.13508015],
   [ 2.07013638]]),
 'Lstar': array([[ 2.86958324],
   [ 1.95259007]]),
 'MLT': array([ 11.97222034,  12.13378624]),
 'Xj': array([[ 0.00081949],
   [ 0.00270321]])}
```

Other function wrapped with the IRBEM library include:

- `find_Bmirror`

- find_magequator
- corrd_trans

1.1.8 Pycdf - Python Access to NASA CDF Library

pycdf provides a "pythonic" interface to the NASA CDF library. It requires that the base C library be properly installed. The module can then be imported, e.g.:

```
>>> import spacepy.pycdf as cdf
```

Extensive documentation is provided in epydoc format in docstrings.

To open and close a CDF file:

```
>>> cdf_file = cdf.CDF('filename.cdf')
>>> cdf_file.close()
```

CDF files, like standard Python files, act as context managers:

```
>>> with cdf.CDF('filename.cdf') as cdf_file:
...     #do brilliant things with cdf_file
>>> #cdf_file is automatically closed here
```

CDF files act as Python dictionaries, holding CDF variables keyed by the variable name:

```
>>> var_names = keys(cdf_file) #list of all variables
>>> for var_name in cdf_file:
...     print(len(cdf_file[var_name])) #number of records in each variable

    #list comprehensions work, too
>>> lengths = [len(cdf_file[var_name]) for var_name in cdf_file]
```

Each CDF variable acts as a Python list, one element per record. Multidimensional CDF variables are represented as nested lists and can be subscripted using a multidimensional slice notation similar to numpy. Creating a Python Var object does not read the data from disc; data are only read as they are accessed:

```
>>> epoch = cdf_file['Epoch'] #Python object created, nothing read from disc
>>> epoch[0] #time of first record in CDF (datetime object)
>>> a = epoch[...] #copy all times to list a
>>> a = epoch[-5:] #copy last five times to list a
>>> b_gse = cdf_file['B_GSE'] #B_GSE is a 1D, three-element array
>>> bz = b_gse[0,2] #Z component of first record
>>> bx = b_gse[:,0] #copy X component of all records to bx
>>> bx = cdf_file['B_GSE'][:,0] #same as above
```

1.1.9 The datamodel Module

The SpacePy datamodel module implements classes that are designed to make implementing a standard data model easy. The concepts are very similar to those used in standards like HDF5, netCDF and NASA CDF.

The basic container type is analogous to a folder (on a filesystem; HDF5 calls this a group): Here we implement this as a dictionary-like object, a datamodel.SpaceData object, which also carries attributes. These attributes can be considered to be global, i.e. relevant for the entire folder. The next container type is for storing data and is based on a numpy array, this class is datamodel.dmarray and also carries attributes. The dmarray class is analogous to an HDF5 dataset.

Guide for NASA CDF users

By definition, a NASA CDF only has a single 'layer'. That is, a CDF contains a series of records (stored variables of various types) and a set of attributes that are either global or local in scope. Thus to use SpacePy's datamodel to capture the functionality of CDF the two basic data types are all that is required, and the main constraint is that datamodel.SpaceData objects cannot be nested (more on this later, if conversion from a nested datamodel to a flat datamodel is required).

This is best illustrated with an example. Imagine representing some satellite data within a CDF – the global attributes might be the mission name and the instrument PI, the variables might be the instrument counts [n-dimensional array], timestamps[1-dimensional array and an orbit number [scalar]. Each variable will have one attribute (for this example).

```
>>> import spacepy.datamodel as dm
>>> mydata = dm.SpaceData(attrs={'MissionName': 'BigSat1'})
>>> mydata['Counts'] = dm.dmarray([[42, 69, 77], [100, 200, 250]], attrs={'Units': 'cnts/s'})
>>> mydata['Epoch'] = dm.dmarray([1, 2, 3], attrs={'units': 'minutes'})
>>> mydata['OrbitNumber'] = dm.dmarray(16, attrs={'StartsFrom': 1})
>>> mydata.attrs['PI'] 'Prof. Big Shot'
```

This has now populated a structure that can map directly to a NASA CDF. To visualize our datamodel, we can use the toolbox function dictree (which works for any dictionary-like object, including PyCDF file objects).

```
>>> import spacepy.toolbox as tb
>>> tb.dictree(mydata, attrs=True)
+
:|____MissionName
:|____PI
|____Counts
    :|____Units
|____Epoch
    :|____units
|____OrbitNumber
    :|____StartsFrom
```

Attributes are denoted by a leading colon. The global attributes are those in the base level, and the local attributes are attached to each variable.

If we have data that has nested 'folders', allowed by HDF5 but not by NASA CDF, then how can this be represented such that the data structure can be mapped directly to a NASA CDF? The data will need to be flattened so that it is single layered. Let us now store some ephemerides in our data structure:

```
>>> mydata['Ephemeris'] = dm.SpaceData()
>>> mydata['Ephemeris']['GSM'] = dm.dmarray([[1,3,3], [1.2,4,2.5], [1.4,5,1.9]])
>>> tb.dictree(mydata, attrs=True)
+
:|____MissionName
:|____PI
|____Counts
    :|____Units
|____Ephemeris
    |____GSM
|____Epoch
    :|____units
|____OrbitNumber
    :|____StartsFrom
```

Nested dictionary-like objects is not uncommon in Python (an can be exceptionally useful for representing data, so to make this compatible with NASA CDF we call the object method 'flatten'.

```
>>> mydata.flatten()
>>> tb.dictree(mydata, attrs=True)
+
:|____MissionName
:|____PI
|____Counts
    :|____Units
|____Ephemeris<--GSM
|____Epoch
    :|____units
|____OrbitNumber
    :|____StartsFrom
```

Note that the nested SpaceData has been moved to a variable with a new name reflecting its origin. The data structure is now flat again and can be mapped directly to NASA CDF.

Converters to/from datamodel

Currently converters to HDF5 and NASA CDF are under development, as are extractors that unpack data from these formats into a SpacePy datamodel. Also under development is the reverse of the SpaceData.flatten method, so that flattened objects can be restored to their former glory.

1.1.10 Empiricals Module

The empiricals module provides access to some useful empirical models. As of SpacePy 0.1.0, the models available are:

- An empirical parametrization of the L* of the last closed drift shell (Lmax)
- The plasmapause location, following either Carpenter and Anderson (1992) or Moldwin et al. (2002)
- The magnetopause standoff location (i.e. the sub-solar point), using the Shue et al. (1997) model

Each model is called by passing it a Ticktock object (see above) which then calculates the model output using the 1-hour Qin-Denton OMNI data (from the OMNI module; see above). For example:

```
>>> import spacepy.time as spt
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2002-01-01T12:00:00','2002-01-04T00:00:00',.25)
```

calls the tickrange function from spacepy.time and makes a Ticktock object with times from midday on January 1st 2002 to midnight January 4th 2002, incremented 6-hourly:

```
>>> Lpp = emp.getPlasmaPause(ticks)
```

then returns the model plasmapause location using the default setting of the Moldwin et al. (2002) model. The Carpenter and Anderson model can be used by setting the Lpp_model keyword to 'CA1992'.

The magnetopause standoff location can be called using this syntax, or can be called for specific solar wind parameters (ram pressure, P, and IMF Bz) passed through in a Python dictionary:

```
>>> data = {'P': [2,4], 'Bz': [-2.4, -2.4]}
>>> emp.getMPstandoff(data)
array([ 10.29156018,   8.96790412])
```

1.1.11 SeaPy - Superposed Epoch Analysis in Python

Superposed epoch analysis is a technique used to reveal consistent responses, relative to some repeatable phenomenon, in noisy data . Time series of the variables under investigation are extracted from a window around the epoch and all data at a given time relative to epoch forms the sample of events at that lag. The data at each time lag are then averaged so that fluctuations not consistent about the epoch cancel. In many superposed epoch analyses the mean of the data at each time u relative to epoch, is used to represent the central tendency. In SeaPy we calculate both the mean and the median, since the median is a more robust measure of central tendency and is less affected by departures from normality. SeaPy also calculates a measure of spread at each time relative to epoch when performing the superposed epoch analysis; the interquartile range is the default, but the median absolute deviation and bootstrapped confidence intervals of the median (or mean) are also available.

As an example we fetch OMNI data for 4 years and perform a superposed epoch analysis of the solar wind radial velocity, with a set of epoch times read from a text file:

```
>>> import spacepy.seapy as se
>>> import spacepy.omni as om
>>> import spacepy.toolbox as tb
    #now read the epochs for the analysis
>>> epochs = se.readepochs('epochs_OMNI.txt', iso=True)
>>> st, en = datetime.datetime(2005,1,1), datetime.datetime(2009,1,1)
```

The readepochs function can handle multiple formats by a user-specified format code. ISO 8601 format is directly supported. As an alternative to the getOMNI function used above, we can get the hourly data directly from the OMNI module using a toolbox function:

```
>>> einds, oinds = tb.tOverlap([st, en], om.omnidata['UTC'])
>>> omni1hr = array(om.omnidata['UTC'])[oinds]
>>> omniVx = om.omnidata['velo'][oinds]
```

and these data are used for the superposed epoch analysis. the temporal resolution is 1 hr and the window is +/- 3 days

```
>>> delta = datetime.timedelta(hours=1)
>>> window= datetime.timedelta(days=3)
>>> sevx = se.Sea(omniVx, omni1hr, epochs, window, delta)
    #rather than quartiles, we calculate the 95% confidence interval on the median
>>> sevx.sea(ci=True)
>>> sevx.plot()
```

1.2 Documentation Standard

SpacePy is trying to be a high quality product and as such we are striving for a certain amount of uniformity among modules in the package. Please take the time to make you code compliant with the documentation standard.

SpacePy uses Sphinx as its documentation system

On top of Sphinx SpacePy uses the following extensions:

- 'sphinx.ext.autodoc'
- 'sphinx.ext.doctest''
- 'sphinx.ext.intersphinx'
- 'sphinx.ext.todo'
- 'sphinx.ext.pngmath'
- 'sphinx.ext.ifconfig'

- 'sphinx.ext.viewcode'
- 'numpydoc'
- 'sphinx.ext.inheritance_diagram'
- 'sphinx.ext.autosummary'
- 'sphinx.ext.extlinks'

1.2.1 So what do I need to do in my code?

Since we are using the 'numpydoc' extension there are fixed headings that may appear in your documentation block, there are a few things to note:

- No other headings can appear there
- I believe that no reStructuredText commands can either (e.g. .. Note:)

Allowed headings

Always use

- Parameters
- Returns

Use as needed

- Attributes
- Raises
- Warns
- Other Parameters
- See Also
- Notes
- Warnings
- References
- Examples
- Methods

No need to use

- Summary
- Extended Summary
- index

Do not use

- Signature

Examples

- Use them, but they must be fully stand alone, the user can just copy-paste exactly what is there and it should work (think, and probably run doctest on them)

1.2.2 Method Example

This code from toolbox shows what a method should look like in your code

```
def logspace(min, max, num, **kwargs):
    """
    Returns log spaced bins.  Same as numpy logspace except the min and max are the ,min and max
    not log10(min) and log10(max)

    Parameters
    ==========
    min : float
        minimum value
    max : float
        maximum value
    num : integer
        number of log spaced bins

    Other Parameters
    ================
    kwargs : dict
        additional keywords passed into matplotlib.dates.num2date

    Returns
    =======
    out : array
        log spaced bins from min to max in a numpy array

    Notes
    =====
    This function works on both numbers and datetime objects

    Examples
    ========
    >>> import spacepy.toolbox as tb
    >>> tb.logspace(1, 100, 5)
    array([   1.        ,    3.16227766,   10.        ,   31.6227766 ,  100.        ])
    """
    from numpy import logspace, log10
    if isinstance(min, datetime.datetime):
        from matplotlib.dates import date2num, num2date
        return num2date(logspace(log10(date2num(min)), log10(date2num(max)), num, **kwargs))
    else:
        return logspace(log10(min), log10(max), num, **kwargs)
```

Which then renders as:

toolbox.**logspace**(*min*, *max*, *num*, ***kwargs*)

Returns log-spaced bins. Same as numpy.logspace except the min and max are the min and max not log10(min) and log10(max)

Parameters **min** : float

minimum value

max : float

maximum value

num : integer

number of log spaced bins

Returns **out** : array

log-spaced bins from min to max in a numpy array

Other Parameters **kwargs** : dict

additional keywords passed into matplotlib.dates.num2date

Notes

This function works on both numbers and datetime objects

Examples

```
>>> import sapcepy.toolbox as tb
>>> tb.logspace(1, 100, 5)
array([   1.        ,    3.16227766,   10.        ,   31.6227766 ,  100.        ])
```

1.3 SpacePy Python Programming Tips

One often hears that interpreted languages are too slow for whatever task someone needs to do. In many cases this is exactly wrong-headed. As the time spent programming/debugging in an interpreted language is less than a compiled language the programmer has time to figure out where code is slow and make it faster. This page is dedicated to that idea, providing examples of code speedup and best practices.

1.3.1 Lists, for loops, and arrays

This example teaches the lesson that every IDL or Matlab programmer already knows; do everything in arrays and never use a for loop if there is another choice.

The following bit of code takes in a series of points, computes their magnitude, and drops the largest 100 of them.

This is how the code started out, Shell_x0_y0_z0 is an Nx3 numpy array, ShellCenter is a 3 element list or array, and Num_Pts_Removed is the number of points to drop:

```
import numpy as np
def SortRemove_HighFluxPts_(Shell_x0_y0_z0, ShellCenter, Num_Pts_Removed):
    #Sort the Shell Points based on radial distance (Flux prop to 1/R^2) and remove Num_Pts_Removed p
    Num_Pts_Removed = np.abs(Num_Pts_Removed)  #make sure the number is positive
    #Generate an array of radial distances of points from origin
    R = []
    for xyz in Shell_x0_y0_z0:
        R.append(1/np.linalg.norm(xyz + ShellCenter)) #Flux prop to 1/r^2, but don't need the ^2
    R = np.asarray(R)
    ARG = np.argsort(R)    # array of sorted indies based on flux in 1st column
    Shell_x0_y0_z0 = np.take(Shell_x0_y0_z0, ARG, axis = 0)  # sort based on index order
    return Shell_x0_y0_z0[:-Num_Pts_Removed,:]   #remove last points that have the "anomalously" high
```

A cProfile of this yields a lot of time spent just in the function itself; this is the for loop (list comprehension is a little faster but not much in this case):

```
Tue Jun 14 10:10:56 2011    SortRemove_HighFluxPts_.prof

         700009 function calls in 4.209 seconds

   Ordered by: cumulative time
   List reduced from 14 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.002    0.002    4.209    4.209 <string>:1(<module>)
        1    2.638    2.638    4.207    4.207 test1.py:235(SortRemove_HighFluxPts_)
   100000    0.952    0.000    1.529    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
   100001    0.099    0.000    0.240    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
   100000    0.229    0.000    0.229    0.000 {method 'reduce' of 'numpy.ufunc' objects}
   100001    0.141    0.000    0.141    0.000 {numpy.core.multiarray.array}
   100000    0.082    0.000    0.082    0.000 {method 'ravel' of 'numpy.ndarray' objects}
   100000    0.042    0.000    0.042    0.000 {method 'conj' of 'numpy.ndarray' objects}
   100000    0.016    0.000    0.016    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.005    0.005 /opt/local/Library/Frameworks/Python.framework/Versions
```

Simply pulling out the addition inside the for loop makes an amazing difference (2.3x speedup). We believe the difference is that pulling out the addition lets numpy do its thing in C once only (saving a massive overhead as array operations are done as for loops in C) and not in python for each element:

```
def SortRemove_HighFluxPts_(Shell_x0_y0_z0, ShellCenter, Num_Pts_Removed):
    #Sort the Shell Points based on radial distance (Flux prop to 1/R^2) and remove Num_Pts_Removed p
    Num_Pts_Removed = np.abs(Num_Pts_Removed)  #make sure the number is positive
    #Generate an array of radial distances of points from origin
    R = []
    Shell_xyz = Shell_x0_y0_z0 + ShellCenter
    for xyz in Shell_xyz:
        R.append(1/np.linalg.norm(xyz)) #Flux prop to 1/r^2, but don't need the ^2
    R = np.asarray(R)
    ARG = np.argsort(R)    # array of sorted indies based on flux in 1st column
    Shell_x0_y0_z0 = np.take(Shell_x0_y0_z0, ARG, axis = 0)  # sort based on index order
    return Shell_x0_y0_z0[:-Num_Pts_Removed,:]   #remove last points that have the "anomalously" high
```

A quick profile:

```
Tue Jun 14 10:18:39 2011    SortRemove_HighFluxPts_.prof

         700009 function calls in 1.802 seconds

   Ordered by: cumulative time
   List reduced from 14 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.001    0.001    1.802    1.802 <string>:1(<module>)
        1    0.402    0.402    1.801    1.801 test1.py:235(SortRemove_HighFluxPts_)
   100000    0.862    0.000    1.361    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
   100000    0.207    0.000    0.207    0.000 {method 'reduce' of 'numpy.ufunc' objects}
   100001    0.080    0.000    0.199    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
   100001    0.120    0.000    0.120    0.000 {numpy.core.multiarray.array}
   100000    0.067    0.000    0.067    0.000 {method 'ravel' of 'numpy.ndarray' objects}
   100000    0.041    0.000    0.041    0.000 {method 'conj' of 'numpy.ndarray' objects}
   100000    0.014    0.000    0.014    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.005    0.005 /opt/local/Library/Frameworks/Python.framework/Versions
```

A closer look here reveals that all of this can be done on the arrays without the for loop (or list comprehension):

```
def SortRemove_HighFluxPts_(Shell_x0_y0_z0, ShellCenter, Num_Pts_Removed):
    #Sort the Shell Points based on radial distance (Flux prop to 1/R^2) and remove # points with the
    Num_Pts_Removed = np.abs(Num_Pts_Removed)  #make sure the number is positive
    #Generate an array of radial distances of points from origin
    R = 1 / np.sum((Shell_x0_y0_z0 + ShellCenter) ** 2, 1)
    ARG = np.argsort(R)   # array of sorted indies based on flux in 1st column
    Shell_x0_y0_z0 = np.take(Shell_x0_y0_z0, ARG, axis = 0)  # sort based on index order
    return Shell_x0_y0_z0[:-Num_Pts_Removed,:]   #remove last points that have the "anomalously" high
```

The answer is exactly the same and from where we started this is a 382x speedup:

```
Tue Jun 14 10:21:54 2011    SortRemove_HighFluxPts_.prof

         10 function calls in 0.011 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.011    0.011 <string>:1(<module>)
        1    0.002    0.002    0.011    0.011 test1.py:236(SortRemove_HighFluxPts_)
        1    0.000    0.000    0.004    0.004 /opt/local/Library/Frameworks/Python.framework/Versions
        1    0.004    0.004    0.004    0.004 {method 'argsort' of 'numpy.ndarray' objects}
        1    0.000    0.000    0.003    0.003 /opt/local/Library/Frameworks/Python.framework/Versions
        1    0.003    0.003    0.003    0.003 {method 'take' of 'numpy.ndarray' objects}
        1    0.000    0.000    0.002    0.002 /opt/local/Library/Frameworks/Python.framework/Versions
        1    0.002    0.002    0.002    0.002 {method 'sum' of 'numpy.ndarray' objects}
        1    0.000    0.000    0.000    0.000 {isinstance}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Overall think really hard before you write a for loop or a list comprehension.

1.3.2 Zip

the zip function is a great thing but it is really slow, if you find yourself using it then you probably need to reexamine the algorithm that you are using A good alternative, if you do need the functionality of zip, is in `itertools.izip()`.

This example generate evenly distributed N points on the unit sphere centered at (0,0,0) using the "Golden Spiral" method.

The original code:

```
import numpy as np
def PointsOnSphere_(N):
# Generate evenly distributed N points on the unit sphere centered at (0,0,0)
# Uses "Golden Spiral" method
    x0 = np.array((N,), dtype= float)
    y0 = np.array((N,), dtype= float)
    z0 = np.array((N,), dtype= float)
    phi = (1 + np.sqrt(5)) / 2. # the golden ratio
    long_incr = 2.0*np.pi / phi # how much to increment the longitude
    dz = 2.0 / float(N)     # a unit sphere has diameter 2
    bands = np.arange(0, N, 1) # each band will have one point placed on it
    z0 = bands * dz - 1 + (dz/2)  # the z location of each band/point
    r = np.sqrt(1 - z0*z0)     # the radius can be directly determined from height
    az = bands * long_incr # the azimuth where to place the point
    x0 = r * np.cos(az)
    y0 = r * np.sin(az)
```

```
    x0_y0_z0 = np.array(zip(x0,y0,z0))      #combine into 3 column (x,y,z) file
    return (x0_y0_z0)
```

Profiling this with cProfile one can see a lot of time in zip():

```
Tue Jun 14 09:54:41 2011    PointsOnSphere_.prof

         9 function calls in 8.132 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.010    0.010    8.132    8.132 <string>:1(<module>)
        1    0.470    0.470    8.122    8.122 test1.py:192(PointsOnSphere_)
        4    6.993    1.748    6.993    1.748 {numpy.core.multiarray.array}
        1    0.654    0.654    0.654    0.654 {zip}
        1    0.005    0.005    0.005    0.005 {numpy.core.multiarray.arange}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

So lets try and do a few simple rewrites to make this faster. Using numpy.vstack is the first one that came to mind. The change here is to replace building up the array from the elements made by zip to just appending the data we already have to an array that we already have:

```
def PointsOnSphere_(N):
# Generate evenly distributed N points on the unit sphere centered at (0,0,0)
# Uses "Golden Spiral" method
    x0 = np.array((N,), dtype= float)
    y0 = np.array((N,), dtype= float)
    z0 = np.array((N,), dtype= float)
    phi = (1 + np.sqrt(5)) / 2. # the golden ratio
    long_incr = 2.0*np.pi / phi # how much to increment the longitude
    dz = 2.0 / float(N)     # a unit sphere has diameter 2
    bands = np.arange(0, N, 1) # each band will have one point placed on it
    z0 = bands * dz - 1 + (dz/2)  # the z location of each band/point
    r = np.sqrt(1 - z0*z0)     # the radius can be directly determined from height
    az = bands * long_incr # the azimuth where to place the point
    x0 = r * np.cos(az)
    y0 = r * np.sin(az)
    x0_y0_z0 = np.vstack((x0, y0, z0)).transpose()
    return (x0_y0_z0)
```

Profiling this with cProfile one can see that this is now fast enough for me, no more work to do. We picked up a 48x speed increase, I'm sure this can still be made better and let the spacepy team know if you rewrite it and it will be included:

```
Tue Jun 14 09:57:41 2011    PointsOnSphere_.prof

         32 function calls in 0.168 seconds

   Ordered by: cumulative time
   List reduced from 13 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.010    0.010    0.168    0.168 <string>:1(<module>)
        1    0.123    0.123    0.159    0.159 test1.py:217(PointsOnSphere_)
        1    0.000    0.000    0.034    0.034 /opt/local/Library/Frameworks/Python.framework/Versions
        1    0.034    0.034    0.034    0.034 {numpy.core.multiarray.concatenate}
        1    0.002    0.002    0.002    0.002 {numpy.core.multiarray.arange}
        1    0.000    0.000    0.000    0.000 {map}
```

```
     3    0.000    0.000    0.000    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
     6    0.000    0.000    0.000    0.000 {numpy.core.multiarray.array}
     3    0.000    0.000    0.000    0.000 /opt/local/Library/Frameworks/Python.framework/Versions
     1    0.000    0.000    0.000    0.000 {method 'transpose' of 'numpy.ndarray' objects}
```

Release 0.1.0

Doc generation date August 23, 2011

For additions or fixes to this page contact Brian Larsen at Los Alamos.

CHAPTER

TWO

SPACEPY CODE

2.1 coordinates - module for coordinate transforms

Implementation of Coords class functions for coordinate transformations

Authors: Josef Koller Institution: Los ALamos National Laboratory Contaact: jkoller@lanl.gov

class coordinates.**Coords**(*data*, *dtype*, *carsph*[, *units*, *ticks*])

A class holding spatial coordinates in Cartesian/spherical in units of Re and degrees

Parameters **data** : list or ndarray, dim = (n,3)

coordinate points [X,Y,Z] or [rad, lat, lon]

dtype : string

coordinate system, possible are GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL

carsph : string

Cartesian or spherical, 'car' or 'sph'

units : list of strings, optional

standard are ['Re', 'Re', 'Re'] or ['Re', 'deg', 'deg'] depending on the carsph content

ticks : Ticktock instance, optional

used for coordinate transformations (see a.convert)

Returns **out** : Coords instance

instance with a.data, a.carsph, etc.

See Also:

spacepy.time.Ticktock

Examples

```
>>> from spacepy import coordinates as coord
>>> cvals = coord.Coords([[1,2,4],[1,2,2]], 'GEO', 'car')
>>> cvals.x  # returns all x coordinates
array([1, 1])
>>> from spacepy.time import Ticktock
>>> cvals.ticks = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:00:00'], 'ISO') # add ticks
```

```
>>> newcoord = cvals.convert('GSM', 'sph')
>>> newcoord
```

Methods

append
convert

append (*other*)
Will be called when another Coords instance has to be appended to the current one

Parameters **other** : Coords instance

Coords instance to append

convert (*returntype*, *returncarsph*)
Can be used to create a new Coords instance with new coordinate types

Parameters **returntype** : string

coordinate system, possible are GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL

returncarsph : string

coordinate type, possible 'car' for Cartesian and 'sph' for spherical

Returns **out** : Coords object

Coords object in the new coordinate system

Examples

```
>>> from spacepy.coordinates import Coords
>>> y = Coords([[1,2,4],[1,2,2]], 'GEO', 'car')
>>> from spacepy.time import Ticktock
>>> y.ticks = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:00:00'], 'ISO')
>>> x = y.convert('SM','car')
>>> x
Coords( [[ 0.81134097  2.6493305   3.6500375 ]
 [ 0.92060408  2.30678864  1.68262126]] ), dtype=SM,car, units=['Re', 'Re', 'Re']
```

2.2 datamodel - easy to use general data model

The datamodel classes consitute a data model implementation meant to mirror the functionality of the data model output from pycdf, though implemented slightly differently.

This contains the following classes:

- dmarray - numpy arrays that support .attrs for information about the data
- **SpaceData - base class that extends dict, to be extended by others** Currently used in GPScode and other projects

Authors: Steve Morley and Brian Larsen Additional Contributors: Charles Kiyanda and Miles Engel Institution: Los Alamos National Laboratory Contact: smorley@lanl.gov; balarsen@lanl.gov

2.2.1 About datamodel

The SpacePy datamodel module implents classes that are designed to make implementing a standard data model easy. The concepts are very similar to those used in standards like HDF5, netCDF and NASA CDF.

The basic container type is analagous to a folder (on a filesystem; HDF5 calls this a group): Here we implement this as a dictionary-like object, a datamodel.SpaceData object, which also carries attributes. These attributes can be considered to be global, i.e. relevant for the entire folder. The next container type is for storing data and is based on a numpy array, this class is datamodel.dmarray and also carries attributes. The dmarray class is analagous to an HDF5 dataset.

In fact, HDF5 can be loaded directly into a SpacePy datamodel, carrying across all attributes, using the function fromHDF5:

```
>>> import spacepy.datamodel as dm
>>> data = dm.fromHDF5('test.h5')
```

2.2.2 Guide for NASA CDF users

By definition, a NASA CDF only has a single 'layer'. That is, a CDF contains a series of records (stored variables of various types) and a set of attributes that are either global or local in scope. Thus to use SpacePy's datamodel to capture the functionality of CDF the two basic data types are all that is required, and the main constraint is that datamodel.SpaceData objects cannot be nested (more on this later, if conversion from a nested datamodel to a flat datamodel is required).

This is best illustrated with an example. Imagine representing some satellite data within a CDF – the global attributes might be the mission name and the instrument PI, the variables might be the instrument counts [n-dimensional array], timestamps[1-dimensional array and an orbit number [scalar]. Each variable will have one attribute (for this example).

```
>>> import spacepy.datamodel as dm
>>> mydata = dm.SpaceData(attrs={'MissionName': 'BigSat1'})
>>> mydata['Counts'] = dm.dmarray([[42, 69, 77], [100, 200, 250]], attrs={'Units': 'cnts/s'})
>>> mydata['Epoch'] = dm.dmarray([1, 2, 3], attrs={'units': 'minutes'})
>>> mydata['OrbitNumber'] = dm.dmarray(16, attrs={'StartsFrom': 1})
>>> mydata.attrs['PI'] 'Prof. Big Shot'
```

This has now populated a structure that can map directly to a NASA CDF. To visualize our datamodel, we can use the toolbox function dictree (which works for any dictionary-like object, including PyCDF file objects).

```
>>> import spacepy.toolbox as tb
>>> tb.dictree(mydata, attrs=True)
+
:|____MissionName
:|____PI
|____Counts
     :|____Units
|____Epoch
     :|____units
|____OrbitNumber
     :|____StartsFrom
```

Opening a CDF and working directly with the contents can be easily done using the PyCDF module, however, if you wish to load the entire contents of a CDF directly into a datamodel (complete with attributes) the following will make life easier:

```
>>> import spacepy.datamodel as dm
>>> data = dm.fromCDF('test.cdf')
```

class datamodel.**SpaceData**(**args*, ***kwargs*)
Datamodel class extending dict

Methods

flatten

flatten()
Method to collapse datamodel to one level deep

Examples

```
>>> import spacepy.datamodel as dm
>>> import spacepy.toolbox as tb
>>> a = dm.SpaceData()
>>> a['1'] = dm.SpaceData(dog = 5, pig = dm.SpaceData(fish=dm.SpaceData(a='carp', b='perch')
>>> a['4'] = dm.SpaceData(cat = 'kitty')
>>> a['5'] = 4
>>> tb.dictree(a)
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5

>>> b = dm.flatten(a)
>>> tb.dictree(b)
+
|____1<--dog
|____1<--pig<--fish<--a
|____1<--pig<--fish<--b
|____4<--cat
|____5

>>> a.flatten()
>>> tb.dictree(a)
+
|____1<--dog
|____1<--pig<--fish<--a
|____1<--pig<--fish<--b
|____4<--cat
|____5
```

class datamodel.**dmarray**
Container for data within a SpaceData object

Raises **NameError** :

raised is the request name was not added to the allowed attributes list

Examples

```
>>> import spacepy.datamodel as datamodel
>>> position = datamodel.dmarray([1,2,3], attrs={'coord_system':'GSM'})
>>> position
dmarray([1, 2, 3])
>>> position.attrs
{'coord_system': 'GSM'}a
```

The dmarray, like a numpy ndarray, is versatile and can store any datatype; dmarrays are not just for arrays.

```
>>> name = datamodel.dmarray('TestName')
dmarray('TestName')
```

To extract the string (or scalar quantity), use the tolist method

```
>>> name.tolist()
'TestName'
```

Methods

addAttribute

addAttribute(*name*, *value=None*)
Method to add an attribute to a dmarray equilivant to a = datamodel.dmarray([1,2,3]) a.Allowed_Attributes = a.Allowed_Attributes + ['blabla']

datamodel.**flatten**(*dobj*)
Function to collapse datamodel to one level deep

See Also:

SpaceData.flatten

Examples

```
>>> import spacepy.datamodel as dm
>>> import spacepy.toolbox as tb
>>> a = dm.SpaceData()
>>> a['1'] = dm.SpaceData(dog = 5, pig = dm.SpaceData(fish=dm.SpaceData(a='carp', b='perch')))
>>> a['4'] = dm.SpaceData(cat = 'kitty')
>>> a['5'] = 4
>>> tb.dictree(a)
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5

>>> b = dm.flatten(a)
>>> tb.dictree(b)
```

```
+
|____1<--dog
|____1<--pig<--fish<--a
|____1<--pig<--fish<--b
|____4<--cat
|____5

>>> a.flatten()
>>> tb.dictree(a)
+
|____1<--dog
|____1<--pig<--fish<--a
|____1<--pig<--fish<--b
|____4<--cat
|____5
```

datamodel.**fromCDF**(*fname*, ***kwargs*)

Create a SpacePy datamodel representation of a NASA CDF file

Parameters **file** : string

the name of the cdf file to be loaded into a datamodel

Returns **out** : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Examples

```
>>> import spacepy.datamodel as dm
>>> data = dm.fromCDF('test.cdf')
```

datamodel.**fromHDF5**(*fname*, ***kwargs*)

Create a SpacePy datamodel representation of an HDF5 file

Parameters **file** : string

the name of the HDF5 file to be loaded into a datamodel

Returns **out** : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Notes

Known issues – zero-sized datasets will break in h5py This is kluged by returning a dmarray containing a None

Examples

```
>>> import spacepy.datamodel as dm
>>> data = dm.fromHDF5('test.hdf')
```

datamodel.**toHDF5**(*fname*, *SDobject*, ***kwargs*)

Create an HDF5 file from a SpacePy datamodel representation

Parameters **fname** : str

Filename to write to

SDobject : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Returns **None** :

2.3 empiricals - module with heliospheric empirical modules

Module with some useful empirical models (plasmapause, magnetopause, Lmax)

Authors: Steve Morley, Josef Koller Institution: Los Alamos National Laboratory Contact: smorley@lanl.gov

empiricals.**ShueMP**(*ticks*)

Calculates the Shue et al. (1997) subsolar magnetopause radius

Lets put the full reference here

Parameters **ticks** : spacepy.time.Ticktock

TickTock object of desired times (will be interpolated from hourly OMNI data) OR dictionary of form {'P': [], 'Bz': []} Where P is SW ram pressure [nPa] and Bz is IMF Bz (GSM) [nT]

Returns **out** : float

Magnetopause (sub-solar point) standoff distance [Re]

Examples

```
>>> import spacepy.time as spt
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2002-01-01T12:00:00','2002-01-04T00:00:00',.25)
>>> emp.ShueMP(ticks)
array([ 10.57319537,  10.91327764,  10.75086873,  10.77577207,
         9.78180261,  11.0374474 ,  11.4065    ,  11.27555451,
        11.47988573,  11.8202582 ,  11.23834814])
>>> data = {'P': [2,4], 'Bz': [-2.4, -2.4]}
>>> emp.ShueMP(data)
array([ 9.96096838,  8.96790412])
```

empiricals.**getDststar**(*ticks*, *model='OBrien'*)

Calculate the pressure-corrected Dst index, Dst*

We need to add in the references to the models here!

Parameters **ticks** : spacepy.time.Ticktock

TickTock object of desired times (will be interpolated from hourly OMNI data) OR dictionary including 'Pdyn' and 'Dst' keys where data are lists or arrays and Dst is in [nT], and Pdyn is in [nPa]

Returns **out** : float

Dst* - the pressure corrected Dst index from OMNI [nT]

Examples

Coefficients are applied to the standard formulation e.g. Burton et al., 1975 of Dst* = Dst - b*sqrt(Pdyn) + c The default is the O'Brien and McPherron model (2002). Other options are Burton et al. (1975) and Borovsky and Denton (2010)

```
>>> import spacepy.time as spt
>>> import spacepy.omni as om
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2000-10-16T00:00:00', '2000-10-31T12:00:00', 1/24.)
>>> dststar = emp.getDststar(ticks)
>>> dststar[0]
-21.317220132108943
```

User-determined coefficients can also be supplied as a two-element list or tuple of the form (b,c), e.g.

```
>>> dststar = emp.getDststar(ticks, model=(2,11)) #b is extreme driving from O'Brien
```

We have chosen the OBrien model as the default here as this was rigorously determined from a very long data set and is pertinent to most conditions. It is, however, the most conservative correction. Additionally, Siscoe, McPherron and Jordanova (2005) argue that the pressure contribution to Dst diminishes during magnetic storms.

To show the relative differences, run the following example:

```
>>> import matplotlib.pyplot as plt
>>> params = [('Burton','k-'), ('OBrien','r-'), ('Borovsky','b-')]
>>> for model, col in params:
        dststar = getDststar(ticks, model=model)
        plt.plot(ticks.UTC, dststar, col)
```

empiricals.**getLmax**(*ticks*, *model='JKemp'*)

calculate a simple empirical model for Lmax - last closed drift-shell

What is the paper this model is from? Put it here!

Parameters **ticks** : spacepy.time.Ticktock

Ticktock object of desired times

model : string, optional

'JKemp' (default - empirical model of J. Koller)

Returns **out** : np.ndarray

Lmax - L* of last closed drift shell

Examples

```
>>> from spacepy.empiricals import getLmax
>>> import spacepy.time as st
>>> import datetime
>>> ticks = st.tickrange(datetime.datetime(2000, 1, 1), datetime.datetime(2000, 1, 3), deltadays
array([ 7.4928412,  8.3585632,  8.6463423])
```

empiricals.**getMPstandoff**(*ticks*)

Calculates the Shue et al. (1997) subsolar magnetopause radius

Lets put the full reference here

Parameters **ticks** : spacepy.time.Ticktock

TickTock object of desired times (will be interpolated from hourly OMNI data) OR dictionary of form {'P': [], 'Bz': []} Where P is SW ram pressure [nPa] and Bz is IMF Bz (GSM) [nT]

Returns **out** : float

Magnetopause (sub-solar point) standoff distance [Re]

Examples

```
>>> import spacepy.time as spt
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2002-01-01T12:00:00','2002-01-04T00:00:00',.25)
>>> emp.ShueMP(ticks)
array([ 10.57319537,  10.91327764,  10.75086873,  10.77577207,
         9.78180261,  11.0374474 ,  11.4065    ,  11.27555451,
        11.47988573,  11.8202582 ,  11.23834814])
>>> data = {'P': [2,4], 'Bz': [-2.4, -2.4]}
>>> emp.ShueMP(data)
array([ 9.96096838,  8.96790412])
```

empiricals.**getPlasmaPause**(*ticks*, *model='M2002'*, *LT='all'*)

Plasmapause location model(s)

We need to list the references here!

Parameters **ticks** : spacepy.time.Ticktock

TickTock object of desired times

Lpp_model : string, optional

'CA1992' or 'M2002' (default) CA1992 returns the Carpenter and Anderson model, M2002 returns the Moldwin et al. model

LT : int, float

requested local time sector, 'all' is valid option

Returns **out** : float

Plasmapause radius in Earth radii

Examples

```
>>> import spacepy.time as spt
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2002-01-01T12:00:00','2002-01-04T00:00:00',.25)
>>> emp.getPlasmaPause(ticks)
array([ 6.42140002,  6.42140002,  6.42140002,  6.42140002,  6.42140002,
        6.42140002,  6.42140002,  6.26859998,  5.772     ,  5.6574    ,
        5.6574    ])
```

empiricals.**get_Lmax**(*ticks*, *model='JKemp'*)

calculate a simple empirical model for Lmax - last closed drift-shell

What is the paper this model is from? Put it here!

Parameters **ticks** : spacepy.time.Ticktock

Ticktock object of desired times

model : string, optional

'JKemp' (default - empirical model of J. Koller)

Returns **out** : np.ndarray

Lmax - L* of last closed drift shell

Examples

```
>>> from spacepy.empiricals import getLmax
>>> import spacepy.time as st
>>> import datetime
>>> ticks = st.tickrange(datetime.datetime(2000, 1, 1), datetime.datetime(2000, 1, 3), deltadays
array([ 7.4928412,  8.3585632,  8.6463423])
```

empiricals.**get_plasma_pause**(*ticks*, *model='M2002'*, *LT='all'*)

Plasmapause location model(s)

We need to list the references here!

Parameters **ticks** : spacepy.time.Ticktock

TickTock object of desired times

Lpp_model : string, optional

'CA1992' or 'M2002' (default) CA1992 returns the Carpenter and Anderson model, M2002 returns the Moldwin et al. model

LT : int, float

requested local time sector, 'all' is valid option

Returns **out** : float

Plasmapause radius in Earth radii

Examples

```
>>> import spacepy.time as spt
>>> import spacepy.empiricals as emp
>>> ticks = spt.tickrange('2002-01-01T12:00:00','2002-01-04T00:00:00',.25)
>>> emp.getPlasmaPause(ticks)
array([ 6.42140002,  6.42140002,  6.42140002,  6.42140002,  6.42140002,
    6.42140002,  6.42140002,  6.26859998,  5.772     ,  5.6574    ,
    5.6574    ])
```

2.4 omni - module top read and process NASA OMNIWEB data

tools to read and process omni data

Authors: Josef Koller Institution: Los Alamos National Laboratory Contact: jkoller@lanl.gov

omni.**get_G123**(*TAI*, *omnidata*)

get specific G1, G2, G3 for this TAI

omni.**get_omni**(*ticks*)

will load the pickled omni file, interpolate to the given ticks time and return the omni values as dictionary with Kp, Dst, dens, velo, Pdyn, ByIMF, BzIMF, G1, G2, G3, etc. (see also http://www.dartmouth.edu/~rdenton/magpar/index.html and http://www.agu.org/pubs/crossref/2007/2006SW000296.shtml)

Note carefully about Qbits: If the status variable is 2, the quantity you are using is fairly well determined. If it is 1, the value has some connection to measured values, but is not directly measured. These values are still better than just using an average value, but not as good as those with the status variable equal to 2. If the status variable is 0, the quantity is based on average quantities, and the values listed are no better than an average value. The lower the status variable, the less confident you should be in the value.

Parameters **ticks** : Ticktock class

containing time information

Returns **out** : dict

containing all omni values as a dictionary

Examples

```
>>> import spacepy.time as spt
>>> import spacepy.omni as om
>>> ticks = spt.Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> d = om.get_omni(ticks)
['velo', 'Bz6', 'Bz5', 'Bz4', 'Bz3', 'Bz2', 'Bz1', 'RDT', 'Dst',
'akp3', 'DOY', 'Qbits', 'G3', 'G2', 'G1', 'Hr', 'ticks', 'BzIMF',
'UTC', 'Kp', 'Pdyn', 'dens', 'ByIMF', 'W6', 'W5', 'W4', 'W3', 'W2',
'W1', 'Year']
```

2.5 PoPPy - Point Processes in Python

PoPPy – Point Processes in Python.

This module contains point process class types and a variety of functions for association analysis. The routines given here grew from work presented by Morley and Freeman (Geophysical Research Letters, 34, L08104, doi:10.1029/2006GL028891, 2007), which were originally written in IDL. This module is intended for application to discrete time series of events to assess statistical association between the series and to calculate confidence limits. Any mis-application or mis-interpretation by the user is the user's own fault.

```
>>> import datetime as dt
>>> import spacepy.time as spt
```

Since association analysis is rather computationally expensive, this example shows timing.

```
>>> t0 = dt.datetime.now()
>>> onsets = spt.Ticktock(onset_epochs, 'CDF')
>>> ticksR1 = spt.Ticktock(tr_list, 'CDF')
```

Each instance must be initialized

```
>>> lags = [dt.timedelta(minutes=n) for n in xrange(-400,401,2)]
>>> halfwindow = dt.timedelta(minutes=10)
>>> pp1 = poppy.PPro(onsets.UTC, ticksR1.UTC, lags, halfwindow)
```

To perform association analysis

```
>>> pp1.assoc()
Starting association analysis
calculating association for series of length [3494, 1323] at 401 lags
>>> t1 = dt.datetime.now()
>>> print("Elapsed:  " + str(t1-t0))
Elapsed:  0:35:46.927138
```

Note that for calculating associations between long series at a large number of lags is SLOW!!

To plot

```
>>> pp1.plot(dpi=80)
Error: No confidence intervals to plot - skipping
```

To add 95% confidence limits (using 4000 bootstrap samples)

```
>>> pp1.aa_ci(95, n_boots=4000)
```

The plot method will then add the 95% confidence intervals as a semi- transparent patch.

Authors: Steve Morley and Jon Niehof Institution: Los Alamos National Laboratory Contact: smorley@lanl.gov, jniehof@lanl.gov

class poppy.**PPro**(*process1*, *process2*, *lags=None*, *winhalf=None*, *verbose=False*)

PoPPy point process object

Initialize object with series1 and series2. These should be timeseries of events, given as lists, arrays, or lists of datetime objects. Includes method to perform association analysis of input series

Output can be nicely plotted with plot method

Methods

aa_ci
assoc
assoc_mult
plot
plot_mult
swap

aa_ci(*inter*, *n_boots=1000*, *seed=None*)

Get bootstrap confidence intervals for association number

Requires input of desired confidence interval, e.g., >>> obj.aa_ci(95)

Upper and lower confidence limits are added to the ci attribute

Attribute conf_above will contain the confidence (in percent) that the association number at that lag is I{above} the asymptotic association number. (The confidence of being below is 100 - conf_above) For minor variations in conf_above to be meaningful, a I{large} number of bootstraps is required. (Rougly, 1000 to be meaningful to the nearest percent; 10000 to be meaningful to a tenth of a percent.) A conf_above of 100 usually indicates an insufficient sample size to resolve, I{not} perfect certainty.

Note also that a 95% chance of being above indicates an exclusion from the *90*% confidence interval!

Parameters **inter** : float

percentage confidence interval to calculate

n_boots : int, optional

number of bootstrap iterations to run

seed : int, optional

seed for the random number generator. If not specified, Python code will use numpy's RNG and its current seed; C code will seed from the clock.

assoc(*u=None*, *h=None*)

Perform association analysis on input series

Parameters **u** : list, optional

the time lags to use

h : :

association window half-width, same type as process1

assoc_mult(*windows*, *inter=95*, *n_boots=1000*, *seed=None*)

Association analysis w/confidence interval on multiple windows

Using the time sequence and lags stored in this object, perform full association analysis, including bootstrapping of confidence intervals, for every listed window half-size

Parameters **windows** : sequence

window half-size for each analysis

inter : float, optional

desired confidence interval, default 95

n_boots : int, optional

number of bootstrap iterations, default 1000

seed : int, optional

Random number generator seed. It is STRONGLY recommended not to specify (i.e. leave None) to permit multithreading.

Returns **out** : three numpy array

Three numpy arrays, (windows x lags), containing (in order) low values of confidence interval, high values of ci, percentage confidence above the asymptotic association number

Warning: This function is likely to take a LOT of time.

plot(*figsize=None*, *dpi=80*, *asympt=True*, *show=True*, *norm=True*, *xlabel='Time lag'*, *xscale=None*, *ylabel=None*, *title=None*, *transparent=True*)

Create basic plot of association analysis.

Uses object attributes created by the assoc method and, optionally, aa_ci.

Parameters **figsize** : , optional

passed through to matplotlib.pyplot.figure

dpi : int, optional

passed through to matplotlib.pyplot.figure

asympt : boolean, optional

True to overplot the line of asymptotic association number

show : boolean, optional

Show the plot? (if false, will create without showing)

norm : boolean, optional

Normalize plot to the asymptotic association number

title : string, optional

label/title for the plot

xlabel : string, optional

label to put on the X axis of the resulting plot

xscale : float, optional

scale x-axis by this factor (e.g. 60.0 to convert seconds to minutes)

ylabel : string, optional

label to put on the Y axis of the resulting plot

transparent : boolean, optional

make c.i. patch transparent (default)

plot_mult(*windows*, *data*, *min=None*, *max=None*, *cbar_label=None*, *figsize=None*, *dpi=80*, *xlabel='Lag'*, *ylabel='Window Size'*)

Plots a 2D function of window size and lag

Parameters **windows** : list

list of window sizes (y axis)

data : list

list of data, dimensioned (windows x lags)

min : float, optional

clip L{data} to this minimum value

max : float, optional

clip L{data} to this maximum value

swap()

Swaps process 1 and process 2

poppy.**boots_ci**(*data*, *n*, *inter*, *func*, *seed=None*, *target=None*, *sample_size=None*)

Construct bootstrap confidence interval

The bootstrap is a statistical tool that uses multiple samples derived from the original data (called surrogates) to estimate a parameter of the population from which the sample was drawn. This assumes that the sample is randomly drawn and hence is representative of the underlying distribution. The benefit of the bootstrap is that it is non-parametric and can be applied in situations where there is reasonable doubt about the characteristics of the underlying distribution. This routine uses the boot- strap for its most common application - the estimation of confidence intervals.

Parameters **data** : array like

data to bootstrap

n : int

number of surrogate series to select, i.e. number of bootstrap iterations.

inter : numerical

desired percentage confidence interval

func : callable

Function to apply to each surrogate series

sample_size : int

number of samples in the surrogate series, default length of L{data}. This will change the statistical properties of the bootstrap and should only be used for good reason!

seed : int

Optional seed for the random number generator. If not specified, numpy generator will not be reseeded; C generator will be seeded from the clock.

target : same as data

a 'target' value. If specified, will also calculate percentage confidence of being at or above this value.

Returns **out** : sequence of float

inter percent confidence interval on value derived from func applied to the population sampled by data. If target is specified, also the percentage confidence of being above that value.

Examples

```
>>> data, n = numpy.random.lognormal(mean=5.1, sigma=0.3, size=3000), 4000.
>>> myfunc = lambda x: numpy.median(x)
>>> ci_low, ci_high = poppy.boots_ci(data, n, 95, myfunc)
>>> ci_low, numpy.median(data), ci_high
(163.96354196633686, 165.2393331896551, 166.60491435416566) iter. 1
... repeat
(162.50379144492726, 164.15218265100233, 165.42840588032755) iter. 2
```

For comparison

```
>>> data = numpy.random.lognormal(mean=5.1, sigma=0.3, size=90000)
>>> numpy.median(data)
163.83888237895815
```

Note that the true value of the desired quantity may lie outside the 95% confidence interval one time in 20 realizations. This occurred for the first iteration here.

For the lognormal distribution, the median is found exactly by taking the exponential of the "mean" parameter. Thus here, the theoretical median is 164.022 (6 s.f.) and this is well captured by the above bootstrap confidence interval.

poppy.**plot_two_ppro**(*pprodata, pproref, ratio=None, norm=False, title=None, xscale=None, figsize=None, dpi=80, ylim=[None, None], log=False, xticks=None, yticks=None*)

Overplots two PPro objects

Parameters **pprodata** : PPro

first point process to plot (in blue)

pproref : PPro

second process to plot (in red)

ratio : float

multiply L{pprodata} by this ratio before plotting, useful for comparing processes of different magnitude

norm : boolean

normalize everything to L{pproref}, i.e. the association number for L{pproref} will always plot as 1.

title : string

title to put on the plot

xscale : float

scale x-axis by this factor (e.g. 60.0 to convert seconds to minutes)

figsize : :

passed through to matplotlib.pyplot.figure

dpi : int

passed through to matplotlib.pyplot.figure

ylim : list

[minimum, maximum] values of y for the axis

log : bollean

True for a log plot

xticks : sequence or float

if provided, a list of tickmarks for the X axis

yticks : sequance or float

if provided, a list of tickmarks for the Y axis

poppy.**value_percentile**(*sequence*, *target*)

Find the percentile of a particular value in a sequence

Parameters **sequence** : sequence

a sequence of values, sorted in ascending order

target : same type as sequence

a target value

Returns **out** : float

the percentile of target in sequence

2.6 pyCDF - Python interface to CDF files

pyCDF: Python interface to NASA's Common Data Format library

This package provides a Python interface to the Common Data Format (CDF) library used for many NASA missions, available at http://cdf.gsfc.nasa.gov/. It is targeted at Python 2.6+ and should work without change on either Python 2 or Python 3.

The interface is intended to be quite 'pythonic' rather than reproducing the C interface. To open or close a CDF and access its variables, see the `pycdf.CDF` class documentation. Accessing data within the variables is via the `pycdf.Var` class. The `pycdf.lib` object (of type `pycdf.lib`) provides access to some routines that affect the functionality of the library in general.

The base CDF library must be properly installed in order to use this package. pycdf will search for the library in this order:

1. A directory named by the environment variable CDF_LIB (which should be set if using the definitions file provided with the CDF library).
2. A subdirectory C{lib} in a directory named by the environment variable CDF_BASE.
3. The standard system library search path.

If pycdf has trouble finding the library, try setting CDF_LIB before importing the module, e.g. if the library is in CDF/lib in the user's home directory:

```
import os
os.putenv("CDF_LIB", "~/CDF/lib")
import pycdf
```

The `pycdf.const` module contains constants useful for accessing the underlying library.

Authors: Jon Niehof Institution: Los Alamos National Laboratory Contact: jniehof@lanl.gov

2.6.1 Contents

2.6.2 Quickstart

Create a CDF

This quickstart guide should walk uses through the basics of CDF manipulation using pyCDF

To create a new CDF from some data for your own use or to send to a colleague. We will show the example then explain the part

```
>>> from spacepy import pycdf
>>> import numpy as np
>>> import datetime
>>> time = [datetime.datetime(2000, 10, 1, 1, val) for val in range(60)]
>>> data = np.random.random_sample(len(time))
>>> cdf = pycdf.CDF('MyCDF.cdf', '')
>>> cdf['Epoch'] = time
>>> cdf['data'] = data
```

```
>>> cdf.attrs['Author'] = 'John Doe'
>>> cdf.attrs['CreateDate'] = datetime.datetime.now()
>>> cdf['data'].attrs['units'] = 'MeV'
>>> cdf.close()
```

Import the pyCDF module. This can be done however you like.

```
>>> from spacepy import pycdf
```

Make a datetime data set, these are automatically converted into CDF_EPOCH types.

```
>>> import datetime
>>> # make a dataset every minute for a hour
>>> time = [datetime.datetime(2000, 10, 1, 1, val) for val in range(60)]
```

Warning: If you create a CDF in backwards compatibility mode (default) then `datetime.datetime` objects are degraded to CDF_EPOCH, not CDF_EPOCH16 type. This means millisecond resolution vs microsecond resolution.

Create some data of an arbitrary type

```
>>> data = np.random.random_sample(len(time))
```

Create a new empty CDF. The '' is the name of the CDF to use as a master. Note that the data is copied form the master to the n

```
>>> cdf = pycdf.CDF('MyCDF.cdf', '')
```

Note: You cannot create a new CDF with a name that already exists on disk. It will throw an `exceptions.NameError`

To put that data into the CDF just do it, CDF objects behave like Python dictionaries.

```
>>> # put time into the cdf as 'Epoch'
>>> cdf['Epoch'] = time
>>> # and the same with data (note that the smallest data type that fits the data is used by def
>>> cdf['data'] = data
```

Adding attributes is done the same way. CDF variables are also treated as dictionaries.

```
>>> # add some attributes to the variable data and to the global cdf
>>> cdf.attrs['Author'] = 'John Doe'
>>> cdf.attrs['CreateDate'] = datetime.datetime.now()
>>> cdf['data'].attrs['units'] = 'MeV'
```

It is best to close the CDF manually to be sure you know that it will be written.

```
>>> # and be sure to close the cdf to assure it is written
>>> cdf.close()
```

CDF files, like standard Python files, act as context managers

```
>>> with cdf.CDF('filename.cdf', '') as cdf_file:
...     #do brilliant things with cdf_file
>>> #cdf_file is automatically closed here
```

Read a CDF

Reading a CDF is done in much the same way, the CDF object behaves like a dictionary and only goes to disk when you request

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('MyCDF.cdf')
>>> print(cdf)
    Epoch: CDF_EPOCH [60]
    data: CDF_FLOAT [60]
>>> cdf['data'][4]
    0.8609974384307861
>>> data = cdf['data'][...] # don't forget the [...]
>>> cdf_dat = cdf.copy()
>>> cdf_dat.keys()
    ['Epoch', 'data']
>>> cdf.close()
```

Again import the pycdf module

```
>>> from spacepy import pycdf
```

Then open the CDF, this looks the same and creation, but without mention of a master CDF.

```
>>> cdf = pycdf.CDF('MyCDF.cdf')
```

The default `__str__()` and `__repr__()` behavior explains the contents, type, and size but not the data.

```
>>> print(cdf)
    Epoch: CDF_EPOCH [60]
    data: CDF_FLOAT [60]
```

To access the data one has to request specific elements of the variable that behaves like a list in this respect.

```
>>> cdf['data'][4]
    0.8609974384307861
>>> data = cdf['data'][...] # don't forget the [...]
```

One can also grab the entire contest of a CDF using the convenience routine `pycdf.CDF.copy()`

```
>>> cdf_dat = cdf.copy()
```

Since CDF objects behave like dictionaries they have a `keys()` method and iterations are over the names in `keys()`

```
>>> cdf_dat.keys()
    ['Epoch', 'data']
```

and with writing it is Best to close the CDF (or use context managers)

```
>>> cdf.close()
```

Modify a CDF

Again using the CDF created above a variable can be added or the contents of a variable changed.

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('MyCDF.cdf')
>>> cdf.readonly(False)
```

```
    False
>>> cdf['newVar'] = [1.0, 2.0]
>>> print(cdf)
    Epoch: CDF_EPOCH [60]
    data: CDF_FLOAT [60]
    newVar: CDF_FLOAT [2]
>>> cdf.close()
```

The parts of the example are straightforward. A particular open CDF must be made write-able

```
>>> cdf.readonly(False)
    False
```

Then new variables can be added

```
>>> cdf['newVar'] = [1.0, 2.0]
```

Or contents changed

```
>>> cdf['data'][0] = 8675309
```

And the new variable shows up

```
>>> print(cdf)
    Epoch: CDF_EPOCH [60]
    data: CDF_FLOAT [60]
    newVar: CDF_FLOAT [2]
```

As with writing be sure to close the CDF

```
>>> cdf.close()
```

Non record varying

Creating a variable that is non record varying is really useful in the conversion of text files to CDF where whole columns do not change.

To create a variable that is non-record varying one has to manually create the variable using `pycdf.CDF.new()`.

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('MyCDF2.cdf', '')
>>> # create a variable manually
>>> cdf.new('data2', [1], recVary=False)
    <Var:
    CDF_BYTE [1] NRV
    >
```

Slicing and indexing

This example is redundant to the above but worth a call out as it is a very common operation.

If one has the hourly data file created above and only wants to read in a portion of the data follow this recipe. Using `bisect` ca

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('MyCDF.cdf')
>>> start = datetime.datetime(2000, 10, 1, 1, 9)
```

```
>>> stop = datetime.datetime(2000, 10, 1, 1, 35)
>>> import bisect
>>> start_ind = bisect.bisect_left(cdf['Epoch'], start)
>>> stop_ind = bisect.bisect_left(cdf['Epoch'], stop)
>>> # then grab the data we want
>>> time = cdf['Epoch'][start_ind:stop_ind]
>>> data = cdf['data'][start_ind:stop_ind]
>>> cdf.close()
```

Class reference

pyCDF class reference

pyCDF: Python interface to NASA's Common Data Format library

This package provides a Python interface to the Common Data Format (CDF) library used for many NASA missions, available at http://cdf.gsfc.nasa.gov/. It is targeted at Python 2.6+ and should work without change on either Python 2 or Python 3.

The interface is intended to be quite 'pythonic' rather than reproducing the C interface. To open or close a CDF and access its variables, see the `pycdf.CDF` class documentation. Accessing data within the variables is via the `pycdf.Var` class. The `pycdf.lib` object (of type `pycdf.lib`) provides access to some routines that affect the functionality of the library in general.

The base CDF library must be properly installed in order to use this package. pycdf will search for the library in this order:

1. A directory named by the environment variable CDF_LIB (which should be set if using the definitions file provided with the CDF library).
2. A subdirectory C{lib} in a directory named by the environment variable CDF_BASE.
3. The standard system library search path.

If pycdf has trouble finding the library, try setting CDF_LIB before importing the module, e.g. if the library is in CDF/lib in the user's home directory:

```
import os
os.putenv("CDF_LIB", "~/CDF/lib")
import pycdf
```

The `pycdf.const` module contains constants useful for accessing the underlying library.

Authors: Jon Niehof Institution: Los Alamos National Laboratory Contact: jniehof@lanl.gov

class `pycdf.`**`CDF`**(*pathname*, *masterpath=None*)
Python object representing a CDF file.

Methods

checksum
clear
clone
close
col_major
compress
copy
get
items
iteritems
iterkeys
itervalues
keys
new
pop
popitem
readonly
save
setdefault
update
values
version

attrs
Descriptor to get attribute list for a pycdf.CDF or pycdf.Var.

checksum(*new_val=None*)
Set or check the checksum status of this CDF

Parameters **new_val** : Boolean (optional)

True to enable checksum, False to disable, or leave out to simply check.

Returns **out** : Boolean

True if the checksum is enabled or False if disabled

clone(*zVar*, *name=None*, *data=True*)
Clone a zVariable (from another CDF or this) into this CDF

Parameters **zVar** : pycdf.Var

variable to clone

name : str

Name of the new variable (default: name of the original)

data : Boolean (optional)

Copy data, or only type, dimensions, variance, attributes? (default: copy data as well)

close()
Closes the CDF file

Although called on object destruction pycdf.CDF.__del__(), to ensure all data are saved the user should explicitly call pycdf.CDF.close() or pycdf.CDF.save().

Raises **CDFError** : if CDF library reports an error

Warns **CDFWarning** : if CDF library reports a warning and interpreter is set to error on warnings.

col_major(*new_col=None*)
Finds the majority of this CDF file

Parameters **new_col** : Boolean

Specify True to change to column-major, False to change to row major, or do not specify to leave majority alone (check only)

Returns **out** : Boolean

True if column-major, false if row-major

compress(*comptype=None*, *param=None*)
Set or check the compression of this CDF

Sets compression on entire *file*, not per-variable.

Parameters **comptype** : ctypes.c_long

type of compression to change to, see CDF C reference manual section 4.10. Constants for this parameter are in `pycdf.const`. If not specified, will not change compression.

param : ctypes.c_long

Compression parameter, see CDF CRM 4.10 and `pycdf.const`. If not specified, will choose reasonable default (5 for gzip; other types have only one possible parameter.)

Returns **out** : tuple

(comptype, param) currently in effect

See Also:

`pycdf.Var.compress`

copy()
Make a copy of all data and attributes in this CDF

Returns **out** : `pycdf.CDFCopy`

dict of all data

new(*name*, *data=None*, *type=None*, *recVary=True*, *dimVarys=None*, *dims=None*, *n_elements=None*)
Create a new zVariable in this CDF

Parameters **name** : str

name of the new variable

data : CDF type of the variable, from `pycdf.const` (optional)

data to store in the new variable

type : ctypes.c_long

CDF type of the variable, from `pycdf.const`

Returns **out** : pycdf.Var

the newly-created zVariable

Other Parameters **recVary** : Boolean (optional)

record variance of the variable (default True)

dimVarys : list of Boolean (optional)

dimension variance of each dimension

dims : list of int (optional)

size of each dimension of this variable, default zero-dimensional

n_elements : int

number of elements, should be 1 except for CDF_CHAR, for which it's the length of the string.

Raises **ValueError** : if neither data nor sufficient typing information is provided.

readonly(*ro=None*)

Sets or check the readonly status of this CDF

If the CDF has been changed since opening, setting readonly mode will have no effect.

Parameters **ro** : Boolean

True to set the CDF readonly, False to set it read/write, otherwise to check

Returns **out** : Boolean

True if CDF is read-only, else False

Raises **pycdf.CDFError** : if bad mode is set

save()

Saves the CDF file but leaves it open.

If closing the CDF, `pycdf.CDF.close()` is sufficient, there is no need to call `pycdf.CDF.save()` before `pycdf.CDF.close()`.

Relies on an undocumented call of the CDF C library, which is also used in the Java interface.

Raises **CDFError** : if CDF library reports an error

Warns **CDFWarning** : if CDF library reports a warning and interpreter is set to error on warnings.

version()

Get version of library that created this CDF

Returns **out** : tuple

version of CDF library, in form (version, release, increment) :

class pycdf.**gAttrList**(*cdf_file*, *special_entry=None*)

Object representing *all* the gAttributes in a CDF.

Normally accessed as an attribute of an open `pycdf.CDF`

```
>>> global_attribs = cdffile.attrs
```

Appears as a dictionary: keys are attribute names; each value is an attribute represented by a `pycdf.gAttr` object. To access the global attribute TEXT

```
>>> text_attr = cdffile.attrs['TEXT']
```

Attributes

attr_name
global_scope

Methods

AttrType
clear
clone
copy
get
items
iteritems
iterkeys
itervalues
keys
new
pop
popitem
rename
setdefault
update
values

AttrType
alias of gAttr

class pycdf.**AttrList**(*cdf_file*, *special_entry=None*)
Object representing a list of attributes.

Only used in its subclasses, pycdf.gAttrList and pycdf.zAttrList

@ivar _cdf_file: CDF these attributes are in @type _cdf_file: pycdf.CDF @ivar special_entry: callable which returns a "special"

entry number, used to limit results for zAttrs to those which match the zVar

@type special_entry: callable @cvar AttrType: type of attribute in this list, L{zAttr} or L{gAttr} @type AttrType: type @cvar attr_name: name of attribute type, 'zAttribute' or 'gAttribute' @type attr_name: str @cvar global_scope: is this list scoped global (True) or variable (False) @type global_scope: bool

Methods

clear
clone
copy
get
items
iteritems
iterkeys
itervalues
keys
new
pop
popitem
rename
setdefault
update
values

clone(*master*, *name=None*, *new_name=None*)
Clones this attribute list, or one attribute in it, from another

Parameters **master** : pycdf.AttrList

the attribute list to copy from

name : str (optional)

name of attribute to clone (default: clone entire list)

new_name : str (optional)

name of the new attribute, default L{name}

copy()
Create a copy of this attribute list

Returns **out: dict** :

copy of the entries for all attributes in this list

new(*name*, *data=None*, *type=None*)
Create a new Attr in this AttrList

Parameters **name** : str

name of the new Attribute

data : CDF type of the first entry from `pycdf.const`. Only used if data are specified.

data to put into the first entry in the new Attribute

Raises **KeyError** : if the name already exists in this list

rename(*old_name*, *new_name*)
Rename an attribute in this list

Renaming a zAttribute renames it for *all* zVariables in this CDF!

Parameters **old_name** : str

the current name of the attribute

new_name : str

the new name of the attribute

class `pycdf.`**CDFCopy**(*cdf*)
A copy of all data and attributes in a `pycdf.CDF`

Data are `pycdf.VarCopy` objects, keyed by variable name (i.e. data are accessed much like from a `pycdf.CDF`).

@ivar attrs: attributes for the CDF @type attrs: dict

Methods

clear
copy
fromkeys
get
has_key
items
iteritems
iterkeys
itervalues
keys
pop
popitem
setdefault
update
values

class pycdf.**Library**

Abstraction of the base CDF C library and its state.

Not normally intended for end-user use. An instance of this class is created at package load time as the pycdf.lib variable, providing access to the underlying C library if necessary.

Calling the C library directly requires knowledge of the ctypes http://docs.python.org/library/ctypes.html package.

__init__() searches for and loads the C library; details on the search are documented there.

@ivar _del_middle_rec_bug: does this version of the library have a bug when deleting a record from the middle of a variable?

@type _del_middle_rec_bug: Boolean @ivar _library: ctypes connection to the library @type _library: ctypes.WinDLL or ctypes.CDLL @ivar version: version of the CDF library, in order version, release,

increment, subincrement

@type version: tuple

Methods

call
check_status
datetime_to_epoch
datetime_to_epoch16
epoch16_to_datetime
epoch_to_datetime
set_backward

call(**args*, ***kwargs*)

Call the CDF internal interface

Passes all parameters directly through to the CDFlib routine of the CDF library's C internal interface. Checks the return value with pycdf.Library.check_status().

Terminal **NULL_** is automatically added to args.

Parameters **args** : various, see ctypes

Passed directly to the CDF library interface. Useful constants are defined in the `pycdf.const` module of this package.

Returns **out** : int

CDF status from the library

Other Parameters **ignore** : sequence of CDF statuses

sequence of CDF statuses to ignore. If any of these is returned by CDF library, any related warnings or exceptions will I{not} be raised.

Raises **CDFError** : if CDF library reports an error

Warns **CDFWarning** : if CDF library reports a warning and interpreter is set to error on warnings.

check_status(*status*, *ignore=()*)

Raise exception or warning based on return status of CDF call

Parameters **status** : int

status returned by the C library, equivalent to C{CDFStatus}

ignore : sequence of ctypes.c_long

CDF statuses to ignore. If any of these is returned by CDF library, any related warnings or exceptions will I{not} be raised. (Default none).

Returns **out** : int

status (unchanged)

Raises **CDFError** : if status < CDF_WARN, indicating an error

Warns **CDFWarning** : if CDF_WARN <= status < CDF_OK, indicating a warning,

and interpreter is set to error on warnings.

datetime_to_epoch(*dt*)

Converts a Python datetime to a CDF Epoch value

Parameters **dt** : `datetime.datetime`

date and time to convert

Returns **out** : float

epoch corresponding to dt

datetime_to_epoch16(*dt*)

Converts a Python datetime to a CDF Epoch16 value

Parameters **dt** : `datetime.datetime`

date and time to convert

Returns **out** : list of float

epoch16 corresponding to dt

epoch16_to_datetime(*epoch*)

Converts a CDF epoch16 value to a datetime

Parameters **epoch** : list of two floats

epoch16 value from CDF

Returns **out** : `datetime.datetime`

date and time corresponding to epoch. Invalid values are set to usual epoch invalid value, i.e. last moment of year 9999.

Raises **EpochError** : if input invalid

epoch_to_datetime(*epoch*)
Converts a CDF epoch value to a datetime

Parameters **epoch** : float

epoch value from CDF

Returns **out** : datetime.datetime

date and time corresponding to epoch. Invalid values are set to usual epoch invalid value, i.e. last moment of year 9999.

set_backward(*backward=True*)
Set backward compatibility mode for new CDFs

Unless backward compatible mode is set, CDF files created by the version 3 library can not be read by V2.

Parameters **backward** : Boolean

Set backward compatible mode if True; clear it if False.

Raises **ValueError** : if backward=False and underlying CDF library is V2

exception pycdf.**CDFError**(*status*)
Raised for an error in the CDF library.

exception pycdf.**CDFException**(*status*)
Base class for errors or warnings in the CDF library.

Not normally used directly (see subclasses pycdf.CDFError and pycdf.CDFWarning).

Error messages provided by this class are looked up from the underlying C library.

@ivar status: CDF library status code @type status: ctypes.c_long @ivar string: CDF library error message for L{status} @type string: string

exception pycdf.**CDFWarning**(*status*)
Used for a warning in the CDF library.

exception pycdf.**EpochError**
Used for errors in epoch routines

2.7 radbelt - Functions supporting radiation belt diffusion codes

Functions supporting radiation belt diffusion codes

Authors: Josef Koller Institution: Los Alamos National Laboratory Contact: jkoller@lanl.gov

class radbelt.**RBmodel**(*grid='L'*, *NL=91*, *const_kp=False*)
1-D Radial diffusion class

This module contains a class for performing and visualizing 1-D radial diffusion simulations of the radiation belts.

Here is an example using the default settings of the model. Each instance must be initialized with (assuming import radbelt as rb):

```
>>> rmod = rb.RBmodel()
```

Next, set the start time, end time, and the size of the timestep:

```
>>> import datetime
>>> start = datetime.datetime(2003,10,14)
>>> end = datetime.datetime(2003,12,26)
>>> delta = datetime.timedelta(hours=1)
>>> rmod.setup_ticks(start, end, delta, dtype='UTC')
```

Now, run the model over the entire time range using the evolve method:

```
>>> rmod.evolve()
```

Finally, visualize the results:

```
>>> rmod.plot_summary()
```

Methods

Gaussian_source
add_Lmax
add_Lpp
add_PSD_obs
add_PSD_twin
add_omni
add_source
assimilate
evolve
get_DLL
plot
plot_obs
set_lgrid
setup_ticks

Gaussian_source()
Gaussian source term added to radiation belt model. The source term is given by the equation:

S = A exp{-(L-mu)^2/(2*sigma^2)}

with A=10^(-8), mu=5.0, and sigma=0.5 as default values

add_Lmax(*Lmax_model*)
add last closed drift shell Lmax

add_Lpp(*Lpp_model*)
add last closed drift shell Lmax

add_PSD_obs(*time=None*, *PSD=None*, *Lstar=None*, *satlist=None*)
add PSD observations

Parameters **time** : Ticktock datetime array

array of observation times

PSD : list of numpy arrays

PSD observational data for each time. Each entry in the list is a numpy array with the observations for the corresponding time

Lstar : list of numpy arrays

Lstar location of each PSD observations. Each entry in the list is a numpy array with the location of the observations for the corresponding time

satlist : list of satellite names

Returns **out** : radbeltmodel.PSDdata

List with information of the observational data, where each entry contains the observations and locations of observations for each time specified in the time array. Each list entry is a dictionary with the following information:

'Ticks' [Ticktock array] time of observations

'Lstar' [numpy array] location of observations

'PSD' [numpy array] PSD observation values

'sat' [list of strings] satellite names

'MU' [scalar value] Mu value for the observations

'K' [scalar value] K value for the observations

add_PSD_twin(*dt=0*, *Lt=1*)

add observations from PSD database using the ticks list the arguments are the following:

dt = observation time delta in seconds Lt = observation space delta

add_omni(*keylist=None*)

add omni data to instance according to the tickrange in ticks

add_source(*source=True*, *A=1e-08*, *mu=5.0*, *sigma=0.5*)

add source parameters A, mu, and sigma for the Gaussian source function

assimilate(*method='EnKF'*, *inflation=0*)

Assimilates data for the radiation belt model using the Ensemble Kalman Filter. The algorithm used is the SVD method presented by Evensen in 2003 (Evensen, G., Ocean dynamics, 53, pp.343–367, 2003). To compensate for model errors, three inflation algorithms are implemented. The inflation methodology is specified by the 'inflation' argument, and the options are the following:

inflation == 0: Add model error (perturbation for the ensemble) around model state values only where observations are available (DEFAULT).

inflation == 1: Add model error (perturbation for the ensemble) around observation values only where observations are available.

inflation == 2: Inflate around ensemble average for EnKF.

Prior to assimilation, a set of data values has to be speficied by setting the start and end dates, and time step, using the setup_ticks funcion of the radiation belt model:

>> import spacepy >> import datetime >> from spacepy import radbelt

>> start = datetime.datetime(2002,10,23) >> end = datetime.datetime(2002,11,4) >> delta = datetime.timedelta(hours=0.5) >> rmod.setup_ticks(start, end, delta, dtype='UTC')

Once the dates and time step are specified, the data is added using the add_PSD function:

>> rmod.add_PSD()

The observations are averaged over the time windows, whose interval is give by the time step.

Once the dates and data are set, the assimiation is performed using the 'assimilate' function:

>> rmod.assimilate(inflation=1)

This function will add the PSDa values, which are the analysis state of the radiation belt using the observations within the dates. To plot the analysis simply use the plot funtion:

>> rmod.plot(values=rmod.PSDa,clims=[-10,-6],Lmax=False,Kp=False,Dst=False)

evolve()
calculate the diffusion in L at constant mu,K coordinates

get_DLL(*Lgrid*, *params*, *DLL_model='BA2000'*)
Calculate DLL as a simple power law function (alpha*L**Bbta) using alpha/beta values from popular models found in the literature and chosen with the kwarg "DLL_model".

The calculated DLL is returned, as is the alpha and beta values used in the calculationp.

The output DLL is in units of units/day.

plot(*Lmax=True, Lpp=False, Kp=True, Dst=True, clims=[0, 10], title=None, values=None*)
Create a summary plot of the RadBelt object distribution function. For reference, the last closed drift shell, Dst, and Kp are all included. These can be disabled individually using the corresponding Boolean kwargs.

The clims kwarg can be used to manually set the color bar range. To use, set it equal to a two-element list containing minimum and maximum Log_10 value to plot. Default action is to use [0,10] as the log_10 of the color range. This is good enough for most applications.

The title of the top most plot defaults to 'Summary Plot' but can be customized using the title kwarg.

The figure object and all three axis objects (PSD axis, Dst axis, and Kp axis) are all returned to allow the user to further customize the plots as necessary. If any of the plots are excluded, None is returned in their stead.

Examples

```
>>> rb.plot(Lmax=False, Kp=False, clims=[2,10], title='Good work!')
```

This command would create the summary plot with a color bar range of 100 to 10^10. The Lmax line and Kp values would be excluded. The title of the topmost plot (phase space density) would be set to 'Good work!'.

plot_obs(*Lmax=True, Lpp=False, Kp=True, Dst=True, clims=[0, 10], title=None, values=None*)
Create a summary plot of the observations. For reference, the last closed drift shell, Dst, and Kp are all included. These can be disabled individually using the corresponding boolean kwargs.

The clims kwarg can be used to manually set the color bar range. To use, set it equal to a two-element list containing minimum and maximum Log_10 value to plot. Default action is to use [0,10] as the log_10 of the color range. This is good enough for most applications.

The title of the top most plot defaults to 'Summary Plot' but can be customized using the title kwarg.

The figure object and all three axis objects (PSD axis, Dst axis, and Kp axis) are all returned to allow the user to further customize the plots as necessary. If any of the plots are excluded, None is returned in their stead.

Examples

```
>>> rb.plot_obs(Lmax=False, Kp=False, clims=[2,10], title='Observations Plot')
```

This command would create the summary plot with a color bar range of 100 to 10^10. The Lmax line and Kp values would be excluded. The title of the topmost plot (phase space density) would be set to 'Good work!'.

set_lgrid(*NL=91*)
Using NL grid points, create grid in L. Default number of points is 91 (dL=0.1).

setup_ticks(*start*, *end*, *delta*, *dtype='ISO'*)
Add time information to the simulation by specifying a start and end time, timestep, and time type (optional).

Examples

```
>>> start = datetime.datetime(2003,10,14)
>>> end = datetime.datetime(2003,12,26)
>>> delta = datetime.timedelta(hours=1)
>>> rmod.setup_ticks(start, end, delta, dtype='UTC')
```

radbelt.**diff_LL**(*r*, *grid*, *f*, *Tdelta*, *Telapsed*, *params=None*)
calculate the diffusion in L at constant mu,K coordinates time units

radbelt.**get_local_accel**(*Lgrid*, *params*, *SRC_model='JK1'*)
calculate the diffusion coefficient D_LL

radbelt.**get_modelop_L**(*f*, *L*, *Dm_old*, *Dm_new*, *Dp_old*, *Dp_new*, *Tdelta*, *NL*)
Advance the distribution function, f, discretized into the Lgrid, L, forward in time by a timestep, Tdelta. The off-grid current and next diffusion coefficients, D[m,p]_[old,new] will be used. The number of grid points is set by NL.

This function performs the same calculation as the C-based code, spacepy.lib.solve_cnp. This code is very slow and should only be used when the C code fails to compile.

2.8 SeaPy - Superposed Epoch in Python

SeaPy – Superposed Epoch in Python.

This module contains superposed epoch class types and a variety of functions for using on superposed epoch objects. Each instance must be initialized with (assuming import seapy as se):

```
>>> obj = se.Sea(data, times, epochs)
```

To perform a superposed epoch analysis

```
>>> obj.sea()
```

To plot

```
>>> obj.plot()
```

If multiple SeaPy objects exist, these can be combined into a single object

```
>>> objdict = seadict([obj1, obj2],['obj1name','obj2name'])
```

and then used to create a multipanel plot

```
>>> multisea(objdict)
```

For two-dimensional superposed epoch analyses, initialize an Sea2d() instance

```
>>> obj = se.Sea2d(data, times, epochs, y=[4., 12.])
```

All object methods are the same as for the 1D object. Also, the multisea() function should accept both 1D and 2D objects, even mixed together. Currently, the plot() method is recommended for 2D SEA.

–++– By Steve Morley –++–

smorley@lanl.gov Los Alamos National Laboratory

class seapy.**Sea**(*data*, *times*, *epochs*, *window=3.0*, *delta=1.0*, *verbose=True*)
SeaPy Superposed epoch analysis object

Initialize object with data, times, epochs, window (half-width) and delta (optional). 'times' and epochs should be in some useful format Includes method to perform superposed epoch analysis of input data series

Output can be nicely plotted with plot method, or for multiple objects use the seamulti function

Methods

plot
restoreepochs
sea

plot(*xquan='Time Since Epoch'*, *yquan=''*, *xunits=''*, *yunits=''*, *epochline=False*, *usrlimy=*[], *show=True*, *figsize=None*, *dpi=None*, *transparent=True*)
Method called to create basic plot of superposed epoch analysis.

Parameters **Uses object attributes created by the obj.sea() method.** :

Other Parameters **xquan** : str

(default = 'Time since epoch') - x-axis label.

yquan : str

default None - yaxus label

xunits : str

(default = None) - x-axis units.

yunits : str

(default = None) - y-axis units.

epochline : boolean

(default = False) - put vertical line at zero epoch.

usrlimy : list

(default = []) - override automatic y-limits on plot.

transparent : boolean

(default True): make patch for low/high bounds transparent

If both quan and units are supplied, axis label will read :

'Quantity Entered By User [Units]' :

restoreepochs()
Replaces epoch times stored in obj.badepochs in the epochs attribute

sea(***kwargs*)
Method called to perform superposed epoch analysis on data in object.

Parameters Uses object attributes obj.data, obj.times, obj.epochs, obj.delta, obj.window, :

all of which must be available on instantiation. :

Other Parameters storedata : boolean

saves matrix of epoch windows as obj.datacube (default = False)

quartiles : list

calculates the quartiles as the upper and lower bounds (and is default);

ci : float

will find the bootstrapped confidence intervals (and requires ci_quan to be set);

mad : flaot

will use +/- the median absolute deviation for the bounds;

ci_quan : string

can be set to 'median' or 'mean'

A basic plot can be raised with the obj.plot() method :

class seapy.**Sea2d**(*data*, *times*, *epochs*, *window=3.0*, *delta=1.0*, *verbose=False*, *y=*[ ])
SeaPy 2D Superposed epoch analysis object

Initialize object with data, times, epochs, window (half-width), delta (optional), and y (two-element vector with max and min of y;optional) 'times' and epochs should be in some useful format Includes method to perform superposed epoch analysis of input data series

Output can be nicely plotted with plot method, or for multiple objects use the seamulti function

Methods

plot
sea

plot(*xquan='Time Since Epoch'*, *yquan=''*, *xunits=''*, *yunits=''*, *zunits=''*, *epochline=False*, *usrlimy=*[ ], *show=True*, *zlog=True*, *figsize=None*, *dpi=300*)
Method called to create basic plot of 2D superposed epoch analysis.

Parameters Uses object attributes created by the obj.sea() method. :

Other Parameters - x(y)quan (default = 'Time since epoch' (None)) - x(y)-axis label. :

- x(y/z)units (default = None (None)) - x(y/z)-axis units.
- epochline (default = False) - put vertical line at zero epoch.
- usrlimy (default = []) - override automatic y-limits on plot.

- show (default = True) - shows plot; set to false to output plot object to variable
- figsize - tuple of (width, height) in inches
- dpi (default=300) - figure resolution in dots per inch

If both ?quan and ?units are supplied, axis label will read :

'Quantity Entered By User [Units]' :

sea (*storedata=False*, *quartiles=True*, *ci=False*, *mad=False*, *ci_quan='median'*, *nmask=1*)
Method called to perform 2D superposed epoch analysis on data in object.

Parameters Uses object attributes obj.data, obj.times, obj.epochs, obj.delta, obj.window, :

all of which must be available on instantiation. :

Other Parameters - storedata (default = False) - saves matrix of epoch windows as obj.datacube :

- quartiles calculates the inter-quartile range to show the spread (and is default);
- ci will find the bootstrapped confidence interval (and requires ci_quan to be set);
- mad will use the median absolute deviation for the spread;
- ci_quan can be set to 'median' or 'mean'

A basic plot can be raised with the obj.plot() method :

class seapy.**SeaBase** (*data*, *times*, *epochs*, ***kwargs*)
SeaPy Superposed epoch analysis base class

Do not use directly – subclass it!

seapy.**multisea** (*dictobj*, *n_cols=1*, *epochline=False*, *usrlimx=[]*, *usrlimy=[]*, *xunits=''*, *show=True*, *zunits=''*, *zlog=True*, *figsize=None*, *dpi=300*)
Function to create multipanel plot of superposed epoch analyses.

Parameters Dictionary of Sea objects (from superposedepoch.seadict()). :

Returns Plot of input object median and bounds (ci, mad, quartiles - see sea()). :

If keyword 'show' is False, output is a plot object. :

Other Parameters - epochline (default = False) - put vertical line at zero epoch. :

- usrlimy (default = []) - override automatic y-limits on plot (same for all plots).
- show (default = True) - shows plot; set to false to output plot object to variable
- x/zunits - Units for labeling x and z axes, if required
- figsize - tuple of (width, height) in inches
- dpi (default=300) - figure resolution in dots per inch
- n_cols - Number of columns: not yet implemented.

seapy.**readepochs** (*fname*, *iso=False*, *isofmt='%Y-%m-%dT%H:%M:%S'*)
Read epochs from text file assuming YYYY MM DD hh mm ss format

Parameters Filename (include path) :

Returns epochs (type=list) :

Other Parameters iso (default = False), read in ISO date format :

isofmt (default is YYYY-mm-ddTHH:MM:SS, code is %Y-%m-%dT%H:%M:%S) :

seapy.**sea_signif**(*obj1*, *obj2*, *test='KS'*, *show=True*, *xquan='Time Since Epoch'*, *yquan=''*, *xunits=''*, *yunits=''*, *epochline=True*, *usrlimy=[]*)

Test for similarity between distributions at each lag in two 1-D SEAs

Two seapy.Sea() instances for comparison

Other Parameters - show (default = True) :

- x(y)quan (default = 'Time since epoch' (None)) - x(y)-axis label.
- x(y)units (default = None (None)) - x(y)-axis units.
- epochline (default = True) - put vertical line at zero epoch.
- usrlimy (default = []) - override automatic y-limits on plot.

Examples

```
>>> obj1 = seapy.Sea(data1, times1, epochs1)
>>> obj2 = seapy.Sea(data2, times2, epochs2)
>>> obj1.sea(storedata=True)
>>> obj2.sea(storedata=True)
>>> seapy.sea_signif(obj1, obj2)
```

seapy.**seadict**(*objlist*, *namelist*)

Function to create dictionary of SeaPy.Sea objects.

Parameters - objlist: List of Sea objects. :

- namelist: List of variable labels for input objects.

Other Parameters namelist = List containing names for y-axes. :

2.9 time - Time conversion, manipulation and implementation of Ticktock class

Time conversion, manipulation and implementation of Ticktock class

2.9.1 Examples:

```
>>> import spacepy.time as spt
>>> import datetime as dt
```

Day of year calculations

```
>>> dts = spt.doy2date([2002]*4, range(186,190), dtobj=True)
>>> dts
[datetime.datetime(2002, 7, 5, 0, 0),
datetime.datetime(2002, 7, 6, 0, 0),
datetime.datetime(2002, 7, 7, 0, 0),
datetime.datetime(2002, 7, 8, 0, 0)]
```

```
>>> dts = spt.Ticktock(dts,'UTC')
>>> dts.DOY
array([ 186.,  187.,  188.,  189.])
```

Ticktock object creation

```
>>> isodates = ['2009-12-01T12:00:00', '2009-12-04T00:00:00', '2009-12-06T12:00:00']
>>> dts = spt.Ticktock(isodates, 'ISO')
```

OR

```
>>> dtdates = [dt.datetime(2009,12,1,12), dt.datetime(2009,12,4), dt.datetime(2009,12,6,12)]
>>> dts = spt.Ticktock(dtdates, 'UTC')
```

ISO time formatting

```
>>> dts = spt.tickrange('2009-12-01T12:00:00','2009-12-06T12:00:00',2.5)
```

OR

```
>>> dts = spt.tickrange(dt.datetime(2009,12,1,12),dt.datetime(2009,12,6,12),     dt.timedelta(days=2,
```

```
>>> dts
Ticktock( ['2009-12-01T12:00:00', '2009-12-04T00:00:00', '2009-12-06T12:00:00'] ), dtype=ISO
```

```
>>> dts.isoformat()
Current ISO output format is %Y-%m-%dT%H:%M:%S
Options are: [('seconds', '%Y-%m-%dT%H:%M:%S'), ('microseconds', '%Y-%m-%dT%H:%M:%S.%f')]
```

```
>>> dts.isoformat('microseconds')
>>> dts.ISO
['2009-12-01T12:00:00.000000',
 '2009-12-04T00:00:00.000000',
 '2009-12-06T12:00:00.000000']
```

Time manipulation

```
>>> tdelt  = spt.Tickdelta(days=1, hours=6)
>>> tdelt
Tickdelta( days=1.25 )
```

```
>>> new_dts = dts + tdelt
>>> new_dts.UTC
[datetime.datetime(2009, 12, 2, 18, 0),
 datetime.datetime(2009, 12, 5, 6, 0),
 datetime.datetime(2009, 12, 7, 18, 0)]
```

Other time formats

```
>>> dts.RDT  # Gregorian ordinal time
array([ 733742.5,  733745. ,  733747.5])


>>> dts.GPS # GPS time
array([  9.43704015e+08,   9.43920015e+08,   9.44136015e+08])


>>> dts.JD # Julian day
array([ 2455167. ,  2455169.5,  2455172. ])
```

And so on.

Authors:

Josef Koller, Brian Larsen, Steve Morley, Jon Niehof

jkoller@lanl.gov, Los Alamos National Laboratory

class spacepy.time.**Tickdelta**(***kwargs*)

Tickdelta class holding timedelta similar to datetime.timedelta This can be used to add/substract from Ticktock objects

Parameters **days** : float

number of days in for the delta

hours : float

number of hours for the delta

minutes : float

number of minutes for the delta

seconds : float

number of secondes for the delta

Returns **out** : Tickdelta

instance with self.days, self.secs, self.timedelta

See Also:

spacepy.time.Ticktock

Examples

```
>>> dt = Tickdelta(days=3.5, hours=12)
>>> dt
Tickdelta( days=4.0 )
```

class spacepy.time.**Ticktock**(*data*, *dtype*)

Ticktock class holding various time coordinate systems (TAI, UTC, ISO, JD, MJD, UNX, RDT, CDF, DOY, eDOY)

Possible data types: ISO: ISO standard format like '2002-02-25T12:20:30' UTC: datetime object with UTC time TAI: elapsed seconds since 1958/1/1 (includes leap seconds) UNX: elapsed seconds since 1970/1/1 (all days have 86400 secs sometimes unequal lenghts) JD: Julian days elapsed MJD: Modified Julian days RDT: Rata Die days elapsed since 1/1/1 CDF: CDF epoch: milliseconds since 1/1/0000

Parameters **data** : array_like (int, datetime, float, string)

time stamp

dtype : string {*CDF*, *ISO*, *UTC*, *TAI*, *UNX*, *JD*, *MJD*, *RDT*}

data type for data

Returns **out** : Ticktock

instance with self.data, self.dtype, self.UTC etc

See Also:

`a.getCDF`, `a.getISO`, `a.getUTC`

Examples

```
>>> x=Ticktock([2452331.0142361112, 2452332.0142361112], 'JD')
>>> x.ISO
['2002-02-25T12:20:30', '2002-02-26T12:20:30']
>>> x.DOY # Day of year
array([ 56.,  57.])
```

Methods

append
argsort
convert
getCDF
getDOY
getGPS
getISO
getJD
getMJD
getRDT
getTAI
getUNX
getUTC
geteDOY
getleapsecs
isoformat
now
sort
update_items

append(*other*)

Will be called when another Ticktock instance has to be appended to the current one

Parameters **other** : Ticktock

other (Ticktock instance)

argsort()
This will return the indices that would sort the Ticktock values

Returns **out** : list

indices that would sort the Ticktock values

convert(*dtype*)
convert a Ticktock instance into a new time coordinate system provided in dtype

Parameters **dtype** : string

data type for new system, possible values are {*CDF*, *ISO*, *UTC*, *TAI*, *UNX*, *JD*, *MJD*, *RDT*}

Returns **out** : Ticktock

Ticktock instance with new time coordinates

See Also:

`a.CDF`, `a.ISO`, `a.UTC`

Examples

```
>>> a = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:00:00'], 'ISO')
>>> s = a.convert('TAI')
>>> type(s)
<class 'time.Ticktock'>
>>> s
Ticktock( [1391342432 1391342432] ), dtype=TAI
```

getCDF()
a.getCDF() or a.CDF

Return CDF time which is milliseconds since 01-Jan-0000 00:00:00.000. "Year zero" is a convention chosen by NSSDC to measure epoch values. This date is more commonly referred to as 1 BC. Remember that 1 BC was a leap year. The CDF date/time calculations do not take into account the changes to the Gregorian calendar, and cannot be directly converted into Julian date/times.

Returns **out** : numpy array

days elapsed since Jan. 1st

See Also:

`getUTC`, `getUNX`, `getRDT`, `getJD`, `getMJD`, `getISO`, `getTAI`, `getDOY`, `geteDOY`

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.CDF
array([  6.31798704e+13])
```

getDOY()
a.DOY or a.getDOY()

extract DOY (days since January 1st of given year)

Returns **out** : numpy array

day of the year

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getISO, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.DOY
array([ 33])
```

getGPS()

a.GPS or a.getGPS()

return GPS epoch (0000 UT (midnight) on January 6, 1980)

Returns **out** : numpy array

elapsed secs since 6Jan1980 (excludes leap secs)

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getCDF, getISO, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.GPS
array([])
```

getISO()

a.ISO or a.getISO()

convert dtype data into ISO string

Returns **out** : list of strings

date in ISO format

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getCDF, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.ISO
['2002-02-02T12:00:00']
```

getJD()

a.JD or a.getJD()

convert dtype data into Julian Date (JD)

Returns **out** : numpy array

elapsed days since 12:00 January 1, 4713 BC

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getISO, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.JD
array([ 2452308.])
```

getMJD()

a.MJD or a.getMJD()

convert dtype data into MJD (modified Julian date)

Returns **out** : numpy array

elapsed days since November 17, 1858 (Julian date was 2,400 000)

See Also:

getUTC, getUNX, getRDT, getJD, getISO, getCDF, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.MJD
array([ 52307.5])
```

getRDT()

a.RDT or a.RDT()

convert dtype data into Rata Die (lat.) Time (days since 1/1/0001)

Returns **out** : numpy array

elapsed days since 1/1/1

See Also:

getUTC, getUNX, getISO, getJD, getMJD, getCDF, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.RDT
array([ 730883.5])
```

getTAI()

a.TAI or a.getTAI()

return TAI (International Atomic Time)

Returns **out** : numpy array

elapsed secs since 1958/1/1 (includes leap secs, i.e. all secs have equal lengths)

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getCDF, getISO, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.TAI
array([1391342432])
```

getUNX()

a.UNX or a.getUNX()

convert dtype data into Unix Time (Posix Time) seconds since 1970-Jan-1 (not counting leap seconds)

Returns **out** : numpy array

elapsed secs since 1970/1/1 (not counting leap secs)

See Also:

getUTC, getISO, getRDT, getJD, getMJD, getCDF, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.UNX
array([  1.01265120e+09])
```

getUTC()

a.UTC or a.getUTC()

convert dtype data into UTC object a la datetime()

Returns **out** : list of datetime objects

datetime object in UTC time

See Also:

getISO, getUNX, getRDT, getJD, getMJD, getCDF, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.UTC
[datetime.datetime(2002, 2, 2, 12, 0)]
```

geteDOY()

a.eDOY or a.geteDOY()

extract eDOY (elapsed days since midnight January 1st of given year)

Returns **out** : numpy array

days elapsed since midnight bbedJan. 1st

See Also:

getUTC, getUNX, getRDT, getJD, getMJD, getISO, getTAI, getDOY, geteDOY

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.eDOY
array([ 32.5])
```

getleapsecs()
a.leaps or a.getleapsecs()

retrieve leapseconds from lookup table, used in getTAI

Returns **out** : numpy array

leap seconds

See Also:

getTAI

Examples

```
>>> a = Ticktock('2002-02-02T12:00:00', 'ISO')
>>> a.leaps
array([32])
```

isoformat(*b*, *attrib*)
This changes the self.__isofmt attribute by and subsequently this function will update the ISO attribute.

Parameters **fmt** : string, optional

classmethod now()
Creates a Ticktock object with the current time, equivalent to datetime.now()

Returns **out** : ticktock

Ticktock object with the current time, equivalent to datetime.now()

See Also:

datetime.datetime.now

sort()
This will sort the Ticktock values in place

update_items(*b*, *attrib*)
After changing the self.data attribute by either __setitem__ or __add__ etc this function will update all other attributes. This function is called automatically in __add__ and __setitem__

Parameters **cls** : Ticktock

attrib : string

attribute to update

See Also:

spacepy.Ticktock.__setitem__, spacepy.Ticktock.__add__, spacepy.Ticktock.__sub__

spacepy.time.**doy2date**(*year*, *doy*, *dtobj=False*, *flAns=False*)
convert integer day-of-year doy into a month and day after http://pleac.sourceforge.net/pleac_python/datesandtimes.html

Parameters **year** : int or array of int

year

doy [int or array of int] day of year

Returns **month** : int or array of int

month as integer number

day [int or array of int] as integer number

See Also:

`getDOY`

Examples

```
>>> month, day = doy2date(2002, 186)
>>> dts = doy2date([2002]*4, range(186,190), dtobj=True)
```

spacepy.time.**sec2hms**(*sec*, *rounding=True*, *days=False*, *dtobj=False*)

Convert seconds of day to hours, minutes, seconds

Parameters **sec** : float

Seconds of day

Returns **out** : [hours, minutes, seconds] or datetime.timedelta

Other Parameters **rounding** : boolean

set for integer seconds

days [boolean] set to wrap around day (i.e. modulo 86400)

dtobj [boolean] set to return a timedelta object

spacepy.time.**test**()

test all time conversions

Returns **out** : int

number of failures

Examples

```
>>> test()
testing ticks: PASSED TEST 1
testing ticks: PASSED TEST 2
0
```

spacepy.time.**tickrange**(*start*, *end*, *deltadays*, *dtype='ISO'*)

return a Ticktock range given the start, end, and delta

Parameters **start** : string or number

start time

end [string or number] end time (inclusive)

deltadays [float or timedelta] step in units of days (float); or datetime timedelta object

dtype [string (optional)] data type for start, end; e.g. ISO, UTC, RTD, etc. see Ticktock for all options

Returns **out** : Ticktock instance

ticks

See Also:

`Ticktock`

Examples

```
>>> ticks = st.tickrange('2002-02-01T00:00:00', '2002-02-10T00:00:00', deltadays = 1)
>>> ticks
Ticktock( ['2002-02-01T00:00:00', '2002-02-02T00:00:00', '2002-02-03T00:00:00',
'2002-02-04T00:00:00'] ), dtype=ISO
```

2.10 toolbox - Toolbox of various functions and generic utilities

Toolbox of various functions and generic utilities.

Authors: Steve Morley, Jon Niehof, Brian Larsen, Josef Koller, Dan Welling Institution: Los Alamos National Laboratory Contact: smorley@lanl.gov, jniehof@lanl.gov, balarsen@lanl.gov, jkoller@lanl.gov, dwelling@lanl.gov Los Alamos National Laboratory

`toolbox.`**`applySmartTimeTicks`**(*ax*, *time*, *dolimit=True*)

Given an axis 'ax' and a list/array of datetime objects, 'time', use the smartTimeTicks function to build smart time ticks and then immediately apply them to the given axis. The first and last elements of the time list will be used as bounds for the x-axis range.

The range of the 'time' input value will be used to set the limits of the x-axis as well. Set kwarg 'dolimit' to False to override this behavior.

Parameters **ax** : matplotlib.pyplot.Axes

A matplotlib Axis object.

time : list

list of datetime objects

dolimit : boolean (optional)

The range of the 'time' input value will be used to set the limits of the x-axis as well. Setting this overrides this behavior.

`toolbox.`**`arraybin`**(*array*, *bins*)

Split a sequence into subsequences based on value.

Given a sequence of values and a sequence of values representing the division between bins, return the indices grouped by bin.

Parameters **array** : array_like

the input sequence to slice, must be sorted in ascending order

bins : array_like

dividing lines between bins. Number of bins is len(bins)+1, value that exactly equal a dividing value are assigned to the higher bin

Returns **out** : list

indices for each bin (list of lists)

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.arraybin(range(10), [4.2])
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
```

toolbox.**assemble**(*fln_pattern*, *outfln*, *sortkey='ticks'*, *verbose=True*)

assembles all pickled files matching fln_pattern into single file and save as outfln. Pattern may contain simple shell-style wildcards *? a la fnmatch file will be assembled along time axis given by Ticktock (key: 'ticks') in dictionary If sortkey = None, then nothing will be sorted

Parameters **fln_pattern** : string

pattern to match filenames

outfln : string

filename to save combined files to

Returns **out** : dict

dictionary with combined values

Examples

```
>>> import spacepy.toolbox as tb
>>> a, b, c = {'ticks':[1,2,3]}, {'ticks':[4,5,6]}, {'ticks':[7,8,9]}
>>> tb.savepickle('input_files_2001.pkl', a)
>>> tb.savepickle('input_files_2002.pkl', b)
>>> tb.savepickle('input_files_2004.pkl', c)
>>> a = tb.assemble('input_files_*.pkl', 'combined_input.pkl')
('adding ', 'input_files_2001.pkl')
('adding ', 'input_files_2002.pkl')
('adding ', 'input_files_2004.pkl')
('\n writing: ', 'combined_input.pkl')
>>> print(a)
{'ticks': array([1, 2, 3, 4, 5, 6, 7, 8, 9])}
```

toolbox.**binHisto**(*data*, *verbose=False*)

Calculates bin width and number of bins for histogram using Freedman-Diaconis rule if rule fails, defaults to square-root method

Parameters **data** : array_like

list/array of data values

verbose : boolean (optional)

print out some more information

Returns **out** : tuple

calculated width of bins using F-D rule, number of bins (nearest integer) to use for histogram

Examples

```
>>> import numpy, spacepy
>>> import matplotlib.pyplot as plt
>>> numpy.random.seed(8675301)
>>> data = numpy.random.randn(1000)
>>> binw, nbins = spacepy.toolbox.binHisto(data)
>>> print(nbins)
19.0
>>> p = plt.hist(data, bins=nbins, histtype='step', normed=True)
```

toolbox.**bin_center_to_edges**(*centers*)

Convert a list of bin centers to their edges

Given a list of center values for a set of bins, finds the start and end value for each bin. (start of bin n+1 is assumed to be end of bin n). Useful for e.g. matplotlib.pyplot.pcolor.

Edge between bins n and n+1 is arithmetic mean of the center of n and n+1; edge below bin 0 and above last bin are established to make these bins symmetric about their center value.

Parameters **centers** : list

list of center values for bins

Returns **out** : list

list of edges for bins

****note:** returned list will be one element longer than centers** :

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.bin_center_to_edges([1,2,3])
[0.5, 1.5, 2.5, 3.5]
```

toolbox.**dictree**(*in_dict*, *verbose=False*, *spaces=None*, *levels=True*, *attrs=False*, ***kwargs*)

pretty print a dictionary tree

Parameters **in_dict** : dict

a complex dictionary (with substructures)

verbose : boolean (optional)

print more info

spaces : string (optional)

string will added for every line

levels : integer (optional)

number of levels to recurse through (True means all)

attrs : boolean (optional)

display information for attributes

Examples

```
>>> import spacepy.toolbox as tb
>>> d = {'grade':{'level1':[4,5,6], 'level2':[2,3,4]}, 'name':['Mary', 'John', 'Chris']}
>>> tb.dictree(d)
+
|____grade
     |____level1
     |____level2
|____name
```

More complicated example using a datamodel:

```
>>> from spacepy import datamodel
>>> counts = datamodel.dmarray([2,4,6], attrs={'units': 'cts/s'})
>>> data = {'counts': counts, 'PI': 'Dr Zog'}
>>> tb.dictree(data)
+
|____PI
|____counts
>>> tb.dictree(data, attrs=True, verbose=True)
+
|____PI (str [6])
|____counts (spacepy.datamodel.dmarray (3,))
    :|____units (str [5])
```

Attributes of, e.g., a CDF or a datamodel type object (obj.attrs) are denoted by a colon.

toolbox.**dist_to_list**(*func*, *length*, *min=None*, *max=None*)

Convert a probability distribution function to a list of values

This is a deterministic way to produce a known-length list of values matching a certain probability distribution. It is likely to be a closer match to the distribution function than a random sampling from the distribution.

Parameters **func** : callable

function to call for each possible value, returning probability density at that value (does not need to be normalized.)

length : int

number of elements to return

min : float

minimum value to possibly include

max : float

maximum value to possibly include

Examples

```
>>> import matplotlib
>>> import numpy
>>> import spacepy.toolbox as tb
>>> gauss = lambda x: math.exp(-(x ** 2) / (2 * 5 ** 2)) /                    (5 * math.s
```

```
>>> vals = tb.dist_to_list(gauss, 1000, -numpy.inf, numpy.inf)
>>> print vals[0]
-16.45263...
>>> p1 = matplotlib.pyplot.hist(vals, bins=[i - 10 for i in range(21)],
>>> matplotlib.pyplot.hold(True)
>>> x = [i / 100.0 - 10.0 for i in range(2001)]
>>> p2 = matplotlib.pyplot.plot(x, [gauss(i) * 1000 for i in x], 'red')
>>> matplotlib.pyplot.draw()
```

toolbox.**feq**(*x*, *y*, *precision=4.9999999999999998e-07*)

compare two floating point values if they are equal after: http://www.lahey.com/float.htm

Parameters **x** : float

a number

y : float or array of flaots

otehr numbers to compare

precision : float (optional)

precision for equal (default 0.0000005)

Returns **out** : bool

True (equal) or False (not equal)

Examples

```
>>> import spacepy.toolbox as tb
>>> x = 1 + 1e-4
>>> y = 1 + 2e-4
>>> tb.feq(x, y)
False
>>> tb.feq(x, y, 1e-3)
True
```

toolbox.**geomspace**(*start*, *ratio=None*, *stop=False*, *num=50*)

Returns geometrically spaced numbers.

Parameters **start** : float

The starting value of the sequence. :

ratio : float (optional)

The ratio between subsequent points :

stop: float (optional) :

End value, if this is selected 'num' is overridden :

num : int (optional)

Number of samples to generate. Default is 50. :

Returns

======= :

seq : array

Examples

To get a geometric progression between 0.01 and 3 in 10 steps

```
>>> import spacepy.toolbox as tb
>>> tb.geomspace(0.01, stop=3, num=10)
[0.01,
 0.018846716378431192,
 0.035519871824902655,
 0.066943295008216955,
 0.12616612944575134,
 0.23778172582285118,
 0.44814047465571644,
 0.84459764235318191,
 1.5917892219322083,
 2.9999999999999996]
```

To get a geometric progression with a specified ratio, say 10

```
>>> import spacepy.toolbox as tb
>>> tb.geomspace(0.01, ratio=10, num=5)
[0.01, 0.10000000000000001, 1.0, 10.0, 100.0]
```

toolbox.**human_sort**(*l*)

Sort the given list in the way that humans expect. http://www.codinghorror.com/blog/2007/12/sorting-for-humans-natural-sort-order.html

Parameters **l** : list

list of objects to human sort

Returns **out** : list

sorted list

Examples

```
>>> import spacepy.toolbox as tb
>>> dat = ['r1.txt', 'r10.txt', 'r2.txt']
>>> dat.sort()
>>> print dat
['r1.txt', 'r10.txt', 'r2.txt']
>>> tb.human_sort(dat)
['r1.txt', 'r2.txt', 'r10.txt']
```

toolbox.**hypot**(**vals*)

Compute sqrt(vals[0] **2 + vals[1] **2 ...), ie. n-dimensional hypoteneuse

Parameters **vals** : float (arbitary number)

arbitary number of float values

Returns **out** : float

the Euclidian distance of the points ot the origin

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.hypot(3,4)
5.0
>>> a = [3, 4]
>>> tb.hypot(*a)
5.0
>>> tb.hypot(*range(10))
16.88194...
```

toolbox.**interpol**(*newx*, *x*, *y*, *wrap=None*, ***kwargs*)

1-D linear interpolation with interpolation of hours/longitude

Parameters **newx** : array_like

x values where we want the interpolated values

x : array_like

x values of the origional data

y : array_like

7 values of the origional data

wrap : string, optional

for continous x data that wraps in y at 'hours' (24), 'longitude' (360), or arbitary value (int, float)

kwargs : dict

additional keywords, currently accepts baddata that sets baddata for masked arrays

Returns **out** : numpy.masked_array

interpolated data values for new abscissa values

Examples

For a simple interpolation

```
>>> import spacepy.toolbox as tb
>>> import numpy
>>> x = numpy.arange(10)
>>> y = numpy.arange(10)
>>> tb.interpol(numpy.arange(5)+0.5, x, y)
array([ 0.5,  1.5,  2.5,  3.5,  4.5])
```

To use the wrap functionality, without the wrap keyword you get the wrong answer

```
>>> y = range(24)*2
>>> x = range(len(y))
>>> tb.interpol([1.5, 10.5, 23.5], x, y, wrap='hour').compressed() # compress removed the masked
array([  1.5,  10.5,  23.5])
>>> tb.interpol([1.5, 10.5, 23.5], x, y)
array([  1.5,  10.5,  11.5])
```

toolbox.**intsolve**(*func*, *value*, *start=None*, *stop=None*, *maxit=1000*)

Find the function input such that definite integral is desired value.

Given a function, integrate from an (optional) start point until the integral reached a desired value, and return the end point of the integration.

Parameters **func** : callable

function to integrate, must take single parameter

value : float

desired final value of the integral

start : float (optional)

value at which to start integration, default -Infinity

stop : float (optional)

value at which to stop integration, default +Infinity

maxit : integer

maximum number of iterations

Returns **out** : float

x such that the integral of L{func} from L{start} to x is L{value}

****Note:** Assumes func is everywhere positive, otherwise solution may** :

be multi-valued.

`toolbox.`**`leap_year`**(*year*, *numdays=False*)

return an array of boolean leap year, a lot faster than the mod method that is normally seen

Parameters **year** : array_like

array of years

numdays : boolean (optional)

optionally return the number of days in the year

Returns **out** : numpy array

an array of boolean leap year, or array of number of days

Examples

```
>>> import numpy
>>> import spacepy.toolbox as tb
>>> leap_year(numpy.arange(15)+1998)
array([False, False,  True, False, False, False,  True, False, False,
... False,  True, False, False, False,  True], dtype=bool)
```

`toolbox.`**`leapyear`**(*year*, *numdays=False*)

return an array of boolean leap year, a lot faster than the mod method that is normally seen

Parameters **year** : array_like

array of years

numdays : boolean (optional)

optionally return the number of days in the year

Returns **out** : numpy array

an array of boolean leap year, or array of number of days

Examples

```
>>> import numpy
>>> import spacepy.toolbox as tb
>>> leap_year(numpy.arange(15)+1998)
array([False, False,  True, False, False, False,  True, False, False,
... False,  True, False, False, False,  True], dtype=bool)
```

toolbox.**linspace**(*min*, *max*, *num=50*, *endpoint=True*, *retstep=False*)

Returns linearly spaced numbers. Same as numpy.linspace except allows for support of datetime objects

Parameters **start** : float

The starting value of the sequence.

stop : float

The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : int (optional)

Number of samples to generate. Default is 50.

endpoint : bool, optional

If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

retstep : bool (optional)

If True, return (*samples*, *step*), where *step* is the spacing between samples.

Returns **samples** : array

There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

step : float (only if *retstep* is True)

Size of spacing between samples.

toolbox.**listUniq**(*inVal*)

Given an input iterable (list, deque) return a list of the unique elements. Maintains order (keeps the first of non-unique elements

Parameters **inVal** : iterable

Input iterable

Returns **out** : list

list of unique elements from iterable

Examples

```
>>> import spacepy.toolbox as tb
>>> a = [1,1,2,3,3,4,5,5]
>>> tb.listUniq(a)
[1, 2, 3, 4, 5]
```

toolbox.**loadpickle**(*fln*)

load a pickle and return content as dictionary

Parameters **fln** : string

filename

Returns **out** : dict

dictionary with content from file

See Also:

savepickle

Examples

note: If fln is not found, but the same filename with '.gz' is found, will attempt to open the .gz as a gzipped file.

```
>>> d = loadpickle('test.pbin')
```

toolbox.**logspace**(*min*, *max*, *num*, ***kwargs*)

Returns log-spaced bins. Same as numpy.logspace except the min and max are the min and max not log10(min) and log10(max)

Parameters **min** : float

minimum value

max : float

maximum value

num : integer

number of log spaced bins

Returns **out** : array

log-spaced bins from min to max in a numpy array

Other Parameters **kwargs** : dict

additional keywords passed into matplotlib.dates.num2date

Notes

This function works on both numbers and datetime objects

Examples

```
>>> import sapcepy.toolbox as tb
>>> tb.logspace(1, 100, 5)
array([   1.        ,    3.16227766,   10.        ,   31.6227766 ,  100.        ])
```

toolbox.**makePoly**(*x*, *y1*, *y2*, *face='blue'*, *alpha=0.5*)

Make filled polygon for plotting

Parameters **x** : list

List of x start and stop values

y1 : list

List of y lower bounds for fill

y2 : list

List of y upper bounds for fill

face : string (optional)

color of the fill (default blue)

alpha : float (optional)

alpha of the fill (default 0.5)

.. deprecated:: vesion 0.1 :

Equivalent functionality to built-in matplotlib function fill_between :

Examples

```
>>> import spacepy.toolbox as tb
>>> poly0c = tb.makePoly(x, ci_low, ci_high, face='red', alpha=0.8)
>>> ax0.add_patch(poly0qc)
```

toolbox.**medAbsDev**(*series*)

Calculate median absolute deviation of a given input series

Median absolute deviation (MAD) is a robust and resistant measure of the spread of a sample (same purpose as standard deviation). The MAD is preferred to the inter-quartile range as the inter-quartile range only shows 50% of the data whereas the MAD uses all data but remains robust and resistant. See e.g. Wilks, Statistical methods for the Atmospheric Sciences, 1995, Ch. 3.

Parameters **series** : array_like

the input data series

Returns **out** : float

the median absolute deviation

Examples

Find the median absolute deviation of a data set. Here we use the log- normal distribution fitted to the population of sawtooth intervals, see Morley and Henderson, Comment, Geophysical Research Letters, 2009.

```
>>> import numpy
>>> import spacepy.toolbox as tb
>>> numpy.random.seed(8675301)
>>> data = numpy.random.lognormal(mean=5.1458, sigma=0.302313, size=30)
>>> print data
array([ 181.28078923,  131.18152745, ... , 141.15455416, 160.88972791])
>>> tb.medabsdev(data)
28.346646721370192
```

note This implementation is robust to presence of NaNs

toolbox.**mlt2rad**(*mlt*, *midnight=False*)

Convert mlt values to radians for polar plotting transform mlt angles to radians from -pi to pi referenced from noon by default

Parameters **mlt** : numpy array

array of mlt values

midnight : boolean (optional)

reference to midnight instead of noon

Returns **out** : numpy array

array of radians

Examples

```
>>> from numpy import array
>>> mlt2rad(array([3,6,9,14,22]))
array([-2.35619449, -1.57079633, -0.78539816,  0.52359878,  2.61799388])
```

toolbox.**normalize**(*vec*)

Given an input vector normalize the vector

Parameters **vec** : array_like

input vector to normalize

Returns **out** : array_like

normalized vector

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.normalize([1,2,3])
[0.0, 0.5, 1.0]
```

toolbox.**pmm**(*a*, **b*)

print min and max of input arrays

Parameters **a** : numpy array

input array

b : list argumants

some additional number of arrays

Returns **out** : list

list of min, max for each array

Examples

```
>>> import spacepy.toolbox as tb
>>> from numpy import arange
>>> tb.pmm(arange(10), arange(10)+3)
[[0, 9], [3, 12]]
```

toolbox.**printfig**(*fignum*, *saveonly=False*, *pngonly=False*, *clean=False*, *filename=None*)

save current figure to file and call lpr (print).

This routine will create a total of 3 files (png, ps and c.png) in the current working directory with a sequence number attached. Also, a time stamp and the location of the file will be imprinted on the figure. The file ending with c.png is clean and no directory or time stamp are attached (good for powerpoint presentations).

Parameters **fignum** : integer

matplotlib figure number

saveonly : boolean (optional)

True (don't print and save only to file) False (print and save)

pngolny : boolean (optional)

True (only save png files and print png directly) False (print ps file, and generate png, ps; can be slow)

clean : boolean (optional)

True (print and save only clean files without directory info) False (print and save directory location as well)

filename : string (optional)

None (If specified then the filename is set and code does not use the sequence number)

Examples

```
>>> import spacepy.toolbox as tb
>>> import matplotlib.pyplot as plt
>>> p = plt.plot([1,2,3],[2,3,2])
>>> tb.printfig(1, pngonly=True, saveonly=True)
```

toolbox.**progressbar**(*count*, *blocksize*, *totalsize*)

print a progress bar with urllib.urlretrieve reporthook functionality

Examples

```
>>> import spacepy.toolbox as tb
>>> import urllib
>>> urllib.urlretrieve(PSDDATA_URL, PSDdata_fname, reporthook=tb.progressbar)
```

toolbox.**query_yes_no**(*question*, *default='yes'*)

Ask a yes/no question via raw_input() and return their answer.

"question" is a string that is presented to the user. "default" is the presumed answer if the user just hits <Enter>. It must be "yes" (the default), "no" or None (meaning an answer is required of the user).

The "answer" return value is one of "yes" or "no".

Parameters **question** : string

the question to ask

default : string (optional)

Returns **out** : string

answer ('yes' or 'no')

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.query_yes_no('Ready to go?')
Ready to go? [Y/n] y
'yes'
```

toolbox.**rad2mlt**(*rad*, *midnight=False*)

Convert radian values to mlt transform radians from -pi to pi to mlt referenced from noon by default

Parameters **rad** : numpy array

array of radian values

midnight : boolean (optional)

reference to midnight instead of noon

Returns **out** : numpy array

array of mlt values

Examples

```
>>> rad2mlt(array([0,pi, pi/2.]))
array([ 12.,  24.,  18.])
```

toolbox.**savepickle**(*fln*, *dict*, *compress=None*)

save dictionary variable dict to a pickle with filename fln

Parameters **fln** : string

filename

dict : dict

container with stuff

compress : bool

write as a gzip-compressed file (.gz will be added to L{fln}). If not specified, defaults to uncompressed, unless the compressed file exists and the uncomprssed does not.

See Also:

loadpickle

Examples

```
>>> d = {'grade':[1,2,3], 'name':['Mary', 'John', 'Chris']}
>>> savepickle('test.pbin', d)
```

toolbox.**smartTimeTicks**(*time*)

Returns major ticks, minor ticks and format for time-based plots

smartTimeTicks takes a list of datetime objects and uses the range to calculate the best tick spacing and format. Returned to the user is a tuple containing the major tick locator, minor tick locator, and a format string – all necessary to apply the ticks to an axis.

It is suggested that, unless the user explicitly needs this info, to use the convenience function applySmartTimeTicks to place the ticks directly on a given axis.

Parameters **time** : list

list of datetime objects

Returns **out** : tuple

tuple of Mtick - major ticks, mtick - minor ticks, fmt - format

toolbox.**tCommon**(*ts1*, *ts2*, *mask_only=True*)

Finds the elements in a list of datetime objects present in another

Parameters **ts1** : list

first set of datetime objects

ts2 : list

second set of datetime objects

Returns **out** : tuple

Two element tuple of truth tables (of 1 present in 2, & vice versa)

toolbox.**tOverlap**(*ts1*, *ts2*, **args*, ***kwargs*)

Finds the overlapping elements in two lists of datetime objects

Parameters **ts1** : datetime

first set of datetime object

ts2: **datetime** :

datatime object

args: :

additional arguments passed to tOverlapHalf

Returns **out** : list

indices of ts1 within interval of ts2, & vice versa

Examples

Given two series of datetime objects, event_dates and omni['Time']:

```
>>> import spacepy.toolbox as tb
>>> from spacepy import omni
>>> import datetime
>>> event_dates = st.tickrange(datetime.datetime(2000, 1, 1), datetime.datetime(2000, 10, 1), de
>>> onni_dates = st.tickrange(datetime.datetime(2000, 1, 1), datetime.datetime(2000, 10, 1), del
>>> omni = omni.get_omni(onni_dates)
>>> [einds,oinds] = tb.tOverlap(event_dates, omni['ticks'])
>>> omni_time = omni['ticks'][oinds[0]:oinds[-1]+1]
>>> print omni_time
[datetime.datetime(2000, 1, 1, 0, 0), datetime.datetime(2000, 1, 1, 12, 0),
... , datetime.datetime(2000, 9, 30, 0, 0)]
```

toolbox.**tOverlapHalf**(*ts1*, *ts2*, *presort=False*)

Find overlapping elements in two lists of datetime objects

This is one-half of tOverlap, i.e. it finds only occurances where ts2 exists within the bounds of ts1, or the second element returnd by tOverlap.

Parameters **ts1** : list

first set of datetime object

ts2 : list

datatime object

presort : bool

Set to use a faster algorithm which assumes ts1 and ts2 are both sorted in ascending order. This speeds up the overlap comparison by about 50x, so it is worth sorting the list if one sort can be done for many calls to tOverlap

Returns **out** : list

indices of ts2 within interval of ts1

note: Returns empty list if no overlap found

toolbox.**thread_job**(*job_size*, *thread_count*, *target*, **args*, ***kwargs*)

Split a job into subjobs and run a thread for each

Each thread spawned will call L{target} to handle a slice of the job.

This is only useful if a job:

1. Can be split into completely independent subjobs
2. Relies heavily on code that does not use the Python GIL, e.g. numpy or ctypes code
3. Does not return a value. Either pass in a list/array to hold the result, or see L{thread_map}

Parameters **job_size** : int

Total size of the job. Often this is an array size.

thread_count : int

Number of threads to spawn. If =0 or None, will spawn as many threads as there are cores available on the system. (Each hyperthreading core counts as 2.) Generally this is the Right Thing to do. If NEGATIVE, will spawn abs(thread_count)

threads, but will run them sequentially rather than in parallel; useful for debugging.

target : callable

Python callable (generally a function, may also be an imported ctypes function) to run in each thread. The *last* two positional arguments passed in will be a "start" and a "subjob size," respectively; frequently this will be the start index and the number of elements to process in an array.

args : sequence

Arguments to pass to L{target}. If L{target} is an instance method, self must be explicitly pssed in. start and subjob_size will be appended.

kwargs : dict

keyword arguments to pass to L{target}.

Examples

squaring 100 million numbers:

```
>>> import numpy
>>> import spacepy.toolbox as tb
>>> numpy.random.seed(8675301)
>>> a = numpy.random.randint(0, 100, [100000000])
>>> b = numpy.empty([100000000], dtype='int64')
>>> def targ(in_array, out_array, start, count):                out_array[start:start + count] = i
>>> tb.thread_job(len(a), 0, targ, a, b)
>>> print(b[0:5])
[2704 7225  196 1521   36]
```

This example:

- Defines a target function, which will be called for each thread. It is usually necessary to define a simple "wrapper" function like this to provide the correct call signature.
- The target function receives inputs C{in_array} and C{out_array}, which are not touched directly by C{thread_job} but are passed through in the call. In this case, C{a} gets passed as C{in_array} and C{b} as C{out_array}
- The target function also receives the start and number of elements it needs to process. For each thread where the target is called, these numbers are different.

toolbox.**thread_map**(*target*, *iterable*, *thread_count=None*, **args*, ***kwargs*)

Apply a function to every element of a list, in separate threads

Interface is similar to multiprocessing.map, except it runs in threads

Parameters **target** : callable

Python callable to run on each element of iterable. For each call, an element of iterable is appended to args and both args and kwargs are passed through. Note that this means the iterable element is always the *last* positional argument; this allows the specification of self as the first argument for method calls.

iterable : iterable

elements to pass to each call of L{target}

args : sequence

arguments to pass to target before each element of iterable

thread_count : integer

Number of threads to spawn; see L{thread_job}.

kwargs : dict

keyword arguments to pass to L{target}.

Returns **out** : list

return values of L{target} for each item from L{iterable}

Examples

find totals of several arrays

```
>>> import numpy
>>> from spacepy import toolbox
>>> inputs = range(100)
>>> totals = toolbox.thread_map(numpy.sum, inputs)
>>> print(totals[0], totals[50], totals[99])
(0, 50, 99)
```

toolbox.**timestamp**(*position=[1.0029999999999999, 0.01], size='xx-small', draw=True, **kwargs*)

print a timestamp on the current plot, vertical lower right

Parameters **position** : list

position for the timestamp

size : string (optional)

text size

draw : bollean (optional)

call draw to make sure it appears

kwargs : keywords

other keywords to axis.annotate

Examples

```
>>> import spacepy.toolbox as tb
>>> from pylab import plot, arange
>>> plot(arange(11))
[<matplotlib.lines.Line2D object at 0x49072b0>]
>>> tb.timestamp()
```

toolbox.**update**(*all=True*, *omni=False*, *leapsecs=False*, *PSDdata=False*)

Download and update local database for omni, leapsecs etc

Parameters **all** : boolean (optional)

if True, update all of them

omni : boolean (optional)

if True. update only onmi

leapsecs : boolean (optional)

if True, update only leapseconds

Returns **out** : string

data directory where things are saved

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.update(omni=True)
```

toolbox.**windowMean**(*data*, *time=[]*, *winsize=0*, *overlap=0*, *st_time=None*)

Windowing mean function, window overlap is user defined

Parameters **data** : array_like

1D series of points;

time : list (optional)

series of timestamps, optional (format as numeric or datetime) For non-overlapping windows set overlap to zero.

winsize : integer or datetime.timedelta (optional)

window size

overlap : integer or datetime.timedelta (optional)

amount of window overlap

st_time : datetime.datetime (optional)

for time-based averaging, a start-time other than the first point can be specified

Returns **out** : tuple

the windowed mean of the data, and an associated reference time vector

Examples

For non-overlapping windows set overlap to zero. e.g. (time-based averaging) Given a data set of 100 points at hourly resolution (with the time tick in the middle of the sample), the daily average of this, with half-overlapping windows is calculated:

```
>>> import spacepy.toolbox as tb
>>> from datetime import datetime, timedelta
>>> wsize = datetime.timedelta(days=1)
>>> olap = datetime.timedelta(hours=12)
>>> data = [10, 20]*50
>>> time = [datetime.datetime(2001,1,1) + datetime.timedelta(hours=n, minutes = 30) for n in ran
>>> outdata, outtime = tb.windowMean(data, time, winsize=wsize, overlap=olap, st_time=datetime.d
>>> outdata, outtime
([15.0, 15.0, 15.0, 15.0, 15.0, 15.0, 15.0],
 [datetime.datetime(2001, 1, 1, 12, 0),
  datetime.datetime(2001, 1, 2, 0, 0),
  datetime.datetime(2001, 1, 2, 12, 0),
```

```
  datetime.datetime(2001, 1, 3, 0, 0),
  datetime.datetime(2001, 1, 3, 12, 0),
  datetime.datetime(2001, 1, 4, 0, 0),
  datetime.datetime(2001, 1, 4, 12, 0)])
```

When using time-based averaging, ensure that the time tick corresponds to the middle of the time-bin to which the data apply. That is, if the data are hourly, say for 00:00-01:00, then the time applied should be 00:30. If this is not done, unexpected behaviour can result.

e.g. (pointwise averaging),

```
>>> outdata, outtime = tb.windowMean(data, winsize=24, overlap=12)
>>> outdata, outtime
([15.0, 15.0, 15.0, 15.0, 15.0, 15.0, 15.0], [12.0, 24.0, 36.0, 48.0, 60.0, 72.0, 84.0])
```

where winsize and overlap are numeric, in this example the window size is 24 points (as the data are hourly) and the overlap is 12 points (a half day). The output vectors start at winsize/2 and end at N-(winsize/2), the output time vector is basically a reference to the nth point in the original series.

note This is a quick and dirty function - it is NOT optimized, at all.

2.11 Indices and tables

- *genindex*
- *modindex*
- *search*

Release 0.1.0

Doc generation date August 23, 2011

PYTHON MODULE INDEX

PYTHON MODULE INDEX