SpacePy Documentation

Release 0.2.1

The SpacePy Team

October 02, 2019

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 (details in the `__citation__` attribute) before publication or public presentation of analysis performed by that module, or in case of questions about the module. This allows the author to validate the analysis and receive appropriate credit for his or her work.

CHAPTER

ONE

SPACEPY DOCUMENTS

SpacePy Help

The best way to get help is to open an issue. Searching existing issues may find other people who have had similar questions. Try changing the filters (at the top) to include closed issues, as they may have been addressed.

Most modules also have a contact person listed in the docstring. These are built into in the main SpacePy documentation, or you can view it from within Python/IPython:

```
>>> import spacepy.pycdf
>>> help(spacepy.pycdf)
>>> print(spacepy.pycdf.__contact__)
```

A web version of this documentation is currently hosted at spacepy.github.io.

Contributing

Contributions to SpacePy are welcome! Development is managed via our github. If you're interesting in contributing a new feature or bugfix, it is recommended to open an issue to discuss your plans. Once your code is ready, you can open a pull request. Thanks for helping to improve SpacePy!

Installing SpacePy

The simplest way from zero to a working SpacePy setup is:

1. Install the Anaconda Python environment. Python 3 is recommended (as is, usually, 64-bit).
2. `pip install --upgrade spacepy`

If you are familiar with installing Python packages, have particular preferences for managing an installation, or if the above doesn't work, refer to platform-specific instructions and the details below. (In particular, if you will use *LANLstar* on Windows, see Windows Installation.)

For installing the NASA CDF library to support *pycdf*, see the platform-specific instructions linked below.

If you need further assistance, you can open an issue.

SpacePy Dependencies

SpacePy relies on several other pieces of software for complete functionality. Installing SpacePy links to details on installing the required software for each platform.

Unless otherwise noted, a dependency may be installed *after* SpacePy, and the new functionality will be available the next time SpacePy is imported.

Hard Dependencies

Without these packages installed, SpacePy will not function.

Python 2.6+

Python is the core language for SpacePy. Python 2.6 or later is required. SpacePy works under Python 3, but Python 2 is still recommended for most users. *Python 3 is not simply a "newer" Python 2; there are substantial differences in the language*. See Should I use Python 2 or Python 3?.

Required to install SpacePy.

NumPy 1.4+

NumPy provides the high-performance array data structure used throughout SpacePy. Version 1.4 or later is required; 1.6 or later recommended.

Required to install SpacePy. f2py is part of NumPy, but is sometimes packaged separately; it is required (at installation time) if `irbempy` is to be used.

dateutil

If you choose not to install *matplotlib*, dateutil is required. (Installing matplotlib will fulfill this dependency.)

C compiler

If you are installing SpacePy from source, a working C compiler is required. (Not necessary for the Windows binary installer.)

Soft Dependencies

Without these packages, SpacePy will install, but certain features may not be available. Usually an ImportError means a dependency is missing.

These are simply marked as dependencies in SpacePy metadata and thus will be automatically installed when using dependency-resolving methods such as pip.

SciPy

SciPy provides several useful scientific and numerical functions build on top of NumPy. It is highly recommended. Version 0.7.0 or later should suffice. The following modules may have limited functionality without SciPy:

- `empiricals`
- `seapy`
- `toolbox`

matplotlib

matplotlib is the preferred plotting package for Python. It is highly recommended. Without it, you will not be able to effectively visualize data, and the following modules may have limited functionality:

- *data_assimilation*
- *plot*
- *poppy*
- *pybats*
- *radbelt*
- *time*
- *toolbox*

ffnet

ffnet is a neural network package, required for *LANLstar*. The latest release of ffnet at this writing, 0.7.1, does not support Python 3. We have submitted patches for Python 3 support and they have been integrated into SVN as of r364, so they should be in the next version. In the meantime, users who require Python 3 support will need to install ffnet from a subversion checkout.

networkx

networkx is a requirement for ffnet, and thus *LANLstar*.

h5py

h5py provides a Python interface to HDF5 files. It is required for the HDF import/export capability of *datamodel* and for use of the *omni* module.

CDF

NASA's CDF library provides access to Common Data Format files. It is required for *pycdf*, and thus for the CDF import/export capability of *datamodel*.

Warning: Unlike the Python-based dependencies, the CDF library must be installed if pycdf support is needed; it will not be automatically installed.

Fortran compiler

If installing from source, *irbempy* requires a Fortran compiler. (This is not required for the Windows binary installer). Supported compilers are the GNU compiler `gfortran`, the older GNU compiler `g77`, and the Portland Group PGI compiler.

If *irbempy* is to be used, the Fortran compiler (and f2py) must be installed before SpacePy.

Linux Installation

Installation on Linux requires both a C and a Fortran compiler; a recent GCC is recommended (the C compiler is likely included with your distribution). On Debian and Ubuntu:

```
sudo apt-get install gfortran
```

Once this is set up, `pip install spacepy` should Just Work. If you're installing as a single user (not in a virtual environment) then add the `--user` flag.

You will also need the *NASA CDF library* to use `pycdf`.

Our recommended (but not required) Python distribution is Anaconda running 64-bit Python 3. Anaconda includes much of the scientific Python stack. Another excellent distribution is Canopy.

You may need to install the dependencies some way other than pip; for example, if you are running an earlier version of Python. The latest version of many dependencies requires Python 3.6 and pip will not install older versions to get around this. See *Dependencies via conda* and *Dependencies via system packages*.

Dependencies via conda

Installation via `pip` will automatically install most Python dependencies (but not the *NASA CDF library*). They can also be installed from conda:

```
conda install numpy scipy matplotlib networkx h5py
```

Dependencies via system packages

SpacePy usually works with the system Python on Linux. To install dependencies via the package manager on Debian or Ubuntu:

```
sudo apt-get install python-dev python-h5py python-matplotlib python-networkx python-numpy python-sc
```

For Python 3, use:

```
sudo apt-get install python3-dev python3-h5py python3-matplotlib python3-networkx python3-numpy pytho
```

For other distributions, check SpacePy Dependencies and install by hand or via your package manager.

CDF

It is recommended to install the ncurses library; on Ubuntu and Debian:

```
sudo apt-get install ncurses-dev
```

Download the latest CDF library. Choose the file ending in `-dist-all.tar.gz` from the `linux` directory. Untar and cd into the resulting directory. Then build:

```
make OS=linux ENV=gnu CURSES=yes FORTRAN=no UCOPTIONS=-O2 SHARED=yes all
```

Use `CURSES=no` if the curses library is not installed. (The distribution-specific directions above will install curses.)

Install:

```
sudo make install
```

This will install the library into the default location `/usr/local/cdf`, where SpacePy can find it. If you choose to install elsewhere, see the CDF documentation, particularly the notes on the `CDF_BASE` and `CDF_LIB` environment variables. SpacePy uses these variables to find the library.

ffnet

ffnet can be installed from `pip`; if the wrong Fortran compiler is found try:

```
FC_VENDOR=gfortran pip install ffnet
```

It can also be installed from sourcea. Compilation requires f2py (from numpy), installed per the distribution-specific directions above. Untar and cd into the resulting directory. Then build:

```
python setup.py build
```

Either install just for one user:

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

Or install for all users on the system:

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

Normally the correct Fortran compiler will be found; if compilation fails, try specifying the older GNU compiler at the build step:

```
python setup.py build --fcompiler=gnu
```

Compiling

With the dependencies installed, SpacePy can be built from source. This uses the same basic setup as ffnet (standard Python distutils). You can always get the latest source code for SpacePy from our github repository and the latest release from PyPI

Build:

```
python setup.py build
```

If this fails, specify a Fortran compiler:

```
python setup.py build --fcompiler=gnu95
```

`python setup.py build --help-fcompiler` will list options for Fortran compilers. Currently available compilers are `pg`, `gnu95`, `gnu`, `intelem`, `intel` or `none` (to skip all Fortran); `gnu95` (the GNU gfortran compiler) is recommended.

Install for one user:

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

If you're using conda, installation as user isn't recommended:

```
python setup.py install
```

Or install for all users on the system:

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

If you want to build the documentation yourself (rather than using the documentation shipped with SpacePy), install sphinx and numpydoc. The easiest way is via pip:

```
pip install sphinx numpydoc
```

They are also available via conda:

```
conda install sphinx numpydoc
```

Or the package manager:

> sudo apt-get install python-sphinx python-numpydoc

For Python 3:

> sudo apt-get install python3-sphinx python3-numpydoc

MacOS Installation

The following are performed from the command line on OSX.

Installation on OSX requires a C compiler:

```
xcode-select --install
```

Our recommended (but not required) Python distribution is Anaconda running 64-bit Python 3. Anaconda includes much of the scientific Python stack. Another excellent distribution is Canopy.

Once python is installed (Anaconda assumed) install the Fortran compiler:

```
conda install gfortran_osx-64
```

Dependencies via conda and pip

Installation via `pip` will automatically install most Python dependencies (but not the *NASA CDF library*). They can also be installed from conda:

```
conda install numpy scipy matplotlib networkx h5py
pip install ffnet
```

Once this is set up, `pip install spacepy` should just work. If you're installing as a single user (not in a virtual environment) then add the `--user` flag.

You will also need the *NASA CDF library* to use *`pycdf`*.

To install the latest code from the repository, rather than the latest stable release, use:

```
pip install git+https://github.com/spacepy/spacepy
```

Windows Installation

The SpacePy team currently provides binary "wheels" via PyPI so it can be installed on Windows without a compiler. Binaries are provided for Python 2.7, 3.6, and 3.7 in 64-bit and 32-bit variants for each. `pip install spacepy` should find and install these binaries.

Our recommended (but not required) Python distribution is Anaconda running 64-bit Python 3. Anaconda includes much of the scientific Python stack. Another excellent distribution is Canopy.

You may need to install the dependencies some way other than pip; for example, if you are running an earlier version of Python. The latest version of many dependencies requires Python 3.6 and pip will not install older versions to get around this. See *Dependencies via conda*.

Fortran and ffnet

ffnet is required for *`LANLstar`*. It can be installed either before or after SpacePy. Binary wheels are not provided, so a Fortran compiler is required

With Anaconda, the compiler and ffnet can be installed with:

```
conda install m2w64-gcc-fortran libpython
SET FC_VENDOR=gfortran
pip install ffnet
```

The *FC_VENDOR* line is necessary because ffnet defaults to the Microsoft compiler.

Some standalone binary installers (no `pip` support) are also available on the ffnet site. These do not require a compiler but support only a limited set of Python versions.

Compiling

If a binary wheel is not available for your version of Python, `pip` will try to compile SpacePy. The only supported compiler is `mingw32`. Install it with:

```
conda install m2w64-gcc-fortran libpython
```

This is also required if installing from a source distribution or git checkout.

`irbempy` requires Fortran to compile and the only supported compiler is `gnu95`; this is the default and provided by `m2w64-gcc-fortran`.

If you have difficulties, it may be useful to reference the *build scripts <https://github.com/spacepy/spacepy/tree/master/developer/scripts>* the SpacePy developers use.

NASA CDF

`pycdf` requires the NASA CDF library . Binary installers are available for Windows; be sure to pick the version that matches your Python installation. The current 32-bit version is cdf37_1_0-setup-32.exe; for 64-bit, cdf37_1_0-setup-64.exe.

This is a simple self-extracting installer that can be installed either before or after installing SpacePy.

Standalone installers

Self-extracting and self-installing executables are also available for download direct from our github. Be sure to choose the file that matches your Python installation, both in Python version and word size (64-bit vs. 32-bit.) E.g. `spacepy-0.2.0.win-amd64.py36.exe` is the installer for Python 3.6 on 64-bit Windows and `spacepy-0.2.0.win32.py27.exe` is the installer for Python 2.7 on 32-bit Windows.

Once downloaded, these can be installed without an internet connection.

If using these installers, the SpacePy Dependencies will not be installed automatically.

Dependencies via conda

Installation via `pip` will automatically install most Python dependencies (but not *ffnet* or the *NASA CDF library*). They can also be installed from conda:

```
conda install numpy scipy matplotlib networkx h5py
```

Standalone dependencies

Most of the SpacePy Dependencies have Windows installers available via their pages, but `pip` or `conda` are recommended instead.

Developers

If you want to build the documentation yourself (rather than using the documentation shipped with SpacePy), install sphinx and numpydoc. The easiest way is via pip:

```
pip install sphinx numpydoc
```

They are also available via conda:

```
conda install sphinx numpydoc
```

SpacePy installs with the common Python distutils and pip.

The latest stable release is provided via PyPI To install from PyPI, make sure you have pip installed:

```
pip install --upgrade spacepy
```

If you are installing for a single user, and are not working in a virtual environment, add the `--user` flag when installing with pip.

Source releases are available from PyPI and our github. Development versions are on github.

After downloading and unpacking, run (a virtual environment, such as a conda environment, is recommended):

```
python setup.py install
```

or, to install for all users (not in a virtual environment):

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

or, to install for a single user (not in a virtual environment):

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

If you do not have administrative privileges, or you will be developing for SpacePy, we strongly recommend using virtual environments.

To install in custom location, e.g.:

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

Installs using `setup.py` do not require setuptools. SpacePy: Space Science Tools for Python

SpacePy is a package of tools primarily aimed at the space science community. This __init__.py file sets the parameters for import statements.

If running the ipython shell, simply type '?' after any command for help. ipython also offers tab completion, so hitting tab after '<module name>.' will list all available functions, classes and variables.

Detailed HTML documentation is available locally in the spacepy/doc directory and can be launched by typing:

```
>>> spacepy.help()
```

Most functionality is in spacepy's submodules. Each module has specific help available:

coordinates	Implementation of Coords class functions for coordinate transformations
data_assimilation	
datamodel	The datamodel classes constitute a data model implementation meant to mirror the functionality of the da
empiricals	Module with some useful empirical models (plasmapause, magnetopause, Lmax)
irbempy	module wrapper for irbem_lib
LANLstar	Lstar and Lmax calculation using artificial neural network (ANN) technique.
omni	Tools to read and process omni data (Qin-Denton, etc.)
poppy	PoPPy – Point Processes in Python.
pybats	PyBats! An open source Python-based interface for reading, manipulating, and visualizing BATS-R-US a
pycdf	This package provides a Python interface to the Common Data Format (CDF) library used for many NAS
radbelt	Functions supporting radiation belt diffusion codes
seapy	SeaPy – Superposed Epoch in Python.
time	Time conversion, manipulation and implementation of Ticktock class
toolbox	Toolbox of various functions and generic utilities.
ae9ap9	Module for reading and dealing with AE9AP9 data files.

SpacePy - A Quick Start Documentation

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

Installation

See Installing SpacePy.

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:

- `windowMean()`: windowing mean with variable window size and overlap
- `dictree()`: pretty prints the contents of dictionaries (recursively)
- `loadpickle()`: single line convenience routine for loading Python pickles
- `savepickle()`: same as loadpickle, but for saving
- `update()`: updates the OMNI database and the leap seconds database (internet connection required)
- `tOverlap()`: find interval of overlap between two time series
- `tCommon()`: find times common to two time series
- `binHisto()`: calculate number of bins for a histogram
- `medAbsDev()`: find the median absolute deviation of a data series
- `normalize()`: normalize a data series
- `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)
```

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`
- 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()
```

Timedelta Class

You can add/subtract time from a Ticktock class instance by using an instance of `datetime.timedelta`:

```
>>> dt = datetime.timedelta(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 spherical forms.

- 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.

The radbelt Module

The radiation belt module currently includes a simple radial diffusion code as a class. Import the module and instatiate a radbelt object:

```
>>> 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()
```

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 (NOTE: This requires a database available from the SpacePy team):

```
>>> 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’.

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, at 1-hourly resolution, on request on first import; if not downloaded when SpacePy is first used then any 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'])
```

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,
```

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()*

- *coord_trans()*

pyCDF - Python Access to NASA CDF Library

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

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

To open and close a CDF file, we use the *CDF* class:

```
>>> 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 like a numpy array, where the first dimension is the record number. Multidimensional CDF variables can be subscripted using numpy's multidimensional slice notation. Many common list operations are also implemented, where each record acts as one element of the list and can be independently deleted, inserted, etc. 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
```

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 `tree()` method, which is equivalent to `toolbox.dictree()` (which works for any dictionary-like object, including PyCDF file objects).

```
>>> mydata.tree(attrs=True)
+
:|____MissionName
:|____PI
|____Counts
    :|____Units
|____Epoch
    :|____units
|____OrbitNumber
    :|____StartsFrom
>>> 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 (and can be exceptionally useful for representing data, so to make this compatible with NASA CDF we call the *flatten()* method .

```
>>> 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 exist to read HDF5 and NASA CDF files directly to a SpacePy datamodel. This capability also exists for JSON-headed ASCII files (RBSP/AutoPlot-compatible). A converter from the datamodel to HDF5 is now available and a converter to NASA CDF is under development. Also under development is the reverse of the SpaceData.flatten method, so that flattened objects can be restored to their former glory.

Empiricals Module

The empiricals module provides access to some useful empirical models. As of SpacePy 0.1.2, the models available are:

- *getLmax()* An empirical parametrization of the L* of the last closed drift shell (Lmax)
- *getPlasmaPause()* The plasmapause location, following either Carpenter and Anderson (1992) or Moldwin et al. (2002)
- *getMPstandoff()* The magnetopause standoff location (i.e. the sub-solar point), using the Shue et al. (1997) model
- *vampolaPA()* A conversion of omnidirectional electron flux to pitch-angle dependent flux, using the sin ^{n} model of Vampola (1996)

Each of the first three models 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 *tickrange()* 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])
```

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 datetime as dt
>>> import spacepy.seapy as sea
>>> import spacepy.omni as om
>>> import spacepy.toolbox as tb
>>> import spacepy.time as spt
>>> # now read the epochs for the analysis (the path specified is the default
>>> # install location on linux, different OS will have this elsewhere)
>>> epochs = sea.readepochs('~/.local/lib/python2.7/site-packages/spacepy/data/SEA_epochs_OMNI.txt')
```

The readepochs function can handle multiple formats by a user-specified format code. ISO 8601 format is directly supported though it is not used here. The the readepochs docstring for more information. As above, we use the get_omni function to retrieve the hourly data from the OMNI module:

```
>>> ticks = spt.tickrange(dt.datetime(2005,1,1), dt.datetime(2009,1,1), dt.timedelta(hours=1))
>>> omni1hr = om.get_omni(ticks)
>>> omni1hr.tree(levels=1, verbose=True)
```

```
+
|____ByIMF (spacepy.datamodel.dmarray (35065,))
|____Bz1 (spacepy.datamodel.dmarray (35065,))
|____Bz2 (spacepy.datamodel.dmarray (35065,))
|____Bz3 (spacepy.datamodel.dmarray (35065,))
|____Bz4 (spacepy.datamodel.dmarray (35065,))
|____Bz5 (spacepy.datamodel.dmarray (35065,))
|____Bz6 (spacepy.datamodel.dmarray (35065,))
|____BzIMF (spacepy.datamodel.dmarray (35065,))
|____DOY (spacepy.datamodel.dmarray (35065,))
|____Dst (spacepy.datamodel.dmarray (35065,))
```

```
|____G (spacepy.datamodel.dmarray (35065, 3))
|____Hr (spacepy.datamodel.dmarray (35065,))
|____Kp (spacepy.datamodel.dmarray (35065,))
|____Pdyn (spacepy.datamodel.dmarray (35065,))
|____Qbits (spacepy.datamodel.SpaceData [7])
|____RDT (spacepy.datamodel.dmarray (35065,))
|____UTC (spacepy.datamodel.dmarray (35065,))
|____W (spacepy.datamodel.dmarray (35065, 6))
|____Year (spacepy.datamodel.dmarray (35065,))
|____akp3 (spacepy.datamodel.dmarray (35065,))
|____dens (spacepy.datamodel.dmarray (35065,))
```

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

```
>>> delta = dt.timedelta(hours=1)
>>> window= dt.timedelta(days=3)
>>> sevx = sea.Sea(omni1hr['velo'], omni1hr['UTC'], epochs, window, delta)
    #rather than quartiles, we calculate the 95% confidence interval on the median
>>> sevx.sea(ci=True)
>>> sevx.plot()
```

Documentation Standard

SpacePy aims to be a high quality product, and as such we (the SpacePy Team) encourage a a high degree of uniformity in the documentation across included modules. If you are contributing to SpacePy, or hope to, please take the time to make your code compliant with the documentation standard.

SpacePy uses Sphinx to generate its documentation. This allows most of the documentation to be built from docstrings in the code, with additional information being provided in reStructured Text files. This allows easy generation of high-quality, searchable HTML documentation.

In addition to Sphinx, SpacePy uses the following extensions:

- 'sphinx.ext.autodoc'
- 'sphinx.ext.doctest' '
- 'sphinx.ext.intersphinx'
- 'sphinx.ext.todo'
- 'sphinx.ext.imgmath' (falls back to 'sphinx.ext.pngmath' if imgmath is not available)
- 'sphinx.ext.ifconfig'
- 'sphinx.ext.viewcode'
- 'numpydoc'
- 'sphinx.ext.inheritance_diagram'
- 'sphinx.ext.autosummary'
- 'sphinx.ext.extlinks'

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 in your docstrings * Most reStructuredText commands cannot appear in your docstrings either (e.g. .. Note:) * Since 'numpydoc' is not well documented, the best way of finding out what you can do in your docstrings is to look at the source for the SpacePy documentation or the numpy documentation.

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 should be able to type the exact code in the example and it should work as shown (doctest can help with this)

Function Example

This code from toolbox shows what a function 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:

`spacepy.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

See also:

`geomspace`, `linspace`

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.        ])
```

SpacePy Python Programming Tips

One often hears that interpreted languages are too slow for whatever task someone needs to do. In many cases this belief is unfounded. As the time spent programming and debugging in an interpreted language is of far less than for a compiled language, the programmer has more time to identify bottlenecks in the code and optimize it where necessary. This page is dedicated to that idea, providing examples of code speedup and best practices.

One often neglected way to speed up development time is to use established libraries, and the time spent finding existing code that does what you want can be more productive than trying to write and optimize your own algorithms. We recommend exploring the SpacePy documentation, as well as taking the time to learn some of the vast functionality already in numpy and the Python standard library.

Basic examples

Though there are some similarities, Python does not look like (or work like) Matlab or IDL. As (most of us) are, or have been, Matlab or IDL programmers, we have to rethink how we do things – what is efficient in one language may not be the most efficient in another. One truth that Python shares with these other languages, however, is that if you are using a for loop there is likely to be a faster way...

Take the simple case of a large data array where you want to add one to each element. Here wa show four of the possible ways to do this, and by using a profiling tool, we can show that the speeds of the different methods can vary substantially.

Create the data

```
>>> data = range(10000000)
```

The most C-like way is just a straight up for loop

```
>>> for i in range(len(data)):
>>>     data[i] += 1
```

This is 6 function calls in 2.590 CPU seconds (from `cProfile`)

The next, more Pythonic, way is with a list comprehension

```
>>> data = [val+1 for val in data]
```

This is 4 function calls in 1.643 CPU seconds (~1.6x)

Next we introduce numpy and change our list to an array then add one

```
>>> data = np.asarray(data)
>>> data += 1
```

This is 6 function calls in 1.959 CPU seconds (~1.3x), better than the for loop but worse than the list comprehension

Next we do this the *right* way and just create it in numpy and never leave

```
>>> data = np.arange(10000000)
>>> data += 1
```

this is 4 function calls in 0.049 CPU seconds (~53x).

While this is a really simple example it shows the basic premise, if you need to work with numpy, start in numpy and stay in numpy. This will usually be true for array-based manipulations.

If in doubt, and speed is not a major consideration, use the most human-readable form. This will make your code more maintainable and encourage its use by others.

Lists, for loops, and arrays

This example teaches the lesson that most advanced IDL or Matlab programmers already know; do everything in arrays and never use a for loop if there is another choice. The language has optimized array manipulation and it is unlikely that you will find a faster way with your own code.

The following bit of code takes in a series of coordinates, computes their displacement, 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" hig
```

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 moving the addition outside the for-loop gives a factor of 2.3 speedup. We believe that the difference arising from moving the addition lets numpy (which works primarily in C) operate once only. This massively reduces the calling overhead as array operations are done as for loops in C, and not in element-wise in python:

```
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" hig
```

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" hig
```

The answer is exactly the same and comparing to the initial version of this code we have managed a speedup of 382x:

```
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}
```

In summary, when working on arrays it's worth taking the time to think about whether you can get the results you need without for-loops or list comprehensions. The small amount of development time will likely be recouped very quickly.

Zip

The `zip()` function is extremely useful, but it is really slow. If you find yourself using it on large amounts of data then significant time-savings might be achieved by re-writing your code to make the `zip()` operation unnecessary. A good alternative, if you do need the functionality of `zip()`, is in `itertools.izip()`. This is far more efficient as it builds an interator.

This example generates N points, evenly distributed 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` shows that a lot of time is spent 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 concatenating the arrays 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}
```

External links

To learn about NumPy from a MatLab user's perspective

- NumPy for MatLab users
- Mathesaurus

And if you're coming from an IDL, or R, background

- Mathesaurus

Here is a collection of links that serve as a decent reference for Python and speed

- PythonSpeed PerformanceTips
- scipy array tip sheet
- Python Tips, Tricks, and Hacks

Release 0.2.1

Doc generation date October 02, 2019

For additions or fixes to this page contact the SpacePy team at Los Alamos.

SpacePy Configuration

SpacePy has a few tunable options that can be altered through the `spacepy.rc` configuration file. All options have defaults which will be used if not specified in the configuration file. These defaults are usually fine for most people and may change between SpacePy releases, so we do not recommend changing the configuration file without substantial reason.

`spacepy.rc` lives in the per-user SpacePy directory, called `.spacepy`. On Unix-like operating systems, it is in a user's home directory; on Windows, in the user's Documents and Settings folder. If it doesn't exist, this directory (and `spacepy.rc`) is automatically created when SpacePy is imported.

`spacepy.rc` has an INI-style format, parsed by `ConfigParser`. It contains a single section, `[spacepy]`.

- *The spacepy directory*
- *Available configuration options*
- *Developer documentation*

The spacepy directory

When first imported, spacepy 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

If you change the default location, make sure you add the environment variable `$SPACEPY` to your `.cshrc`, `.tcshrc`, or `.bashrc` script.

Available configuration options

enable_deprecation_warning SpacePy raises `DeprecationWarning` when deprecated functions are called. Starting in Python 2.7, these are ignored. SpacePy adds a warnings filter to force display of deprecation warnings from SpacePy the first time a deprecated function is called. Set this option to False to retain the default Python behavior. (See `warnings` module for details on custom warning filters.)

leapsec_url URL of the leapsecond database used by time conversions. `update()` will download from the URL. The default should almost always be acceptable.

ncpus Number of CPUs to use for computations that can be multithreaded/multiprocessed. By default, they will use the number of CPUs reported by `multiprocessing.cpu_count()`. You may wish to set this to a lower number if you need to reserve other processors on your machine.

notice True to display the SpacePy license and other information on import (default); False to omit.

omni2_url URL containing the OMNI2 data. `update()` will download from the URL. The default should almost always be acceptable.

qindenton_url URL containing Qin-Denton packaging of OMNI data. `update()` will download from the URL. The default should almost always be acceptable.

psddata_url URL containing PSD data. `update()` will download from the URL if requested. The default should almost always be acceptable.

support_notice True to display a notice on import if not a release version of SpacePy (default); False to omit. Those regularly installing from git instead of a release may want to set this to False.

user_agent User Agent for network access. If this is set, `update()` will use this User Agent string on all HTTP requests. Normally leaving this unset should be fine.

Developer documentation

`spacepy.rc` is loaded into a dictionary (`spacepy.config`) by SpacePy's main `__init__.py`. All options from the `[spacepy]` section are loaded, with no developer intervention needed. Each key is the option's name; the associated value is the option's value. To specify a default, add to the `defaults` dictionary at the top of `_read_config`; each default, if not overridden by the config file, will be included in the config dict. Values are assumed to be strings. The `caster` dictionary is keyed by option name; the value for each key is a function to be applied to the value with the same key to produce a different type from a string.

Release 0.2.1

Doc generation date October 02, 2019

For additions or fixes to this page, contact the SpacePy Team at Los Alamos.

SpacePy Case Studies

The SpacePy team has prepared case studies showing how to reproduce the results from published papers using Python-based tools, including SpacePy. It is hoped that these extensively-documented examples will ease the transition to Python for space scientists.

Basic familiarity with programming and general computing tasks in your chosen environment is assumed, including editing text files, copying and deleting files, etc. No Python-specific knowledge is assumed, although it is recommended to at least skim the excellent Python tutorial.

Paulikas and Blake revisited (Reeves et al. 2011)

This case study reproduces the figures of Reeves et al. (2011), "On the relationship between relativistic electron flux and solar wind velocity: Paulikas and Blake revisited" (doi:10.1029/2010JA015735).

Setup

Create a directory to hold files for this case study. Within this directory, create subdirectories `code`, `data`, and `plots`. (Using version control on the code directory is recommended; the SpacePy team uses git.)

Obtaining energetic particle data

We require the 1.8-3.5 MeV electron flux from the LANL-GEO ESP detector, available in the paper's auxiliary material (scroll down to "Supporting information" on the paper's page. The ESP data are in Data Set S1. Save this file to the `data` directory; the filename is assumed to be `jgra20797-sup-0003-ds01.txt`.

The data file was corrupted on upload to AGU, and the code to fix it is non-trivial, so this is a good chance to learn how to run someone else's code. (*Appendix: Fixing the ESP data file* has step-by-step information on each portion of this process.) Copy all of the following and paste it into a file called `fix_esp_data.py` in the `code` directory.

```python
import os.path


datadir = os.path.join('..', 'data')
in_name = os.path.join(datadir, 'jgra20797-sup-0003-ds01.txt')
out_name = os.path.join(datadir, 'jgra20797-sup-0003-ds01_FIXED.txt')
infile = open(in_name, 'r')
outfile = open(out_name, 'w')
data = infile.read()
infile.close()

data = data.replace('\r', '\n')
data = data.replace('\n\n', '\n')
data = data.split('\n')

for i in range(15):
    outfile.write(data.pop(0) + '\n')
oldline = None
for line in data:
    if line[0:2] in ['19', '20', '2']:
        if not oldline is None:
            outfile.write(oldline + '\n')
        oldline = line
    else:
        oldline += line
outfile.write(oldline + '\n')
outfile.close()
```

Now this script can be run with `python fix_esp_data.py`. It should create a file called `jgra20797-sup-0003-ds01_FIXED.txt` in the `data` directory.

File fixed, we can load and begin examining the data. Change to the `code` directory and start your Python interpreter. (IPython is recommended, but not required.)

In the following examples, do not type the leading >>>; this is the Python interpreter prompt. IPython has a different prompt that looks like `In [1]`.

```
>>> import os.path
>>> datadir = os.path.join('..', 'data')
>>> print(datadir)
../data
```

The first line imports the `os.path` module from the Python standard library. Python has a huge standard library. To keep this code organized, it is divided into many modules, and a module must be imported before it can be used. (The Python module of the week is a great way to explore the standard library.)

The second line makes a variable, `datadir`, which will contain the path of the data directory. The `os.path.join()` function provides a portable way of "gluing" together directories in a path, and will use backslashes on Windows and forward slashes on Unix. The third line then prints out the value of this variable for confirmation; note this is a Unix system.

Note that string constants in Python can use single or double quotes; we could just as well have written:

```
>>> datadir = os.path.join("..", "data")
```

or even:

```
>>> datadir = os.path.join('..', "data")
```

The full path can also be used (and this is a better case for using a variable.) For example, I am preparing this example in a directory `reeves_morley_friedel_2011` in my home directory, so I could use:

```
>>> datadir = os.path.join('home', 'jniehof', 'reeves_morley_friedel_2011',
...                        'data')
```

This very long line can be typed across two lines in Python, and because the line break happens within parentheses, a line continuation character is not required.

Returning to reading the ESP data file:

```
>>> fname = os.path.join(datadir, 'jgra20797-sup-0003-ds01_FIXED.txt')
```

creates a variable holding the full path to the fixed file.

```
>>> import numpy
```

The import statement imports any installed module, just as if it were in the standard library. Here we import the very useful `numpy` module, which is a prerequisite for SpacePy and useful in its own right.

```
>>> esp_fluxes = numpy.loadtxt(fname, skiprows=14, usecols=[1])
```

`loadtxt()` makes it easy to load data from a file into a numpy `ndarray`, a very useful data container. `skiprows` skips the header information, and specifying only column 1 (first column is column 0) with `usecols` will only load the fluxes for 1.8-3.5MeV. We only load the fluxes at this point because they can be represented as floats, which numpy arrays store very efficiently.

```
>>> import datetime
```

The `datetime` module provides Python objects which can manipulate dates and times and have some understanding of the meanings of dates, making for easy comparisons between dates, date arithmetic, and other useful features.

```
>>> convert = lambda x: datetime.datetime.strptime(x, '%Y-%m-%d')
```

This line sets up a converter to be used later. `strptime()` creates a `datetime` from a string, given a format definition (here specified as year-month-day). So:

```
>>> print(datetime.datetime.strptime('2010-01-02', '%Y-%m-%d'))
2010-01-02 00:00:00
```

lambda is a simple shortcut for a one-liner function; wherever `convert(x)` is used after the definition, it functions like `datetime.datetime.strptime(x, '%Y-%m-%d')`. This makes it easier to parse a date string without specifying the format all the time:

```
>>> print(convert('2010-01-02'))
```

This converter can be used with `loadtxt()`:

```
>>> esp_times = numpy.loadtxt(fname, skiprows=14, usecols=[0],
...                           converters={0: convert}, dtype=numpy.object)
```

The `converters` option takes a Python dictionary. The default dtype is float, which cannot store datetimes; using `numpy.object` allows storage of any Python object.

Since it would be useful to be able to load the data without typing so many lines, create a file called `common.py` in the `code` directory with the following contents:

```
import datetime
import os.path

import numpy


datadir = os.path.join('..', 'data')

def load_esp():
    fname = os.path.join(datadir, 'jgra20797-sup-0003-ds01_FIXED.txt')
    esp_fluxes = numpy.loadtxt(fname, skiprows=14, usecols=[1])
    convert = lambda x: datetime.datetime.strptime(x, '%Y-%m-%d')
    esp_times = numpy.loadtxt(fname, skiprows=14, usecols=[0],
                              converters={0: convert}, dtype=numpy.object)
    return (esp_times, esp_fluxes)
```

All needed imports are at the top of the file, with one blank line between standard library imports and other imports and two blank lines after them. `datadir` is defined as a global variable, outside of the function (but notice that it is available to the `load_esp` function.)

The rest of the file defines a function which returns the dates and fluxes in a tuple. The next section shows how to use this function.

Solar Wind data and averaging

The top panel of figure 1 shows the ESP fluxes overplotted with the solar wind velocity. Fortunately, the *omni* module of SpacePy provides an interface to the hourly solar wind dataset, OMNI. *get_omni()* returns data for a particular set of times. In this case, we want hourly data, covering 1989 through 2010 (we'll cut it down to size later). *tickrange()* allows us to specify a start time, stop time, and time step.

```
>>> import spacepy.omni
>>> import spacepy.time
>>> times = spacepy.time.tickrange('1989-01-01', '2011-01-01',
...                                 datetime.timedelta(hours=1))
>>> d = spacepy.omni.get_omni(times)
>>> vsw = d['velo']
>>> vsw_times = d['UTC']
```

We'll also load the esp data:

```
>>> import common
>>> esp_times, esp_flux = common.load_esp()
```

Even though we have not installed `common.py`, the `import` statement finds it because it is in the current directory.

`load_esp` returns a tuple, which can be *unpacked* into separate variables.

Now we need to produce 27-day running averages of both the flux and the solar wind speed. Fortunately there are no gaps in the time series:

```
>>> import numpy
>>> d = numpy.diff(vsw_times)
>>> print(d.min())
1:00:00
>>> print(d.max())
1:00:00
>>> d = numpy.diff(esp_times)
>>> print(d.min())
1 day, 0:00:00
>>> print(d.max())
1 day, 0:00:00
```

`numpy.diff()` returns the difference between every element of an array and the previous element. `min()` and `max()` do exactly what they sound like. So this code confirms that every time in the vsw data is on a continuous one hour cadence, and the ESP data is on a continuous one day cadence.

```
>>> import scipy.stats
>>> esp_flux_av = numpy.empty(shape=esp_flux.shape, dtype=esp_flux.dtype)
>>> for i in range(len(esp_flux_av)):
...     esp_flux_av[i] = scipy.stats.nanmean(esp_flux[max(i - 13, 0):i + 14])
```

`numpy.empty()` creates an empty array, taking the `shape` and `dtype` from the `esp_flux` array. `empty` does not initialize the data in the array, so it is essentially random junk; use `zeros()` to create an array filled with zeros.

`len()` returns the length of an array, and `range()` then iterates over each number from 0 to length minus 1, i.e. the entire array. Each element is then set to a 27-day average: from 13 days before a day's measurement through 13 days after. (Python slices do not include the last element listed; they are half-open). Note that these slices can happily run off the end of the `esp_flux` array, but we use `max()` to ensure the first index does not go negative. (Negative indices have special meaning in Python.)

`nanmean()` takes the mean of a numpy array, but skips any elements with a value of "not a number" (nan), which is often used for fill. (This is our first exposure to the `scipy` module.)

For the solar wind averaging, the times need to cover the 24 * 13.5 = 324 hours previous, and 324 hours following (non-inclusive). There is also a more efficient way than using an explicit loop:

```
>>> vsw_av = numpy.fromiter((scipy.stats.nanmean(vsw[max(0, i - 324):i + 324])
...                          for i in range(len(vsw))),
...                          count=len(vsw), dtype=vsw.dtype)
```

`fromiter()` makes a numpy array from an iterator, which is like a list except that it holds information on generating each element in a sequence rather than creating the entire sequence. `count` provides numpy with the number of elements in the output (so it can make the entire array at once); `dtype` here is just copied from the input.

The type of iterator used here is a generator expression, closely related to a list comprehension. These are among the most powerful and most difficult to understand concepts in Python. An illustrative, although not useful, example:

```
>>> for i in (x + 1 for x in range(10)):
...     print(i)
```

Here `(x + 1 for x in range(10))` is a generator expression that creates an iterator, which will return the numbers 1 through 10. At no point is the complete list of all numbers constructed, saving memory.

In our calculation of `esp_flux_av`, we created an explicit loop in Python. The generator expression used to compute `vsw_av` has no explicit loop, and the actual looping is handled in (much faster) compiled C code.

Making Figure 1

To actually plot, we need access to the `pyplot` module:

```
>>> import matplotlib.pyplot as plt
>>> plt.ion()
```

This alternate form of the import statement shouldn't be overused (it can make code harder to read by masking the origin of functions), but is conventional for matplotlib.

`ion()` turns on interactive mode so plots appear and are updated as they're created.

```
>>> plt.semilogy(esp_times, 10 ** esp_flux_av, 'b')
>>> plt.draw()
>>> plt.draw()
```

`semilogy()` creates a semilog plot, log on the Y axis. The first two arguments are a list of X and Y values; after that there are many options to specify formatting (such as the color, used here.)

The ESP fluxes are stored as the log of the flux; `**` is the exponentiation operator so the (geometric!) average is plotted properly.

`draw()` draws the updated plot; sometimes it needs to be called repeatedly. Use it whenever you want the plot updated; it will not be included from here on.

```
>>> plt.xlabel('Year', weight='bold')
>>> plt.ylabel('Electron Flux\n1.8-3.5 MeV', color='blue', weight='bold')
>>> plt.ylim(1e-2, 10)
(0.01, 10)
```

`xlabel()` and `ylabel()` set the labels for the axes. Note the newline (`\n`) in the string for the Y label. `ylim()` sets the lower and upper limits for the Y axis; there is, of course, `xlim()` as well.

These are the simplest, although not most flexible, ways to work with plots. To produce the full Figure 1, we'll move out of interactive mode:

```
>>> plt.ioff()
>>> plt.show()
```

`ioff()` turns off interactive mode. Once interactive mode is off, `show()` displays the full plot, including controls for panning, zooming, etc. Until the plot is closed, nothing further can happen in the Python window.

```
>>> fig = plt.figure(figsize=[11, 8.5])
```

`figure()` creates a new `Figure`; the size specified here is US-letter paper, landscape orientation.

```
>>> ax = fig.add_subplot(111)
```

`add_subplot()` creates an `Axes` object, which can contain an actual plot. `111` here means that the figure will have 1 subplot and the new subplot should be in position (1, 1); more on this later.

```
>>> fluxline = ax.plot(esp_times, 10 ** esp_flux_av, 'b')
```

`plot()` puts the relevant data into the plot; again specifying a blue line. It returns a list of `Line2D` objects, which we save for later use.

```
>>> ax.set_yscale('log')
```

set_yscale() switches the Y axis between log and linear (set_xscale() for the X axis).

```
>>> ax.set_ylim(1e-2, 10)
>>> ax.set_xlabel('Year', weight='bold')
>>> ax.set_ylabel('Electron Flux\n1.8-3.5 MeV', color='b', weight='bold')
```

set_ylim() (and set_xlim()), set_xlabel(), and set_ylabel() function much as above, but operate on a particular Axes object.

```
>>> ax2 = ax.twinx()
```

twinx() establishes a second Y axis (two values twinned on one X axis) on the same plot.

```
>>> vswline = ax2.plot(vsw_times, vsw_av, 'r')
>>> ax2.set_ylim(300, 650)
>>> ax2.set_ylabel('Solar Wind Speed', color='r', rotation=270, weight='bold')
```

The resulting Axes object has all the methods that we've used before. Note rotation on set_ylabel() to make the text run top-to-bottom rather than bottom-to-top.

```
>>> ax.set_xlim(esp_times[0], esp_times[-1])
```

Since the solar wind data extends beyond the ESP data, this sets the X axis to match the ESP data. Note -1 to refer to the last element of the array.

```
>>> leg = ax.legend([fluxline[0], vswline[0]], ['Flux', 'Vsw'],
...                 loc='upper left', frameon=False)
```

legend(), as may be expected, creates a Legend on the axes. The first parameter is a list of the matplotlib objects to make a legend for; since the plotting commands return these, we can pass them back in. Each plotting command returns a *list*. In this case we just take the 0th element of each list since we know there's only one line from each plotting command. The second parameter is the text used to annotate each line.

```
>>> fluxtext, vswtext = leg.get_texts()
>>> fluxtext.set_color(fluxline[0].get_color())
>>> vswtext.set_color(vswline[0].get_color())
```

The default text color is black, so we use get_texts() to get the Text objects for the annotations. Again, we know there are two (we just created the legend). Then set_color() sets the color based on the the existing color for each line (get_color()).

To see the results:

```
>>> plt.show()
```

Close the window when done. Now we want to save the output:

```
>>> fig_fname = os.path.join('..', 'plots', 'fig1a.eps')
>>> fig.savefig(fig_fname)
```

savefig() saves the figure, in this case as an encapsulated PostScript file (to the plots directory).

Let's tweak a few things. For one, there's a lot of padding around the figure, which can make it difficult to properly scale for publication. The way around this is to specify a Bbox (bounding box), basically the lower left and upper right corners (in inches) to include in the saved figure. Getting this right tends to be a matter of trial and error. (get_tightbbox() is supposed to help with this, but it doesn't quite work yet.)

```
>>> import matplotlib.transforms
>>> bob = matplotlib.transforms.Bbox([[0.52, 0.35], [10.5, 7.95]])
>>> fig.savefig(fig_fname, bbox_inches=bob, pad_inches=0.0)
```

Better, but all the text is awfully small. Once the figure is fit in the paper it'll be really small. And the font isn't that great.

```
>>> import matplotlib
>>> matplotlib.rcParams['axes.unicode_minus'] = False
>>> matplotlib.rcParams['text.usetex']= True
>>> matplotlib.rcParams['font.family'] = 'serif'
>>> matplotlib.rcParams['font.size'] = 14
>>> bob = matplotlib.transforms.Bbox([[0.4, 0.35], [10.7, 7.95]])
>>> fig.savefig(fig_fname, bbox_inches=bob, pad_inches=0.0)
```

Now the font is bigger and it's rendered using TeX, which should match the body of the paper better (assuming the paper is in LaTeX). The larger font means tweaking the bounding box. `unicode_minus` fixes a problem where negative numbers on the axis don't render properly in TeX. Matplotlib has many more options for customization.

The end result is a nice figure that can be printed full-size, put in a PDF, or included directly in a paper.

Now we need the bottom half of Figure 1. From SIDC, download the "Monthly mean total sunspot number" (`monthssn.dat`). Put it in the `data` directory.

```
>>> import bisect
>>> import datetime
>>> monthfile = os.path.join(common.datadir, 'monthssn.dat')
>>> convert = lambda x: datetime.datetime.strptime(x, '%Y%m')
>>> ssn_data = numpy.genfromtxt(monthfile, skip_header=2400, usecols=[0, 2, 3],
...                             converters={0: convert}, dtype=numpy.object,
...                             skip_footer=24)
>>> idx = bisect.bisect_left(ssn_data[:, 0], datetime.datetime(1989, 1, 1))
>>> ssn_data = ssn_data[idx:]
>>> ssn_times = ssn_data[:, 0]
>>> ssn = numpy.asarray(ssn_data[:, 1], dtype=numpy.float64)
>>> smooth_ssn = numpy.asarray(ssn_data[:, 2], dtype=numpy.float64)
>>> ssn_times += datetime.timedelta(days=15)
```

Much of this should be familiar. `genfromtxt()` is a little more flexible than `loadtxt()`; here it allows the skipping of lines at the end as well as the beginning (skipping 200 years at the start, 2 at the end, where data are provisional.) Here we load both times and the sunspot numbers in the same command so that if any lines don't load, they will not wind up in any of the arrays.

`bisect` provides fast functions for searching in sorted data; `bisect_left()` is roughly a find-the-position-of function. Having found the position of the start of 1989, we then keep times from then on (specifying a start index without a stop index in Python means "from start to end of the list.") Note that, although `bisect` is meant to work on lists, it also works fine on numpy arrays; this is a common feature of Python known as duck typing.

We then use `asarray()` to convert the `ssn` and `smooth_ssn` columns to float arrays. Note the slice notation: `[:, 0]` means take all indices of the first dimension (line number) and only the 0th index of the second dimension (column in the line). Finally, we use `timedelta` to shift the date associated with a month from the beginning to roughly the middle of the month. Adding a scalar to an array does an element-wise addition.

```
>>> import matplotlib.figure
>>> fig = plt.figure(figsize=[11, 8.5],
...                  subplotpars=matplotlib.figure.SubplotParams(hspace=0.1))
>>> ax = fig.add_subplot(211)
```

When creating the figure this time, we use `SubplotParams` to choose a slightly smaller vertical spacing between adjacent subplots. Tweaking `SubplotParams` also provides an alternative to tweaking bounding boxes.

Then we create a subplot with the information that there will be 2 rows, 1 column, and this is the first subplot. Now everything acting on ax, above, can be repeated, although we skip setting the xlabel since only the bottom axis will be labeled.

```
>>> fluxline = ax.plot(esp_times, 10 ** esp_flux_av, 'b')
>>> ax.set_yscale('log')
>>> ax.set_ylim(1e-2, 10)
>>> ax.set_ylabel('Electron Flux\n1.8-3.5 MeV', color='b', weight='bold')
>>> ax2 = ax.twinx()
>>> vswline = ax2.plot(vsw_times, vsw_av, 'r')
>>> ax2.set_ylim(300, 650)
>>> ax2.set_ylabel('Solar Wind Speed', color='r', rotation=270, weight='bold')
>>> ax.set_xlim(esp_times[0], esp_times[-1])
>>> leg = ax.legend([fluxline[0], vswline[0]], ['Flux', 'Vsw'],
...                 loc='upper left', frameon=False)
>>> fluxtext, vswtext = leg.get_texts()
>>> fluxtext.set_color(fluxline[0].get_color())
>>> vswtext.set_color(vswline[0].get_color())
```

Then we move on to adding the solar wind:

```
>>> ax3 = fig.add_subplot(212, sharex=ax)
```

This adds another subplot, the second in the 2x1 array. Its x axis is shared with the existing `ax`. (This is poorly documented; see this example)

```
>>> plt.setp(ax.get_xticklabels(), visible=False)
>>> plt.setp(ax2.get_xticklabels(), visible=False)
```

`setp()` sets a property. `get_xticklabels()` returns all the tick labels (`Text`) for the x axis; `setp` then sets `visible` to `False` for all of them. This hides the labeling on the axis for the upper subfigure.

```
>>> ax3.set_xlabel('Year', weight='bold')
>>> ax3.set_ylabel('Sunspot Number', weight='bold')
>>> smoothline = ax3.plot(ssn_times, smooth_ssn, lw=2.0, color='k')
>>> ssnline = ax3.plot(ssn_times, ssn, color='k', linestyle='dotted')
```

There is nothing new here except for the specifications of `linewidth` and `linestyle`; see `plot()` for details. Note `k` as the abbreviation for black (to avoid confusion with blue.)

```
>>> leg2 = ax3.legend([ssnline[0], smoothline[0]],
...                   ['Sunspot Number', 'Smoothed SSN'],
...                   loc='upper right', frameon=False)
>>> ax3.set_ylim(0, 200)
>>> ax3.set_xlim(esp_times[0], esp_times[-1])
```

```
>>> fig_fname = os.path.join('..', 'plots', 'fig1.eps')
>>> fig.savefig(fig_fname, bbox_inches=bob, pad_inches=0.0)
```

All of this has been seen for the top half of figure 1.

Following is the complete code to reproduce Figure 1.

```
import bisect
import datetime
import os.path

import common
import matplotlib
import matplotlib.figure
import matplotlib.pyplot as plt
import matplotlib.transforms
import numpy
import scipy
import scipy.stats
import spacepy.omni
import spacepy.time


matplotlib.rcParams['axes.unicode_minus'] = False
matplotlib.rcParams['text.usetex']= True
matplotlib.rcParams['font.family'] = 'serif'
matplotlib.rcParams['font.size'] = 14
bob = matplotlib.transforms.Bbox([[0.4, 0.35], [10.7, 7.95]])

times = spacepy.time.tickrange('1989-01-01', '2011-01-01',
                               datetime.timedelta(hours=1))
d = spacepy.omni.get_omni(times)
vsw = d['velo']
vsw_times = d['UTC']
esp_times, esp_flux = common.load_esp()
esp_flux_av = numpy.empty(shape=esp_flux.shape, dtype=esp_flux.dtype)
for i in range(len(esp_flux_av)):
    esp_flux_av[i] = scipy.stats.nanmean(esp_flux[max(i - 13, 0):i + 14])
vsw_av = numpy.fromiter((scipy.stats.nanmean(vsw[max(0, i - 324):i + 324])
                         for i in range(len(vsw))),
                         count=len(vsw), dtype=vsw.dtype)
monthfile = os.path.join(common.datadir, 'monthssn.dat')
convert = lambda x: datetime.datetime.strptime(x, '%Y%m')
ssn_data = numpy.genfromtxt(monthfile, skip_header=2400, usecols=[0, 2, 3],
                            converters={0: convert}, dtype=numpy.object,
                            skip_footer=24)
idx = bisect.bisect_left(ssn_data[:, 0], datetime.datetime(1989, 1, 1))
ssn_data = ssn_data[idx:]
ssn_times = ssn_data[:, 0]
ssn = numpy.asarray(ssn_data[:, 1], dtype=numpy.float64)
smooth_ssn = numpy.asarray(ssn_data[:, 2], dtype=numpy.float64)
ssn_times += datetime.timedelta(days=15)

fig = plt.figure(figsize=[11, 8.5],
                 subplotpars=matplotlib.figure.SubplotParams(hspace=0.1))
ax = fig.add_subplot(211)
fluxline = ax.plot(esp_times, 10 ** esp_flux_av, 'b')
ax.set_yscale('log')
ax.set_ylim(1e-2, 10)
ax.set_ylabel('Electron Flux\n1.8-3.5 MeV', color='b', weight='bold')
ax2 = ax.twinx()
vswline = ax2.plot(vsw_times, vsw_av, 'r')
ax2.set_ylim(300, 650)
ax2.set_ylabel('Solar Wind Speed', color='r', rotation=270, weight='bold')
ax.set_xlim(esp_times[0], esp_times[-1])
leg = ax.legend([fluxline[0], vswline[0]], ['Flux', 'Vsw'],
                loc='upper left', frameon=False)
fluxtext, vswtext = leg.get_texts()
fluxtext.set_color(fluxline[0].get_color())
vswtext.set_color(vswline[0].get_color())

ax3 = fig.add_subplot(212, sharex=ax)
```

```
plt.setp(ax.get_xticklabels(), visible=False)
plt.setp(ax2.get_xticklabels(), visible=False)
ax3.set_xlabel('Year', weight='bold')
ax3.set_ylabel('Sunspot Number', weight='bold')
smoothline = ax3.plot(ssn_times, smooth_ssn, lw=2.0, color='k')
ssnline = ax3.plot(ssn_times, ssn, color='k', linestyle='dotted')
leg2 = ax3.legend([ssnline[0], smoothline[0]],
                  ['Sunspot Number', 'Smoothed SSN'],
                  loc='upper right', frameon=False)
ax3.set_ylim(0, 200)
ax3.set_xlim(esp_times[0], esp_times[-1])

fig_fname = os.path.join('..', 'plots', 'fig1.eps')
fig.savefig(fig_fname, bbox_inches=bob, pad_inches=0.0)
```

Appendix: Fixing the ESP data file

This appendix provides a detailed explanation of the script that fixes the ESP data file.

First set up a variable to hold the location of the data, as above:

```
>>> import os.path
>>> datadir = os.path.join('..', 'data')
```

Examining the data file, it is clear that something is odd: lines appear to have been broken inappropriately; for example, the data for 1989-10-12 are split across two lines. So the first task is to fix this file, first opening the original (broken) file and an output (fixed) file:

```
>>> in_name = os.path.join(datadir, 'jgra20797-sup-0003-ds01.txt')
>>> out_name = os.path.join(datadir, 'jgra20797-sup-0003-ds01_FIXED.txt')
>>> infile = open(in_name, 'r')
>>> outfile = open(out_name, 'w')
```

These lines `open()` the original file for reading (`r`), and a new file for writing (`w`). Note that opening a file for writing will destroy any existing contents.

The file happens to contain a mixture of carriage returns and proper newlines, so to begin all the carriage returns need to be rewritten as newlines:

```
>>> data = infile.read()
>>> infile.close()
>>> data = data.replace('\r', '\n')
>>> data = data.replace('\n\n', '\n')
```

`read()` reads *all* data from the file at once, so this is not recommended for large files. In this case it makes things easier. Once the data are read, `close()` the file. Calling the `replace()` method on `data` replaces all instances of the first parameter (`'\r'`) with the second (`'\n'`). `\r` is the special code indicating a carriage return; `\n`, a newline. For a literal backslash, use `\\`. Once the carriage returns have been replaced with newlines, a second round of replacement eliminates duplicates.

Now that the line endings have been cleaned up, it's time to rejoin the erroneously split lines. First copy over the 15 lines of header verbatim:

```
>>> data = data.split('\n')
>>> for i in range(15):
...     outfile.write(data.pop(0) + '\n')
```

`split()` splits a string into a list, with the split between elements happening wherever the provided parameter occurs. A simple example:

```
>>> foo = 'a.b.c'.split('.')
>>> print(foo)
['a', 'b', 'c']
```

The splitting character is not present in the output.

The advantage of a list is that it makes it easy to access individual elements: >>> print(foo[1]) b

The first element of a Python list is numbered zero.

`range()` returns a list of numbers, starting from 0, with the parameter specifying how many elements are in the list:

```
>>> print(range(5))
[0, 1, 2, 3, 4]
```

The last number is 4 (not 5 as might be expected), but there are 5 elements in the list.

The for executes the following indented statement once for every element in the `in` list:

```
>>> for i in ['a', 'b', 'c']:
...     print i
a
b
c
```

Indentation is significant in Python! Normally indents are four spaces and the tab key will do the job. (In the above example, you may need to hit enter twice after the print statement, the second to terminate the indentation.)

pop returns one element from a list, and deletes it from the list. Using `0` pops off the first element, and `write()` writes a string to a file. `+` can be used to concatenate two strings together. Since `split()` removed the newlines, they need to be readded.

So this little block of code splits the data into a list on newlines and, repeating fifteen times, takes the first element of that list and writes it, with a newline, to the output. Now `data` contains only the actual lines of data.

```
>>> oldline = None
>>> for line in data:
...     if line[0:2] in ['19', '20', '2']:
...         if not oldline is None:
...             outfile.write(oldline + '\n')
...         oldline = line
...     else:
...         oldline += line
>>> outfile.write(oldline + '\n')
>>> outfile.close()
```

`None` is a special Python value specifically indicating nothing; it's used here to mark the first time around the loop.

`line[0:2]` gets the first two characters in the string *line*, and the `in` operator compares the resulting string to see if it is present in the following list. This will return `True` if the line begins with `19` or `20`. The if statement executes the following indented block if the condition is True. So, if this is True, the previous line probably ended properly and it can be written out. First there is an additional check that this isn't the first time around the loop, and then the *previous* line (which we know ended cleanly) is written out. The currently-read line then becomes the new "previous" line.

The `2` is a special case: if the line is less than two characters long, `line[0:2]` will return the entire line, and it so happens that these cases always correspond to the previous line being whole.

If this test fails, everything under `else` is executed. Here the assumption is that the previous line didn't end cleanly and the current line is actually a continuation of it, so the current line is appended to the previous. `a += b` is a shortcut for `a = a + b`.

Once the loop terminates, the last line is written out, and the file closed.

Publication List

The following publications have been prepared using SpacePy. If you have published a paper using SpacePy, contact the SpacePy team to be added to this list. Please also provide a citation or acknowledgment, as appropriate, in your paper.

Papers using SpacePy

Peer-reviewed papers

- Niehof, J. T., S. K. Morley, and R. H. W. Friedel (2012), Association of cusp energetic ions with geomagnetic storms and substorms, Ann. Geophys., 30 (12), 1633-1643, doi:10.5194/angeo-30-1633-2012.
- Turner, D. L., V. Angelopoulos, Y. Shprits, A. Kellerman, P. Cruce and D. Larson (2012), Radial distributions of equatorial phase space density for outer radiation belt electrons, Geophys. Res. Lett., 39, L09101, doi:10.1029/2012GL051722.
- Welling, D. T. and A. J. Ridley (2010), Exploring sources of magnetospheric plasma using multispecies MHD, Journal of Geophysical Research, 115, 4201, doi:10.1029/2009JA014596.
- Morley, S. K., R. H. W. Friedel, E. L. Spanswick, G. D. Reeves, J. T. Steinberg, J. Koller, T. Cayton and E. Noveroske (2010), Dropouts of the outer electron radiation belt in response to solar wind stream interfaces: Global Positioning System observations, Proceedings of the Royal Society A, doi:10.1098/rspa.2010.0078.

Other publications and presentations

- Niehof, J. T. and S. K. Morley (2012), Determining the significance of associations between two series of discrete events: bootstrap methods, Tech Report LA-14453, Los Alamos National Laboratory, Los Alamos, NM, doi:10.2172/1035497.

Papers about SpacePy

Peer-reviewed papers

- Morley, S. K., D. T. Welling, J. Koller, B. A. Larsen, M. G. Henderson and J. Niehof (2010), SpacePy - A Python-based library of tools for the space sciences, Proceedings of the 9th Python in Science Conference (SciPy 2010), presented in Austin, TX, June 30 - July 1, 2010 pdf. full proceedings

Other publications and presentations

- Niehof, J. T., M. G. Henderson, J. Koller, B. A. Larsen, S. Morley, D. T. Welling, Y. Yu (2012), Space Science with the SpacePy Toolkit, Abstract IN53C-1746 presented at 2012 Fall Meeting, AGU, San Francisco, Calif., 3-7 Dec. (pdf)
- The SpacePy Developer Team (2010), SpacePy - Python-Based of Tools for the Space Science Community, A Tri-Fold pdf.
- Morley, S. K., D. T. Welling, J. Koller, B. A. Larsen, M. G. Henderson (2010), SpacePy - Data Analysis and Visualization Tools for the Space Sciences, presented at GEM 2010 Summer Workshop, Snowmass, CO, June 20-25. (pdf)

Writing Pythonic Code

Code is often described as "Pythonic" or "not Pythonic" (with the implication that "Pythonic" is better.) The description is often applied to refer to code that reflects best practices which have emerged from the Python community and have become almost second nature to experienced programmers.

Reading lots of Python code (particularly from well-respected long maintained community projects) is the best way to develop this sense, but some principles are described here.

Good coding practice

Familiarity with modern coding practices that apply across most languages is a good start:

- Compact but descriptive names for variables, functions, etc.
- Succinct comments where necessary
- Encapsulation of data and abstraction through functions and classes
- Use of existing libraries rather than reimplementing

Using language features

Where Python or its standard library provides a means of accomplishing a task, it is generally preferred to use that means rather than reimplementing the wheel. The canonical example is using list comprehensions rather than for loops to transform a list:

```
>>> newlist = [i + 1 for i in oldlist]
```

not:

```
>>> newlist = []
>>> for i in range(len(oldlist)):
...     newlist.append(oldlist[i] + 1)
```

Note there are several non-Pythonic ways to perform this task.

For those not familiar with the features of the lanaguage and the standard library, this does represent a barrier to entry. However once that knowledge is built, using the features of the language makes one's intention much more clear. It often also results in shorter code that is easier to comprehend.

See several examples in SpacePy Python Programming Tips.

Idiom and communication

Because "code is more often read than written," anything that improves clarity is beneficial. A list comprehension and a for loop may have the same result, but the use of a list comprehension immediately makes it apparent to the reader that the code is intended to create a new list based on some element-by-element translation of the input list. It is a pattern with a common solution, and sticking to the common solution helps make the pattern apparent so the reader of the code understands the underlying problem.

Generally this choice of the common way is referred to as "idiomatic Python." This can be expanded to conventions such as the use of "self" as the first argument in instance methods, even though such choice is generally free.

Further Reading

A web search for “pythonic” will give a wealth of opinions. These references are a good starting point.

- Python Style Guide (PEP 8)
- Zen of Python (PEP 20)
- What is Pythonic?
- Examples of idiomatic and nonidiomatic Python
- Idomatic Python from Wikibooks

Release 0.2.1

Doc generation date October 02, 2019

SpacePy Scripts

Some scripts using SpacePy are included in the `scripts` directory of the source distribution. At the moment they are not installed by the installer.

istp_checks.py

Checks for various ISTP compliance issues in a file and prints any issues found. This script is supplemental to the checker included with the ISTP skeleton editor; it primarily checks for errors that the skeleton editor does not.

Badly noncompliant files generally result in the error “Test x did not complete”. This means the test crashed due to some failure in the assumptions about the CDF structure. Please run the individual test to get a traceback and open an issue.

This is just a thin wrapper to *`spacepy.pycdf.istp.FileChecks.all()`*; that documentation (plus other methods in the class) describes the actual checks.

`cdffile`
Name of the CDF file to check (required)

CHAPTER

TWO

SPACEPY CODE

ae9ap9 - Handle AE9/AP9 data files

Module for reading and dealing with AE9AP9 data files.

See https://www.vdl.afrl.af.mil/programs/ae9ap9/ to download the model. This is not a AE9AP9 runner.

Authors: Brian Larsen, Steve Morley Institution: Los Alamos National Laboratory Contact: balarsen@lanl.gov

This module provides a convenient class for handling data from AE9/AP9 (and legacy models provided by the software).

Class

Ae9Data(*args, **kwargs)	Dictionary-like container for AE9/AP9/SPM data, derived from SpacePy's datamodel

spacepy.ae9ap9.Ae9Data

class `spacepy.ae9ap9.`**`Ae9Data`**(**args*, ***kwargs*)

Dictionary-like container for AE9/AP9/SPM data, derived from SpacePy's datamodel

To inspect the variables within this class, use the tree method. To export the data to a CDF, HDF5 or JSON-headed ASCII file use the relevant "to" method (toCDF, toHDF5, toJSONheadedASCII).

getLm([alpha, model])	Calculate McIlwain L for the imported AE9/AP9 run and add to object
plotOrbit([timerange, coord_sys, landscape, ...])	Plot X-Y and X-Z projections of satellite orbit in requested coordinate system
plotSummary([timerange, coord_sys, ...])	Generate summary plot of AE9/AP9/SPM data loaded
plotSpectrogram([ecol, pvars])	Plot a spectrogram of the flux along the requested orbit, as a function of Lm an
setUnits([per])	Set units of energy and flux/fluence

`getLm`(*alpha=[90], model='T89'*)

Calculate McIlwain L for the imported AE9/AP9 run and add to object

`plotOrbit`(*timerange=None*, *coord_sys=None*, *landscape=True*, *fig_target=None*)

Plot X-Y and X-Z projections of satellite orbit in requested coordinate system

`plotSummary`(*timerange=None*, *coord_sys=None*, *fig_target=None*, *spec=False*, *orbit_params=(False*, *True)*, ***kwargs*)

Generate summary plot of AE9/AP9/SPM data loaded spec : if True, plot spectrogram instead of flux/fluence lineplot, requires 'ecol' keyword

plotSpectrogram(*ecol=0*, *pvars=None*, ***kwargs*)
Plot a spectrogram of the flux along the requested orbit, as a function of Lm and time

Other Parameters zlim : list

2-element list with upper and lower bounds for color scale

colorbar_label : string

text to appear next to colorbar (default is 'Flux' plus the units)

ylabel : string

text to label y-axis (default is 'Lm' plus the field model name)

title : string

text to appear above spectrogram (default is climatology model name, data type and energy)

pvars : list

list of plotting variable names in order [Epoch-like (X axis), Flux-like (Z axis), Energy (Index var for Flux-like)]

ylim : list

2-element list with upper and lower bounds for y axis

setUnits(*per=None*)
Set units of energy and flux/fluence

If keyword 'per' is set to None, this method reports the units currently set. To set energy in MeV and flux/fluence in 'per MeV', set 'per=MeV'. Valid options are 'eV', 'keV', 'Mev' and 'GeV'.

Other Parameters per : string (optional)

Energy units for both energy and flux/fluence

Though the class is derived from SpacePy's SpaceData, the class also provides several methods targeted at the AE9/AP9 output. Additional functions for working with the data are provided.

Functions

readFile(fname[, comments])	read a model generated file into a datamodel.SpaceData object
parseHeader(fname)	given an AE9AP9 output test file parse the header and return the information in a

spacepy.ae9ap9.readFile

spacepy.ae9ap9.**readFile**(*fname*, *comments='#'*)
read a model generated file into a datamodel.SpaceData object

Parameters fname : str

filename of the file

Returns out : *SpaceData*

Data contained in the file

Other Parameters comments : str (optional)

String that is the comments in the data file

Examples

```
>>> from spacepy import ae9ap9
>>> ae9ap9.readFile('ephem_sat.dat').tree(verbose=1)
+
|____Epoch (spacepy.datamodel.dmarray (121,))
|____Coords (spacepy.datamodel.dmarray (121, 3))
|____MJD (spacepy.datamodel.dmarray (121,))
|____posComp (spacepy.datamodel.dmarray (3,))
```

spacepy.ae9ap9.parseHeader

`spacepy.ae9ap9.`**`parseHeader`**(*fname*)

given an AE9AP9 output test file parse the header and return the information in a dictionary

Parameters fname : str

filename of the file

Returns out : dict

Dictionary of the header information in the file

coordinates - module for coordinate transforms

Implementation of Coords class functions for coordinate transformations

Authors: Josef Koller and Steven Morley Institution: Los ALamos National Laboratory Contact: smorley@lanl.gov

Copyright 2010 Los Alamos National Security, LLC.

Coords(data, dtype, carsph, [units, ticks])	A class holding spatial coordinates in Cartesian/spherical

spacepy.coordinates.Coords

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

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

Coordinate transforms are based on the IRBEM library; its manual may prove useful. For a good reference on heliospheric and magnetospheric coordinate systems, see Franz & Harper, “Heliospheric Coordinate Systems”, Planet. Space Sci., 50, pp 217-233, 2002 (https://doi.org/10.1016/S0032-0633(01)00119-2).

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

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

dtype : string

coordinate system; possible values are:

- **GDZ** (Geodetic; WGS84),
- **GEO** (Geographic Coordinate System),

- **GSM** (Geocentric Solar Magnetospheric),
- **GSE** (Geocentric Solar Ecliptic),
- **SM** (Solar Magnetic),
- **GEI** (Geocentric Equatorial Inertial; True-of-Date),
- **MAG** (Geomagnetic Coordinate System),
- **SPH** (Spherical Coordinate System),
- **RLL** (Radius, Latitude, Longitude; Geodetic)

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
```

append(other)	Append another Coords instance to the current one
convert(returntype, returncarsph)	Create a new Coords instance with new coordinate types

append(*other*)

Append another Coords instance to the current one

Parameters other : Coords instance

Coords instance to append

convert(*returntype*, *returncarsph*)

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']
```

datamanager - easy access to and manipulation of data

The datamanager classes and functions are useful for locating the correct data file for a particular day and manipulating data and subsets in a generic way.

Authors: Jon Niehof

Institution: University of New Hampshire

Contact: Jonathan.Niehof@unh.edu

About datamanager

Examples

Examples go here

Classes

DataManager(directories, file_fmt[, ...])	THIS CLASS IS NOT YET COMPLETE, doesn't do much useful.

Functions

apply_index(data, idx)	Apply an array of indices to data.
array_interleave(array1, array2, idx)	Create an array containing all elements of both array1 and array2
axis_index(shape[, axis])	Returns array of indices along axis, for all other axes
flatten_idx(idx[, axis])	Convert multidimensional index into index on flattened array.
insert_fill(times, data[, fillval, tol, ...])	Populate gaps in data with fill.
rev_index(idx[, axis])	From an index, return an index that reverses the action of that index
values_to_steps(array[, axis])	Transform values along an axis to their order in a unique sequence.

class `spacepy.datamanager.`**`DataManager`**(*directories*, *file_fmt*, *descend=False*, *period=None*)

THIS CLASS IS NOT YET COMPLETE, doesn't do much useful.

Will have to do something that allows the config file to specify regex and other things, and then just the directory to be changed (since regex, etc.

Parameters **directories** : list

A list of directories that might contain the data

file_fmt : string

Regular expression that matches the files desired. Will also recognize strftime parameters %w %d %m %y %Y %H %M %s %j %U %W, all zero-pad. https://docs.python.org/2/library/datetime.html#strftime-strptime-behavior Can have subdirectory reference, but separator should be unix-style, with no leading slash.

period : string

Size of file; can be a number followed by one of d, m, y, H, M, s. Anything else is assumed to be "irregular" and files treated as if there are neither gaps nor overlaps in the sequence. If not specified, will be assumed to match one count of the smallest unit in the format string.

Examples

=======

`files_matching`(*dt=None*)

Return all the files matching this file format

Parameters **dt** : datetime

Optional; if specified, match only files for this date.

Returns **out** : generator

Iterates over every file matching the format specified at creation. Note this is specified in native path format!

`get_filename`(*dt*)

Returns the filename corresponding to a particular point in time

`spacepy.datamanager.`**`apply_index`**(*data*, *idx*)

Apply an array of indices to data.

Most useful in dealing with the output from `numpy.argsort()`, and best explained by the example.

Parameters **data** : array

Input data, at least two dimensional. The 0th dimension is treated as a "time" or "record" dimension.

idx : sequence

2D index to apply to the import data. The 0th dimension must be the same size as `data`'s 0th dimension. Dimension 1 must be the same size as one other dimension in data (the first match found is used); this is referred to as the "index dimension."

Returns **data** : sequence

View of `data`, with index applied. For each index of the 0th dimension, the values along the index dimension are obtained by applying the value of `idx` at the same index in the 0th dimension. This is repeated across any other dimensions in `data`.

Warning: No guarantee is made whether the returned data is a copy of the input data. Modifying values in the input may change the values of the input. Call `copy()` if a copy is required.

Raises ValueError : if can't match the shape of data and indices

Examples

Assume `flux` is a 3D array of fluxes, with a value for each of time, pitch angle, and energy. Assume energy is not necessarily constant in time, nor is ordered in the energy dimension. If `energy` is a 2D array of the energies as a function of energy step for each time, then the following will sort the flux at each time and pitch angle in energy order.

```
>>> idx = numpy.argsort(energy, axis=1)
>>> flux_sorted = spacepy.datamanager.apply_index(flux, idx)
```

`spacepy.datamanager.`**`array_interleave`**`(`*array1*, *array2*, *idx*`)`

Create an array containing all elements of both array1 and array2

`idx` is an index on the output array which indicates which elements will be populated from `array1`, i.e., `out[idx] == array1` (in order.) The other elements of `out` will be filled, in order, from `array2`.

Parameters array1 : array

Input data.

array2 : array

Input data. Must have same number of dimensions as `array1`, and all dimensions except the zeroth must also have the same length.

idx : array

A 1D array of indices on the zeroth dimension of the output array. Must have the same length as the zeroth dimension of `array1`.

Returns out : array

All elements from `array1` and `array2`, interleaved according to `idx`.

Examples

```
>>> import numpy
>>> import spacepy.datamanager
>>> a = numpy.array([10, 20, 30])
>>> b = numpy.array([1, 2])
>>> idx = numpy.array([1, 2, 4])
>>> spacepy.datamanager.array_interleave(a, b, idx)
array([ 1, 10, 20,  2, 30])
```

`spacepy.datamanager.`**`axis_index`**`(`*shape*, *axis=-1*`)`

Returns array of indices along axis, for all other axes

Parameters shape : tuple

Shape of the output array

Returns idx : array

An array of indices. The value of each element is that element's index along `axis`.

Other Parameters **axis** : int

Axis along which to return indices, defaults to the last axis.

See also:

numpy.mgrid This function is a special case

Examples

For a shape of `(i, j, k, l)` and `axis` = -1, `idx[i, j, k, :] = range(l)` for all `i`, `j`, `k`.

Similarly, for the same shape and `axis = 1`, `idx[i, :, k, l] = range(j)` for all `i`, `k`, `l`.

```
>>> import numpy
>>> import spacepy.datamanager
>>> spacepy.datamanager.axis_index((5, 3))
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
>>> spacepy.datamanager.axis_index((5, 3), 0)
    array([[0, 0, 0],
           [1, 1, 1],
           [2, 2, 2],
           [3, 3, 3],
           [4, 4, 4]])
```

`spacepy.datamanager.`**`flatten_idx`**(*idx*, *axis=-1*)

Convert multidimensional index into index on flattened array.

Convert a multidimensional index, that is values along a particular axis, so that it can derefence the flattened array properly. Note this is not the same as `ravel_multi_index()`.

Parameters **idx** : array

Input index, i.e. a list of elements along a particular axis, in the style of `argsort()`.

Returns **flat** : array

A 1D array of indices suitable for indexing the flat version of the array

Other Parameters **axis** : int

Axis along which `idx` operates, defaults to the last axis.

See also:

apply_index

Examples

```
>>> import numpy
>>> import spacepy.datamanager
>>> data = numpy.array([[3, 1, 2], [3, 2, 1]])
>>> idx = numpy.argsort(data, -1)
>>> idx_flat = spacepy.datamanager.flatten_idx(idx)
>>> data.ravel() #flat array
array([3, 1, 2, 3, 2, 1])
```

```
>>> idx_flat #indices into the flat array
array([1, 2, 0, 5, 4, 3])
>>> data.ravel()[idx_flat] #index applied to the flat array
array([1, 2, 3, 1, 2, 3])
```

`spacepy.datamanager.`**`insert_fill`**(*times*, *data*, *fillval=nan*, *tol=1.5*, *absolute=None*, *doTimes=True*)

Populate gaps in data with fill.

Continuous data are often treated differently from discontinuous data, e.g., matplotlib will draw lines connecting data points but break the line at fill. Often data will be irregularly sampled but also contain large gaps that are not explicitly marked as fill. This function adds a single record of explicit fill to each gap, defined as places where the spacing between input times is a certain multiple of the median spacing.

Parameters **times** : sequence

Values representing when the data were taken. Must be one-dimensional, i.e., each value must be scalar. Not modified

data : sequence

Input data.

Returns **times, data** : tuple of sequence

Copies of input times and data, fill added in gaps (`doTimes` True)

data : sequence

Copy of input data, with fill added in gaps (`doTimes` False)

Other Parameters **fillval :**

Fill value, same type as `data`. Default is `numpy.nan`. If scalar, will be repeated to match the shape of `data` (minus the time axis).

Note: The default value of `nan` will not produce good results with integer input.

tol : float

Tolerance. A single fill value is inserted between adjacent values where the spacing in `times` is strictly greater than `tol` times the median of the spacing across all `times`. The inserted time for fill is halfway between the time on each side. (Default 1.5)

absolute :

An absolute value for maximum spacing, of a type that would result from a difference in `times`. If specified, `tol` is ignored and any gap strictly larger than `absolute` will have fill inserted.

doTimes : boolean

If True (default), will return a tuple of the times (with new values inserted for the fill records) and the data with new fill values. If False, will only return the data – useful for applying fill to multiple arrays of data on the same timebase.

Raises **ValueError** : if can't identify the time axis of data

Try using `numpy.rollaxis()` to put the time axis first in both `data` and `times`.

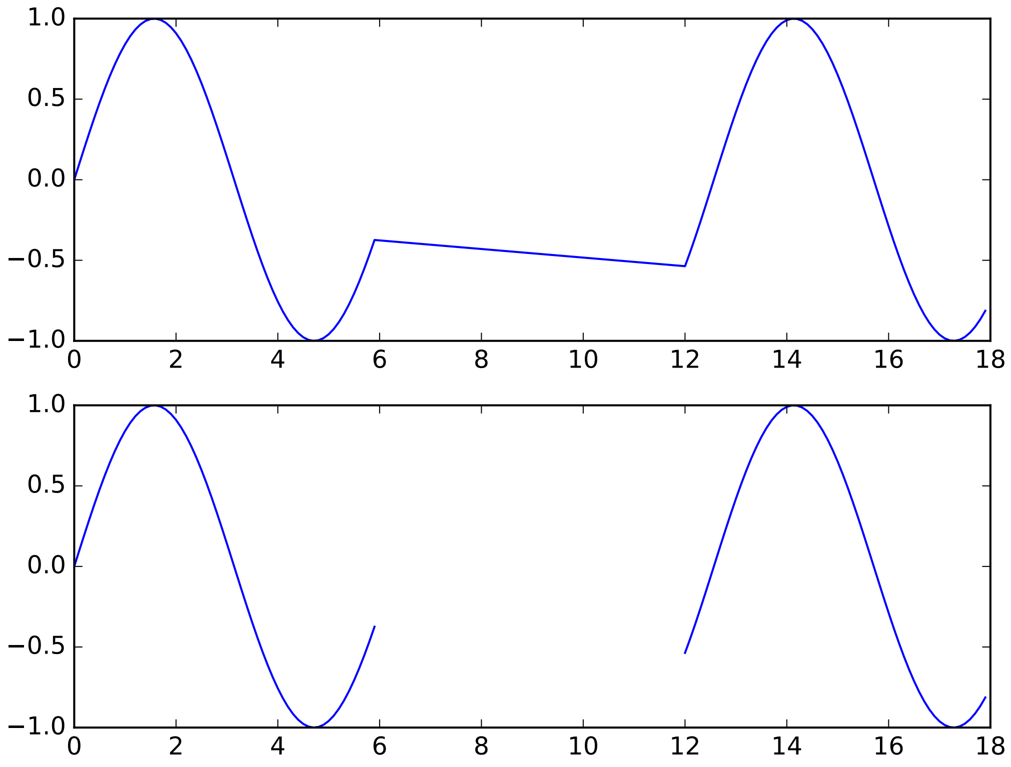
Examples

This example shows simple hourly data with a gap, populated with fill. Note that only a single fill value is inserted, to break the sequence of valid data rather than trying to match the existing cadence.

```
>>> import datetime
>>> import numpy
>>> import spacepy.datamanager
>>> t = [datetime.datetime(2012, 1, 1, 0),
         datetime.datetime(2012, 1, 1, 1),
         datetime.datetime(2012, 1, 1, 2),
         datetime.datetime(2012, 1, 1, 5),
         datetime.datetime(2012, 1, 1, 6)]
>>> temp = [30.0, 28, 27, 32, 35]
>>> filled_t, filled_temp = spacepy.datamanager.insert_fill(t, temp)
>>> filled_t
array([datetime.datetime(2012, 1, 1, 0, 0),
       datetime.datetime(2012, 1, 1, 1, 0),
       datetime.datetime(2012, 1, 1, 2, 0),
       datetime.datetime(2012, 1, 1, 3, 30),
       datetime.datetime(2012, 1, 1, 5, 0),
       datetime.datetime(2012, 1, 1, 6, 0)], dtype=object)
>>> filled_temp
array([ 30.,  28.,  27.,  nan,  32.,  35.])
```

This example plots “gappy” data with and without explicit fill values.

```
>>> import matplotlib.pyplot as plt
>>> import numpy
>>> import spacepy.datamanager
>>> x = numpy.append(numpy.arange(0, 6, 0.1), numpy.arange(12, 18, 0.1))
>>> y = numpy.sin(x)
>>> xf, yf = spacepy.datamanager.insert_fill(x, y)
>>> fig = plt.figure()
>>> ax0 = fig.add_subplot(211)
>>> ax0.plot(x, y)
>>> ax1 = fig.add_subplot(212)
>>> ax1.plot(xf, yf)
>>> plt.show()
```



`spacepy.datamanager.`**`rev_index`**(*idx*, *axis=-1*)

From an index, return an index that reverses the action of that index

Essentially, `a[idx][rev_index(idx)] == a`

Note: This becomes more complicated in multiple dimensions, due to the vagaries of applying a multidimensional index.

Parameters **idx** : array

Indices onto an array, often the output of `argsort()`.

Returns **rev_idx** : array

Indices that, when applied to an array after `idx`, will return the original array (before the application of `idx`).

Other Parameters **axis** : int

Axis along which to return indices, defaults to the last axis.

See also:

`apply_index`

Examples

```
>>> import numpy
>>> import spacepy.datamanager
>>> data = numpy.array([7, 2, 4, 6, 3])
>>> idx = numpy.argsort(data)
>>> data[idx] #sorted
array([2, 3, 4, 6, 7])
>>> data[idx][spacepy.datamanager.rev_index(idx)] #original
array([7, 2, 4, 6, 3])
```

`spacepy.datamanager.`**`values_to_steps`**(*array*, *axis=-1*)

Transform values along an axis to their order in a unique sequence.

Useful in, e.g., converting a list of energies to their steps.

Parameters **array** : array

Input data.

Returns **steps** : array

An array, the same size as `array`, with values along `axis` corresponding to the position of the value in `array` in a unique, sorted, set of the values in `array` along that axis. Differs from `argsort()` in that identical values will have identical step numbers in the output.

Other Parameters **axis** : int

Axis along which to find the steps.

Examples

```
>>> import numpy
>>> import spacepy.datamanager
>>> data = [[10., 12., 11., 9., 10., 12., 11., 9.],
            [10., 12., 11., 9., 14., 16., 15., 13.]]
>>> spacepy.datamanager.values_to_steps(data)
array([[1, 3, 2, 0, 1, 3, 2, 0],
       [1, 3, 2, 0, 5, 7, 6, 4]])
```

datamodel - easy to use general data model

The datamodel classes constitute 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

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

About datamodel

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.

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')
```

Functions are also available to directly load data and metadata into a SpacePy datamodel from NASA CDF as well as JSON-headed ASCII. Writers also exist to output a SpacePy datamodel directly to HDF5 or JSON-headed ASCII. See `datamodel.fromCDF()`, `datamodel.readJSONheadedASCII()`, `datamodel.toHDF5()`, and `datamodel.toJSONheadedASCII()` for more details.

Examples

Imagine representing some satellite data within 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, HDF5 or JSON-headed ASCII file. To visualize our datamodel, we can use tree method (which can be applied to any dictionary-like object using *dictree()*).

```
>>> mydata.tree(attrs=True)
```

```
+
:|____MissionName
:|____PI
|____Counts
     :|____Units
|____Epoch
     :|____units
|____OrbitNumber
     :|____StartsFrom
```

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).

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('inFile.cdf')
```

A quick guide to JSON-headed ASCII

In many cases it is preferred to have a human-readable ASCII file, rather than a binary file like CDF or HDF5. To make it easier to carry all the same metadata that is available in HDF5 or CDF we have developed an ASCII data storage format that encodes the metadata using JSON (JavaScript Object Notation). This notation supports two basic datatypes: key/value collections (like a SpaceData) and ordered lists (which can represent arrays). JSON is human-readable, but if large arrays are stored in metadata is quickly becomes difficult to read. For this reason we use JSON to encode the metadata (usually smaller datasets) and store the data in a standard flat-ASCII format. The metadata is provided as a header that describes the contents of the file.

To use JSON for storing only metadata associated with the data to be written to an ASCII file a minimal metadata standard must be implemented. We use the following attribute names: DIMENSION and START_COLUMN. We also recommend using the NASA ISTP metadata standard to assign attribute names. The biggest limitation of flat ASCII is that sensibly formatting datasets of more than 2-dimensions (i.e. ranks greater than 2) is not possible. For this reason if you have datasets of rank 3 or greater then we recommend using HDF5. If text is absolutely required then it is possible to encode multi-dimensional arrays in the JSON metadata, but this is not recommended.

This format is best understood by illustration. The following example builds a toy SpacePy datamodel and writes it to a JSON-headed ASCII file. The contents of the file are then shown.

```
>>> import spacepy.datamodel as dm
>>> data = dm.SpaceData()
>>> data.attrs['Global'] = 'A global attribute'
>>> data['Var1'] = dm.dmarray([1,2,3,4,5], attrs={'Local1': 'A local attribute'})
>>> data['Var2'] = dm.dmarray([[8,9],[9,1],[3,4],[8,9],[7,8]])
>>> data['MVar'] = dm.dmarray([7.8], attrs={'Note': 'Metadata'})
>>> dm.toJSONheadedASCII('outFile.txt', data, depend0='Var1', order=['Var1'])
#Note that not all field names are required, those not given will be listed
#alphabetically after those that are specified
```

The file looks like:

```
#{
#    "MVar": {
#        "Note": "Metadata",
#        "VALUES": [7.8]
#    },
#    "Global": "A global attribute",
#    "Var1": {
#        "Local1": "A local attribute",
#        "DIMENSION": [1],
#        "START_COLUMN": 0
#     },
#     "Var2": {
#         "DIMENSION": [2],
#         "START_COLUMN": 2
#     }
#}
1 8 9
2 9 1
3 3 4
4 8 9
5 7 8
```

Classes

SpaceData(*args, **kwargs)	Datamodel class extending dict by adding attributes.
dmarray	Container for data within a SpaceData object
DMWarning	Warnings class for datamodel, subclassed so it can be set to always

spacepy.datamodel.SpaceData

class spacepy.datamodel.**SpaceData**(**args*, ***kwargs*)

Datamodel class extending dict by adding attributes.

flatten()	Method to collapse datamodel to one level deep
tree(**kwargs)	Print the contents of the SpaceData object in a visual tree
SpaceData.toCDF	
SpaceData.toHDF5	
SpaceData.toJSONheadedASCII	

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
>>> a.tree()
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5
```

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

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

tree(***kwargs*)

Print the contents of the SpaceData object in a visual tree

Other Parameters **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

See also:

`toolbox.dictree`

Examples

```
>>> import spacepy.datamodel as dm
>>> import spacepy.toolbox as tb
>>> a = dm.SpaceData()
>>> a['1'] = dm.SpaceData(dog = 5)
>>> a['4'] = dm.SpaceData(cat = 'kitty')
>>> a['5'] = 4
>>> a.tree()
+
|____1
     |____dog
|____4
     |____cat
|____5
```

spacepy.datamodel.dmarray

class `spacepy.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'
```

`addAttribute`(name[, value])	Method to add an attribute to a dmarray

`addAttribute`(*name*, *value=None*)

Method to add an attribute to a dmarray equivalent to a = datamodel.dmarray([1,2,3]) a.Allowed_Attributes = a.Allowed_Attributes + ['blabla']

spacepy.datamodel.DMWarning

class `spacepy.datamodel.`**`DMWarning`**

Warnings class for datamodel, subclassed so it can be set to always

Functions

`convertKeysToStr`(SDobject)	
`createISTPattrs`(datatype[, ndims, vartype, ...])	Return set of unpopulated attributes for ISTP compliant variable
`dmcopy`(dobj)	Generic copy utility to return a copy of a (datamodel) object
`dmfilled`(shape[, fillval, dtype, order, attrs])	Return a new dmarray of given shape and type, filled with a specified va
`flatten`(dobj)	Collapse datamodel to one level deep
`fromCDF`(fname, **kwargs)	Create a SpacePy datamodel representation of a NASA CDF file
`fromHDF5`(fname, **kwargs)	Create a SpacePy datamodel representation of an HDF5 file or netCDF4
`fromRecArray`(recarr)	Takes a numpy recarray and returns each field as a dmarray in a SpaceD
`toCDF`(fname, SDobject, **kwargs)	Create a CDF file from a SpacePy datamodel representation
`toHDF5`(fname, SDobject, **kwargs)	Create an HDF5 file from a SpacePy datamodel representation

Table 2.11 – continued from previous page

toHTML(fname, SDobject[, attrs, varLinks, ...])	Create an HTML dump of the structure of a spacedata
toJSONheadedASCII(fname, insd[, metadata, ...])	Write JSON-headed ASCII file of data with metadata from SpaceData o
toRecArray(sdo)	Takes a SpaceData and creates a numpy recarray
unflatten(dobj[, marker])	Collapse datamodel to one level deep
readJSONMetadata(fname, **kwargs)	Read JSON metadata from an ASCII data file
readJSONheadedASCII(fname[, mdata, comment, ...])	read JSON-headed ASCII data files into a SpacePy datamodel
resample(data[, time, winsize, overlap, ...])	resample a SpaceData to a new time interval
writeJSONMetadata(fname, insd[, depend0, ...])	Scrape metadata from SpaceData object and make a JSON header

spacepy.datamodel.convertKeysToStr

spacepy.datamodel.**convertKeysToStr**(*SDobject*)

spacepy.datamodel.createISTPattrs

spacepy.datamodel.**createISTPattrs**(*datatype*, *ndims=1*, *vartype=None*, *units=None*, *NRV=False*)

Return set of unpopulated attributes for ISTP compliant variable

spacepy.datamodel.dmcopy

spacepy.datamodel.**dmcopy**(*dobj*)

Generic copy utility to return a copy of a (datamodel) object

Parameters dobj : object

object to return a copy of

Returns copy_obj: object (same type as input)

copy of input oibject

Examples

```
>>> import spacepy.datamodel as dm
>>> dat = dm.dmarray([2,3], attrs={'units': 'T'})
>>> dat1 = dm.dmcopy(dat)
>>> dat1.attrs['copy': True]
>>> dat is dat1
False
>>> dat1.attrs
{'copy': True, 'units': 'T'}
>>> dat.attrs
{'units': 'T'}
```

spacepy.datamodel.dmfilled

spacepy.datamodel.**dmfilled**(*shape*, *fillval=0*, *dtype=None*, *order='C'*, *attrs=None*)

Return a new dmarray of given shape and type, filled with a specified value (default=0).

See also:

numpy.ones

Examples

```
>>> import spacepy.datamodel as dm
>>> dm.dmfilled(5, attrs={'units': 'nT'})
dmarray([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> dm.dmfilled((5,), fillval=1, dtype=np.int)
dmarray([1, 1, 1, 1, 1])
```

```
>>> dm.dmfilled((2, 1), fillval=np.nan)
dmarray([[ nan],
       [ nan]])
```

```
>>> a = dm.dmfilled((2, 1), np.nan, attrs={'units': 'nT'})
>>> a
dmarray([[ nan],
       [ nan]])
>>> a.attrs
{'units': 'nT'}
```

spacepy.datamodel.flatten

spacepy.datamodel.**flatten**(*dobj*)

Collapse datamodel to one level deep

See also:

unflatten, *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
>>> a.tree()
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5
```

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

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

spacepy.datamodel.fromCDF

spacepy.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

See also:

spacepy.pycdf.CDF.copy

Examples

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

spacepy.datamodel.fromHDF5

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

Create a SpacePy datamodel representation of an HDF5 file or netCDF4 file which is HDF5 compliant

Parameters **file** : string

the name of the HDF5/netCDF4 file to be loaded into a datamodel

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

SpaceData with associated attributes and variables in dmarrays

Notes

Zero-sized datasets will break in h5py. This is kluged by returning a dmarray containing a None.

This function is expected to work with any HDF5-compliant files, including netCDF4 (not netCDF3) and MatLab save files from v7.3 or later, but some datatypes are not supported, e.g., non-string vlen datatypes, and will raise a warning.

Examples

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

spacepy.datamodel.fromRecArray

spacepy.datamodel.**fromRecArray**(*recarr*)
Takes a numpy recarray and returns each field as a dmarray in a SpaceData container

Parameters **recarr** : numpy record array

object to parse into SpaceData container

Returns sd: spacepy.datamodel.SpaceData

dict-like containing arrays of named records in recarr

Examples

```
>>> import numpy as np
>>> import spacepy.datamodel as dm
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', float), ('y', int)])
>>> print(x, x.dtype)
array([(1.0, 2), (3.0, 4)], dtype=[('x', '<f8'), ('y', '<i4')])
>>> sd = dm.fromRecArray(x)
>>> sd.tree(verbose=1)
+
|____x (spacepy.datamodel.dmarray (2,))
|____y (spacepy.datamodel.dmarray (2,))
```

spacepy.datamodel.toCDF

spacepy.datamodel.**toCDF**(*fname*, *SDobject*, ***kwargs*)
Create a CDF 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

Other Parameters **skeleton** : str (optional)

create new CDF from a skeleton file (default ‘’)

flatten : bool (optional)

flatten incoming datamodel - if SpaceData objects are nested (default False)

overwrite : bool (optional)

allow overwrite of an existing target file (default False)

autoNRV : bool (optional)

attempt automatic identification of non-record varying entries in CDF

backward : bool (optional)

create CDF in backward-compatible format (default is v3+ compatibility only)

TT2000 : bool (optional)

write variables beginning with 'Epoch' as datatype CDF_TT2000 (default is automatic selection of EPOCH or EPOCH16)

verbose : bool (optional)

verbosity flag

spacepy.datamodel.toHDF5

`spacepy.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

Other Parameters **overwrite** : bool (optional)

allow overwrite of an existing target file (default True)

mode : str (optional)

HDF5 file open mode (a, w, r) (default 'a')

compression : str (optional)

compress all the variables using this method (default None) (gzip, shuffle, fletcher32, szip, lzf)

compression_opts : str (optional)

options to the compression, see h5py documentation for more details

Examples

```
>>> import spacepy.datamodel as dm
>>> a = dm.SpaceData()
>>> a['data'] = dm.dmarray(range(100000), dtype=float)
>>> dm.toHDF5('test_gzip.h5', a, overwrite=True, compression='gzip')
>>> dm.toHDF5('test.h5', a, overwrite=True)
>>> # test_gzip.h5 was 118k, test.h5 was 785k
```

spacepy.datamodel.toHTML

spacepy.datamodel.**toHTML**(*fname*, *SDobject*, *attrs=()*, *varLinks=False*, *linkFormat=None*, *echo=False*, *tableTag='<table border="1">'*)

Create an HTML dump of the structure of a spacedata

Parameters fname : str

Filename to write to

SDobject : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Other Parameters overwrite : bool (optional)

allow overwrite of an existing target file (default True)

mode : str (optional)

HDF5 file open mode (a, w, r) (default 'a')

echo : bool

echo the html to the screen

varLinks : bool

make the variable name a link to a stub page

spacepy.datamodel.toJSONheadedASCII

spacepy.datamodel.**toJSONheadedASCII**(*fname*, *insd*, *metadata=None*, *depend0=None*, *order=None*, ***kwargs*)

Write JSON-headed ASCII file of data with metadata from SpaceData object

Parameters fname : str

Filename to write to (can also use a file-like object) None can be given in conjunction with the returnString keyword to skip writing output

insd : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Returns None

Other Parameters depend0 : str (optional)

variable name to use to indicate parameter on which other data depend (e.g. Time)

order : list (optional)

list of key names in order of start column in output JSON file

metadata: str or file-like (optional)

filename with JSON header to use (or file-like with JSON metadata)

delimiter: str

delimiter to use in ASCII output (default is whitespace), for tab, use ' '

Examples

```
>>> import spacepy.datamodel as dm
>>> data = dm.SpaceData()
>>> data.attrs['Global'] = 'A global attribute'
>>> data['Var1'] = dm.dmarray([1,2,3,4,5], attrs={'Local1': 'A local attribute'})
>>> data['Var2'] = dm.dmarray([[8,9],[9,1],[3,4],[8,9],[7,8]])
>>> data['MVar'] = dm.dmarray([7.8], attrs={'Note': 'Metadata'})
>>> dm.toJSONheadedASCII('outFile.txt', data, depend0='Var1', order=['Var1'])
#Note that not all field names are required, those not given will be listed
#alphabetically after those that are specified
```

spacepy.datamodel.toRecArray

spacepy.datamodel.**toRecArray**(*sdo*)

Takes a SpaceData and creates a numpy recarray

Parameters sdo : SpaceData

SpaceData to change to a numpy recarray

Returns recarr: numpy record array

numpy.recarray object with the same values (attributes are lost)

Examples

```
>>> import numpy as np
>>> import spacepy.datamodel as dm
>>> sd = dm.SpaceData()
>>> sd['x'] = dm.dmarray([1.0, 2.0])
>>> sd['y'] = dm.dmarray([2,4])
>>> sd.tree(verbose=1)
+
|____x (spacepy.datamodel.dmarray (2,))
|____y (spacepy.datamodel.dmarray (2,))
>>> ra = dm.toRecArray(sd)
>>> print(ra, ra.dtype)
[(2, 1.0) (4, 2.0)] (numpy.record, [('y', '<i8'), ('x', '<f8')])
```

spacepy.datamodel.unflatten

spacepy.datamodel.**unflatten**(*dobj*, *marker='<–'*)

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
>>> a.tree()
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5
```

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

```
>>> c = dm.unflatten(b)
>>> c.tree()
+
|____1
     |____dog
     |____pig
          |____fish
               |____a
               |____b
|____4
     |____cat
|____5
```

spacepy.datamodel.readJSONMetadata

`spacepy.datamodel.`**`readJSONMetadata`**(*fname*, ***kwargs*)

Read JSON metadata from an ASCII data file

Parameters **fname** : str

Filename to read metadata from

Returns mdata: spacepy.datamodel.SpaceData

SpaceData with the metadata from the file

Other Parameters **verbose** : bool (optional)

set verbose output so metadata tree prints on read (default False)

spacepy.datamodel.readJSONheadedASCII

`spacepy.datamodel.`**`readJSONheadedASCII`**(*fname*, *mdata=None*, *comment='#'*, *convert=False*, *restrict=None*)

read JSON-headed ASCII data files into a SpacePy datamodel

Parameters **fname** : str or list

Filename(s) to read data from

Returns mdata: spacepy.datamodel.SpaceData

SpaceData with the data and metadata from the file

Other Parameters **mdata** : spacepy.datamodel.SpaceData (optional)

supply metadata object, otherwise is read from fname (default None)

comment: str (optional)

comment string in file to be read; lines starting with comment are ignored (default '#')

convert: bool or dict-like (optional)

If True, uses common names to try conversion from string. If a dict- like then uses the functions specified as the dict values to convert each element of 'key' to a non-string

restrict: list of strings (optional)

If present, restrict the variables stored to only those on this list

spacepy.datamodel.resample

`spacepy.datamodel.`**`resample`**`(`*data*, *time=[]*, *winsize=0*, *overlap=0*, *st_time=None*, *outtimename='Epoch'*`)`

resample a SpaceData to a new time interval

Parameters **data** : SpaceData or dmarray

SpaceData with data to resample or dmarray with data to resample, variables can only be 1d or 2d, if time is specified only variables the same length as time are resampled, otherwise only variables with length equal to the longest length are resampled

time : array-like

dmarray of times the correspond to the data

winsize : datetime.timedelta

Time frame to average the data over

overlap : datetime.timedelta

Overlap in the moving average

st_time : datetime.datetime

Starting time for the resample, if not specified the time of the first data point is used (see spacepy.toolbox.windowMean)

Returns **ans** : SpaceData

Resampled data, included keys are in the input keys (with the data caveats above) and Epoch which contains the output time

Examples

```
>>> import datetime
>>> import spacepy.datamodel as dm
>>> a = dm.SpaceData()
>>> a.attrs['foo'] = 'bar'
```

```
    >>> a['a'] = dm.dmarray(range(10*2)).reshape(10,2)
    >>> a['b'] = dm.dmarray(range(10)) + 4
    >>> a['c'] = dm.dmarray(range(3)) + 10
    >>> times = [datetime.datetime(2010, 1, 1) + datetime.timedelta(hours=i) for i in range(10)]
    >>> out = dm.resample(a, times, winsize=datetime.timedelta(hours=2), overlap=datetime.timedelta(
    >>> out.tree(verbose=1, attrs=1)
    # +
    # :|____foo (str [3])
    # |____Epoch (spacepy.datamodel.dmarray (4,))
    # |____a (spacepy.datamodel.dmarray (4, 2))
    # :|____DEPEND_0 (str [5])
    #
    # Things to note:
    #    - attributes are preserved
    #    - the output variables have their DEPEND_0 changed to Epoch (or outtimename)
    #    - each dimension of a 2d array is resampled individually
```

spacepy.datamodel.writeJSONMetadata

spacepy.datamodel.**writeJSONMetadata**(*fname*, *insd*, *depend0=None*, *order=None*, *verbose=False*, *returnString=False*)

Scrape metadata from SpaceData object and make a JSON header

Parameters **fname** : str

Filename to write to (can also use a file-like object) None can be given in conjunction with the returnString keyword to skip writing output

insd : spacepy.datamodel.SpaceData

SpaceData with associated attributes and variables in dmarrays

Returns None (unless returnString keyword is True)

Other Parameters **depend0** : str (optional)

variable name to use to indicate parameter on which other data depend (e.g. Time)

order : list (optional)

list of key names in order of start column in output JSON file

verbose: bool (optional)

verbose output

returnString: bool (optional)

return JSON header as string instead of returning None

data assimilation - data assimilation module

Classes

ensemble([ensembles])	Ensemble-based data assimilation subroutines for the Radiation Belt Model

spacepy.data_assimilation.ensemble

class `spacepy.data_assimilation.`**`ensemble`**(*ensembles=50*)
Ensemble-based data assimilation subroutines for the Radiation Belt Model

`EnKF`(A, Psi, Inn, HAp)	analysis subroutine after code example in
`EnKF_oneobs`(A, Psi, Inn, HAp)	analysis subroutine for a single observations
`add_model_error`(model, A, PSDdata)	this routine will add a standard error to the ensemble states
`add_model_error_obs`(model, A, Lobs, y)	this routine will add a standard error to the ensemble states
`getHA`(model, Lobs, A)	compute HA provided L vector of observations
`getHAprime`(HA)	calculate ensemble perturbation of HA
`getHPH`(Lobs, Pfxx)	compute HPH
`getInnovation`(y, Psi, HA)	compute innovation ensemble D’
`getperturb`(model, y)	compute perturbations of observational vector

`EnKF`(*A*, *Psi*, *Inn*, *HAp*)
analysis subroutine after code example in Evensen 2003 this will take the prepared matrices and calculate the analysis most efficiently, A will be returned

Parameters **A :**

Psi :

Inn :

HAp :

Returns out :

`EnKF_oneobs`(*A*, *Psi*, *Inn*, *HAp*)
analysis subroutine for a single observations with the EnKF. This is a special case.

Parameters **A :**

Psi :

Inn :

HAp :

Returns out :

`add_model_error`(*model*, *A*, *PSDdata*)
this routine will add a standard error to the ensemble states

Parameters **model :**

A :

PSDdata :

Returns out :

`add_model_error_obs`(*model*, *A*, *Lobs*, *y*)
this routine will add a standard error to the ensemble states

Parameters **model :**

A :

Lobs :

y :

Returns out :

getHA(*model*, *Lobs*, *A*)
compute HA provided L vector of observations and ensemble matrix A

Parameters **model :**

Lobs :

A :

Returns out :

getHAprime(*HA*)
calculate ensemble perturbation of HA HA' = HA-HA_mean

Parameters **HA :**

Returns out :

getHPH(*Lobs*, *Pfxx*)
compute HPH

Parameters **Lobs**

Pfxx

Returns out

getInnovation(*y*, *Psi*, *HA*)
compute innovation ensemble D'

Parameters **y :**

Psi :

HA :

Returns out :

getperturb(*model*, *y*)
compute perturbations of observational vector

Parameters **model :**

y :

Returns out :

Functions

`average_window`(PSDdata, Lgrid)	combine observations on same L shell in
`getobs4window`(dd, Tnow)	get observations in time window [Tnow - Twindow, Tnow]
`output`(init, result)	write results to file and be done
`forecast`(Tnow+Twindow)	
`assimilate_JK`(dd)	this version is currently not working
`addmodelerror_old2`(dd, A, y, L)	this routine will add a standard error to the ensemble states
`addmodelerror_old`(dd, A, y, L)	this routine will add a standard error to the ensemble states

spacepy.data_assimilation.average_window

spacepy.data_assimilation.**average_window**(*PSDdata*, *Lgrid*)

combine observations on same L shell in

Parameters **model :**

PSDdata :

HAp :

Returns out :

spacepy.data_assimilation.getobs4window

spacepy.data_assimilation.**getobs4window**(*dd*, *Tnow*)

get observations in time window [Tnow - Twindow, Tnow] from all satellites lumped together into one y vector

Parameters **model :**

PSDdata :

HAp :

Returns out :

spacepy.data_assimilation.output

spacepy.data_assimilation.**output**(*init*, *result*)

write results to file and be done

Parameters **model :**

PSDdata :

HAp :

Returns out :

spacepy.data_assimilation.forecast

spacepy.data_assimilation.**forecast**(*Tnow+Twindow*)

spacepy.data_assimilation.assimilate_JK

spacepy.data_assimilation.**assimilate_JK**(*dd*)

this version is currently not working main function to assimilate all data provided in init

Parameters **model :**

PSDdata :

HAp :

Returns out :

spacepy.data_assimilation.addmodelerror_old2

`spacepy.data_assimilation.`**`addmodelerror_old2`**(*dd*, *A*, *y*, *L*)

this routine will add a standard error to the ensemble states

spacepy.data_assimilation.addmodelerror_old

`spacepy.data_assimilation.`**`addmodelerror_old`**(*dd*, *A*, *y*, *L*)

this routine will add a standard error to the ensemble states

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

Copyright 2010 Los Alamos National Security, LLC.

getDststar(ticks[, model, dbase])	Calculate the pressure-corrected Dst index, Dst*
getExpectedSWTemp(velo[, model, units])	Return the expected solar wind temperature based on the bulk velocity
getLmax(ticks[, model, dbase])	calculate a simple empirical model for Lmax - last closed drift-shell
getMPstandoff(ticks[, dbase, alpha])	Calculates the Shue et al.
getPlasmaPause(ticks[, model, LT, omnivals])	Plasmapause location model(s)
getSolarProtonSpectra([norm, gamma, E0, ...])	Returns a SpaceData with energy and fluence spectra of solar particle ev
getSolarRotation(ticks[, rtype, fp, reverse])	Calculates solar rotation number (Carrington or Bartels) for a given date
getVampolaOrder(L)	Empirical lookup of power for sin^n pitch angle model from Vampola (1
omniFromDirectionalFlux(fluxarr, alphas[, norm])	Calculate omnidirectional flux [(s cm^2 kev)^-1] from directional flux [(
vampolaPA(omniflux, **kwargs)	Pitch angle model of sin^n form

spacepy.empiricals.getDststar

`spacepy.empiricals.`**`getDststar`**(*ticks*, *model='OBrien'*, *dbase='QDhourly'*)

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)
```

spacepy.empiricals.getExpectedSWTemp

spacepy.empiricals.**getExpectedSWTemp**(*velo*, *model='XB15'*, *units='K'*)

Return the expected solar wind temperature based on the bulk velocity

The formulations used by this function are those given by, L87 – Lopez, R.E., J. Geophys. Res., 92, 11189-11194, 1987 BS06 – Borovsky, J.E. and J.T. Steinberg, Geophysical Monograph Series 167, 59-76, 2006 XB15 – Xu, F. and J.E. Borovsky, J. Geophys. Res., 120, 70-100, 2015

Parameters **velo** : array-like

Array like of solar wind bulk velocity values [km/s]

model : str [optional]

Name of model to use. Valid choices are L87, BS06 and XB15. Default is XB15

units : str [optional]

Units for output temperature, options are eV or K. Default is Kelvin [K]

Returns **Texp** : array-like

The expected solar wind temperature given the bulk velocity [K] or [eV]

spacepy.empiricals.getLmax

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

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

Uses the parametrized Lmax from: Koller and Morley (2010) 'Magnetopause shadowing effects for radiation belt models during high-speed solar wind streams' American Geophysical Union, Fall Meeting 2010, abstract #SM13A-1787

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

See also:

spacepy.LANLstar.LANLmax

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])
```

spacepy.empiricals.getMPstandoff

spacepy.empiricals.**getMPstandoff**(*ticks*, *dbase='QDhourly'*, *alpha=[]*)

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

Shue et al. (1997), A new functional form to study the solar wind control of the magnetopause size and shape, J. Geophys. Res., 102(A5), 9497–9511, doi:10.1029/97JA00196.

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]

alpha : list

Used as an optional return value to obtain the flaring angles. To use, assign an empty list and pass to this function through the keyword argument. The list will be modified in place, adding the flaring angles for each time step.

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.getMPstandoff(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.getMPstandoff(data)
array([ 9.96096838,  8.96790412])
```

spacepy.empiricals.getPlasmaPause

spacepy.empiricals.**getPlasmaPause**(*ticks*, *model='M2002'*, *LT='all'*, *omnivals=None*)

Plasmapause location model(s)

CA1992 – Carpenter, D. L., and R. R. Anderson, An ISEE/whistler model of equatorial electron density in the magnetosphere, J. Geophys. Res., 97, 1097, 1992. M2002 – Moldwin, M. B., L. Downward, H. K. Rassoul, R. Amin, and R. R. Anderson, A new model of the location of the plasmapause: CRRES results, J. Geophys. Res., 107(A11), 1339, doi:10.1029/2001JA009211, 2002. RT1970 – Rycroft, M. J., and J. O. Thomas, The magnetospheric plasmapause and the electron density trough at the alouette i orbit, Planetary and Space Science, 18(1), 65-80, 1970

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

omnivals : spacepy.datamodel.SpaceData, dict

dict-like containing UTC (datetimes) and Kp keys

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    ])
```

spacepy.empiricals.getSolarProtonSpectra

spacepy.empiricals.**getSolarProtonSpectra**(*norm=32000000.0*, *gamma=-0.96*, *E0=15.0*, *Emin=0.1*, *Emax=600*, *nsteps=100*)

Returns a SpaceData with energy and fluence spectra of solar particle events

The formulation follows that of: Ellison and Ramaty ApJ 298: 400-408, 1985 dJ/dE = K^{-gamma}exp(-E/E0)

and the defualt values are the 10/16/2003 SEP event of: Mewaldt, R. A., et al. (2005), J. Geophys. Res., 110, A09S18, doi:10.1029/2005JA011038.

Returns **data** : dm.SpaceData

SpaceData with the energy and fluence values

Other Parameters **norm** : float

Normilization factor for the intensity of the SEP event

gamma : float

Power law index

E0 : float

Expoential scaling factor

Emin : float

Minimum energy for fit

Emax : float

Maximum energy for fit

nsteps : int

The number of log spaced energy steps to return

spacepy.empiricals.getSolarRotation

spacepy.empiricals.**getSolarRotation**(*ticks*, *rtype='carrington'*, *fp=False*, *reverse=False*)

Calculates solar rotation number (Carrington or Bartels) for a given date/time

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

Returns **rnumber** : integer or array

Carrington (or Bartels) rotation number

spacepy.empiricals.getVampolaOrder

spacepy.empiricals.**getVampolaOrder**(*L*)

Empirical lookup of power for sin^n pitch angle model from Vampola (1996)

Vampola, A.L. Outer zone energetic electron environment update, Final Report of ESA/ESTEC/WMA/P.O. 151351, ESA-ESTEC, Noordwijk, The Netherlands, 1996.

Parameters **L** : arraylike or float

Returns **order** : array

coefficient for sin^n model corresponding to McIlwain L (computed for OP77?)

Examples

Apply Vampola pitch angle model at L=[4, 6.6]

```
>>> from spacepy.empiricals import vampolaPA, getVampolaOrder
>>> order = getVampolaOrder([4,6.6])
>>> order
array([ 3.095 ,  1.6402])
>>> vampolaPA([3000, 3000], alpha=[45, 90], order=order)
(array([[ 140.08798878,  192.33572182],
    [ 409.49143136,  339.57417256]]), [45, 90])
```

spacepy.empiricals.omniFromDirectionalFlux

spacepy.empiricals.**omniFromDirectionalFlux**(*fluxarr*, *alphas*, *norm=True*)

Calculate omnidirectional flux [(s cm^2 kev)^-1] from directional flux [(s sr cm^2 keV)^-1] array

J = 2.pi integ(j sin(a) da) If kwarg 'norm' is True (default), the omni flux is normalized by 4.pi to make it per steradian, in line with the PRBEM guidelines

Parameters **fluxarr** : arraylike

Array of directional flux values

alphas : arraylike

Array of pitch angles corresponding to fluxarr

Returns **omniflux** : float

Omnidirectional flux value

Examples

Roundtrip from omni flux, to directional flux (Vampola model), integrate to get back to omni flux.

```
>>> from spacepy.empiricals import vampolaPA, omniFromDirectionalFlux
>>> dir_flux, pa  = vampolaPA(3000, alpha=range(0,181,2), order=4)
>>> dir_flux[:10], pa[:10]
(array([  0.00000000e+00,   6.64032473e-04,   1.05986545e-02,
         5.34380898e-02,   1.67932162e-01,   4.06999226e-01,
         8.36427502e-01,   1.53325140e+00,   2.58383611e+00,
         4.08170975e+00]), [0, 2, 4, 6, 8, 10, 12, 14, 16, 18])
>>> omniFromDirectionalFlux(dir_flux, pa, norm=False)
3000.0000008112293
```

Calculate "spin-averaged" flux, giving answer per steradian

```
>>> omniFromDirectionalFlux(dir_flux, pa, norm=True)
238.73241470239859
```

spacepy.empiricals.vampolaPA

spacepy.empiricals.**vampolaPA**(*omniflux*, ***kwargs*)

Pitch angle model of sin^n form

Parameters **omniflux** : arraylike or float

omnidirectional number flux data

order : integer or float (optional)

order of sin^n functional form for distribution (default=2)

alphas : arraylike (optional)

pitch angles at which to evaluate the differential number flux (default is 5 to 90 degrees in 36 steps)

Returns **dnflux** : array

differential number flux corresponding to pitch angles alphas

alphas : array

pitch angles at which the differential number flux was evaluated

Notes

Directional number flux integrated over pitch angle from 0 to 90 degrees is a factor of 4*pi lower than omnidirectional number flux.

Examples

Omnidirectional number flux of [3000, 6000]

```
>>> from spacepy.empiricals import vampolaPA
>>> vampolaPA(3000, alpha=[45, 90])
(array([ 179.04931098,  358.09862196]), [45, 90])
>>> data, pas = vampolaPA([3000, 6000], alpha=[45, 90])
>>> pas
[45, 90]
>>> data
array([[ 179.04931098,  358.09862196],
   [ 358.09862196,  716.19724391]])
```

irbempy - Python interface to irbem/ONERA library

module wrapper for irbem_lib Reference for this library https://sourceforge.net/projects/irbem/ D. Boscher, S. Bourdarie, P. O'Brien, T. Guild, IRBEM library V4.3, 2004-2008

Most functions in this module use an options list to define the models used and the settings that define the quality level of the result. The options list is a 5-element list and is defined as follows.

Options

- 1st element: 0 - don't compute L* or phi ; 1 - compute L*; 2- compute phi
- **2nd element: 0 - initialize IGRF field once per year (year.5);** n - n is the frequency (in days) starting on January 1st of each year (i.e. if options(2nd element)=15 then IGRF will be updated on the following days of the year: 1, 15, 30, 45 ...)
- **3rd element: resolution to compute L* (0 to 9) where 0 is the recomended value to ensure a** good ratio precision/computation time (i.e. an error of ~2% at L=6). The higher the value the better will be the precision, the longer will be the computing time. Generally there is not much improvement for values larger than 4. Note that this parameter defines the integration step (theta) along the field line such as dtheta=(2pi)/(720*[options(3rd element)+1])
- **4th element: resolution to compute L* (0 to 9). The higher the value the better will be** the precision, the longer will be the computing time. It is recommended to use 0 (usually sufficient) unless L* is not computed on a LEO orbit. For LEO orbit higher values are recommended. Note that this parameter defines the integration step (phi) along the drift shell such as dphi=(2pi)/(25*[options(4th element)+1])
- **5th element: allows to select an internal magnetic field model (default is set to IGRF)**
  - 0 = IGRF
  - 1 = Eccentric tilted dipole

  - 2 = Jensen&Cain 1960
  - 3 = GSFC 12/66 updated to 1970
  - 4 = User-defined model (Default: Centred dipole + uniform [Dungey open model])
  - 5 = Centred dipole

The routines also require specification of the external magnetic field model. The default is the Tsyganenko 2001 storm-time model. The external model is always specified using the extMag keyword and the following options exist.

extMag

- ‘0’ = No external field model
- ‘MEAD’ = Mead and Fairfield
- ‘T87SHORT’ = Tsyganenko 1987 short (inner magnetosphere)
- ‘T87LONG’ = Tsyganenko 1987 long (valid in extended tail region)
- ‘T89’ = Tsyganenko 1989
- ‘OPQUIET’ = Olsen-Pfitzer static model for quiet conditions
- ‘OPDYN’ = Olsen-Pfitzer static model for active conditions
- ‘T96’ = Tsyganenko 1996
- ‘OSTA’ = Ostapenko and Maltsev
- ‘T01QUIET’ = Tsyganenko 2001 model for quiet conditions
- ‘T01STORM’ = Tsyganenko 2001 model for active conditions
- ‘T05’ = Tsyganenko and Sitnov 2005 model
- ‘ALEX’ = Alexeev model
- ‘TS07’ = Tsyganenko and Sitnov 2007 model

Authors

Josef Koller, Steve Morley

Copyright 2010 Los Alamos National Security, LLC.

This module provides a Python interface to the IRBEM library

Reference for this library https://sourceforge.net/projects/irbem/

D. Boscher, S. Bourdarie, P. O’Brien, T. Guild, IRBEM library V4.3, 2004-2008

Authors: Josef Koller, Steve Morley

Copyright 2010 Los Alamos National Security, LLC.

`get_AEP8`(energy, loci[, model, fluxtype, ...])	will return the flux from the AE8-AP8 model
`get_Bfield`(ticks, loci[, extMag, options, ...])	call get_bfield in irbem lib and return a dictionary with the B-field vector and
`get_Lm`(ticks, loci, alpha[, extMag, intMag, ...])	Return the MacIlwain L value for a given location, time and model
`get_Lstar`(ticks, loci[, alpha, extMag, ...])	This will call make_lstar1 or make_lstar_shell_splitting_1 from the irbem library
`find_Bmirror`(ticks, loci, alpha[, extMag, ...])	call find_mirror_point from irbem library and return a dictionary with values for

Table 2.16 – continued from previous page

find_magequator(ticks, loci[, extMag, ...])	call find_magequator from irbem library and return a dictionary with values for
car2sph(CARin)	coordinate transformation from cartesian to spherical
sph2car(SPHin)	coordinate transformation from spherical to cartesian
coord_trans(loci, returntype, returncarsph)	thin layer to call coor_trans1 from irbem lib
get_sysaxes(dtype, carsph)	will return the sysaxes according to the irbem library
get_dtype(sysaxes)	will return the coordinate system type as string
prep_irbem([ticks, loci, alpha, extMag, ...])	Prepare inputs for direct IRBEM-LIB calls.

spacepy.irbempy.get_AEP8

spacepy.irbempy.**get_AEP8**(*energy*, *loci*, *model='min'*, *fluxtype='diff'*, *particles='e'*)

will return the flux from the AE8-AP8 model

Parameters - **energy (float)** : center energy in MeV; if fluxtype=RANGE, then needs to be a list [Emin, Emax]

- **loci (Coords)** [a Coords instance with the location inside the magnetosphere] optional instead of a Coords instance, one can also provide a list with [BBo, L] combination
- model (str) : MIN or MAX for solar cycle dependence
- fluxtype (str) : DIFF, RANGE, INT are possible types
- particles (str): e or p or electrons or protons

Returns - **float** : flux from AE8/AP8 model

Examples

```
>>> spacepy.irbempy.get_aep8()
```

```
>>> import spacepy.time as spt
>>> import spacepy.coordinates as spc
>>> import spacepy.irbempy as ib
>>> t = spt.Ticktock(['2017-02-02T12:00:00'], 'ISO')
>>> y = spc.Coords([3,0,0], 'GEO', 'car')
>>> y.ticks = t
>>> energy = 1.0 #MeV
>>> ib.get_AEP8(energy, y, model='max')
1932209.4427359989
```

spacepy.irbempy.get_Bfield

spacepy.irbempy.**get_Bfield**(*ticks*, *loci*, *extMag='T01STORM'*, *options=[1, 0, 0, 0, 0]*, *omnivals=None*)

call get_bfield in irbem lib and return a dictionary with the B-field vector and strenght.

Parameters - **ticks (Ticktock class)** : containing time information

- loci (Coords class) : containing spatial information
- **extMag (string)** [optional; will choose the external magnetic field model] possible values ['0', 'MEAD', 'T87SHORT', 'T87LONG', 'T89', 'OPQUIET', 'OPDYN', 'T96', 'OSTA', 'T01QUIET', 'T01STORM', 'T05', 'ALEX', 'TS07']
- options (optional list or array of integers length=5) : explained in Lstar

- omni values as dictionary (optional) : if not provided, will use lookup table
- (see Lstar documentation for further explanation)

Returns - results (dictionary) : containing keys: Bvec, and Blocal

See also:

get_Lstar, *find_Bmirror*, *find_magequator*

Examples

```
>>> 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]])}
```

spacepy.irbempy.get_Lm

spacepy.irbempy.**get_Lm**(*ticks*, *loci*, *alpha*, *extMag='T01STORM'*, *intMag='IGRF'*, *IGRFset=0*, *omnivals=None*)

Return the MacIlwain L value for a given location, time and model

Parameters - ticks (Ticktock class) : containing time information

- loci (Coords class) : containing spatial information
- alpha (list or ndarray) : pitch angles in degrees
- **extMag (string)** [optional; will choose the external magnetic field model] possible values ['0', 'MEAD', 'T87SHORT', 'T87LONG', 'T89', 'OPQUIET', 'OPDYN', 'T96', 'OSTA', 'T01QUIET', 'T01STORM', 'T05', 'ALEX', 'TS07']
- **intMag (string)** [optional: select the internal field model] possible values ['IGRF','EDIP','JC','GSFC','DUN','CDIP'] For full details see get_Lstar
- omni values as dictionary (optional) : if not provided, will use lookup table

Returns - results (dictionary) : containing keys: Lm, Bmin, Blocal (or Bmirr), Xj, MLT

if pitch angles provided in "alpha" then drift shells are calculated and "Bmirr" is returned if not provided, then "Blocal" at spacecraft is returned. A negative value for Lm indicates the field line is closed but particles are lost to the atmosphere; the absolute value indicates the L value.

spacepy.irbempy.get_Lstar

spacepy.irbempy.**get_Lstar**(*ticks, loci, alpha=90, extMag='T01STORM', options=[1, 0, 0, 0, 0], omnivals=None*)

This will call make_lstar1 or make_lstar_shell_splitting_1 from the irbem library and will lookup omni values for given time if not provided (optional). If pitch angles are provided, drift shell splitting will be calculated and "Bmirr" will be returned. If they are not provided, then no drift shell splitting is calculated and "Blocal" is returned.

Parameters **- ticks (Ticktock class)** : containing time information

- loci (Coords class) : containing spatial information
- **alpha (list or ndarray)** [optional pitch angles in degrees (default is 90);] if provided will calculate shell splitting; max 25 values
- **extMag (string)** [optional; will choose the external magnetic field model] possible values [‘0’, ‘MEAD’, ‘T87SHORT’, ‘T87LONG’, ‘T89’, ‘OPQUIET’, ‘OPDYN’, ‘T96’, ‘OSTA’, ‘T01QUIET’, ‘T01STORM’, ‘T05’, ‘ALEX’, ‘TS07’]
- options (optional list or array of integers length=5) : explained below
- omni values as dictionary (optional) : if not provided, will use lookup table

Returns **- results (dictionary)** : containing keys: Lm, Lstar, Bmin, Blocal (or Bmirr), Xj, MLT

if pitch angles provided in “alpha” then drift shells are calculated and “Bmirr” is returned if not provided, then “Blocal” at spacecraft is returned. A negative value for Lm indicates the field line is closed but particles are lost to the atmosphere; the absolute value indicates the L value. A negative value for Lstar indicates the field line is closed but particles are lost to the atmosphere before completing a drift orbit; the absolute value indicates the drift shell.

Notes

External Field

- 0 : no external field
- MEAD : Mead & Fairfield [1975] (uses 0<=Kp<=9 - Valid for rGEO<=17. Re)
- T87SHORT: Tsyganenko short [1987] (uses 0<=Kp<=9 - Valid for rGEO<=30. Re)
- T87LONG : Tsyganenko long [1987] (uses 0<=Kp<=9 - Valid for rGEO<=70. Re)
- T89 : Tsyganenko [1989] (uses 0<=Kp<=9 - Valid for rGEO<=70. Re)
- OPQUIET : Olson & Pfitzer quiet [1977] (default - Valid for rGEO<=15. Re)
- **OPDYN** [Olson & Pfitzer dynamic [1988] (uses 5.<=dens<=50., 300.<=velo<=500.,] -100.<=Dst<=20. - Valid for rGEO<=60. Re)
- **T96** [Tsyganenko [1996] (uses -100.<=Dst (nT)<=20., 0.5<=Pdyn (nPa)<10.,] abs(ByIMF) (nT)<1=0., abs(BzIMF) (nT)<=10. - Valid for rGEO<=40. Re)
- **OSTA** [Ostapenko & Maltsev [1997] (uses dst,Pdyn,BzIMF, Kp)] T01QUIET: Tsyganenko [2002a,b] (uses -50.<Dst (nT)<20., 0.5<Pdyn (nPa)<=5., abs(ByIMF) (nT)<=5., abs(BzIMF) (nT)<=5., 0.<=G1<=10., 0.<=G2<=10. - Valid for xGSM>=-15. Re)
- **T01STORM: Tsyganenko, Singer & Kasper [2003] storm (uses Dst, Pdyn, ByIMF, BzIMF, G2, G3 -** there is no upper or lower limit for those inputs - Valid for xGSM>=-15. Re)
- **T05** [Tsyganenko & Sitnov [2005] storm (uses Dst, Pdyn, ByIMF, BzIMF,] W1, W2, W3, W4, W5, W6 - no upper or lower limit for inputs - Valid for xGSM>=-15. Re)

OMNI contents

- Kp: value of Kp as in OMNI2 files but has to be double instead of integer type
- Dst: Dst index (nT)
- dens: Solar Wind density (cm-3)

- velo: Solar Wind velocity (km/s)
- Pdyn: Solar Wind dynamic pressure (nPa)
- ByIMF: GSM y component of IMF mag. field (nT)
- BzIMF: GSM z component of IMF mag. field (nT)
- **G1: G1=< Vsw*(Bperp/40)2/(1+Bperp/40)*sin3(theta/2) > where the <> mean an average over the** previous 1 hour, Vsw is the solar wind speed, Bperp is the transverse IMF component (GSM) and theta it's clock angle.
- **G2: G2=< a*Vsw*Bs > where the <> mean an average over the previous 1 hour,** Vsw is solar wind speed, Bs=|IMF Bz| when IMF Bz < 0 and Bs=0 when IMF Bz > 0, a=0.005.
- **G3: G3=< Vsw*Dsw*Bs /2000.> where the <> mean an average over the previous 1 hour,** Vsw is the solar wind speed, Dsw is the solar wind density, Bs=|IMF Bz| when IMF Bz < 0 and Bs=0 when IMF Bz > 0.
- W1 - W6: see definition in (JGR-A, v.110(A3), 2005.) (PDF 1.2MB)
- AL: the auroral index

Options

- 1st element: 0 - don't compute L* or phi ; 1 - compute L*; 2- compute phi
- **2nd element: 0 - initialize IGRF field once per year (year.5);** n - n is the frequency (in days) starting on January 1st of each year (i.e. if options(2nd element)=15 then IGRF will be updated on the following days of the year: 1, 15, 30, 45 ...)
- **3rd element: resolution to compute L* (0 to 9) where 0 is the recomended value to ensure a** good ratio precision/computation time (i.e. an error of ~2% at L=6). The higher the value the better will be the precision, the longer will be the computing time. Generally there is not much improvement for values larger than 4. Note that this parameter defines the integration step (theta) along the field line such as dtheta=(2pi)/(720*[options(3rd element)+1])
- **4th element: resolution to compute L* (0 to 9). The higher the value the better will be** the precision, the longer will be the computing time. It is recommended to use 0 (usually sufficient) unless L* is not computed on a LEO orbit. For LEO orbit higher values are recommended. Note that this parameter defines the integration step (phi) along the drift shell such as dphi=(2pi)/(25*[options(4th element)+1])
- **5th element: allows to select an internal magnetic field model (default is set to IGRF)**
  - 0 = IGRF
  - 1 = Eccentric tilted dipole
  - 2 = Jensen&Cain 1960
  - 3 = GSFC 12/66 updated to 1970
  - 4 = User-defined model (Default: Centred dipole + uniform [Dungey open model])
  - 5 = Centred dipole

Examples

```
>>> t = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> y = Coords([[3,0,0],[2,0,0]], 'GEO', 'car')
>>> spacepy.irbempy.Lstar(t,y)
{'Blocal': array([ 1020.40493286,  3446.08845227]),
    'Bmin': array([ 1019.98404311,  3437.63865243]),
    'Lm': array([ 3.08948304,  2.06022102]),
    'Lstar': array([ 2.97684043,  1.97868577]),
    'MLT': array([ 23.5728333 ,  23.57287944]),
    'Xj': array([ 0.00112884,  0.00286955])}
```

spacepy.irbempy.find_Bmirror

spacepy.irbempy.**find_Bmirror**(*ticks, loci, alpha, extMag='T01STORM', options=[1, 0, 0, 0, 0], omnivals=None*)

call find_mirror_point from irbem library and return a dictionary with values for Blocal, Bmirr and the GEO (cartesian) coordinates of the mirror point

Parameters **ticks** : Ticktock class

containing time information

loci : Coords class

containing spatial information

alpha : array-like

containing the pitch angles

extMag : str

optional; will choose the external magnetic field model possible values ['0', 'MEAD', 'T87SHORT', 'T87LONG', 'T89', OPQUIET', 'OPDYN', 'T96', 'OSTA', 'T01QUIET', 'T01STORM', 'T05', 'ALEX', 'TS07']

options : array-like (optional)

length=5 : explained in Lstar

omnivals : dict (optional)

if not provided, will use lookup table (see get_Lstar documentation for further explanation)

Returns **results** : dictionary

containing keys: Blocal, Bmirr, GEOcar

See also:

get_Lstar, *get_Bfield*, *find_magequator*

Examples

```
>>> t = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> y = Coords([[3,0,0],[2,0,0]], 'GEO', 'car')
>>> ib.find_Bmirror(t,y,[90,80,60,10])
{'Blocal': array([ 0.,  0.]),
 'Bmirr': array([ 0.,  0.]),
 'loci': Coords( [[ NaN  NaN  NaN]
 [ NaN  NaN  NaN]] ), dtype=GEO,car, units=['Re', 'Re', 'Re']}
```

spacepy.irbempy.find_magequator

`spacepy.irbempy.find_magequator(ticks, loci, extMag='T01STORM', options=[1, 0, 0, 0, 0], omnivals=None)`

call find_magequator from irbem library and return a dictionary with values for Bmin and the GEO (cartesian) coordinates of the magnetic equator

Parameters - **ticks (Ticktock class)** : containing time information

- loci (Coords class) : containing spatial information
- **extMag (string)** [optional; will choose the external magnetic field model] possible values ['0', 'MEAD', 'T87SHORT', 'T87LONG', 'T89', 'OPQUIET', 'OPDYN', 'T96', 'OSTA', 'T01QUIET', 'T01STORM', 'T05', 'ALEX', 'TS07']
- options (optional list or array of integers length=5) : explained in Lstar
- omni values as dictionary (optional) : if not provided, will use lookup table
- (see Lstar documentation for further explanation)

Returns - **results (dictionary)** : containing keys: Bmin, Coords instance with GEO coordinates of

the magnetic equator

See also:

get_Lstar, *get_Bfield*, find_Bmirr

Examples

```
>>> t = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> y = Coords([[3,0,0],[2,0,0]], 'GEO', 'car')
>>> op.find_magequator(t,y)
{'Bmin': array([  945.63652413,  3373.64496167]),
    'loci': Coords( [[ 2.99938371  0.00534151 -0.03213603]
    [ 2.00298822 -0.0073077   0.04584859]] ), dtype=GEO,car, units=['Re', 'Re', 'Re']}
```

spacepy.irbempy.car2sph

`spacepy.irbempy.car2sph(CARin)`

coordinate transformation from cartesian to spherical

Parameters - **CARin (list or ndarray)** : coordinate points in (n,3) shape with n coordinate points in

units of [Re, Re, Re] = [x,y,z]

Returns - **results (ndarray)** : values after conversion to spherical coordinates in

radius, latitude, longitude in units of [Re, deg, deg]

See also:

sph2car

Examples

```
>>> sph2car([1,45,0])
array([ 0.70710678,  0.        ,  0.70710678])
```

spacepy.irbempy.sph2car

spacepy.irbempy.**sph2car**(*SPHin*)
coordinate transformation from spherical to cartesian

Parameters **- SPHin (list or ndarray)** : coordinate points in (n,3) shape with n coordinate points in units of [Re, deg, deg] = [r, latitude, longitude]

Returns **- results (ndarray)** : values after conversion to cartesian coordinates x,y,z

See also:

car2sph

Examples

```
>>> sph2car([1,45,45])
array([ 0.5       ,  0.5       ,  0.70710678])
```

spacepy.irbempy.coord_trans

spacepy.irbempy.**coord_trans**(*loci*, *returntype*, *returncarsph*)
thin layer to call coor_trans1 from irbem lib this will convert between systems GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL

Parameters **- loci (Coords instance)** : containing coordinate information, can contain n points

- returntype (str) : describing system as GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL
- returncarsph (str) : cartesian or spherical units 'car', 'sph'

Returns **- xout (ndarray)** : values after transformation in (n,3) dimensions

See also:

sph2car, *car2sph*

Examples

```
>>> c = Coords([[3,0,0],[2,0,0]], 'GEO', 'car')
>>> c.ticks = Ticktock(['2002-02-02T12:00:00', '2002-02-02T12:10:00'], 'ISO')
>>> coord_trans(c, 'GSM', 'car')
array([[ 2.8639301 , -0.01848784,  0.89306361],
[ 1.9124434 ,  0.07209424,  0.58082929]])
```

spacepy.irbempy.get_sysaxes

`spacepy.irbempy.get_sysaxes(dtype, carsph)`

will return the sysaxes according to the irbem library

Parameters **- dtype (str)** : coordinate system, possible values: GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL

- carsph (str) : cartesian or spherical, possible values: 'sph', 'car'

Returns **- sysaxes (int)** : value after oner_desp library from 0-8 (or None if not available)

See also:

get_dtype

Examples

```
>>> get_sysaxes('GSM', 'car')
2
```

spacepy.irbempy.get_dtype

`spacepy.irbempy.get_dtype(sysaxes)`

will return the coordinate system type as string

Parameters **- sysaxes (int)** : number according to the irbem, possible values: 0-8

Returns **- dtype (str)** : coordinate system GDZ, GEO, GSM, GSE, SM, GEI, MAG, SPH, RLL

- carsph (str) : cartesian or spherical 'car', 'sph'

See also:

get_sysaxes

Examples

```
>>> get_dtype(3)
('GSE', 'car')
```

spacepy.irbempy.prep_irbem

`spacepy.irbempy.prep_irbem(ticks=None, loci=None, alpha=[], extMag='T01STORM', options=[1, 0, 0, 0, 0], omnivals=None)`

Prepare inputs for direct IRBEM-LIB calls. Not expected to be called by the user.

lanlstar - module to calculate Lstar or Lmax using artificial neural network

Lstar and Lmax calculation using artificial neural network (ANN) technique.

This module requires the ffnet package.

Authors: Josef Koller, Yiqun Yu Institution: Los Alamos National Laboratory Contact: jkoller@lanl.gov, yiqun@lanl.gov

LANLstar(inputdict, extMag)	Calculate Lstar
LANLmax(inputdict, extMag)	Calculate last closed drift shell (Lmax)

spacepy.LANLstar.LANLstar

`spacepy.LANLstar.`**`LANLstar`**(*inputdict*, *extMag*)
Calculate Lstar

Based on the L* artificial neural network (ANN) trained from different magnetospheric field models.

Parameters extMag : list of string(s)

containing one or more of the following external magnetic field models: ‘OPDYN’, ‘OPQUIET’, ‘T89’, ‘T96’, ‘T01QUIET’, ‘T01STORM’, ‘T05’

inputdict : dictionary

containing the following keys, each entry is a list or array. Note the keys for the above models are different.

—For OPDYN: [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘dens’, ‘velo’, ‘BzIMF’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

- **For OPQUIET:** [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘dens’, ‘velo’, ‘BzIMF’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

—For T89: [’Year’, ‘DOY’, ‘Hr’, ‘Kp’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

- **For T96:** [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

—For T01QUIET: [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘G1’, ‘G2’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

- **For T01STORM:** [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘G2’, ‘G3’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

—For T05: [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘W1’,’W2’,’W3’,’W4’,’W5’,’W6’, ‘Lm’, ‘Bmirr’, ‘PA’, ‘rGSM’, ‘latGSM’, ‘lonGSM’]

- **For RAMSCB:** [’Year’, ‘DOY’, ‘Hr’, ‘Dst’, ‘Pdyn’, ‘ByIMF’, ‘BzIMF’, ‘PA’, ‘SMx’,’SMy’,’SMz’]

Dictionaries with numpy vectors are allowed.

Returns out : dictionary

Lstar array for each key which corresponds to the specified magnetic field model.

Examples

```
>>> import spacepy.LANLstar as LS
>>>
>>> inputdict = {}
>>> inputdict['Kp']     = [2.7      ]         # Kp index
>>> inputdict['Dst']    = [7.7777   ]         # Dst index (nT)
>>> inputdict['dens']   = [4.1011   ]         # solar wind density (/cc)
>>> inputdict['velo']   = [400.1011 ]         # solar wind velocity (km/s)
>>> inputdict['Pdyn']   = [4.1011   ]         # solar wind dynamic pressure (nPa)
>>> inputdict['ByIMF']  = [3.7244   ]         # GSM y component of IMF magnetic field (nT)
>>> inputdict['BzIMF']  = [-0.1266  ]         # GSM z component of IMF magnetic field (nT)
>>> inputdict['G1']     = [1.029666 ]         # as defined in Tsganenko 2003
>>> inputdict['G2']     = [0.549334 ]
>>> inputdict['G3']     = [0.813999 ]
>>> inputdict['W1']     = [0.122444 ]         # as defined in Tsyganenko and Sitnov 2005
>>> inputdict['W2']     = [0.2514   ]
>>> inputdict['W3']     = [0.0892   ]
>>> inputdict['W4']     = [0.0478   ]
>>> inputdict['W5']     = [0.2258   ]
>>> inputdict['W6']     = [1.0461   ]
>>>
>>> inputdict['Year']   = [1996     ]
>>> inputdict['DOY']    = [6        ]
>>> inputdict['Hr']     = [1.2444   ]
>>>
>>> inputdict['Lm']     = [4.9360   ]          # McIllwain L
>>> inputdict['Bmirr']  = [315.6202 ]          # magnetic field strength at the mirror point
>>> inputdict['rGSM']   = [4.8341   ]          # radial coordinate in GSM [Re]
>>> inputdict['lonGSM'] = [-40.2663 ]          # longitude coodrinate in GSM [deg]
>>> inputdict['latGSM'] = [36.44696 ]          # latitude coordiante in GSM [deg]
>>> inputdict['PA']     = [57.3874  ]          # pitch angle [deg]
>>> inputdict['SMx']    = [3.9783   ]
>>> inputdict['SMy']    = [-2.51335 ]
>>> inputdict['SMz']    = [1.106617 ]
>>>
>>> LS.LANLstar(inputdict, ['OPDYN','OPQUIET','T01QUIET','T01STORM','T89','T96','T05','RAMSCB'])
{'OPDYN': array([4.7171]),
 'OPQUIET': array([4.6673]),
 'T01QUIET': array([4.8427]),
 'T01STORM': array([4.8669]),
 'T89': array([4.5187]),
 'T96': array([4.6439]),
 'TS05': array([4.7174]),
 'RAMSCB','array([5.9609])}
```

spacepy.LANLstar.LANLmax

`spacepy.LANLstar.`**`LANLmax`**(*inputdict*, *extMag*)

Calculate last closed drift shell (Lmax)

Based on the L* artificial neural network (ANN) trained from different magnetospheric field models.

Parameters extMag : list of string(s)

containing one or more of the following external Magnetic field models: 'OPDYN', 'OPQUIET', 'T89', 'T96', 'T01QUIET', 'T01STORM', 'T05'

inputdict : dictionary

containing the following keys, each entry is a list or array. Note the keys for the above models are different.

—For OPDYN: ['Year', 'DOY', 'Hr', 'Dst', 'dens', 'velo', 'BzIMF', 'PA']

– For OPQUIET: ['Year', 'DOY', 'Hr', 'Dst', 'dens', 'velo', 'BzIMF', 'PA']

—For T89: ['Year', 'DOY', 'Hr', 'Kp', 'Pdyn', 'ByIMF', 'BzIMF', 'PA']

– For T96: ['Year', 'DOY', 'Hr', 'Dst', 'Pdyn', 'ByIMF', 'BzIMF','PA']

—For T01QUIET: ['Year', 'DOY', 'Hr', 'Dst', 'Pdyn', 'ByIMF', 'BzIMF', 'G1', 'G2','PA']

– For T01STORM: ['Year', 'DOY', 'Hr', 'Dst', 'Pdyn', 'ByIMF', 'BzIMF', 'G2', 'G3', 'PA']

—For T05: ['Year', 'DOY', 'Hr', 'Dst', 'Pdyn', 'ByIMF', 'BzIMF', 'W1','W2','W3','W4','W5','W6', 'PA']

Dictionaries with numpy vectors are allowed.

Returns out : dictionary

Lmax array for each key which corresponds to the specified magnetic field model.

Examples

```
>>> import spacepy.LANLstar as LS
>>>
>>> inputdict = {}
>>> inputdict['Kp']     = [2.7      ]            # Kp index
>>> inputdict['Dst']    = [7.7777   ]            # Dst index (nT)
>>> inputdict['dens']   = [4.1011   ]            # solar wind density (/cc)
>>> inputdict['velo']   = [400.1011 ]            # solar wind velocity (km/s)
>>> inputdict['Pdyn']   = [4.1011   ]            # solar wind dynamic pressure (nPa)
>>> inputdict['ByIMF']  = [3.7244   ]            # GSM y component of IMF magnetic field (nT)
>>> inputdict['BzIMF']  = [-0.1266  ]            # GSM z component of IMF magnetic field (nT)
>>> inputdict['G1']     = [1.029666 ]            # as defined in Tsganenko 2003
>>> inputdict['G2']     = [0.549334 ]
>>> inputdict['G3']     = [0.813999 ]
>>> inputdict['W1']     = [0.122444 ]            # as defined in Tsyganenko and Sitnov 2005
>>> inputdict['W2']     = [0.2514   ]
>>> inputdict['W3']     = [0.0892   ]
>>> inputdict['W4']     = [0.0478   ]
>>> inputdict['W5']     = [0.2258   ]
>>> inputdict['W6']     = [1.0461   ]
>>>
>>> inputdict['Year']   = [1996     ]
>>> inputdict['DOY']    = [6        ]
>>> inputdict['Hr']     = [1.2444   ]
>>>
>>> inputdict['PA']     = [57.3874  ]             # pitch angle [deg]
>>>
```

```
>>> LS.LANLmax(inputdict, ['OPDYN','OPQUIET','T01QUIET','T01STORM','T89','T96','T05'])
{'OPDYN': array([10.6278]),
 'OPQUIET': array([9.3352]),
 'T01QUIET': array([10.0538]),
 'T01STORM': array([9.9300]),
 'T89': array([8.2888]),
 'T96': array([9.2410]),
 'T05': array([9.9295])}
```

omni - module to read and process NASA OMNIWEB data

Tools to read and process omni data (Qin-Denton, etc.)

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

Copyright 2010-2014 Los Alamos National Security, LLC.

About omni

The omni module primarily manages the hourly OMNI2 and Qin-Denton data, which are sourced from the Virtual Radiation Belt Observatory (ViRBO), who maintain these data sources. The data can be kept up-to-date in SpacePy using the *update()* function in the *spacepy.toolbox* module.

The OMNI2 data combines data from a variety of satellites that sample the solar wind (notably ACE and Wind), and propagates the data to Earth's bow shock nose. The Qin-Denton data is derived from the OMNI2 data and is designed for providing input to the Tsyganenko magnetic field models. The later Tsyganenko magnetic field models require subsidiary parameters (G- and W-parameters) that are pre-calculated in the Qin-Denton data. Further, the Qin-Denton data contains no data gaps – all gaps are filled (for details on the gap filling, see the paper by Qin et al..)

Advanced features

Higher resolution data, or custom data sources, can also be managed/accessed with this module, although this is considered an advanced use for this module. This is achieved using custom names for the dbase keyword in get_omni, which must be defined in the SpacePy configuration file (for a user-install on linux, this is ~/.spacepy/spacepy.rc; see SpacePy Configuration). An example of the formatting required is

qd1min: /usr/somedir/QinDenton/YYYY/QinDenton_YYYYMMDD_1min.txt

In this example the custom data source name is qd1min. Wildcard substitutions can be made for the year (YYYY), month (MM) and day (DD). Future updates will give more flexibility in data storage model, but currently we assume that all custom data sources follow a convention in which the data files are daily, and the files are organized into folders by year. The year, month and day must all be specified in the filename.

Currently there are some restrictions on the data format for custom data sources. The stored data must currently be stored as JSON-headed ASCII. If data conversions are required, then a valid dictionary of conversion functions must be supplied via the convert keyword argument. See *readJSONheadedASCII()* for details. Additionally, by default this will interpolate the data to the requested time ticks. To return only the actual recorded data values for the specified time range set the keyword argument interp to False.

get_omni(ticks[, dbase])	Returns Qin-Denton OMNI values, interpolated to any time-base from a default hourly resolution
omnirange([dbase])	Returns datetimes giving start and end times in the OMNI/Qin-Denton data

spacepy.omni.get_omni

spacepy.omni.**get_omni**(*ticks*, *dbase='QDhourly'*, ***kwargs*)

Returns Qin-Denton OMNI values, interpolated to any time-base from a default hourly resolution

The update function in toolbox retrieves all available hourly Qin-Denton data, and this function accesses that and interpolates to the given times, returning the OMNI values as a SpaceData (dict-like) 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)

Parameters **ticks** : Ticktock class or array-like of datetimes

time values for desired output

dbase : str (optional)

Select data source, options are 'QDhourly', 'OMNI2', 'Mergedhourly' Note - Custom data sources can be specified in the spacepy config file as described in the module documentation.

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

containing all Qin-Denton values at times given by ticks

Notes

Note 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.

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)
>>> d.tree(levels=1)
+
|____ByIMF
|____Bz1
|____Bz2
|____Bz3
|____Bz4
|____Bz5
|____Bz6
|____BzIMF
|____DOY
|____Dst
|____G1
|____G2
|____G3
|____Hr
|____Kp
|____Pdyn
|____Qbits
```

```
    |____RDT
    |____UTC
    |____W1
    |____W2
    |____W3
    |____W4
    |____W5
    |____W6
    |____Year
    |____akp3
    |____dens
    |____ticks
    |____velo
```

spacepy.omni.omnirange

spacepy.omni.**omnirange**(*dbase='QDhourly'*)

Returns datetimes giving start and end times in the OMNI/Qin-Denton data

The update function in toolbox retrieves all available hourly Qin-Denton data, and this function accesses that and looks up the start and end times, returning them as datetime objects.

Parameters **dbase** : string (optional)

name of omni database to check. Currently 'QDhourly' and 'OMNI2hourly'

Returns **omnirange** : tuple

containing two datetimes giving the start and end times of the available data

Examples

```
    >>> import spacepy.omni as om
    >>> om.omnirange()
    (datetime.datetime(1963, 1, 1, 0, 0), datetime.datetime(2011, 11, 30, 23, 0))
    >>> om.omnirange(dbase='OMNI2hourly')
    (datetime.datetime(1963, 1, 1, 0, 0), datetime.datetime(2011, 11, 30, 23, 0))
```

plot - Plot, various specialized plotting functions and associated utilities

plot: SpacePy plotting routines

This package aims to make getting publication ready plots easier. It provides classes and functions for different types of plot (e.g. Spectrogram, levelPlot), for helping make plots more cleanly (e.g. set_target, dual_half_circle), and for making plots convey information more cleanly, with less effort (e.g. applySmartTimeTicks, style).

This plot module now provides style sheets. For most standard plotting we recommend the *default* style sheet (aka *spacepy*). To auto-apply the default plot style the following should be added to your spacepy.rc file:

```
apply_plot_styles: True
```

Different plot types may not work well with this style, so we have provided alternatives. For polar plots, spectrograms, or anything with larger blocks of color, it may be better to use one of the alternatives:

```
import spacepy.plot as splot
splot.style('altgrid') # inverts background from default so it's white
splot.style('polar') # designed for filled polar plots
splot.revert_style() # put the style back to matplotlib defaults
```

For those constrained by institutional computing, we also provide the new colormaps developed for matplotlib v2. These colormaps are designed to be perceptually uniform, and hence colorblind-friendly.

Authors: Brian Larsen and Steve Morley Institution: Los Alamos National Laboratory Contact: balarsen@lanl.gov

Copyright 2011-2016 Los Alamos National Security, LLC.

add_logo(img[, fig, pos, margin])	Add an image (logo) to one corner of a plot.
annotate_xaxis(txt[, ax])	Write text in-line and to the right of the x-axis tick labels
applySmartTimeTicks(ax, time[, dolimit, dolabel])	Given an axis *ax* and a list/array of datetime objects, *time*, use the smartTi
available([returnvals])	List the available plot styles provided by spacepy.plot
collapse_vertical(combine[, others, leave_axis])	Collapse the vertical spacing between two or more subplots.
dual_half_circle([center, radius, ...])	Plot two half circles to a plot with the specified face colors and rotation.
levelPlot(data[, var, time, levels, target, ...])	Draw a step-plot with up to 5 levels following a color cycle (e.g.
plot(*args, **kwargs)	Convenience wrapper for matplotlib's plot function
revert_style()	Revert plot style settings to those in use prior to importing spacepy.plot
set_target(target[, figsize, loc, polar])	Given a *target* on which to plot a figure, determine if that *target* is **None** o
shared_ylabel(axes, txt, *args, **kwargs)	Create a ylabel that spans several subplots
solarRotationPlot(ticks, data[, targ_ax, ...])	Plots a 1-D time series as a Carrington or Bartels plot
spectrogram(data, **kwargs)	This class rebins data to produce a 2D data map that can be plotted as a sp
style([look, cmap])	Apply SpacePy's matplotlib style settings from a known style sheet.
timestamp([position, size, draw, strnow, ...])	print a timestamp on the current plot, vertical lower right

spacepy.plot.add_logo

spacepy.plot.**add_logo**(*img*, *fig=None*, *pos='br'*, *margin=0.05*)

Add an image (logo) to one corner of a plot.

The provided image will be placed in a corner of the plot and sized to maintain its aspect ratio and be as large as possible without overlapping any existing elements of the figure. Thus this should be the last call in constructing a figure.

Parameters img : str or numpy.ndarray

The image to place on the figure. If a string, assumed to be a filename to be read with `imread()`; if a numpy array, assumed to be the image itself (in a simliar format).

Returns (axes, axesimg) : tuple of Axes and AxesImage

The `Axes` object created to hold the iamge, and the `AxesImage` object for the image itself.

Other Parameters fig : matplotlib.figure.Figure

The figure on which to place the logo; if not specified, the `gcf()` function will be used.

pos : str

The position to place the logo. br: bottom right; bl: bottom left; tl: top left; tr: top right

margin : float

Margin to include on each side of figure, as a fraction of the larger dimension of the figure (width or height). Default is 0.05 (5%).

Notes

Calls `draw()` to ensure locations are up to date.

Examples

```
>>> import spacepy.plot.utils
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax0 = fig.add_subplot(211)
>>> ax0.plot([1, 2, 3], [1, 2, 1])
[<matplotlib.lines.Line2D at 0x00000000>]
>>> ax1 = fig.add_subplot(212)
>>> ax1.plot([1, 2, 3], [2, 1, 2])
[<matplotlib.lines.Line2D at 0x00000000>]
>>> spacepy.plot.utils.add_logo('logo.png', fig)
(<matplotlib.axes.Axes at 0x00000000>,
 <matplotlib.image.AxesImage at 0x00000000>)
```

spacepy.plot.annotate_xaxis

spacepy.plot.**annotate_xaxis**(*txt*, *ax=None*)

Write text in-line and to the right of the x-axis tick labels

Annotates the x axis of an `Axes` object with text placed in-line with the tick labels and immediately to the right of the last label. This is formatted to match the existing tick marks.

Parameters **txt** : str

The annotation text.

Returns **out** : matplotlib.text.Text

The `Text` object for the annotation.

Other Parameters **ax** : matplotlib.axes.Axes

The axes to annotate; if not specified, the `gca()` function will be used.

Notes

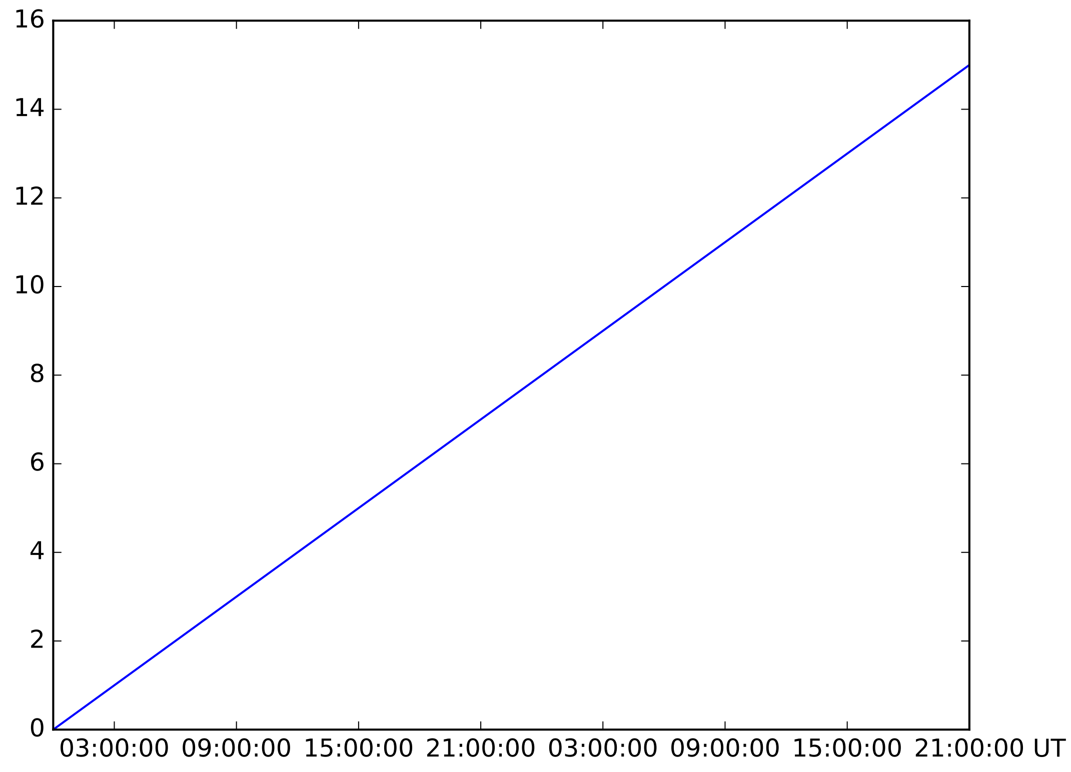
The annotation is placed *immediately* to the right of the last tick label. Generally the first character of `txt` should be a space to allow some room.

Calls `draw()` to ensure tick marker locations are up to date.

Examples

```
>>> import spacepy.plot.utils
>>> import matplotlib.pyplot as plt
>>> import datetime
>>> times = [datetime.datetime(2010, 1, 1) + datetime.timedelta(hours=i)
...     for i in range(0, 48, 3)]
>>> plt.plot(times, range(16))
[<matplotlib.lines.Line2D object at 0x0000000>]
```

```
>>> spacepy.plot.utils.annotate_xaxis(' UT') #mark that times are UT
<matplotlib.text.Text object at 0x0000000>
```



spacepy.plot.applySmartTimeTicks

`spacepy.plot.`**`applySmartTimeTicks`**(*ax*, *time*, *dolimit=True*, *dolabel=False*)

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.

dolabel : boolean (optional)

Sets autolabeling of the time axis with "Time from" time[0]

See also:

`smartTimeTicks`

spacepy.plot.available

`spacepy.plot.`**`available`**(*returnvals=False*)
List the available plot styles provided by spacepy.plot

Note that some of the available styles have multiple aliases. To apply an available style, use *spacepy.plot.style*.

spacepy.plot.collapse_vertical

`spacepy.plot.`**`collapse_vertical`**(*combine*, *others=()*, *leave_axis=False*)
Collapse the vertical spacing between two or more subplots.

Useful for a multi-panel plot where most subplots should have space between them but several adjacent ones should not (i.e., appear as a single plot.) This function will remove all the vertical space between the subplots listed in `combine` and redistribute the space between all of the subplots in both `combine` and `others` in proportion to their current size, so that the relative size of the subplots does not change.

Parameters **combine** : sequence

The `Axes` objects (i.e. subplots) which should be placed together with no vertical space.

Other Parameters **others** : sequence

The `Axes` objects (i.e. subplots) which will keep their vertical spacing, but will be expanded with the space taken away from between the elements of `combine`.

leave_axis : bool

If set to true, will leave the axis lines and tick marks between the collapsed subplots. By default, the axis line ("spine") is removed so the two subplots appear as one.

Notes

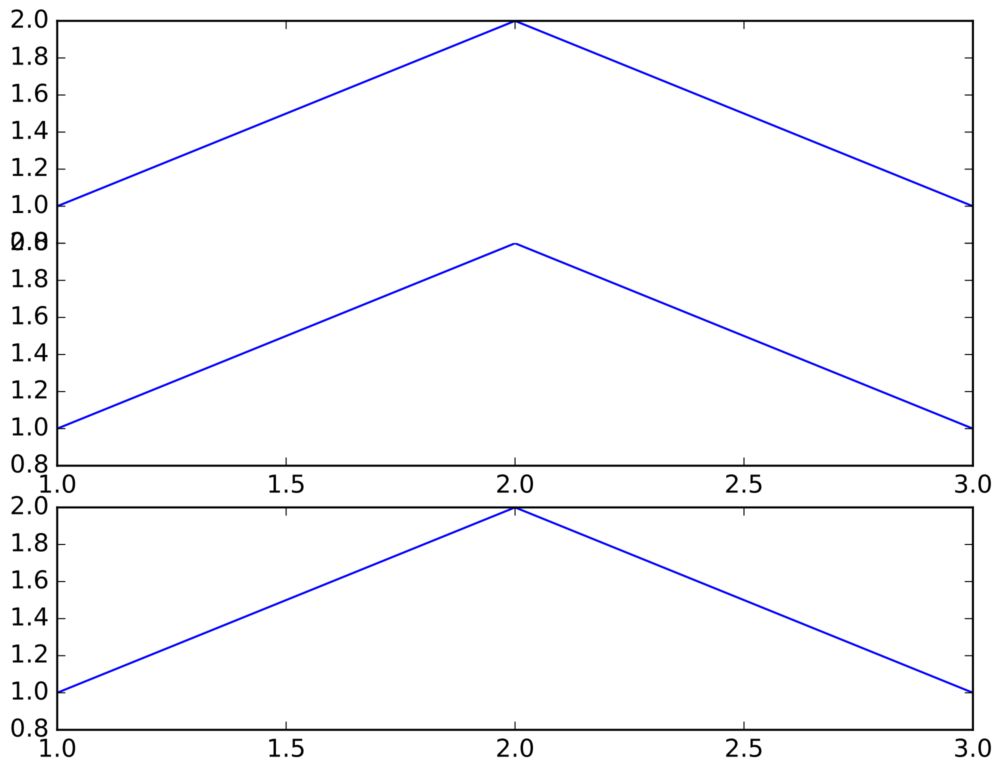
This function can be fairly fragile and should only be used for fairly simple layouts, e.g., a one-column multi-row plot stack.

This may require some clean-up of the y axis labels, as they are likely to overlap.

Examples

```
>>> import spacepy.plot.utils
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> #Make three stacked subplots
>>> ax0 = fig.add_subplot(311)
>>> ax1 = fig.add_subplot(312)
>>> ax2 = fig.add_subplot(313)
>>> ax0.plot([1, 2, 3], [1, 2, 1]) #just make some lines
[<matplotlib.lines.Line2D object at 0x0000000>]
>>> ax1.plot([1, 2, 3], [1, 2, 1])
[<matplotlib.lines.Line2D object at 0x0000000>]
>>> ax2.plot([1, 2, 3], [1, 2, 1])
```

```
    [<matplotlib.lines.Line2D object at 0x0000000>]
    >>> #Collapse space between top two plots, leave bottom one alone
    >>> spacepy.plot.utils.collapse_vertical([ax0, ax1], [ax2])
```



spacepy.plot.dual_half_circle

`spacepy.plot.`**`dual_half_circle`**(*center=(0, 0)*, *radius=1.0*, *sun_direction='right'*, *ax=None*, *colors=('w', 'k')*, ***kwargs*)

Plot two half circles to a plot with the specified face colors and rotation. This is normal to use to denote the sun direction in magnetospheric science plots.

Returns **out** : tuple

Tuple of the two wedge objects

Other Parameters **center** : array-like, 2 elements

Center in data coordinates of the circles, default (0,0)

radius : float

Radius of the circles, defualt 1.0

sun_direction : string or float

The rotation direction of the first (white) circle. Options are ['down', 'down right', 'right', 'up left', 'up right', 'up', 'down left', 'left'] or an angle in degrees counter-clockwise from up. Default right.

ax : matplotlib.axes

Axis to plot the circles on.

colors : array-like, 2 elements

The two colors for the circle fill. The First number is the light and second is the dark.

****kwargs** : other keywords

Other keywords to pass to matplotlib.patches.Wedge

Examples

```
>>> import spacepy.plot
>>> spacepy.plot.dual_half_circle()
```

spacepy.plot.levelPlot

spacepy.plot.**levelPlot**(*data*, *var=None*, *time=None*, *levels=(3, 5)*, *target=None*, *colors=None*, ***kwargs*)

Draw a step-plot with up to 5 levels following a color cycle (e.g. Kp index "stoplight")

Parameters **data** : array-like, or dict-like

Data for plotting. If dict-like, the key providing an array-like to plot must be given to var keyword argument.

Returns **binned** : tuple

Tuple of the binned data and bins

Other Parameters **var** : string

Name of key in dict-like input that contains data

time : array-like or string

Name of key in dict-like that contains time, or arraylike of datetimes

levels : array-like, up to 5 levels

Breaks between levels in data that should be shown as distinct colors

target : figure or axes

Target axes or figure window

colors : array-like

Colors to use for the color sequence (if insufficient colors, will use as a cycle)

****kwargs** : other keywords

Other keywords to pass to spacepy.toolbox.binHisto

Examples

```
>>> import spacepy.plot as splot
>>> import spacepy.time as spt
>>> import spacepy.omni as om
>>> tt = spt.tickrange('2012/09/28','2012/10/2', 3/24.)
>>> omni = om.get_omni(tt)
>>> splot.levelPlot(omni, var='Kp', time='UTC', colors=['seagreen', 'orange', 'crimson'])
```

spacepy.plot.plot

`spacepy.plot.plot(*args, **kwargs)`

Convenience wrapper for matplotlib's plot function

As with matplotlib's plot function, *args* is a variable length argument, allowing for multiple *x*, *y* pairs, each with optional format string. For full details, see matplotlib.pyplot.plot

Other Parameters **smartTimeTicks** : boolean

If True then use applySmartTimeTicks to set x-axis labeling

figsize : array-like, 2 elements

Set figure size directly on call to plot, (width, height)

****kwargs** : other keywords

Other keywords to pass to matplotlib.pyplot.plot

spacepy.plot.revert_style

`spacepy.plot.revert_style()`

Revert plot style settings to those in use prior to importing spacepy.plot

spacepy.plot.set_target

`spacepy.plot.set_target(target, figsize=None, loc=None, polar=False)`

Given a *target* on which to plot a figure, determine if that *target* is **None** or a matplotlib figure or axes object. Based on the type of *target*, a figure and/or axes will be either located or generated. Both the figure and axes objects are returned to the caller for further manipulation. This is used in nearly all *add_plot*-type methods.

Parameters **target** : object

The object on which plotting will happen.

Returns **fig** : object

A matplotlib figure object on which to plot.

ax : object

A matplotlib subplot object on which to plot.

Other Parameters **figsize** : tuple

A two-item tuple/list giving the dimensions of the figure, in inches. Defaults to Matplotlib defaults.

loc : integer

The subplot triple that specifies the location of the axes object. Defaults to matplotlib default (111).

polar : bool

Set the axes object to polar coodinates. Defaults to **False**.

Examples

```
>>> import matplotlib.pyplot as plt
>>> from spacepy.pybats import set_target
>>> fig = plt.figure()
>>> fig, ax = set_target(target=fig, loc=211)
```

spacepy.plot.shared_ylabel

`spacepy.plot.`**`shared_ylabel`**`(`*axes*, *txt*, **args*, ***kwargs*`)`

Create a ylabel that spans several subplots

Useful for a multi-panel plot where several subplots have the same units/quantities on the y axis.

Parameters **axes** : list

The `Axes` objects (i.e. subplots) which should share a single label

txt : str

The label to place in the middle of all the *axes* objects.

Returns **out** : matplotlib.text.Text

The `Text` object for the label.

Other Parameters **Additional arguments and keywords are passed through to**

:meth:‘~matplotlib.axes.Axes.set_ylabel‘

Notes

This function can be fairly fragile and should only be used for fairly simple layouts, e.g., a one-column multi-row plot stack.

The label is associated with the bottommost subplot in `axes`.

Examples

```
>>> import spacepy.plot.utils
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> #Make three stacked subplots
>>> ax0 = fig.add_subplot(311)
>>> ax1 = fig.add_subplot(312)
>>> ax2 = fig.add_subplot(313)
>>> ax0.plot([1, 2, 3], [1, 2, 1]) #just make some lines
[<matplotlib.lines.Line2D object at 0x0000000>]
>>> ax1.plot([1, 2, 3], [1, 2, 1])
[<matplotlib.lines.Line2D object at 0x0000000>]
>>> ax2.plot([1, 2, 3], [1, 2, 1])
[<matplotlib.lines.Line2D object at 0x0000000>]
```

```
>>> #Create a green label across all three axes
>>> spacepy.plot.utils.shared_ylabel([ax0, ax1, ax2],
... 'this is a very long label that spans all three axes', color='g')
```

spacepy.plot.solarRotationPlot

`spacepy.plot.`**`solarRotationPlot`**(*ticks*, *data*, *targ_ax=None*, *rtype='bartels'*, *nbins=27*)

Plots a 1-D time series as a Carrington or Bartels plot

spacepy.plot.spectrogram

`spacepy.plot.`**`spectrogram`**(*data*, ***kwargs*)

This class rebins data to produce a 2D data map that can be plotted as a spectrogram

It is meant to be used on arbitrary data series. The first series "x" is plotted on the abscissa and second series "y" is plotted on the ordinate and the third series "z" is plotted in color.

The series are not passed in independently but instead inside a `SpaceData` container.

Parameters **data** : `SpaceData`

The data for the spectrogram, the variables to be used default to "Epoch" for x, "Energy" for y, and "Flux" for z. Other names are specified using the 'variables' keyword. All keywords override .attrs contents.

Other Parameters **variables** : list

keyword containing the names of the variables to use for the spectrogram the list is a list of the SpaceData keys in x, y, z, order

bins : list

if the name "bins" is not specified in the .attrs of the dmarray variable this specifies the bins for each variable in a [[xbins], [ybins]] format

xlim : list

if the name "lim" is not specified in the .attrs of the dmarray variable this specifies the limit for the x variable [xlow, xhigh]

ylim : list

if the name "lim" is not specified in the .attrs of the dmarray variable this specifies the limit for the y variable [ylow, yhigh]

zlim : list

if the name "lim" is not specified in the .attrs of the dmarray variable this specifies the limit for the z variable [zlow, zhigh]

extended_out : bool (optional)

if this is True add more information to the output data model (default True)

Notes

Helper routines are planned to facilitate the creation of the SpaceData container if the data are not in the format.

Examples

```
>>> import spacepy.datamodel as dm
>>> import numpy as np
>>> import spacepy.plot as splot
>>> sd = dm.SpaceData()
>>> sd['radius'] = dm.dmarray(2*np.sin(np.linspace(0,30,500))+4, attrs={'units':'km'})
>>> sd['day_of_year'] = dm.dmarray(np.linspace(74,77,500))
>>> sd['1D_dataset'] = dm.dmarray(np.random.normal(10,3,500)*sd['radius'])
>>> spec = splot.spectrogram(sd, variables=['day_of_year', 'radius', '1D_dataset'])
>>> ax = spec.plot()
```

`spectrogram.plot`

spacepy.plot.style

`spacepy.plot.`**`style`**(*look=None*, *cmap='plasma'*)

Apply SpacePy's matplotlib style settings from a known style sheet.

Parameters **look** : str

Name of style. For a list of available style names, see 'spacepy.plot.available'.

spacepy.plot.timestamp

`spacepy.plot.`**`timestamp`**(*position=(1.003, 0.01)*, *size='xx-small'*, *draw=True*, *strnow=None*, *rotation='vertical'*, *ax=None*, ***kwargs*)

print a timestamp on the current plot, vertical lower right

Parameters **position** : list

position for the timestamp

size : string (optional)

text size

draw : Boolean (optional)

call draw to make sure it appears

kwargs : keywords

other keywords to axis.annotate

Examples

```
>>> import spacepy.plot.utils
>>> from pylab import plot, arange
>>> plot(arange(11))
[<matplotlib.lines.Line2D object at 0x49072b0>]
>>> spacepy.plot.utils.timestamp()
```

Most of the functionality in the plot module is made available directly through the *plot* namespace. However, the plot module does contain several submodules listed below

`carrington`	Module for plotting data by Carrington or Bartels rotation
`colourmaps`	
`spectrogram`(data, **kwargs)	This class rebins data to produce a 2D data map that can be plotted as a spectrogram
`utils`	Utility routines for plotting and related activities

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 range(-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

Copyright 2010 Los Alamos National Security, LLC.

Classes

`PPro`(process1, process2[, lags, winhalf, ...])	PoPPy point process object

spacepy.poppy.PPro

class `spacepy.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()`.

`aa_ci`(inter[, n_boots, seed])	Get bootstrap confidence intervals for association number
`assoc`([u, h])	Perform association analysis on input series
`assoc_mult`(windows[, inter, n_boots, seed])	Association analysis w/confidence interval on multiple windows
`ci`	
`conf_above`	
`plot`([figsize, dpi, asympt, show, norm, ...])	Create basic plot of association analysis.
`plot_mult`(windows, data[, min, max, ...])	Plots a 2D function of window size and lag
`swap`()	Swaps process 1 and process 2

`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 `ci`.

After calling, `conf_above` will contain the confidence (in percent) that the association number at that lag is *above* the asymptotic association number. (The confidence of being below is 100 - conf_above) For minor variations in conf_above to be meaningful, a *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, *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.

> **Warning:** If `seed` is specified, results may not be reproducible between systems with different sizes for C long type. Note that 64-bit Windows uses a 32-bit long and so results will be the same between 64 and 32-bit Windows, but not between 64-bit Windows and other 64-bit operating systems. If `seed` is not specified, results are not reproducible anyhow.

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.

ci

Contains the upper and lower confidence limits for the association number as a function of lag. The first element is the array of lower limits; the second, the array of upper limits. Not available until after calling `aa_ci()`.

conf_above

Contains the confidence that the association number, as a function of lag, is above the asymptotic association number. (The confidence of being below is 100 - `conf_above`.) Not available until after calling `aa_ci()`.

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 `assoc()` 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

Functions

plot_two_ppro(pprodata, pproref[, ratio, ...])	Overplots two PPro objects

Continued on next page

Table 2.24 – continued from previous page

`boots_ci`(data, n, inter, func[, seed, ...])	Construct bootstrap confidence interval
`value_percentile`(sequence, target)	Find the percentile of a particular value in a sequence

spacepy.poppy.plot_two_ppro

`spacepy.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

spacepy.poppy.boots_ci

`spacepy.poppy.`**`boots_ci`**(*data*, *n*, *inter*, *func*, *seed=None*, *target=None*, *sample_size=None*, *usepy=False*, *nretvals=1*)

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.

nretvals : int

number of return values from input function

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.

spacepy.poppy.value_percentile

`spacepy.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

PyBats - SWMF & BATS-R-US Analysis Tools

PyBats! An open source Python-based interface for reading, manipulating, and visualizing BATS-R-US and SWMF output. For more information on the SWMF, please visit the Center for Space Environment Modeling.

Introduction

At its most fundamental level, PyBats provides access to output files written by the Space Weather Modeling Framework and the codes contained within. The key task performed by PyBats is loading simulation data into a Spacepy data model object so that the user can move on to the important tasks of analyzing and visualizing the values. The secondary goal of PyBats is to make common tasks performed with these data as easy as possible. The result is that most SWMF output can be opened and visualized using only a few lines of code. Many complicated tasks, such as field line integration, is included as well.

Organization

Many output files in the SWMF share a common format. Objects to handle broad formats like these are found in the base module. The base module has classes to handle SWMF *input* files, as well.

The rest of the classes are organized by code, i.e. classes and functions specifically relevant to BATS-R-US can be found in `spacepy.pybats.bats`. Whenever a certain code included in the SWMF requires a independent class or a subclass from the PyBats base module, it will receive its own submodule.

Conventions and Prefixes

Nearly every class in PyBats inherits from *spacepy.datamodel.SpaceData*, so it is important for users to understand how to employ and explore SpaceData objects. There are a few exceptions, so always pay close attention to the docstrings and examples. Legacy code that does not adhere to this pattern is slowly being brought up-to-date with each release.

Visualization methods have two prefixes: *plot_* and *add_*. Whenever a method begins with *plot_*, a quick-look product will be created that is not highly- configurable. These methods are meant to yeild either simple diagnostic plots or static, often-used products. There are few methods that use this prefix. The prefix *add_* is always followed by *plot_type*; it indicates a plotting method that is highly configurable and meant to be combined with other *add_*-like methods and matplotlib commands.

Common calculations, such as calculating Alfven wave speeds of MHD results, are strewn about PyBats' classes. They are always given the method prefix *calc_*, i.e. *calc_alfven*. Methods called *calc_all* will search for all class methods with the *calc_* prefix and call them.

Submodules

There are submodules for most models included within the SWMF. The classes and methods contained within are code-specific, yielding power and convenience at the cost of flexibility. A few of the submodules are helper modules- they are not code specific, but rather provide functionality not related to an SWMF-included code.

`bats`	A PyBats module for handling input, output, and visualization of binary SWMF output files taylored to BATS-R-US-typ
`dgcpm`	The PyBats submodule for handling input and output for the Dynamic Global Core Plasma Model (DGCPM), a plasmas
`dipole`	Some functions for the generation of a dipole field.
`gitm`	PyBats submodule for handling input/output for the Global Ionosphere-Thermosphere Model (GITM), one of the choice
`kyoto`	kyoto is a tool set for obtaining and handling geomagnetic indices stored at the
`pwom`	PyBats submodule for handling input/output for the Polar Wind Outflow Model (PWOM), one of the choices for the PW
`ram`	A module for reading, handling, and plotting RAM-SCB output.
`rim`	Classes, functions, and methods for reading, writing, and plotting output from the Ridley Ionosphere Model (RIM) and t
`trace2d`	A set of routines for fast field line tracing.

spacepy.pybats.bats

A PyBats module for handling input, output, and visualization of binary SWMF output files taylored to BATS-R-US-type data.

Classes

`BatsLog`(filename[, starttime, keep_case])	A specialized version of `LogFile` that includes special methods for plotting common
`Stream`(bats, xstart, ystart, xfield, yfield)	A class for streamlines.
`Bats2d`(filename[, format])	A child class of `IdlFile` taylored to BATS-R-US output.
`Mag`(nlines, time[, gmvars, ievars])	A container for data from a single BATS-R-US virtual magnetometer.
`MagFile`(filename[, ie_name, find_ie])	BATS-R-US magnetometer files are powerful tools for both research and operations.
`GeoIndexFile`(filename[, keep_case])	Geomagnetic Index files are a specialized BATS-R-US output that contain geomagnetic
`VirtSat`(*args, **kwargs)	A `spacepy.pybats.LogFile` object tailored to virtual satellite output; includes s

spacepy.pybats.bats.BatsLog

class `spacepy.pybats.bats.`**`BatsLog`**(*filename*, *starttime=(2000, 1, 1, 0, 0, 0)*, *keep_case=True*, **args*, ***kwargs*)

A specialized version of *`LogFile`* that includes special methods for plotting common BATS-R-US log file values, such as D$_{ST}$.

`add_dst_quicklook`([target, loc, plot_obs, ...])	Create a quick-look plot of Dst (if variable present in file) and compare against

`add_dst_quicklook`(*target=None*, *loc=111*, *plot_obs=False*, *epoch=None*, *add_legend=True*, *plot_sym=False*, *dstvar=None*, *obs_kwargs={'c': 'k', 'ls': '–'}*, ***kwargs*)

Create a quick-look plot of Dst (if variable present in file) and compare against observations.

Like all *add_* * methods in Pybats, the *target* kwarg determines where to place the plot. If kwarg *target* is **None** (default), a new figure is generated from scratch. If *target* is a matplotlib Figure object, a new axis is created to fill that figure at subplot location *loc* (defaults to 111). If target is a matplotlib Axes object, the plot is placed into that axis at subplot location *loc*.

With newer versions of BATS-R-US, new dst-like variables are included, named 'dst', 'dst-sm', 'dstflx', etc. This subroutine will attempt to first use 'dst-sm' as it is calculated consistently with observations. If not found, 'dst' is used. Users may choose which value to use via the *dstvar* kwarg.

Observed Dst and SYM-H is automatically fetched from the Kyoto World Data Center via the *`spacepy.pybats.kyoto`* module. The associated `spacepy.pybats.kyoto.KyotoDst` or `spacepy.pybats.kyoto.KyotoSym` object, which holds the observed Dst/SYM-H, is stored as *self.obs_dst* for future use. The observed line can be customized via the *obs_kwargs* kwarg, which is a dictionary of plotting keyword arguments.

If kwarg *epoch* is set to a datetime object, a vertical dashed line will be placed at that time.

The figure and axes objects are returned to the user.

spacepy.pybats.bats.Stream

class `spacepy.pybats.bats.`**`Stream`**(*bats*, *xstart*, *ystart*, *xfield*, *yfield*, *style='mag'*, *type='streamline'*, *method='rk4'*, *var_list='all'*, *extract=False*, *maxPoints=20000*, **args*, ***kwargs*)

A class for streamlines. Contains all of the information about the streamline, including extracted variables.

Upon instantiation, the object will trace through the vector field determined by the "[x/y]field" values and the Bats object "bats".

Parameters **bats** : Bats

`Bats2d` object through which to trace.

xstart : float

X value of location to start the trace.

ystart : float

Y value of location to start the trace.

xfield : str

Name of variable in `bats` which contains X values of the field

yfield : str

Name of variable in `bats` which contains Y values of the field

Other Parameters **style** : str

Sets line style, including colors. See `set_style()` for details. (Default 'mag')

type : str

(Default 'streamline')

method : str

Integration method. The default is Runge-Kutta 4 ('rk4') which gives a good blend of speed and accuracy. See the test functions in `trace2d` for more info. The other option is a simple Euler's method approach ('eul'). (Default 'rk4')

extract : bool

(Default: False) Extract variables along stream trace and save within object.

maxPoints : int

(Default : 20000) Maximum number of integration steps to take.

var_list : string or sequence of strings

(Default : 'all') List of values to extract from *dataset*. Defaults to 'all', for all values within *bats*.

Notes

Methods

`set_style`(style)	Set the line style either using a simple matplotlib-type style string or using a preset style type.
`treetrace`(bats[, maxPoints])	Trace through the vector field using the quad tree.
`trace`(bats)	Trace through the vector field.
`plot`(ax, *args, **kwargs)	Add streamline to axes object "ax".

`set_style`(*style*)
Set the line style either using a simple matplotlib-type style string or using a preset style type. Current types include:

'mag' [treat line as a magnetic field line. Closed lines are] white, other lines are black.

`treetrace`(*bats*, *maxPoints=20000*)
Trace through the vector field using the quad tree.

`trace`(*bats*)
Trace through the vector field.

`plot`(*ax*, **args*, ***kwargs*)
Add streamline to axes object "ax".

spacepy.pybats.bats.Bats2d

class `spacepy.pybats.bats.`**`Bats2d`**(*filename*, *format='binary'*)
A child class of `IdlFile` taylored to BATS-R-US output.

spacepy.pybats.bats.Mag

class `spacepy.pybats.bats.`**`Mag`**(*nlines*, *time*, *gmvars=()*, *ievars=()*, **args*, ***kwargs*)

A container for data from a single BATS-R-US virtual magnetometer. These work just like a typical *`spacepy.pybats.PbData`* object. Beyond raw magnetometer data, additional values are calculated and stored, including total pertubations (the sum of all global and ionospheric pertubations as measured by the magnetometer). Users will be interested in methods `add_comp_plot()` and `calc_dbdt()`.

Instantiation is best done through :class: *spacepy.pybats.MagFile* objects, which load and parse organize many virtual magnetometers from a single output file into a single object. However, they can be created manually, though painfully. Users must instantiate by handing the new object the number of lines that will be parsed (rather, the number of data points that will be needed), a time vector, and (optionally) the list of variables coming from the GM and IE module. While the latter two are keyword arguments, at least one should be provided. Next, the arrays whose keys were given by the *gmvars* and *ievars* keyword arguments in the instantiation step can either be filled manually or by using the `parse_gmline()` and `parse_ieline()` methods to parse lines of ascii data from a magnetometer output file. Finally, the `recalc()` method should be called to calculate total perturbation.

spacepy.pybats.bats.MagFile

class `spacepy.pybats.bats.`**`MagFile`**(*filename*, *ie_name=None*, *find_ie=False*, **args*, ***kwargs*)

BATS-R-US magnetometer files are powerful tools for both research and operations. *`MagFile`* objects open, parse, and visualize such output.

The $delta B$ calculated by the SWMF requires two components: GM (BATSRUS) and IE (Ridley_serial). The data is spread across two files: GM_mag*.dat and IE_mag*.dat. The former contains $delta B$ caused by gap-region (i.e., inside the inner boundary) FACs and the changing global field. The latter contains the $delta B$ caused by Pederson and Hall currents in the ionosphere. *`MagFile`* objects can open one or both of these files at a time; when both are opened, the total $delta B$ is calculated and made available to the user.

Usage:

```
>>> # Open up the GM magnetometer file only.
>>> obj = spacepy.pybats.bats.MagFile('GM_file.mag')
>>>
>>> # Open up both the GM and IE file [LEGACY SWMF ONLY]
>>> obj = spacepy.pybats.bats.MagFile('GM_file.mag', 'IE_file.mag')
>>>
>>> # Open up the GM magnetometer file; search for the IE file.
>>> obj = spacepy.pybats.bats.MagFile('GM_file.mag', find_ie=True)
```

Note that the **find_ie** kwarg uses a simple search assuming the data remain in a typical SWMF-output organizational tree (i.e., if the results of a simulation are in folder *results*, the GM magnetometer file can be found in *results/GM/* or *results/GM/IO2/* while the IE file can be found in *results/IE/* or *results/IE/ionosphere/*). It will also search the present working directory. This method is not robust; the user must take care to ensure that the two files correspond to each other.

spacepy.pybats.bats.GeoIndexFile

class `spacepy.pybats.bats.`**`GeoIndexFile`**(*filename*, *keep_case=True*, **args*, ***kwargs*)

Geomagnetic Index files are a specialized BATS-R-US output that contain geomagnetic indices calculated from simulated ground-based magnetometers. Currently, the only index instituted is Kp through the faKe_p setup. Future work will expand the system to include Dst, AE, etc.

GeoIndFiles are a specialized subclass of pybats.LogFile. It includes additional methods to quickly visualize the output, perform data-model comparisons, and more.

spacepy.pybats.bats.VirtSat

class `spacepy.pybats.bats.`**`VirtSat`**`(`**args*, ***kwargs*`)`

A *`spacepy.pybats.LogFile`* object tailored to virtual satellite output; includes special satellite-specific plotting methods.

spacepy.pybats.dgcpm

The PyBats submodule for handling input and output for the Dynamic Global Core Plasma Model (DGCPM), a plasmasphere module of the SWMF.

spacepy.pybats.dipole

Some functions for the generation of a dipole field.

Copyright 2010 Los Alamos National Security, LLC.

spacepy.pybats.gitm

PyBats submodule for handling input/output for the Global Ionosphere-Thermosphere Model (GITM), one of the choices for the UA module in the SWMF.

spacepy.pybats.kyoto

kyoto is a tool set for obtaining and handling geomagnetic indices stored at the Kyoto World Data Center (WDC) website. Indices can be loaded from file or fetched from the web.

Instantiation of objects from this module should be done through the constructor functions `fetch()` and `load()`. Use help on these objects for more information.

spacepy.pybats.pwom

PyBats submodule for handling input/output for the Polar Wind Outflow Model (PWOM), one of the choices for the PW module in the SWMF.

spacepy.pybats.ram

A module for reading, handling, and plotting RAM-SCB output.

spacepy.pybats.rim

Classes, functions, and methods for reading, writing, and plotting output from the Ridley Ionosphere Model (RIM) and the similar legacy code, Ridley Serial.

Copyright 2010 Los Alamos National Security, LLC.

spacepy.pybats.trace2d

A set of routines for fast field line tracing. "Number crunching" is performed in C for speed.

Copyright 2010-2014 Los Alamos National Security, LLC.

Top-Level Classes & Functions

Top-level PyBats classes handle common-format input and output from the SWMF and are very flexible. However, they do little beyond open files for the user.

There are several functions found in the top-level module. These are mostly convenience functions for customizing plots.

Classes

`IdlFile`(filename[, format, header, keep_case])	An object class that reads/parses an IDL-formatted output file from the SWMF ar
`ImfInput`([filename, load, npoints])	A class to read, write, manipulate, and visualize solar wind upstream input files fo
`LogFile`(filename[, starttime, keep_case])	An object to read and handle SWMF-type logfiles.
`NgdcIndex`([filename, load])	Many models incorporated into the SWMF rely on National Geophysical Data Ce
`PbData`(*args, **kwargs)	The base class for all PyBats data container classes.
`SatOrbit`([filename])	An class to load, read, write, and handle BATS-R-US satellite orbit input files.

spacepy.pybats.IdlFile

class `spacepy.pybats.`**`IdlFile`**(*filename*, *format=None*, *header='units'*, *keep_case=True*, **args*, ***kwargs*)

An object class that reads/parses an IDL-formatted output file from the SWMF and places it into a `spacepy.pybats.PbData` object.

Usage: >>>data = spacepy.pybats.IdlFile('binary_file.out')

See `spacepy.pybats.PbData` for information on how to explore data contained within the returned object.

This class serves as a parent class to SWMF component-specific derivative classes that do more preprocessing of the data before returning the object. Hence, using this class to read binary files is typically not the most efficient way to proceed. Look for a PyBats sub module that suits your specific needs, or use this base object to write your own.

A note on byte-swapping: PyBats assumes little endian byte ordering because this is what most machines use. However, there is an autodetect feature such that, if PyBats doesn't make sense of the first read (a record length entry, or RecLen), it will proceed using big endian ordering. If this doesn't work, the error will manifest itself through the "struct" package as an "unpack requires a string of argument length 'X'".

spacepy.pybats.ImfInput

class `spacepy.pybats.`**`ImfInput`**(*filename=False*, *load=True*, *npoints=0*, **args*, ***kwargs*)

A class to read, write, manipulate, and visualize solar wind upstream input files for SWMF simulations. More about such files can be found in the SWMF/BATS-R-US documentation for the #SOLARWINDFILE command.

Creating an `ImfInput` object is simple:

```
>>> from spacepy import pybats
>>> obj=pybats.ImfInput(filename='test.dat', load=True)
```

Upon instantiation, if *filename* is a valid file AND kwarg *load* is set to boolean True, the contents of *filename* are loaded into the object and no other work needs to be done.

If *filename* is False or *load* is False, a blank `ImfInput file` is created for the user to manipulate. The user can set the time array and the associated data values (see *obj.attrs['var']* for a list) to any values desired and use the method *obj.write()* to dump the contents to an SWMF formatted input file. See the documentation for the write method for more details.

Like most `pybats` objects, you may interact with `ImfInput` objects as if they were specialized dictionaries. Access data like so:

```
>>> obj.keys()
['bx', 'by', 'bz', 'vx', 'vy', 'vz', 'rho', 'temp']
>>> density=obj['rho']
```

Adding new data entries is equally simple so long as you have the values and the name for the values:

```
>>> import numpy as np
>>> v = np.sqrt(obj['vx']**2 + obj['vy']**2 + obj['vz']**2)
>>> obj['v']=v
```

Kwarg	Description
filename	Set the input/output file name.
load	Read file upon instantiation? Defaults to **True**
npoints	For empty data sets, sets number of points (default is 0)

spacepy.pybats.LogFile

class `spacepy.pybats.`**`LogFile`**(*filename*, *starttime=(2000, 1, 1, 0, 0, 0)*, *keep_case=True*, **args*, ***kwargs*)

An object to read and handle SWMF-type logfiles.

LogFile objects read and hold all information in an SWMF ascii time-varying logfile. The file is read upon instantiation. Many SWMF codes produce flat ascii files that can be read by this class; the most frequently used ones have their own classes that inherit from this.

See `spacepy.pybats.PbData` for information on how to explore data contained within the returned object.

Usage: >>>data = spacepy.pybats.LogFile('filename.log')

kwarg	Description
starttime	Manually set the start time of the data.

Time is handled by Python's datetime package. Given that time may or may not be given in the logfile, there are three options for how time is returned:

1. if the full date and time are listed in the file, `self['time']` is an array of datetime objects corresponding to the entries. The starttime kwarg is ignored.

2. If only the runtime (seconds from start of simulation) is given, self['time'] is an array of datetime objects that starts from the given *starttime* kwarg which defaults to 1/1/1 00:00UT.

3. If neither of the above are given, the time is assumed to advance one second per line starting from either the starttime kwarg or from 2000/1/1 00:00UT + the first iteration (if given in file.) As you can imagine, this is sketchy at best.

This time issue is output dependent: some SWMF models place the full date and time into the log by default while others will almost never include the full date and time. The variable `self['runtime']` contains the more generic seconds from simulation start values.

Example usage:

```
>>> import spacepy.pybats as pb
>>> import pylab as plt
>>> import datetime as dt
>>> time1 = dt.datetime(2009,11,30,9,0)
>>> file1 = pb.logfile('satfile_n000000.dat', starttime=time1)
>>> plt.plot(file1['time'], file1['dst'])
```

spacepy.pybats.NgdcIndex

class `spacepy.pybats.`**`NgdcIndex`**(*filename=None*, *load=True*, **args*, ***kwargs*)

Many models incorporated into the SWMF rely on National Geophysical Data Center (NGDC) provided index files (especially F10.7 and Kp). These files, albeit simple ascii, have a unique format and expansive header that can be cumbersome to handle. Data files can be obtained from http://spidr.ngdc.noaa.gov .

NgdcIndex objects aid in reading, creating, and visualizing these files.

Creating an `NgdcIndex` object is simple:

```
>>> from spacepy import pybats
>>> obj=pybats.NgdcIndex(filename='ngdc_example.dat')
```

Upon instantiation, if *filename* is a valid file AND kwarg *load* is set to boolean True, the contents of *filename* are loaded into the object and no other work needs to be done.

If *filename* is False or *load* is False, a blank `NgdcIndex` is created for the user to manipulate. The user can set the time array and the ssociated data values to any values desired and use the method *obj.write()* to dump the contents to an NGDC formatted input file. See the documentation for the write method for more details.

This class is a work-in-progress. It is especially tuned for SWMF-needs and cannot be considered a general function for the handling of generic NGDC files.

Kwarg	Description
filename	Set the input/output file name.
load	Read file upon instantiation? Defaults to **True**

spacepy.pybats.PbData

class `spacepy.pybats.`**`PbData`**(**args*, ***kwargs*)

The base class for all PyBats data container classes. Inherits from `spacepy.datamodel.SpaceData` but has additional methods for quickly exploring an SWMF dataset.

Just like `spacepy.datamodel.SpaceData` objects, *PbData* objects work just like dictionaries except they have special **attr** dictionary attributes for both the top-level object and most values. This means that the following syntax can be used to explore a generic *PbData* object:

>>>print obj.keys() >>>print obj.attrs >>>value = obj[key]

Printing *PbData* objects will produce a tree of contents and attributes; calling `self.listunits()` will print all values that have the 'units' attribute and the associated units. Hence, it is often most instructive to use the following two lines to quickly learn a *PbData*'s contents:

>>>print obj >>>obj.listunits()

PbData is the main organizational tool for Pybats datasets, so the information here is applicable to nearly all Pybats classes.

spacepy.pybats.SatOrbit

class `spacepy.pybats.`**`SatOrbit`**(*filename=None*, **args*, ***kwargs*)

An class to load, read, write, and handle BATS-R-US satellite orbit input files. These files are used to fly virtual satellites through the MHD domain. Note that the output files should be handled by the *LogFile* and not this satorbit object. The object's required and always present attributes are:

Attribute	Description
head	A list of header lines for the file that contain comments.
coor	The three-letter code (see SWMF doc) of the coord system.
file	Location of the file to read/write.

The object should always have the following two data keys:

Key	Description
time	A list or numpy vector of datetime objects
xyz	A 3 x len(time) numpy array of x,y,z coordinates associated with the time vector.

A "blank" instantiation will create an empty object for the user to fill. This is desired if the user wants to create a new orbit, either from data or from scratch, and save it in a properly formatted file. Here's an example with a "stationary probe" type orbit where the attributes are filled and then dropped to file:

```
>>> from spacepy.pybats import SatOrbit
>>> import datetime as dt
>>> import numpy as np
>>> sat = SatOrbit()
>>> sat['time'] = [ dt.datetime(2000,1,1), dt.datetime(2000,1,2) ]
>>> pos = np.zeros( (3,2) )
>>> pos[:,0]=[6.6, 0, 0]
>>> pos[:,1]=[6.6, 0, 0]
>>> sat['xyz'] = pos
>>> sat.attrs['coor'] = 'SMG'
>>> sat.attrs['file'] = 'noon_probe.dat'
>>> sat.write()
```

If instantiated with a file name, the name is loaded into the object. For example,

```
>>> sat=SatOrbit('a_sat_orbit_file.dat')
```

...will populate all fields with the data in *a_sat_orbit_file.dat*.

Functions

add_body(ax[, rad, facecolor, show_planet, ...])	Creates a circle of radius=self.attrs['rbody'] and returns the MatPlotLib Ellipse pa
add_planet(ax[, rad, ang, add_night, zorder])	Creates a circle of `radius=self.para['rbody']` and returns the MatPlotL
parse_tecvars(line)	Parse the VARIABLES line from a TecPlot-formatted ascii data file.

spacepy.pybats.add_body

`spacepy.pybats.`**`add_body`**(*ax*, *rad=2.5*, *facecolor='lightgrey'*, *show_planet=True*, *ang=0.0*, *add_night=True*, *zorder=1000*, ***extra_kwargs*)

Creates a circle of radius=self.attrs['rbody'] and returns the MatPlotLib Ellipse patch object for plotting. If an axis is specified using the "ax" keyword, the patch is added to the plot. Default color is light grey; extra keywords are handed to the Ellipse generator function.

Because the body is rarely the size of the planet at the center of the modeling domain, add_planet is automatically called. This can be negated by using the show_planet kwarg.

Parameters **ax** : Matplotlib Axes object

Set the axes on which to place planet.

Other Parameters **rad** : float

Set radius of the inner boundary. Defaults to 2.5.

facecolor : string

Set color of face of inner boundary circle via Matplotlib color selectors (name, hex, etc.) Defaults to 'lightgrey'.

show_planet : boolean

Turns on/off planet indicator inside inner boundary. Defaults to **True**

ang : float

Set the rotation of the day-night terminator from the y-axis, in degrees. Defaults to zero (terminator is aligned with Y-axis.)

add_night : boolean

Add night hemisphere. Defaults to **True**

zorder : int

Set the matplotlib zorder of the patch to set how other plot elements order with the inner boundary patch. Defaults to 1000. If a planet is added, it is given a zorder of *zorder*+5.

spacepy.pybats.add_planet

`spacepy.pybats.`**`add_planet`**(*ax*, *rad=1.0*, *ang=0.0*, *add_night=True*, *zorder=1000*, ***extra_kwargs*)

Creates a circle of `radius=self.para['rbody']` and returns the MatPlotLib Ellipse patch object for plotting. If an axis is specified using the *ax* keyword, the patch is added to the plot.

Unlike the add_body method, the circle is colored half white (dayside) and half black (nightside) to coincide with the direction of the sun. Additionally, because the size of the planet is not intrinsically known to the MHD file, the kwarg "rad", defaulting to 1.0, sets the size of the planet. *add_night* can turn off this behavior.

Extra keywords are handed to the Ellipse generator function.

Parameters **ax** : Matplotlib Axes object

Set the axes on which to place planet.

Other Parameters **rad** : float

Set radius of planet. Defaults to 1.

ang : float

Set the rotation of the day-night terminator from the y-axis, in degrees. Defaults to zero (terminator is aligned with Y-axis.)

add_night : boolean

Add night hemisphere. Defaults to **True**

zorder : int

Set the matplotlib zorder of the patch to set how other plot elements order with the inner boundary patch. Defaults to 1000, nightside patch is given zorder of *zorder+5*.

spacepy.pybats.parse_tecvars

`spacepy.pybats.parse_tecvars(line)`

Parse the VARIABLES line from a TecPlot-formatted ascii data file. Create a list of name-unit tuples for each variable.

pycdf - Python interface to CDF files

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 'pythonic' rather than reproducing the C interface. To open or close a CDF and access its variables, see the `CDF` class. Accessing data within the variables is via the `Var` class. The `lib` object provides access to some routines that affect the functionality of the library in general. The `const` module contains constants useful for accessing the underlying library.

The CDF C library must be properly installed in order to use this package. The CDF distribution provides scripts meant to be called in a user's login scripts, `definitions.B` for bash and `definitions.C` for C-shell derivatives. (See the installation instructions which come with the CDF library.) These will set environment variables specifying the location of the library; pycdf will respect these variables if they are set. Otherwise it will search the standard system library path and the default installation locations for the CDF library.

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.environ["CDF_LIB"] = "~/CDF/lib"
>>> from spacepy import pycdf
```

If this works, make the environment setting permanent. Note that on OSX, using plists to set the environment may not carry over to Python terminal sessions; use `.cshrc` or `.bashrc` instead.

Authors: Jon Niehof

Institution: University of New Hampshire

Contact: Jonathan.Niehof@unh.edu

Contents

- *Class reference*
- *Submodules*

Quickstart

Create a CDF

This example presents the entire sequence of creating a CDF and populating it with some data; the parts are explained individually below.

```
>>> from spacepy import pycdf
>>> import datetime
>>> time = [datetime.datetime(2000, 10, 1, 1, val) for val in range(60)]
>>> import numpy as np
>>> 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.

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

Make a data set of `datetime`. These will be 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` objects are degraded to CDF_EPOCH (millisecond resolution), not CDF_EPOCH16 (microsecond resolution).

Create some random data.

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

Create a new empty CDF. The empty string, ‘’, is the name of the CDF to use as a master; given an empty string, an empty CDF will be created, rather than copying from a master CDF. If a master is used, data in the master will be copied to the new CDF.

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

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

To put data into a CDF, assign it directly to an element of the CDF. CDF objects behave like Python dictionaries.

```
>>> # put time into CDF variable Epoch
>>> cdf['Epoch'] = time
>>> # and the same with data (the smallest data type that fits the data is used by default)
>>> cdf['data'] = data
```

Adding attributes is done similarly. CDF attributes are also treated as dictionaries.

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

Closing the CDF ensures the new data are written to disk:

```
>>> 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 very similar: the CDF object behaves like a dictionary. The file is only accessed when data are requested. A full example using the above CDF:

```
>>> 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, similar to a Python list.

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

`CDF.copy()` will return the entire contents of a CDF, including attributes, as a *`SpaceData`* object:

```
>>> 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']
```

Close the CDF when finished:

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

Modify a CDF

An example modifying the CDF created above:

```
>>> 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()
```

As before, each step in this example will now be individually explained. Existing CDF files are opened in read-only mode and must be set to read-write before modification:

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

Then new variables can be added

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

Or contents can be changed

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

The new variables appear immediately:

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

Closing the CDF ensures changes are written to disk:

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

Non record-varying

Non record-varying (NRV) variables are usually used for data that does not vary with time, such as the energy channels for an instrument.

NRV variables need to be created with `CDF.new()`, specifying the keyword 'recVary' as False.

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('MyCDF2.cdf', '')
>>> cdf.new('data2', [1], recVary=False)
    <Var:
```

```
    CDF_BYTE [1] NRV
    >
>>> cdf['data2'][...]
    [1]
```

Slicing and indexing

Subsets of data in a variable can be easily referenced with Python's slicing and indexing notation.

This example uses `bisect` to read a subset of the data from the hourly data file created in earlier examples.

```
>>> 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()
```

The `Var` documentation has several additional examples.

Access to CDF constants and the C library

Constants defined in cdf.h and occasionally useful in accessing CDFs are available in the `const` module.

The underlying C library is represented by the `lib` variable.

Class reference

`CDF`(pathname[, masterpath, create, readonly])	Python object representing a CDF file.
`Var`(cdf_file, var_name, *args)	A CDF variable.
`gAttrList`(cdf_file[, special_entry])	Object representing *all* the gAttributes in a CDF.
`zAttrList`(zvar)	Object representing *all* the zAttributes in a zVariable.
`zAttr`(cdf_file, attr_name[, create])	zAttribute for zVariables within a CDF.
`gAttr`(cdf_file, attr_name[, create])	Global Attribute for a CDF
`AttrList`(cdf_file[, special_entry])	Object representing a list of attributes.
`Attr`(cdf_file, attr_name[, create])	An attribute, g or z, for a CDF
`Library`([libpath, library])	Abstraction of the base CDF C library and its state.
`CDFCopy`(cdf)	A dictionary-like copy of all data and attributes in a `CDF`
`VarCopy`	A list-like copy of the data and attributes in a `Var`
`CDFError`(status)	Raised for an error in the CDF library.
`CDFException`(status)	Base class for errors or warnings in the CDF library.
`CDFWarning`(status)	Used for a warning in the CDF library.
`EpochError`	Used for errors in epoch routines

spacepy.pycdf.CDF

class `spacepy.pycdf.CDF`(*pathname*, *masterpath=None*, *create=None*, *readonly=None*)

Python object representing a CDF file.

Open or create a CDF file by creating an object of this class.

Parameters **pathname** : string

name of the file to open or create

masterpath : string

name of the master CDF file to use in creating a new file. If not provided, an existing file is opened; if provided but evaluates to `False` (e.g., `''`), an empty new CDF is created.

create : bool

Create a new CDF even if masterpath isn't provided

readonly : bool

Open the CDF read-only. Default True if opening an existing CDF; False if creating a new one. A readonly CDF with many variables may be slow to close. See `readonly()`.

Raises **CDFError**

if CDF library reports an error

Warns **CDFWarning**

if CDF library reports a warning and interpreter is set to error on warnings.

Examples

Open a CDF by creating a CDF object, e.g.:

```
>>> cdffile = pycdf.CDF('cdf_filename.cdf')
```

Be sure to `close()` or `save()` when done.

Note: Existing CDF files are opened read-only by default, see `readonly()` to change.

CDF supports the with keyword, like other file objects, so:

```
>>> with pycdf.CDF('cdf_filename.cdf') as cdffile:
...     #do brilliant things with the CDF
```

will open the CDF, execute the indented statements, and close the CDF when finished or when an error occurs. The python docs include more detail on this 'context manager' ability.

CDF objects behave like a python dictionary, where the keys are names of variables in the CDF, and the values, `Var` objects. As a dictionary, they are also iterable and it is easy to loop over all of the variables in a file. Some examples:

1.List the names of all variables in the open CDF `cdffile`:

```
>>> cdffile.keys()
>>> for k in cdffile: #Alternate
...     print(k)
```

2. Get a `Var` object for the variable named `Epoch`:

```
>>> epoch = cdffile['Epoch']
```

3. Determine if a CDF contains a variable named `B_GSE`:

```
>>> if 'B_GSE' in cdffile:
...     print('B_GSE is in the file')
... else:
...     print('B_GSE is not in the file')
```

4. Find how many variables are in the file:

```
>>> print(len(cdffile))
```

5. Delete the variable `Epoch` from the open CDF file `cdffile`:

```
>>> del cdffile['Epoch']
```

6. Display a summary of variables and types in open CDF file `cdffile`:

```
>>> print(cdffile)
```

7. Open the CDF named `cdf_filename.cdf`, read *all* the data from all variables into dictionary `data`, and close it when done or if an error occurs:

```
>>> with pycdf.CDF('cdf_filename.cdf') as cdffile:
...     data = cdffile.copy()
```

This last example can be very inefficient as it reads the entire CDF. Normally it's better to treat the CDF as a dictionary and access only the data needed, which will be pulled transparently from disc. See `Var` for more subtle examples.

Potentially useful dictionary methods and related functions:

- in
- keys
- `len()`
- list comprehensions
- `sorted()`
- `dictree()`

The CDF user's guide section 2.2 has more background information on CDF files.

The `attrs` Python attribute acts as a dictionary referencing CDF attributes (do not confuse the two); all the dictionary methods above also work on the attribute dictionary. See `gAttrList` for more on the dictionary of global attributes.

Creating a new CDF from a master (skeleton) CDF has similar syntax to opening one:

```
>>> cdffile = pycdf.CDF('cdf_filename.cdf', 'master_cdf_filename.cdf')
```

This creates and opens `cdf_filename.cdf` as a copy of `master_cdf_filename.cdf`.

Using a skeleton CDF is recommended over making a CDF entirely from scratch, but this is possible by specifying a blank master:

```
>>> cdffile = pycdf.CDF('cdf_filename.cdf', '')
```

When CDFs are created in this way, they are opened read-write, see `readonly()` to change.

By default, new CDFs (without a master) are created in version 2 (backward-compatible) format. To create a version 3 CDF, use `Library.set_backward()`:

```
>>> pycdf.lib.set_backward(False)
>>> cdffile = pycdf.CDF('cdf_filename.cdf', '')
```

Add variables by direct assignment, which will automatically set type and dimension based on the data provided:

```
>>> cdffile['new_variable_name'] = [1, 2, 3, 4]
```

or, if more control is needed over the type and dimensions, use `new()`.

Although it is supported to assign Var objects to Python variables for convenience, there are some minor pitfalls that can arise when changing a CDF that will not affect most users. This is only a concern when assigning a zVar object to a Python variable, changing the CDF through some other variable, and then trying to use the zVar object via the originally assigned variable.

Deleting a variable:

```
>>> var = cdffile['Var1']
>>> del cdffile['Var1']
>>> var[0] #fail, no such variable
```

Renaming a variable:

```
>>> var = cdffile['Var1']
>>> cdffile['Var1'].rename('Var2')
>>> var[0] #fail, no such variable
```

Renaming via the same variable works:

```
>>> var = cdffile['Var1']
>>> var.rename('Var2')
>>> var[0] #succeeds, aware of new name
```

Deleting a variable and then creating another variable with the same name may lead to some surprises:

```
>>> var = cdffile['Var1']
>>> var[...] = [1, 2, 3, 4]
>>> del cdffile['Var1']
>>> cdffile.new('Var1', data=[5, 6, 7, 8]
>>> var[...]
[5, 6, 7, 8]
```

`attr_num`(attrname)	Get the attribute number and scope by attribute name
`attrs`	Global attributes for this CDF in a dict-like format.
`add_attr_to_cache`(attrname, num, scope)	Add an attribute to the name-to-number cache
`add_to_cache`(varname, num)	Add a variable to the name-to-number cache
`CDF.backward`	
`checksum`([new_val])	Set or check the checksum status of this CDF.
`clear_attr_from_cache`(attrname)	Mark an attribute deleted in the name-to-number cache
`clear_from_cache`(varname)	Mark a variable deleted in the name-to-number cache

Continued on next page

Table 2.32 – continued from previous page

clone(zVar[, name, data])	Clone a zVariable (from another CDF or this) into this CDF
close()	Closes the CDF file
col_major([new_col])	Finds the majority of this CDF file
compress([comptype, param])	Set or check the compression of this CDF
copy()	Make a copy of all data and attributes in this CDF
from_data(filename, sd)	Create a new CDF file from a SpaceData object or similar
new(name[, data, type, recVary, dimVarys, ...])	Create a new zVariable in this CDF
raw_var(name)	Get a "raw" *Var* object.
readonly([ro])	Sets or check the readonly status of this CDF
save()	Saves the CDF file but leaves it open.
var_num(varname)	Get the variable number of a particular variable name
version()	Get version of library that created this CDF

attrs
Global attributes for this CDF in a dict-like format. See *gAttrList* for details.

backward
True if this CDF was created in backward-compatible mode (for opening with CDF library before 3.x)

add_to_cache(*varname*, *num*)
Add a variable to the name-to-number cache

This maintains a cache of name-to-number mappings for zVariables to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters **varname** : bytes

name of the zVariable. Not this is NOT a string in Python 3!

num : int

number of the variable

add_attr_to_cache(*attrname*, *num*, *scope*)
Add an attribute to the name-to-number cache

This maintains a cache of name-to-number mappings for attributes to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters **varname** : bytes

name of the zVariable. Not this is NOT a string in Python 3!

num : int

number of the variable

scope : bool

True if global scope; False if variable scope.

attr_num(*attrname*)
Get the attribute number and scope by attribute name

This maintains a cache of name-to-number mappings for attributes to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters **attrname** : bytes

name of the zVariable. Not this is NOT a string in Python 3!

Returns **out** : tuple

attribute number, scope (True for global) of this attribute

Raises CDFError : if variable is not found

checksum(*new_val=None*)

Set or check the checksum status of this CDF. If checksums are enabled, the checksum will be verified every time the file is opened.

Returns out : boolean

True if the checksum is enabled or False if disabled

Other Parameters new_val : boolean

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

clear_from_cache(*varname*)

Mark a variable deleted in the name-to-number cache

Will remove a variable, and all variables with higher numbers, from the variable cache.

Does NOT delete the variable!

This maintains a cache of name-to-number mappings for zVariables to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters varname : bytes

name of the zVariable. Not this is NOT a string in Python 3!

clear_attr_from_cache(*attrname*)

Mark an attribute deleted in the name-to-number cache

Will remove an attribute, and all attributes with higher numbers, from the attribute cache.

Does NOT delete the variable!

This maintains a cache of name-to-number mappings for attributes to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters attrname : bytes

name of the attribute. Not this is NOT a string in Python 3!

clone(*zVar*, *name=None*, *data=True*)

Clone a zVariable (from another CDF or this) into this CDF

Parameters zVar : `Var`

variable to clone

Returns out : `Var`

The newly-created zVar in this CDF

Other Parameters name : str

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

data : boolean (optional)

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

close()

Closes the CDF file

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

Raises CDFError : if CDF library reports an error

Warns CDFWarning : if CDF library reports a warning

`col_major`(*new_col=None*)

Finds the majority of this CDF file

Returns out : boolean

True if column-major, false if row-major

Other Parameters new_col : boolean

Specify True to change to column-major, False to change to row major, or do not specify to check the majority rather than changing it. (default is check only)

`compress`(*comptype=None*, *param=None*)

Set or check the compression of this CDF

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

See section 2.6 of the CDF user's guide for more information on compression.

Returns out : tuple

(comptype, param) currently in effect

Other 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 `const`. If not specified, will not change compression.

param : ctypes.c_long

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

See also:

`Var.compress()`

Examples

Set file `cdffile` to gzip compression, compression level 9:

```
>>> cdffile.compress(pycdf.const.GZIP_COMPRESSION, 9)
```

`copy`()

Make a copy of all data and attributes in this CDF

Returns out : `CDFCopy`

`SpaceData`-like object of all data

classmethod `from_data`(*filename*, *sd*)

Create a new CDF file from a SpaceData object or similar

The CDF named `filename` is created, opened, filled with the contents of `sd` (including attributes), and closed.

`sd` should be a dictionary-like object; each key will be made into a variable name. An attribute called `attrs`, if it exists, will be made into global attributes for the CDF.

Each value of `sd` should be array-like and will be used as the contents of the variable; an attribute called `attrs`, if it exists, will be made into attributes for that variable.

Parameters **filename** : string

name of the file to create

sd : spacepy.datamodel.SpaceData

data to put in the CDF. This structure cannot be nested, i.e., it must contain only *`dmarray`* and no `Spacedata` objects.

`new`(*name*, *data=None*, *type=None*, *recVary=True*, *dimVarys=None*, *dims=None*, *n_elements=None*, *compress=None*, *compress_param=None*)

Create a new zVariable in this CDF

Note: Either `data` or `type` must be specified. If type is not specified, it is guessed from `data`.

Parameters **name** : str

name of the new variable

Returns **out** : *`Var`*

the newly-created zVariable

Other Parameters **data**

data to store in the new variable. If this has a an `attrs` attribute (e.g., *`dmarray`*), it will be used to populate attributes of the new variable.

type : ctypes.c_long

CDF type of the variable, from *`const`*. See section 2.5 of the CDF user's guide for more information on CDF data types.

recVary : boolean

record variance of the variable (default True)

dimVarys : list of boolean

dimension variance of each dimension, default True for all dimensions.

dims : list of int

size of each dimension of this variable, default zero-dimensional. Note this is the dimensionality as defined by CDF, i.e., for record-varying variables it excludes the leading record dimension. See *`Var`*.

n_elements : int

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

compress : ctypes.c_long

Compression to apply to this variable, default None. See *`Var.compress()`*.

compress_param : ctypes.c_long

Compression parameter if compression used; reasonable default is chosen. See `Var.compress()`.

Raises ValueError : if neither data nor sufficient typing information

is provided.

Notes

Any given data may be representable by a range of CDF types; if the type is not specified, pycdf will guess which the CDF types which can represent this data. This breaks down to:

1.If input data is a numpy array, match the type of that array

2.Proper kind (numerical, string, time)

3.Proper range (stores highest and lowest number provided)

4.Sufficient resolution (EPOCH16 required if datetime has microseconds or below.)

If more than one value satisfies the requirements, types are returned in preferred order:

1.Type that matches precision of data first, then

2.integer type before float type, then

3.Smallest type first, then

4.signed type first, then

5.specifically-named (CDF_BYTE) vs. generically named (CDF_INT1)

So for example, EPOCH_16 is preferred over EPOCH if `data` specifies below the millisecond level (rule 1), but otherwise EPOCH is preferred (rule 2).

For floats, four-byte is preferred unless eight-byte is required:

1.absolute values between 0 and 3e-39

2.absolute values greater than 1.7e38

This will switch to an eight-byte double in some cases where four bytes would be sufficient for IEEE 754 encoding, but where DEC formats would require eight.

`raw_var`(*name*)

Get a "raw" `Var` object.

Normally a `Var` will perform translation of values for certain types (to/from Unicode for CHAR variables on Py3k, and to/from datetime for all time types). A "raw" object does not perform this translation, on read or write.

This does *not* affect the data on disk, and in fact it is possible to maintain multiple Python objects with access to the same zVariable.

Parameters name : str

name or number of the zVariable

`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.

Note: Closing a CDF that has been opened readonly, or setting readonly False, may take a substantial amount of time if there are many variables in the CDF, as a (potentially large) cache needs to be cleared.

Consider specifying `readonly=False` when opening the file if this is an issue. However, this may make some reading operations slower.

Returns out : Boolean

True if CDF is read-only, else False

Other Parameters ro : Boolean

True to set the CDF readonly, False to set it read/write, or leave out to check only.

Raises CDFError : if bad mode is set

save()

Saves the CDF file but leaves it open.

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

Note: 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

var_num(*varname*)

Get the variable number of a particular variable name

This maintains a cache of name-to-number mappings for zVariables to keep from having to query the CDF library constantly. It's mostly an internal function.

Parameters varname : bytes

name of the zVariable. Not this is NOT a string in Python 3!

Returns out : int

Variable number of this zvariable.

Raises CDFError : if variable is not found

version()

Get version of library that created this CDF

Returns out : tuple

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

spacepy.pycdf.Var

class `spacepy.pycdf.`**Var**(*cdf_file*, *var_name*, **args*)

A CDF variable.

This object does not directly store the data from the CDF; rather, it provides access to the data in a format that much like a Python list or numpy `ndarray`. General list information is available in the python docs: 1, 2, 3.

The CDF user's guide, section 2.3, provides background on variables.

Note: Not intended to be created directly; use methods of `CDF` to gain access to a variable.

A record-varying variable's data are viewed as a hypercube of dimensions n_dims+1 (the extra dimension is the record number). They are indexed in row-major fashion, i.e. the last index changes most frequently / is contiguous in memory. If the CDF is column-major, the data are transformed to row-major before return.

Non record-varying variables are similar, but do not have the extra dimension of record number.

Variables can be subscripted by a multidimensional index to return the data. Indices are in row-major order with the first dimension representing the record number. If the CDF is column major, the data are reordered to row major. Each dimension is specified by standard Python slice notation, with dimensions separated by commas. The ellipsis fills in any missing dimensions with full slices. The returned data are lists; Python represents multidimensional arrays as nested lists. The innermost set of lists represents contiguous data.

Note: numpy 'fancy indexing' is *not* supported.

Degenerate dimensions are 'collapsed', i.e. no list of only one element will be returned if a single subscript is specified instead of a range. (To avoid this, specify a slice like 1:2, which starts with 1 and ends before 2).

Two special cases:

1. requesting a single-dimension slice for a record-varying variable will return all data for that record number (or those record numbers) for that variable.
2. Requests for multi-dimensional variables may skip the record-number dimension and simply specify the slice on the array itself. In that case, the slice of the array will be returned for all records.

In the event of ambiguity (e.g., single-dimension slice on a one-dimensional variable), case 1 takes priority. Otherwise, mismatch between the number of dimensions specified in the slice and the number of dimensions in the variable will cause an `IndexError` to be thrown.

This all sounds very complicated but it is essentially attempting to do the 'right thing' for a range of slices.

An unusual case is scalar (zero-dimensional) non-record-varying variables. Clearly they cannot be subscripted normally. In this case, use the `[...]` syntax meaning 'access all data.':

```
>>> from spacepy import pycdf
>>> testcdf = pycdf.CDF('test.cdf', '')
>>> variable = testcdf.new('variable', recVary=False,
...     type=pycdf.const.CDF_INT4)
>>> variable[...] = 10
>>> variable
<Var:
CDF_INT4 [] NRV
>
>>> variable[...]
10
```

Reading any empty non-record-varying variable will return an empty with the same *number* of dimensions, but all dimensions will be of zero length. The scalar is, again, a special case: due to the inability to have a numpy array which is both zero-dimensional and empty, reading an NRV scalar variable with no data will return an empty one-dimensional array. This is really not recommended.

As a list type, variables are also iterable; iterating over a variable returns a single complete record at a time.

This is all clearer with examples. Consider a variable `B_GSM`, with three elements per record (x, y, z components) and fifty records in the CDF. Then:

1. `B_GSM[0, 1]` is the y component of the first record.
2. `B_GSM[10, :]` is a three-element list, containing x, y, and z components of the 11th record. As a shortcut, if only one dimension is specified, it is assumed to be the record number, so this could also be

written `B_GSM[10]`.

3. `B_GSM[...]` reads all data for `B_GSM` and returns it as a fifty-element list, each element itself being a three-element list of x, y, z components.

Multidimensional example: consider fluxes stored as a function of pitch angle and energy. Such a variable may be called Flux and stored as a two-dimensional array, with the first dimension representing (say) ten energy steps and the second, eighteen pitch angle bins (ten degrees wide, centered from 5 to 175 degrees). Assume 100 records stored in the CDF (i.e. 100 different times).

1. `Flux[4]` is a list of ten elements, one per energy step, each element being a list of 18 fluxes, one per pitch bin. All are taken from the fifth record in the CDF.
2. `Flux[4, :, 0:4]` is the same record, all energies, but only the first four pitch bins (roughly, field-aligned).
3. `Flux[..., 0:4]` is a 100-element list (one per record), each element being a ten-element list (one per energy step), each containing fluxes for the first four pitch bins.

This slicing notation is very flexible and allows reading specifically the desired data from the CDF.

Note: The C CDF library allows reading records which have not been written to a file, returning a pad value. pycdf checks the size of a variable and will raise *IndexError* for most attempts to read past the end. If these checks fail, a value is returned with a warning `VIRTUAL_RECORD_DATA`. Please open an issue if this occurs. See pg. 39 and following of the CDF User's Guide for more on virtual records.

All data are, on read, converted to appropriate Python data types; EPOCH, EPOCH16, and TIME_TT2000 types are converted to `datetime`. Data are returned in numpy arrays.

Note: Although pycdf supports TIME_TT2000 variables, the Python `datetime` object does not support leap seconds. Thus, on read, any seconds past 59 are truncated to 59.999999 (59 seconds, 999 milliseconds, 999 microseconds).

Potentially useful list methods and related functions:

- count
- in
- index
- len
- list comprehensions
- sorted

The topic of array majority can be very confusing; good background material is available at IDL Array Storage and Indexing. In brief, *regardless of the majority stored in the CDF*, pycdf will always present the data in the native Python majority, row-major order, also known as C order. This is the default order in NumPy. However, packages that render image data may expect it in column-major order. If the axes seem 'swapped' this is likely the reason.

The *`attrs`* Python attribute acts as a dictionary referencing zAttributes (do not confuse the two); all the dictionary methods above also work on the attribute dictionary. See *`zAttrList`* for more on the dictionary of attributes.

With writing, as with reading, every attempt has been made to match the behavior of Python lists. You can write one record, many records, or even certain elements of all records. There is one restriction: only the record dimension (i.e. dimension 0) can be resized by write, as all records in a variable must have the same dimensions. Similarly, only whole records can be deleted.

Note: Unusual error messages on writing data usually mean that pycdf is unable to interpret the data as a regular array of a single type matching the type and shape of the variable being written. A 5x4 array is supported; an irregular array where one row has five columns and a different row has six columns is not. Error messages of this type include:

- `Data must be well-formed, regular array of number, string, or datetime`
- `setting an array element with a sequence.`
- `shape mismatch: objects cannot be broadcast to a single shape`

For these examples, assume Flux has 100 records and dimensions [2, 3].

Rewrite the first record without changing the rest:

```
>>> Flux[0] = [[1, 2, 3], [4, 5, 6]]
```

Writes a new first record and delete all the rest:

```
>>> Flux[...] = [[1, 2, 3], [4, 5, 6]]
```

Write a new record in the last position and add a new record after:

```
>>> Flux[99:] = [[[1, 2, 3], [4, 5, 6]],
...               [[11, 12, 13], [14, 15, 16]]]
```

Insert two new records between the current number 5 and 6:

```
>>> Flux[5:6] = [[[1, 2, 3], [4, 5, 6]],  [[11, 12, 13],
...                [14, 15, 16]]]
```

This operation can be quite slow, as it requires reading and rewriting the entire variable. (CDF does not directly support record insertion.)

Change the first element of the first two records but leave other elements alone:

```
>>> Flux[0:2, 0, 0] = [1, 2]
```

Remove the first record:

```
>>> del Flux[0]
```

Removes record 5 (the sixth):

```
>>> del Flux[5]
```

Due to the need to work around a bug in the CDF library, this operation can be quite slow.

Delete *all data* from `Flux`, but leave the variable definition intact:

```
>>> del Flux[...]
```

Note: Although this interface only directly supports zVariables, zMode is set on opening the CDF so rVars appear as zVars. See p.24 of the CDF user's guide; pyCDF uses zMode 2.

`attrs`	zAttributes for this zVariable in a dict-like format.
`compress`([comptype, param])	Set or check the compression of this variable
`copy`()	Copies all data and attributes from this variable
`dtype`	Provide the numpy dtype equivalent to the CDF type of this variable.
`dv`([new_dv])	Gets or sets dimension variance of each dimension of variable.
`insert`(index, data)	Inserts a *single* record before an index
`name`()	Returns the name of this variable
`rename`(new_name)	Renames this variable
`rv`([new_rv])	Gets or sets whether this variable has record variance
`shape`	Provides the numpy array-like shape of this variable.
`type`([new_type])	Returns or sets the CDF type of this variable

attrs
zAttributes for this zVariable in a dict-like format. See `zAttrList` for details.

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

Compression may not be changeable on variables with data already written; even deleting the data may not permit the change.

See section 2.6 of the CDF user's guide for more information on compression.

Returns **out** : tuple

the (comptype, param) currently in effect

Other 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 `const`. If not specified, will not change compression.

param : ctypes.c_long

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

copy()
Copies all data and attributes from this variable

Returns **out** : `VarCopy`

list of all data in record order

dtype
Provide the numpy dtype equivalent to the CDF type of this variable.

Data from this variable will be returned in numpy arrays of this type.

See also:

`type`

dv(*new_dv=None*)
Gets or sets dimension variance of each dimension of variable.

If the variance is unknown, True is assumed (this replicates the apparent behavior of the CDF library on variable creation).

Parameters **new_dv** : list of boolean

Each element True to change that dimension to dimension variance, False to change to not dimension variance. (Unspecified to simply check variance.)

Returns **out** : list of boolean

True if that dimension has variance, else false.

insert(*index*, *data*)

Inserts a *single* record before an index

Parameters **index** : int

index before which to insert the new record

data :

the record to insert

name()

Returns the name of this variable

Returns **out** : str

variable's name

rename(*new_name*)

Renames this variable

Parameters **new_name** : str

the new name for this variable

rv(*new_rv=None*)

Gets or sets whether this variable has record variance

If the variance is unknown, True is assumed (this replicates the apparent behavior of the CDF library on variable creation).

Returns **out** : Boolean

True if record varying, False if NRV

Other Parameters **new_rv** : boolean

True to change to record variance, False to change to NRV, unspecified to simply check variance.

shape

Provides the numpy array-like shape of this variable.

Returns a tuple; first element is number of records (RV variable only) And the rest provide the dimensionality of the variable.

Note: Assigning to this attribute will not change the shape.

type(*new_type=None*)

Returns or sets the CDF type of this variable

Parameters **new_type** : ctypes.c_long

the new type from `const`

Returns **out** : int

CDF type

spacepy.pycdf.gAttrList

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

Object representing *all* the gAttributes in a CDF.

Normally accessed as an attribute of an open *`CDF`*:

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

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

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

See also:

`AttrList`

spacepy.pycdf.zAttrList

class `spacepy.pycdf.`**`zAttrList`**(*zvar*)

Object representing *all* the zAttributes in a zVariable.

Normally accessed as an attribute of a *`Var`* in an open CDF:

```
>>> epoch_attribs = cdffile['Epoch'].attrs
```

Appears as a dictionary: keys are attribute names, values are the value of the zEntry associated with the appropriate zVariable. Each vAttribute in a CDF may only have a *single* entry associated with each variable. The entry may be a string, a single numerical value, or a series of numerical values. Entries with multiple values are returned as an entire list; direct access to the individual elements is not possible.

Example: finding the first dependency of (ISTP-compliant) variable `Flux`:

```
>>> print cdffile['Flux'].attrs['DEPEND_0']
```

zAttributes are shared among zVariables, one zEntry allowed per zVariable. (pyCDF hides this detail.) Deleting the last zEntry for a zAttribute will delete the underlying zAttribute.

zEntries are created and destroyed by the usual dict methods on the zAttrlist:

```
>>> epoch_attribs['new_entry'] = [1, 2, 4] #assign a list to new zEntry
>>> del epoch_attribs['new_entry'] #delete the zEntry
```

The type of the zEntry is guessed from data provided. The type is chosen to match the data; subject to that constraint, it will try to match (in order):

1. existing zEntry corresponding to this zVar
2. other zEntries in this zAttribute
3. the type of this zVar
4. data-matching constraints described in *`CDF.new()`*

See also:

`AttrList`

spacepy.pycdf.zAttr

class `spacepy.pycdf.`**`zAttr`**(*cdf_file*, *attr_name*, *create=False*)

zAttribute for zVariables within a CDF.

Warning: Because zAttributes are shared across all variables in a CDF, directly manipulating them may have unexpected consequences. It is safest to operate on zEntries via *`zAttrList`*.

Note: When accessing a zAttr, pyCDF exposes only the zEntry corresponding to the associated zVariable.

See also:

`Attr`

spacepy.pycdf.gAttr

class `spacepy.pycdf.`**`gAttr`**(*cdf_file*, *attr_name*, *create=False*)

Global Attribute for a CDF

Represents a CDF attribute, providing access to the gEntries in a format that looks like a Python list. General list information is available in the python docs: 1, 2, 3.

Normally accessed by providing a key to a *`gAttrList`*:

```
>>> attribute = cdffile.attrs['attribute_name']
>>> first_gentry = attribute[0]
```

Each element of the list is a single gEntry of the appropriate type. The index to the elements is the gEntry number.

A gEntry may be either a single string or a 1D array of numerical type. Entries of numerical type (everything but CDF_CHAR and CDF_UCHAR) with a single element are returned as scalars; multiple-element entries are returned as a list. No provision is made for accessing below the entry level; the whole list is returned at once (but Python's slicing syntax can be used to extract individual items from that list.)

Multi-dimensional slicing is *not* supported; an entry with multiple elements will have all elements returned (and can thus be sliced itself). Example:

```
>>> first_three = attribute[5, 0:3] #will fail
>>> first_three = attribute[5][0:3] #first three elements of 5th Entry
```

gEntries are *not* necessarily contiguous; a gAttribute may have an entry 0 and entry 2 without an entry 1. `len()` will return the *number* of gEntries; use *`max_idx()`* to find the highest defined gEntry number and *`has_entry()`* to determine if a particular gEntry number exists. Iterating over all entries is also supported:

```
>>> entrylist = [entry for entry in attribute]
```

Deleting gEntries will leave a "hole":

```
>>> attribute[0:3] = [1, 2, 3]
>>> del attribute[1]
>>> attribute.has_entry(1)
False
>>> attribute.has_entry(2)
True
>>> print attribute[0:3]
[1, None, 3]
```

Multi-element slices over nonexistent gEntries will return `None` where no entry exists. Single-element indices for nonexistent gEntries will raise `IndexError`. Assigning `None` to a gEntry will delete it.

When assigning to a gEntry, the type is chosen to match the data; subject to that constraint, it will try to match (in order):

1. existing gEntry of the same number in this gAttribute
2. other gEntries in this gAttribute
3. data-matching constraints described in `CDF.new()`.

See also:

`Attr`

spacepy.pycdf.AttrList

class `spacepy.pycdf.`**`AttrList`**(*cdf_file*, *special_entry=None*)

Object representing a list of attributes.

> **Warning:** This class should not be used directly, but only via its subclasses, `gAttrList` and `zAttrList`. Methods listed here are safe to use from the subclasses.

`clone`(master[, name, new_name])	Clones another attribute list, or one attribute from it, into this list.
`copy`()	Create a copy of this attribute list
`from_dict`(in_dict)	Fill this list of attributes from a dictionary
`new`(name[, data, type])	Create a new Attr in this AttrList
`rename`(old_name, new_name)	Rename an attribute in this list

`clone`(*master*, *name=None*, *new_name=None*)

Clones another attribute list, or one attribute from it, into this list.

Parameters **master** : AttrList

the attribute list to copy from. This can be any dict-like object.

Other Parameters **name** : str (optional)

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

new_name : str (optional)

name of the new attribute, default `name`

`copy`()

Create a copy of this attribute list

Returns **out** : dict

copy of the entries for all attributes in this list

`from_dict`(*in_dict*)

Fill this list of attributes from a dictionary

Deprecated since version 0.1.5: Use `clone()` instead; it supports cloning from dictionaries.

Parameters **in_dict** : dict

Attribute list is populated entirely from this dictionary; all existing attributes are deleted.

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

Parameters **name** : str

name of the new Attribute

Other Parameters **data**

data to put into the first entry in the new Attribute

type

CDF type of the first entry from *const*. Only used if data are specified.

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

spacepy.pycdf.Attr

class spacepy.pycdf.**Attr** (*cdf_file*, *attr_name*, *create=False*)
An attribute, g or z, for a CDF

Warning: This class should not be used directly, but only in its subclasses, *gAttr* and *zAttr*. The methods listed here are safe to use in the subclasses.

Represents a CDF attribute, providing access to the Entries in a format that looks like a Python list. General list information is available in the python docs: 1, 2, 3.

An introduction to CDF attributes can be found in section 2.4 of the CDF user's guide.

Each element of the list is a single Entry of the appropriate type. The index to the elements is the Entry number.

Multi-dimensional slicing is *not* supported; an Entry with multiple elements will have all elements returned (and can thus be sliced itself). Example:

```
>>> first_three = attribute[5, 0:3] #will fail
>>> first_three = attribute[5][0:3] #first three elements of 5th Entry
```

append(data)	Add an entry to end of attribute
has_entry(number)	Check if this attribute has a particular Entry number
insert(index, data)	Insert an entry at a particular number
max_idx()	Maximum index of Entries for this Attr
new(data[, type, number])	Create a new Entry in this Attribute
number()	Find the attribute number for this attribute
rename(new_name)	Rename this attribute
type(number[, new_type])	Find or change the CDF type of a particular Entry number

append (*data*)

Add an entry to end of attribute

Puts entry after last defined entry (does not fill gaps)

Parameters **data :**

data for the new entry

`has_entry`(*number*)

Check if this attribute has a particular Entry number

Parameters **number** : int

number of Entry to check or change

Returns **out** : bool

True if `number` is a valid entry number; False if not

`insert`(*index*, *data*)

Insert an entry at a particular number

Inserts entry at particular number while moving all subsequent entries to one entry number later. Does not close gaps.

Parameters **index** : int

index where to put the new entry

data :

data for the new entry

`max_idx`()

Maximum index of Entries for this Attr

Returns **out** : int

maximum Entry number

`new`(*data*, *type=None*, *number=None*)

Create a new Entry in this Attribute

Note: If `number` is provided and an Entry with that number already exists, it will be overwritten.

Parameters **data**

data to put in the Entry

Other Parameters **type** : int

type of the new Entry, from *`const`* (otherwise guessed from `data`)

number : int

Entry number to write, default is lowest available number.

`number`()

Find the attribute number for this attribute

Returns **out** : int

attribute number

rename(*new_name*)

Rename this attribute

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

Parameters new_name : str

the new name of the attribute

type(*number*, *new_type=None*)

Find or change the CDF type of a particular Entry number

Parameters number : int

number of Entry to check or change

Returns out : int

CDF variable type, see `const`

Other Parameters new_type

type to change this Entry to, from `const`. Omit to only check type.

Notes

If changing types, old and new must be equivalent, see CDF User's Guide section 2.5.5 pg. 57

spacepy.pycdf.Library

class spacepy.pycdf.**Library**(*libpath=None*, *library=None*)

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 `lib` variable, providing access to the underlying C library if necessary. The CDF library itself is described in section 2.1 of the CDF user's guide, as well as the CDF C reference manual.

Calling the C library directly requires knowledge of `ctypes`.

Instantiating this object loads the C library, see pycdf - Python interface to CDF files docs for details.

`call`(*args, **kwargs)	Call the CDF internal interface
`check_status`(status[, ignore])	Raise exception or warning based on return status of CDF call
`datetime_to_epoch`(dt)	Converts a Python datetime to a CDF Epoch value
`datetime_to_epoch16`(dt)	Converts a Python datetime to a CDF Epoch16 value
`datetime_to_tt2000`(dt)	Converts a Python datetime to a CDF TT2000 value
`epoch_to_datetime`(epoch)	Converts a CDF epoch value to a datetime
`epoch_to_epoch16`(epoch)	Converts a CDF EPOCH to a CDF EPOCH16 value
`epoch_to_num`(epoch)	Convert CDF EPOCH to matplotlib number.
`epoch_to_tt2000`(epoch)	Converts a CDF EPOCH to a CDF TT2000 value
`epoch16_to_datetime`(epoch0, epoch1)	Converts a CDF epoch16 value to a datetime
`epoch16_to_epoch`(epoch16)	Converts a CDF EPOCH16 to a CDF EPOCH value
`epoch16_to_tt2000`(epoch0, epoch1)	Converts a CDF epoch16 value to TT2000
`get_minmax`(cdftype)	Find minimum, maximum possible value based on CDF type.
`set_backward`([backward])	Set backward compatibility mode for new CDFs
`supports_int8`	
`tt2000_to_datetime`(tt2000)	Converts a CDF TT2000 value to a datetime

Continued on next page

Table 2.36 – continued from previous page

`tt2000_to_epoch`(tt2000)	Converts a CDF TT2000 value to a CDF EPOCH
`tt2000_to_epoch16`(tt2000)	Converts a CDF TT2000 value to a CDF EPOCH16
`v_datetime_to_epoch`	
`v_datetime_to_epoch16`	
`v_datetime_to_tt2000`	
`v_epoch_to_datetime`	
`v_epoch_to_tt2000`	
`v_epoch16_to_datetime`	
`v_epoch16_to_tt2000`	
`v_tt2000_to_datetime`	
`v_tt2000_to_epoch`	
`v_tt2000_to_epoch16`	
`libpath`	
`version`	Version information for NetworkX, created during installation.

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 `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 `const` module.

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 *not* be raised.

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

Warns **CDFWarning** : if CDF library reports a warning

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

Raise exception or warning based on return status of CDF call

Parameters **status** : int

status returned by the C library

Returns **out** : int

status (unchanged)

Other Parameters **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 *not* be raised. (Default none).

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

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

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

See also:

v_datetime_to_epoch

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

See also:

v_datetime_to_epoch16

datetime_to_tt2000(*dt*)

Converts a Python datetime to a CDF TT2000 value

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

date and time to convert

Returns **out** : int

tt2000 corresponding to dt

See also:

v_datetime_to_tt2000

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.

See also:

v_epoch_to_datetime

epoch_to_epoch16(*epoch*)

Converts a CDF EPOCH to a CDF EPOCH16 value

Parameters **epoch** : double

EPOCH to convert. Lists and numpy arrays are acceptable.

Returns **out** : (double, double)

EPOCH16 corresponding to epoch

epoch_to_num(*epoch*)

Convert CDF EPOCH to matplotlib number.

Same output as `date2num()` and useful for plotting large data sets without converting the times through datetime.

Parameters **epoch** : double

EPOCH to convert. Lists and numpy arrays are acceptable.

Returns **out** : double

Floating point number representing days since 0001-01-01.

epoch_to_tt2000(*epoch*)

Converts a CDF EPOCH to a CDF TT2000 value

Parameters **epoch** : double

EPOCH to convert

Returns **out** : int

tt2000 corresponding to epoch

See also:

v_epoch_to_tt2000

epoch16_to_datetime(*epoch0*, *epoch1*)

Converts a CDF epoch16 value to a datetime

Note: The call signature has changed since SpacePy 0.1.2. Formerly this method took a single argument with two values; now it requires two arguments (one for each value). To convert existing code, replace `epoch16_to_datetime(epoch)` with `epoch16_to_datetime(*epoch)`.

Parameters **epoch0** : float

epoch16 value from CDF, first half

epoch1 : float

epoch16 value from CDF, second half

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

See also:

v_epoch16_to_datetime

epoch16_to_epoch(*epoch16*)

Converts a CDF EPOCH16 to a CDF EPOCH value

Parameters **epoch16** : (double, double)

EPOCH16 to convert. Lists and numpy arrays are acceptable. LAST dimension should be 2: the two pairs of EPOCH16

Returns **out** : double

EPOCH corresponding to epoch16

`epoch16_to_tt2000`(*epoch0*, *epoch1*)

Converts a CDF epoch16 value to TT2000

Note: Because TT2000 does not support picoseconds, the picoseconds value in epoch is ignored (i.e., truncated.)

Parameters **epoch0** : float

epoch16 value from CDF, first half

epoch1 : float

epoch16 value from CDF, second half

Returns **out** : long

TT2000 corresponding to epoch.

Raises **EpochError** : if input invalid

See also:

v_epoch16_to_tt2000

`get_minmax`(*cdftype*)

Find minimum, maximum possible value based on CDF type.

This returns the processed value (e.g. datetimes for Epoch types) because comparisons to EPOCH16s are otherwise difficult.

Parameters **cdftype** : int

CDF type number from `const`

Returns **out** : tuple

minimum, maximum value supported by type (of type matching the CDF type).

Raises **ValueError** : if can't match the type

`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

`supports_int8`

True if this library supports INT8 and TIME_TT2000 types; else False.

`tt2000_to_datetime`(*tt2000*)

Converts a CDF TT2000 value to a datetime

Note: Although TT2000 values support leapseconds, Python's datetime object does not. Any times after 23:59:59.999999 will be truncated to 23:59:59.999999.

Parameters **tt2000** : int

TT2000 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

See also:

`v_tt2000_to_datetime`

`tt2000_to_epoch`(*tt2000*)
Converts a CDF TT2000 value to a CDF EPOCH

Note: Although TT2000 values support leapseconds, CDF EPOCH values do not. Times during leapseconds are rounded up to beginning of the next day.

Parameters **tt2000** : int

TT2000 value from CDF

Returns **out** : double

EPOCH corresponding to the TT2000 input time

Raises **EpochError** : if input invalid

See also:

`v_tt2000_to_epoch`

`tt2000_to_epoch16`(*tt2000*)
Converts a CDF TT2000 value to a CDF EPOCH16

Note: Although TT2000 values support leapseconds, CDF EPOCH16 values do not. Times during leapseconds are rounded up to beginning of the next day.

Parameters **tt2000** : int

TT2000 value from CDF

Returns **out** : double, double

EPOCH16 corresponding to the TT2000 input time

Raises **EpochError** : if input invalid

See also:

`v_tt2000_to_epoch16`

v_datetime_to_epoch(*datetime*)
A vectorized version of `datetime_to_epoch()` which takes a numpy array of datetimes as input and returns an array of epochs.

v_datetime_to_epoch16(*datetime*)
A vectorized version of `datetime_to_epoch16()` which takes a numpy array of datetimes as input and returns an array of epoch16.

v_datetime_to_tt2000(*datetime*)
A vectorized version of `datetime_to_tt2000()` which takes a numpy array of datetimes as input and returns an array of TT2000.

v_epoch_to_datetime(*epoch*)
A vectorized version of `epoch_to_datetime()` which takes a numpy array of epochs as input and returns an array of datetimes.

v_epoch_to_tt2000(*epoch*)
A vectorized version of `epoch_to_tt2000()` which takes a numpy array of epochs as input and returns an array of tt2000s.

v_epoch16_to_datetime(*epoch0*, *epoch1*)
A vectorized version of `epoch16_to_datetime()` which takes a numpy array of epoch16 as input and returns an array of datetimes. An epoch16 is a pair of doubles; the input array's last dimension must be two (and the returned array will have one fewer dimension).

v_epoch16_to_tt2000(*epoch16*)
A vectorized version of `epoch16_to_tt2000()` which takes a numpy array of epoch16 as input and returns an array of tt2000s. An epoch16 is a pair of doubles; the input array's last dimension must be two (and the returned array will have one fewer dimension).

v_tt2000_to_datetime(*tt2000*)
A vectorized version of `tt2000_to_datetime()` which takes a numpy array of tt2000 as input and returns an array of datetimes.

v_tt2000_to_epoch(*tt2000*)
A vectorized version of `tt2000_to_epoch()` which takes a numpy array of tt2000 as input and returns an array of epochs.

v_tt2000_to_epoch16(*tt2000*)
A vectorized version of `tt2000_to_epoch16()` which takes a numpy array of tt2000 as input and returns an array of epoch16.

libpath
The path where pycdf found the CDF C library, potentially useful in debugging. If this contains just the name of a file (with no path information), then the system linker found the library for pycdf. On Linux, `ldconfig -p` may be useful for displaying the system's library resolution.

version
Version of the CDF library, (version, release, increment, subincrement)

spacepy.pycdf.CDFCopy

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

Data are `VarCopy` objects, keyed by variable name. CDF attributes are in `attrs`. (I.e., data are accessed much like from a `CDF`).

Do not instantiate this class directly; use `copy()` on an existing `CDF`.

Examples

```
>>> from spacepy import pycdf
>>> with pycdf.CDF('test.cdf') as cdffile:
...     data = cdffile.copy()
```

attrs
Python dictionary containing attributes copied from the CDF.

spacepy.pycdf.VarCopy

class spacepy.pycdf.**VarCopy**
A list-like copy of the data and attributes in a *Var*

Data are in the list elements. CDF attributes are in a dict, accessed through *attrs*. (I.e., data and attributes are accessed like in a *Var*.)

Do not instantiate this class directly; use *copy()* on an existing *Var*.

attrs
Python dictionary containing attributes copied from the zVar

spacepy.pycdf.CDFError

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

spacepy.pycdf.CDFException

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

Not normally used directly, but in subclasses *CDFError* and *CDFWarning*.

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

spacepy.pycdf.CDFWarning

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

spacepy.pycdf.EpochError

class spacepy.pycdf.**EpochError**
Used for errors in epoch routines

spacepy.pycdf.**lib**
Module global *Library* object.

Initalized at *pycdf* load time so all classes have ready access to the CDF library and a common state. E.g:

```
>>> from spacepy import pycdf
>>> pycdf.lib.version
    (3, 3, 0, ' ')
```

Submodules

`const`	Various constants defined in cdf.h and used in pycdf.
`istp`	Support for ISTP-compliant CDFs

spacepy.pycdf.const

Various constants defined in cdf.h and used in pycdf. Most constants referred to in the CDF manuals are provided by this module. E.g., to create a CDF and add a variable of type EPOCH:

```
>>> from spacepy import pycdf
>>> cdf = pycdf.CDF('new.cdf', '')
>>> cdf.new('epoch', type=pycdf.const.CDF_EPOCH)
```

spacepy.pycdf.istp

Support for ISTP-compliant CDFs

The ISTP metadata standard specifies the interpretation of the attributes in a CDF to describe relationships between the variables and their physical interpretation.

This module supports that subset of CDFs.

Authors: Jon Niehof

Additional Contributors: Lorna Ellis, Asher Merrill

Institution: University of New Hampshire

Contact: Jonathan.Niehof@unh.edu

Classes

`FileChecks`	ISTP compliance checks for a CDF file.
`VariableChecks`	ISTP compliance checks for a single variable.

spacepy.pycdf.istp.FileChecks

class `spacepy.pycdf.istp.`**`FileChecks`**

ISTP compliance checks for a CDF file.

Checks a file's compliance with ISTP standards. This mostly performs checks that are not currently performed by the ISTP skeleton editor. All tests return a list, one error string for every noncompliance found (empty list if compliant). `all()` will perform all tests and concatenate all errors.

`all`(f[, catch])	Perform all variable and file-level tests
`filename`(f)	Compare filename to global attributes
`time_monoton`(f)	Checks that times are monotonic
`times`(f)	Compare filename to times

classmethod **`all`** (*f*, *catch=False*)

Perform all variable and file-level tests

In addition to calling every test in this class, will also call *VariableChecks.all()* for every variable in the file.

Parameters f : *CDF*

Open CDF file to check

catch : bool

Catch exceptions in tests (default False). If True, any exceptions in subtests will result in an addition to the validation failures of the form "Test x did not complete." Calling the individual test will reveal the full traceback.

Returns list of str

Description of each validation failure.

Examples

```
>>> import spacepy.pycdf
>>> import spacepy.pycdf.istp
>>> f = spacepy.pycdf.CDF('foo.cdf', create=True)
>>> v = f.new('Var', data=[1, 2, 3])
>>> spacepy.pycdf.istp.FileChecks.all(f)
['No Logical_source in global attrs.',
'No Logical_file_id in global attrs.',
'Cannot parse date from filename foo.cdf.',
'Var: No FIELDNAM attribute.']
```

classmethod filename (*f*)

Compare filename to global attributes

Check global attribute Logical_file_id and Logical_source for consistency with CDF filename.

Parameters f : *CDF*

Open CDF file to check

Returns list of str

Description of each validation failure.

classmethod time_monoton (*f*)

Checks that times are monotonic

Check that all Epoch variables are monotonically increasing.

Parameters f : *CDF*

Open CDF file to check

Returns list of str

Description of each validation failure.

classmethod times (*f*)

Compare filename to times

Check that all Epoch variables only contain times matching filename.

Parameters f : *CDF*

Open CDF file to check

Returns list of str

Description of each validation failure.

Notes

This function assumes daily files and should be extended based on the File_naming_convention global attribute (which itself is another good check to have.)

spacepy.pycdf.istp.VariableChecks

class `spacepy.pycdf.istp.`**`VariableChecks`**

ISTP compliance checks for a single variable.

Checks a variable's compliance with ISTP standards. This mostly performs checks that are not currently performed by the ISTP skeleton editor. All tests return a list, one error string for every noncompliance found (empty list if compliant). `all()` will perform all tests and concatenate all errors.

`all`(v[, catch])	Perform all variable tests
`depends`(v)	Checks that DEPEND and LABL_PTR variables actually exist
`depsize`(v)	Checks that DEPEND has same shape as that dim
`fieldnam`(v)	Check that FIELDNAM attribute matches variable name.
`recordcount`(v)	Check that the DEPEND_0 has same record count as variable
`validdisplaytype`(v)	Check that plottype matches dimensions.
`validrange`(v)	Check that all values are within VALIDMIN/VALIDMAX, or FILLVAL
`validscale`(v)	Check SCALEMIN<=SCALEMAX, and both in range for CDF datatype.

classmethod **`all`**(*v*, *catch=False*)

Perform all variable tests

Parameters **v** : `Var`

Variable to check

catch : bool

Catch exceptions in tests (default False). If True, any exceptions in subtests will result in an addition to the validation failures of the form "Test x did not complete." Calling the individual test will reveal the full traceback.

Returns list of str

Description of each validation failure.

Examples

```
>>> import spacepy.pycdf
>>> import spacepy.pycdf.istp
>>> f = spacepy.pycdf.CDF('foo.cdf', create=True)
>>> v = f.new('Var', data=[1, 2, 3])
>>> spacepy.pycdf.istp.VariableChecks.all(v)
['No FIELDNAM attribute.']
```

classmethod **`depends`**(*v*)

Checks that DEPEND and LABL_PTR variables actually exist

Check that variables specified in the variable attributes for DEPEND and LABL_PTR exist in the CDF.

Parameters **v** : `Var`

Variable to check

Returns list of str

Description of each validation failure.

classmethod **`depsize`**(*v*)

Checks that DEPEND has same shape as that dim

Compares the size of variables specified in the variable attributes for DEPEND and compares to the size of the corresponding dimension in this variable.

Parameters **v** : `Var`

Variable to check

Returns list of str

Description of each validation failure.

classmethod **`fieldnam`**(*v*)

Check that FIELDNAM attribute matches variable name.

Compare FIELDNAM attribute to the variable name; fail validation if they don't match.

Parameters **v** : `Var`

Variable to check

Returns list of str

Description of each validation failure.

classmethod **`recordcount`**(*v*)

Check that the DEPEND_0 has same record count as variable

Checks the record count of the variable specified in the variable attribute for DEPEND_0 and compares to the record count for this variable.

Parameters **v** : `Var`

Variable to check

Returns list of str

Description of each validation failure.

classmethod **`validdisplaytype`**(*v*)

Check that plottype matches dimensions.

Check DISPLAYTYPE of this variable and makes sure it is reasonable for the variable dimensions.

Parameters **v** : `Var`

Variable to check

Returns list of str

Description of each validation failure.

classmethod **`validrange`**(*v*)

Check that all values are within VALIDMIN/VALIDMAX, or FILLVAL

Compare all values of this variable to VALIDMIN and `VALIDMAX`; fails validation if any values are below VALIDMIN or above `VALIDMAX` unless equal to FILLVAL.

Parameters **v** : *Var*

Variable to check

Returns list of str

Description of each validation failure.

classmethod validscale(*v*)

Check SCALEMIN<=SCALEMAX, and both in range for CDF datatype.

Compares SCALEMIN to `SCALEMAX` to make sure it isn't larger and both are within range of the variable CDF datatype.

Parameters **v** : *Var*

Variable to check

Returns list of str

Description of each validation failure.

Functions

fillval(v)	Set ISTP-compliant FILLVAL on a variable
format(v[, use_scaleminmax, dryrun])	Set ISTP-compliant FORMAT on a variable

spacepy.pycdf.istp.fillval

`spacepy.pycdf.istp.`**fillval**(*v*)

Set ISTP-compliant FILLVAL on a variable

Sets a CDF variable's FILLVAL attribute to the value required by ISTP (based on variable type).

Parameters **v** : *Var*

CDF variable to update

Examples

```
>>> import spacepy.pycdf
>>> import spacepy.pycdf.istp
>>> f = spacepy.pycdf.CDF('foo.cdf', create=True)
>>> v = f.new('Var', data=[1, 2, 3])
>>> spacepy.pycdf.istp.fillval(v)
>>> v.attrs['FILLVAL']
-128
```

spacepy.pycdf.istp.format

`spacepy.pycdf.istp.`**format**(*v*, *use_scaleminmax=False*, *dryrun=False*)

Set ISTP-compliant FORMAT on a variable

Sets a CDF variable's FORMAT attribute, which provides a Fortran-like format string that should be useable for printing any valid value in the variable. Sets according to the VALIDMIN/VALIDMAX attributes (or, optionally, SCALEMIN/SCALEMAX) if present, otherwise uses the full range of the type.

Parameters **v** : *Var*

Variable to update

use_scaleminmax : bool, optional

Use SCALEMIN/MAX instead of VALIDMIN/MAX (default False). Note: istpchecks may complain about result.

dryrun : bool, optional

Print the decided format to stdout instead of modifying the CDF (for use in command-line debugging) (default False).

Examples

```
>>> import spacepy.pycdf
>>> import spacepy.pycdf.istp
>>> f = spacepy.pycdf.CDF('foo.cdf', create=True)
>>> v = f.new('Var', data=[1, 2, 3])
>>> spacepy.pycdf.istp.format(v)
>>> v.attrs['FORMAT']
'I4'
```

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

Copyright 2010 Los Alamos National Security, LLC.

Classes

RBmodel([grid, NL, const_kp])	1-D Radial diffusion class

spacepy.radbelt.RBmodel

class spacepy.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()
```

Gaussian_source()	Gaussian source term added to radiation belt model.
add_Lmax(Lmax_model)	add last closed drift shell Lmax
add_Lpp(Lpp_model)	add last closed drift shell Lmax
add_PSD_obs([time, PSD, Lstar, satlist])	add PSD observations
add_PSD_twin([dt, Lt])	add observations from PSD database using the ticks list
add_omni([keylist])	add omni data to instance according to the tickrange in ticks
add_source([source, A, mu, sigma])	add source parameters A, mu, and sigma for the Gaussian source function
assimilate([method, inflation])	Assimilates data for the radiation belt model using the Ensemble Kalman Filter.
evolve()	calculate the diffusion in L at constant mu,K coordinates
get_DLL(Lgrid, params[, DLL_model])	Calculate DLL as a simple power law function (alpha*L**Bbta) using alpha/beta
plot([Lmax, Lpp, Kp, Dst, clims, title, values])	Create a summary plot of the RadBelt object distribution function.
plot_obs([Lmax, Lpp, Kp, Dst, clims, title, ...])	Create a summary plot of the observations.
set_lgrid([NL])	Using NL grid points, create grid in L.
setup_ticks(start, end, delta[, dtype])	Add time information to the simulation by specifying a start and end time, timest

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** : list of dicts

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')
```

Functions

get_modelop_L(f, L, Dm_old, Dm_new, Dp_old, ...)	Advance the distribution function, f, discretized into the Lgrid, L, forward
diff_LL(r, grid, f, Tdelta, Telapsed[, params])	calculate the diffusion in L at constant mu,K coordinates
get_local_accel(Lgrid, params[, SRC_model])	calculate the diffusion coefficient D_LL

spacepy.radbelt.get_modelop_L

spacepy.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.

spacepy.radbelt.diff_LL

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

spacepy.radbelt.get_local_accel

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

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

Copyright 2010 Los Alamos National Security, LLC.

Classes

Sea(data, times, epochs[, window, delta, ...])	SeaPy Superposed epoch analysis object
Sea2d(data, times, epochs[, window, delta, ...])	SeaPy 2D Superposed epoch analysis object

spacepy.seapy.Sea

class `spacepy.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

Parameters **data** : array_like

list or array of data

times : array_like

list of datetime objects (or list of serial times)

epochs : array_like

list of datetime objects (or serial times) for zero epochs in SEA

window : datetime.timedelta

size of the half-window for the SEA (can also be given as serial time)

delta : datetime.timedelta

resolution of the input data series, which must be uniform (can also be given as serial time)

Notes

Output can be nicely plotted with `plot()`, or for multiple objects use the `multisea()` function

`sea`(**kwargs)	Method called to perform superposed epoch analysis on data in object.
`plot`([xquan, yquan, xunits, yunits, ...])	Method called to create basic plot of superposed epoch analysis.

sea(***kwargs*)

Method called to perform superposed epoch analysis on data in object.

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 of ci_quan at the ci percent level (default=95)

mad : float

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

ci_quan : string

can be set to 'median' (default) or 'mean'

Notes

A basic plot can be raised with `plot()`

plot(*xquan='Time Since Epoch', yquan='', xunits='', yunits='', epochline=True, usrlimy=[], show=True, target=None, loc=111, figsize=None, dpi=None, transparent=True, color='#7F7FFF'*)

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 = True) - 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

color : str

Color to use for the patch if not transparent. (default #7F7FFF, a medium blue)

Notes

If both quan and units are supplied, axis label will read ‘Quantity Entered By User [Units]’

spacepy.seapy.Sea2d

class `spacepy.seapy.`**`Sea2d`**(*data*, *times*, *epochs*, *window=3.0*, *delta=1.0*, *verbose=False*, *y=[]*)

SeaPy 2D Superposed epoch analysis object

Initialize object with data (n element vector), times(y*n array), 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

Parameters **data** : array_like

2-D array of data (0th dimension is quantity y, 1st dimension is time)

times : array_like

list of datetime objects (or list of serial times)

epochs : array_like

list of datetime objects (or serial times) for zero epochs in SEA

window : datetime.timedelta

size of the half-window for the SEA (can also be given as serial time)

delta : datetime.timedelta

resolution of the input data series, which must be uniform (can also be given as serial time)

Notes

Output can be nicely plotted with `plot()`, or for multiple objects use the `multisea()` function

`sea`([storedata, quartiles, ci, mad, ...])	Perform 2D superposed epoch analysis on data in object
`plot`([xquan, yquan, xunits, yunits, zunits, ...])	Method called to create basic plot of 2D superposed epoch analysis.

sea (*storedata=False*, *quartiles=True*, *ci=False*, *mad=False*, *ci_quan='median'*, *nmask=1*, ***kwargs*)
Perform 2D superposed epoch analysis on data in object

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 inter-quartile range to show the spread (and is default);

ci : float

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

mad : float

will use the median absolute deviation for the spread;

ci_quan : string

can be set to 'median' or 'mean'

Notes

A basic plot can be raised with `plot()`

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

Uses object attributes created by `sea()`.

Other Parameters **x(y)quan** : str

x(y)-axis label. (default = 'Time since epoch' (None))

x(y/z)units : str

x(y/z)-axis units. (default = None (None))

epochline : boolean

put vertical line at zero epoch. (default = True)

usrlimy : list

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

show : boolean

shows plot; set to false to output plot object to variable (default = True)

figsize : tuple

(width, height) in inches

dpi : int

figure resolution in dots per inch (default=300)

Notes

If both quan and units are supplied, axis label will read 'Quantity Entered By User [Units]'

Functions

seadict(objlist, namelist)	Function to create dictionary of SeaPy.Sea objects.
multisea(dictobj[, n_cols, epochline, ...])	Function to create multipanel plot of superposed epoch analyses.
readepochs(fname[, iso, isofmt])	Read epochs from text file assuming YYYY MM DD hh mm ss format
sea_signif(obj1, obj2[, test, show, xquan, ...])	Test for similarity between distributions at each lag in two 1-D SEAs

spacepy.seapy.seadict

`spacepy.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.

spacepy.seapy.multisea

`spacepy.seapy.`**`multisea`**(*dictobj*, *n_cols=1*, *epochline=True*, *usrlimx=[]*, *usrlimy=[]*, *xunits=''*, *show=True*, *zunits=''*, *zlog=True*, *figsize=None*)

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 = True) - 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.

spacepy.seapy.readepochs

`spacepy.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)

spacepy.seapy.sea_signif

`spacepy.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

Parameters **obj1** : Sea

First instance for comparison

obj2 : Sea

Second instance for comparison

Other Parameters **test**

(default = 'KS') Test to apply at each lag: KS is 2-smaple Kolmogorov-Smirnov; U is Mann-Whitney U-test

show

(default = True)

xquan

(default = 'Time since epoch' (None)) - x-axis label.

yquan

(default = 'Time since epoch' (None)) - y-axis label.

xunits

(default = None (None)) - x-axis units.

yunits

(default = None (None)) - 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)
```

time - Time conversion, manipulation and implementation of Ticktock class

Time conversion, manipulation and implementation of Ticktock class

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

```
>>> 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: Steve Morley, Josef Koller, Brian Larsen, Jon Niehof Institution: Los Alamos National Laboratory Contact: smorley@lanl.gov,

Classes

Ticktock(data, dtype)	Ticktock class holding various time coordinate systems

spacepy.time.Ticktock

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

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

Possible input 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

Possible output data types: All those listed above, plus DOY: Integer day of year, starts with day 1 eDOY: Fractional day of year, starts at day 0

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

time stamp

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

data type for data, if a function it must convert input time format to Python datetime

Returns out : Ticktock

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

Examples

```
>>> x = Ticktock([2452331.0142361112, 2452332.0142361112], 'JD')
>>> x.ISO
dmarray(['2002-02-25T12:20:30', '2002-02-26T12:20:30'], dtype='|S19')
>>> x.DOY # Day of year
dmarray([ 56.,  57.])
>>> y = Ticktock(['01-01-2013', '20-03-2013'], lambda x: datetime.datetime.strptime(x, '%d-%m-%Y
>>> y.UTC
dmarray([2013-01-01 00:00:00, 2013-03-20 00:00:00], dtype=object)
>>> y.DOY # Day of year
dmarray([  1.,  79.])
```

append(other)	Will be called when another Ticktock instance has to be appended to the current one
argsort()	This will return the indices that would sort the Ticktock values
convert(dtype)	convert a Ticktock instance into a new time coordinate system provided in dtype
getCDF()	a.getCDF() or a.CDF
getDOY()	a.DOY or a.getDOY()
getGPS()	a.GPS or a.getGPS()
getISO()	a.ISO or a.getISO()
getJD()	a.JD or a.getJD()
getMJD()	a.MJD or a.getMJD()
getRDT()	a.RDT or a.RDT()
getTAI()	a.TAI or a.getTAI()
getUNX()	a.UNX or a.getUNX()
getUTC()	a.UTC or a.getUTC()
geteDOY()	a.eDOY or a.geteDOY()
getleapsecs()	a.leaps or a.getleapsecs()
isoformat(b, attrib)	This changes the self._isofmt attribute by and subsequently this function will update the ISO attribute.
now()	Creates a Ticktock object with the current time, equivalent to datetime.now()
sort()	This will sort the Ticktock values in place, if you need a stable sort use kind='mergesort'
update_items(b, attrib)	After changing the self.data attribute by either __setitem__ or __add__ etc this function will update all

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:

`CDF`, `ISO`, `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
dmarray(['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, if you need a stable sort use kind='mergesort'

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__

Functions

doy2date(year, doy[, dtobj, flAns])	convert integer day-of-year doy into a month and day

Table 2.51 – continued from previous page

`leapyear`(year[, numdays])	return an array of boolean leap year,
`randomDate`(dt1, dt2[, N, tzinfo, sorted])	Return a (or many) random datetimes between two given dates, this is done under the co
`sec2hms`(sec[, rounding, days, dtobj])	Convert seconds of day to hours, minutes, seconds
`tickrange`(start, end, deltadays[, dtype])	return a Ticktock range given the start, end, and delta

spacepy.time.doy2date

`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:

`Ticktock.getDOY`

Examples

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

spacepy.time.leapyear

`spacepy.time.`**`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.time
>>> spacepy.time.leapyear(numpy.arange(15)+1998)
[False, False,  True, False, False, False,  True, False, False,
       False,  True, False, False, False,  True]
```

spacepy.time.randomDate

`spacepy.time.`**`randomDate`**(*dt1*, *dt2*, *N=1*, *tzinfo=False*, *sorted=False*)

Return a (or many) random datetimes between two given dates, this is done under the convention dt <=1 rand < dt2

Parameters **dt1** : datetime.datetime

start date for the the random date

dt2 : datetime.datetime

stop date for the the random date

Returns **out** : datetime.datetime or numpy.ndarray of datetime.datetime

the new time for the next call to EventTimer

Other Parameters **N** : int (optional)

the number of random dates to generate (defualt=1)

tzinfo : bool (optional)

maintain the tzinfo of the input datetimes (default=False)

sorted : bool (optional)

return the times sorted (default=False)

spacepy.time.sec2hms

`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.tickrange

spacepy.time.**tickrange**(*start*, *end*, *deltadays*, *dtype=None*)

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

Parameters **start** : string or number

start time (ISO standard string and UTC/datetime do not require a dtype)

end : string or number

last possible time in series (excluded unless end=start+n*step for integer n)

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)
```

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

Copyright 2010 Los Alamos National Security, LLC.

Array binning

arraybin(array, bins)	Split a sequence into subsequences based on value.
bin_center_to_edges(centers)	Convert a list of bin centers to their edges
bin_edges_to_center(edges)	Convert a list of bin edges to their centers
binHisto(data[, verbose])	Calculates bin width and number of bins for histogram using Freedman-Diaconis rule, if rule

spacepy.toolbox.arraybin

spacepy.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]]
```

spacepy.toolbox.bin_center_to_edges

spacepy.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]
```

spacepy.toolbox.bin_edges_to_center

`spacepy.toolbox.`**`bin_edges_to_center`**`(`*edges*`)`

Convert a list of bin edges to their centers

Given a list of edge values for a set of bins, finds the center of each bin. (start of bin n+1 is assumed to be end of bin n).

Center of bin n is arithmetic mean of the edges of the adjacent bins.

Parameters **edges** : list

list of edge values for bins

Returns **out** : numpy.ndarray

array of centers for bins

note: returned array will be one element shorter than edges

Examples

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

spacepy.toolbox.binHisto

`spacepy.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

The Freedman-Diaconis method is detailed in: Freedman, D., and P. Diaconis (1981), On the histogram as a density estimator: L2 theory, Z. Wahrscheinlichkeitstheor. Verw. Geb., 57, 453–476

and is also described by: Wilks, D. S. (2006), Statistical Methods in the Atmospheric Sciences, 2nd ed.

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

See also:

`matplotlib.pyplot.hist`

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
>>> p = plt.hist(data, bins=nbins, histtype='step', density=True)
```

Array creation

dist_to_list(func, length[, min, max])	Convert a probability distribution function to a list of values
geomspace(start[, ratio, stop, num])	Returns geometrically spaced numbers.
linspace(min, max, num, **kwargs)	Returns linear-spaced bins.
logspace(min, max, num, **kwargs)	Returns log-spaced bins.

spacepy.toolbox.dist_to_list

spacepy.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.sqrt(2 * math.pi))
>>> 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)], facecolor='green')
>>> 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()
```

spacepy.toolbox.geomspace

`spacepy.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

geometrically spaced sequence

See also:

linspace, *logspace*

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]
```

spacepy.toolbox.linspace

`spacepy.toolbox.linspace(min, max, num, **kwargs)`

Returns linear-spaced bins. Same as numpy.linspace except works with datetime and is faster

Parameters **min** : float, datetime

minimum value

max : float, datetime

maximum value

num : integer

number of linear spaced bins

Returns **out** : array

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

Other Parameters **kwargs** : dict

additional keywords passed into matplotlib.dates.num2date

See also:

geomspace, *logspace*

Notes

This function works on both numbers and datetime objects

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.linspace(1, 10, 4)
array([  1.,   4.,   7.,  10.])
```

spacepy.toolbox.logspace

spacepy.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

See also:

geomspace, *linspace*

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.        ])
```

Array searching and masking

interweave(a, b)	given two array-like variables interweave them together.
isview(array1[, array2])	Returns if an object is a view of another object.
tCommon(ts1, ts2[, mask_only])	Finds the elements in a list of datetime objects present in another
tOverlap(ts1, ts2, *args, **kwargs)	Finds the overlapping elements in two lists of datetime objects
tOverlapHalf(ts1, ts2[, presort])	Find overlapping elements in two lists of datetime objects

spacepy.toolbox.interweave

`spacepy.toolbox.`**`interweave`**`(`*a*, *b*`)`

given two array-like variables interweave them together. Discussed here: http://stackoverflow.com/questions/5347065/interweaving-two-numpy-arrays

Parameters **a** : array-like

first array

b : array-like

second array

Returns **out** : numpy.ndarray

interweaved array

spacepy.toolbox.isview

`spacepy.toolbox.`**`isview`**`(`*array1*, *array2=None*`)`

Returns if an object is a view of another object. More precisely if one array argument is specified True is returned is the arrays owns its data. If two arrays arguments are specified a tuple is returned of if the first array owns its data and the the second if they point at the same memory location

Parameters **array1** : numpy.ndarray

array to query if it owns its data

Returns **out** : bool or tuple

If one array is specified bool is returned, True is the array owns its data. If two arrays are specified a tuple where the second element is a bool of if the array point at the same memory location

Other Parameters **array2** : object (optional) array to query if array1 is a view of this object at the specified memory location

Examples

import numpy import spacepy.toolbox as tb a = numpy.arange(100) b = a[0:10] tb.isview(a) # False tb.isview(b) # True tb.isview(b, a) # (True, True) tb.isview(b, b) # (True, True) # the conditions are met and numpy cannot tell this

spacepy.toolbox.tCommon

`spacepy.toolbox.`**`tCommon`**(*ts1*, *ts2*, *mask_only=True*)

Finds the elements in a list of datetime objects present in another

Parameters **ts1** : list or array-like

first set of datetime objects

ts2 : list or array-like

second set of datetime objects

Returns **out** : tuple

Two element tuple of truth tables (of 1 present in 2, & vice versa)

See also:

tOverlapHalf, *tOverlap*

Examples

```
>>> import spacepy.toolbox as tb
>>> import numpy as np
>>> import datetime as dt
>>> ts1 = np.array([dt.datetime(2001,3,10)+dt.timedelta(hours=a) for a in range(20)])
>>> ts2 = np.array([dt.datetime(2001,3,10,2)+dt.timedelta(hours=a*0.5) for a in range(20)])
>>> common_inds = tb.tCommon(ts1, ts2)
>>> common_inds[0] #mask of values in ts1 common with ts2
array([False, False,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True, False, False, False, False, False, False,
       False, False], dtype=bool)
>>> ts2[common_inds[1]] #values of ts2 also in ts1
```

The latter can be found more simply by setting the mask_only keyword to False

```
>>> common_vals = tb.tCommon(ts1, ts2, mask_only=False)
>>> common_vals[1]
array([2001-03-10 02:00:00, 2001-03-10 03:00:00, 2001-03-10 04:00:00,
       2001-03-10 05:00:00, 2001-03-10 06:00:00, 2001-03-10 07:00:00,
       2001-03-10 08:00:00, 2001-03-10 09:00:00, 2001-03-10 10:00:00,
       2001-03-10 11:00:00], dtype=object)
```

spacepy.toolbox.tOverlap

`spacepy.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

See also:

`tOverlapHalf`, `tCommon`

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)]
```

spacepy.toolbox.tOverlapHalf

`spacepy.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 occurrences where ts2 exists within the bounds of ts1, or the second element returned 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

See also:

`tOverlap`, `tCommon`

Other functions

`assemble`(fln_pattern, outfln[, sortkey, verbose])	assembles all pickled files matching fln_pattern into single file and
`bootHisto`(data[, inter, n, seed, plot, ...])	Bootstrap confidence intervals for a histogram.
`dictree`(in_dict[, verbose, spaces, levels, ...])	pretty print a dictionary tree
`eventTimer`(Event, Time1)	Times an event then prints out the time and the name of the event,
`feq`(x, y[, precision])	compare two floating point values if they are equal
`getNamedPath`(name)	Return the full path of a parent directory with name as the leaf
`human_sort`(l)	Sort the given list in the way that humans expect.
`hypot`(*args)	compute the N-dimensional hypot of an iterable or many arguments
`interpol`(newx, x, y[, wrap])	1-D linear interpolation with interpolation of hours/longitude
`intsolve`(func, value[, start, stop, maxit])	Find the function input such that definite integral is desired value.
`medAbsDev`(series[, scale])	Calculate median absolute deviation of a given input series
`mlt2rad`(mlt[, midnight])	Convert mlt values to radians for polar plotting
`normalize`(vec[, low, high])	Given an input vector normalize the vector to a given range
`pmm`(a, *b)	print min and max of input arrays
`rad2mlt`(rad[, midnight])	Convert radians values to mlt
`windowMean`(data[, time, winsize, overlap, ...])	Windowing mean function, window overlap is user defined

spacepy.toolbox.assemble

`spacepy.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])}
```

spacepy.toolbox.bootHisto

`spacepy.toolbox.bootHisto`(*data*, *inter=90.0*, *n=1000*, *seed=None*, *plot=False*, *target=None*, *figsize=None*, *loc=None*, ***kwargs*)

Bootstrap confidence intervals for a histogram.

All other keyword arguments are passed to `numpy.histogram()` or `matplotlib.pyplot.bar()`.

Parameters **data** : array_like

list/array of data values

inter : float (optional; default 90)

percentage confidence interval to return. Default 90% (i.e. lower CI will be 5% and upper will be 95%)

n : int (optional; default 100)

number of bootstrap iterations

seed : int (optional)

Optional seed for the random number generator. If not specified; numpy generator will not be reseeded.

plot : bool (optional)

Plot the result. Plots if True or `target`, `figsize`, or `loc` specified.

target : (optional)

Target on which to plot the figure (figure or axes). See `spacepy.plot.utils.set_target()` for details.

figsize : tuple (optional)

Passed to `spacepy.plot.utils.set_target()`.

loc : int (optional)

Passed to `spacepy.plot.utils.set_target()`.

Returns **out** : tuple

tuple of bin_edges, low, high, sample[, bars]. Where `bin_edges` is the edges of the bins used; `low` is the histogram with the value for each bin from the bottom of that bin's confidence interval; `high` similarly for the top; `sample` is the histogram of the input sample without resampling. If plotting, also returned is `bars`, the container object returned from matplotlib.

See also:

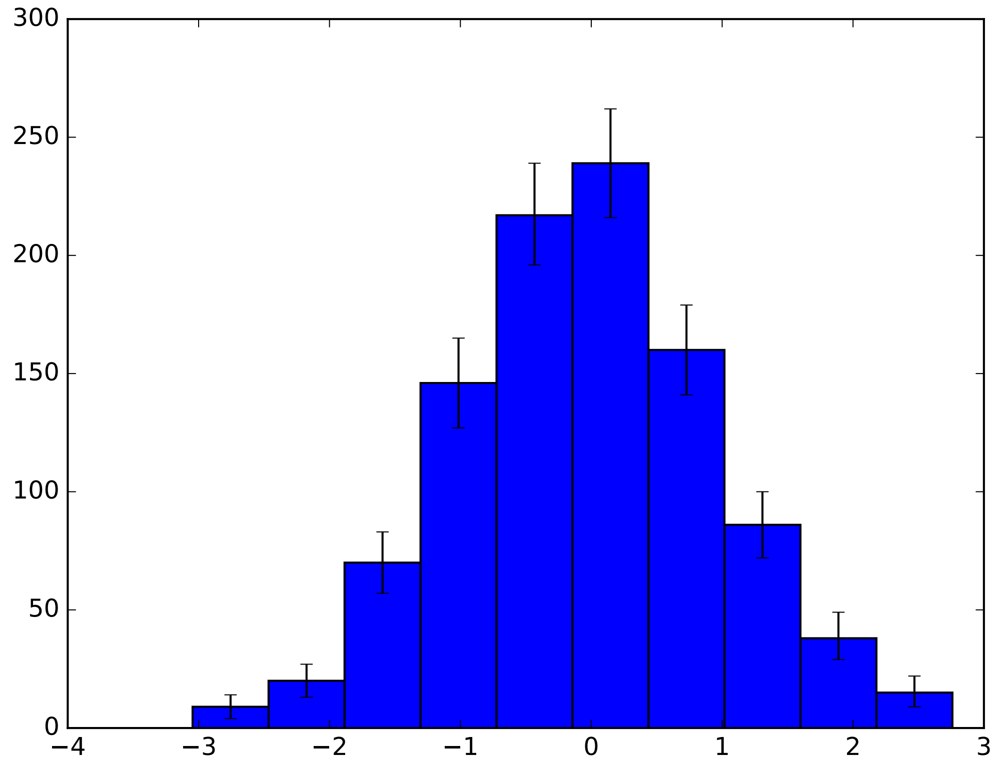
`binHisto`, `plot.utils.set_target`, `numpy.histogram`, `matplotlib.pyplot.hist`

Notes

The confidence intervals are calculated for each bin individually and thus the resulting low/high histograms may not have actually occurred in the calculation from the surrogates. If using a probability density histogram, this can have "interesting" implications for interpretation.

Examples

```
>>> import numpy.random
>>> import spacepy.toolbox
>>> numpy.random.seed(0)
>>> data = numpy.random.randn(1000)
>>> bin_edges, ci_low, ci_high, sample, bars = spacepy.toolbox.bootHisto(
        data, plot=True)
```



spacepy.toolbox.dictree

`spacepy.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.

spacepy.toolbox.eventTimer

`spacepy.toolbox.`**`eventTimer`**`(`*Event*, *Time1*`)`

Times an event then prints out the time and the name of the event, nice for debugging and seeing that the code is progressing

Parameters **Event** : str

Name of the event, string is printed out by function

Time1 : time.time

the time to difference in the function

Returns **Time2** : time.time

the new time for the next call to EventTimer

Examples

```
>>> import spacepy.toolbox as tb
>>> import time
>>> t1 = time.time()
>>> t1 = tb.eventTimer('Test event finished', t1)
('4.40', 'Test event finished')
```

spacepy.toolbox.feq

`spacepy.toolbox.`**`feq`**`(`*x, y, precision=5e-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 floats

other numbers to compare

precision : float (optional)

Relative precision for equal (default 0.0000005) Specified as a fraction of the sum of `x` and `y`.

Returns **out** : bool

True (equal) or False (not equal)

See also:

`numpy.allclose`

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
```

spacepy.toolbox.getNamedPath

`spacepy.toolbox.`**`getNamedPath`**`(`*name*`)`

Return the full path of a parent directory with name as the leaf

Parameters **name** : string

the name of the parent directory to locate

Examples

Run from a directory /mnt/projects/dream/bin/Ephem with ‘dream’ as the name, this function would return ‘/mnt/projects/dream’

spacepy.toolbox.human_sort

spacepy.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']
```

spacepy.toolbox.hypot

spacepy.toolbox.**hypot**(**args*)

compute the N-dimensional hypot of an iterable or many arguments

Parameters **args** : many numbers or array-like

array like or many inputs to compute from

Returns **out** : float

N-dimensional hypot of a number

Notes

This function has a complicated speed function.

- if a numpy array of floats is input this is passed off to C
- if iterables are passed in they are made into numpy arrays and comptaton is done local
- if many scalar agruments are passed in calculation is done in a loop

For max speed:

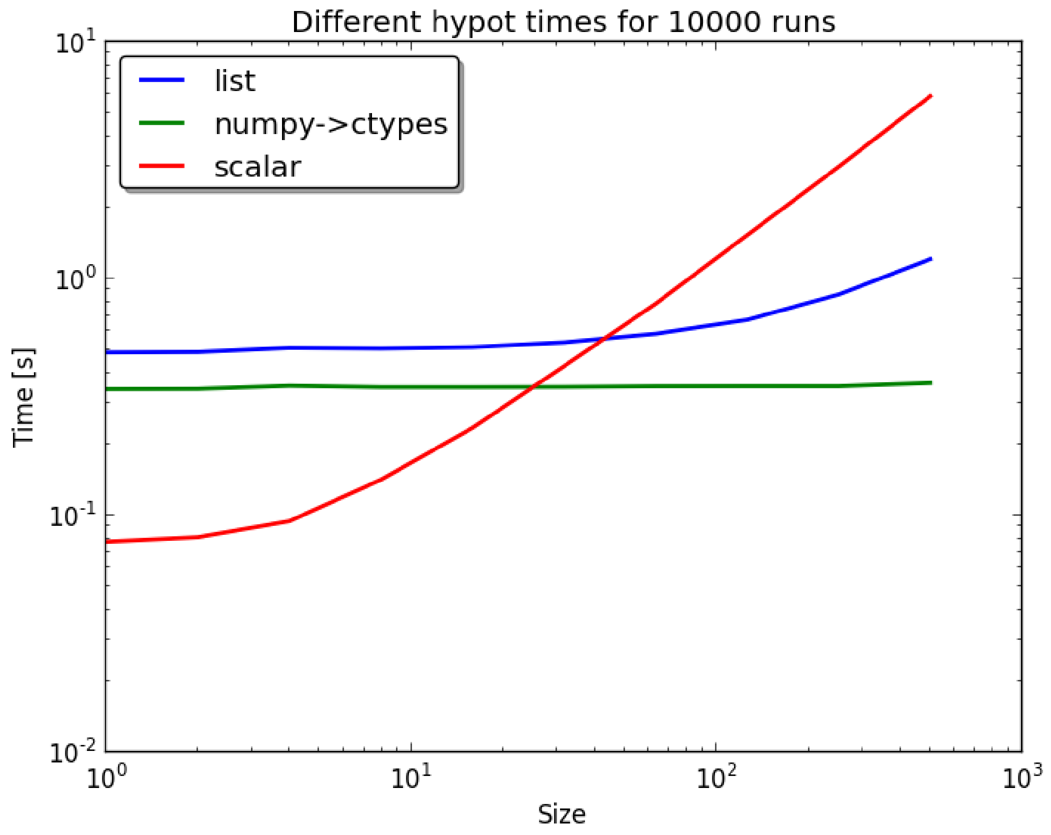
- **<20 elements expand them into scalars**

```
>>> tb.hypot(*vals)
>>> tb.hypot(vals[0], vals[1]...) #alternate
```

- >20 elements premake them into a numpy array of doubles

Examples

```
>>> from spacepy import toolbox as tb
>>> print tb.hypot([3,4])
5.0
>>> print tb.hypot(3,4)
5.0
>>> # Benchmark ####
>>> from spacepy import toolbox as tb
>>> import numpy as np
>>> import timeit
>>> num_list = []
>>> num_np = []
>>> num_np_double = []
>>> num_scalar = []
>>> tot = 500
>>> for num in tb.logspace(1, tot, 10):
>>>     print num
>>>     num_list.append(timeit.timeit(stmt='tb.hypot(a)', setup='from spacepy import toolbox as
>>>     num_np.append(timeit.timeit(stmt='tb.hypot(a)', setup='from spacepy import toolbox as tb
>>>     num_scalar.append(timeit.timeit(stmt='tb.hypot(*a)', setup='from spacepy import toolbox
>>> from pylab import *
>>> loglog(tb.logspace(1, tot, 10),  num_list, lw=2, label='list')
>>> loglog(tb.logspace(1, tot, 10),  num_np, lw=2, label='numpy->ctypes')
>>> loglog(tb.logspace(1, tot, 10),  num_scalar, lw=2, label='scalar')
>>> legend(shadow=True, fancybox=1, loc='upper left')
>>> title('Different hypot times for 10000 runs')
>>> ylabel('Time [s]')
>>> xlabel('Size')
```



spacepy.toolbox.interpol

`spacepy.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 original data (must be monotonically increasing or wrapping)

y : array_like

y values of the original data

wrap : string, optional

for continuous x data that wraps in y at 'hours' (24), 'longitude' (360), or arbitrary 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 maskec
array([  1.5,  10.5,  23.5])
>>> tb.interpol([1.5, 10.5, 23.5], x, y)
array([  1.5,  10.5,  11.5])
```

spacepy.toolbox.intsolve

`spacepy.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.

spacepy.toolbox.medAbsDev

`spacepy.toolbox.`**`medAbsDev`**(*series*, *scale=False*)

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. For additional details on the scaling, see Rousseeuw and Croux, J. Amer. Stat. Assoc., 88 (424), pp. 1273-1283, 1993.

Parameters **series** : array_like

the input data series

Returns **out** : float

the median absolute deviation

Other Parameters **scale** : bool

if True (default: False), scale to standard deviation of a normal distribution

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

spacepy.toolbox.mlt2rad

`spacepy.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

See also:

rad2mlt

Examples

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

spacepy.toolbox.normalize

`spacepy.toolbox.`**`normalize`**(*vec*, *low=0.0*, *high=1.0*)

Given an input vector normalize the vector to a given range

Parameters **vec** : array_like

input vector to normalize

low : float

minimum value to scale to, default 0.0

high : float

maximum value to scale to, default 1.0

Returns **out** : array_like

normalized vector

Examples

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

spacepy.toolbox.pmm

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

print min and max of input arrays

Parameters **a** : numpy array

input array

b : list arguments

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]]
```

spacepy.toolbox.rad2mlt

`spacepy.toolbox.`**`rad2mlt`**(*rad*, *midnight=False*)

Convert radians 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

See also:

`mlt2rad`

Examples

```
>>> rad2mlt(array([0,pi, pi/2.]))
array([ 12.,  24.,  18.])
```

spacepy.toolbox.windowMean

`spacepy.toolbox.`**`windowMean`**(*data*, *time=[]*, *winsize=0*, *overlap=0*, *st_time=None*, *op=<function mean>*)

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

op : callable (optional)

the operator to be called, default numpy.mean

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.

Multithreading and multiprocessing

thread_job(job_size, thread_count, target, ...)	Split a job into subjobs and run a thread for each
thread_map(target, iterable[, thread_count])	Apply a function to every element of a list, in separate threads

spacepy.toolbox.thread_job

spacepy.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 passed 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.

spacepy.toolbox.thread_map

spacepy.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

This is made largely obsolete in python3 by from concurrent import futures

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)
```

```
>>> # in python3
>>> from concurrent import futures
>>> with futures.ThreadPoolExecutor(max_workers=4) as executor:
...:     for ans in executor.map(numpy.sum, [0,50,99]):
...:         print ans
#0
#50
#99
```

System tools

do_with_timeout(timeout, target, *args, **kwargs)	Execute a function (or method) with a timeout.
loadpickle(fln)	load a pickle and return content as dictionary
progressbar(count, blocksize, totalsize[, text])	print a progress bar with urllib.urlretrieve reporthook functionality
query_yes_no(question[, default])	Ask a yes/no question via raw_input() and return their answer.
savepickle(fln, dict[, compress])	save dictionary variable dict to a pickle with filename fln

Continued on next page

Table 2.57 – continued from previous page

timeout_check_call(timeout, *args, **kwargs)	Call a subprocess with a timeout.
TimeoutError	Raised when a time-limited process times out
update([all, QDomni, omni, omni2, leapsecs, ...])	Download and update local database for omni, leapsecs etc

spacepy.toolbox.do_with_timeout

`spacepy.toolbox.`**`do_with_timeout`**(*timeout*, *target*, **args*, ***kwargs*)

Execute a function (or method) with a timeout.

Call the function (or method) `target`, with arguments `args` and keyword arguments `kwargs`. Normally return the return value from `target`, but if `target` takes more than `timeout` seconds to execute, raises `TimeoutError`.

Note: This is, at best, a blunt instrument. Exceptions from `target` may not propagate properly (tracebacks will be hard to follow.) The function which failed to time out may continue to execute until the interpreter exits; trapping the TimeoutError and continuing normally is not recommended.

Parameters **timeout** : float

Timeout, in seconds.

target : callable

Python callable (generally a function, may also be an imported ctypes function) to run.

args : sequence

Arguments to pass to `target`.

kwargs : dict

keyword arguments to pass to `target`.

Returns out :

return value of `target`

Raises **TimeoutError** : If `target` does not return in `timeout` seconds.

Examples

```
>>> import spacepy.toolbox as tb
>>> import time
>>> def time_me_out():
...     time.sleep(5)
>>> tb.do_with_timeout(0.5, time_me_out) #raises TimeoutError
```

spacepy.toolbox.loadpickle

`spacepy.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')
```

spacepy.toolbox.progressbar

spacepy.toolbox.**progressbar**(*count*, *blocksize*, *totalsize*, *text='Download Progress'*)

print a progress bar with urllib.urlretrieve reporthook functionality

Examples

```
>>> import spacepy.toolbox as tb
>>> import urllib
>>> urllib.urlretrieve(config['psddata_url'], PSDdata_fname, reporthook=tb.progressbar)
```

spacepy.toolbox.query_yes_no

spacepy.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'
```

spacepy.toolbox.savepickle

`spacepy.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 uncompressed does not.

See also:

loadpickle

Examples

```
>>> d = {'grade':[1,2,3], 'name':['Mary', 'John', 'Chris']}
>>> savepickle('test.pbin', d)
```

spacepy.toolbox.timeout_check_call

`spacepy.toolbox.timeout_check_call`(*timeout*, **args*, ***kwargs*)

Call a subprocess with a timeout.

Like `subprocess.check_call()`, but will terminate the process and raise *TimeoutError* if it runs for too long.

This will only terminate the single process started; any child processes will remain running (this has implications for, say, spawing shells.)

Parameters **timeout** : float

Timeout, in seconds. Fractions are acceptable but the resolution is of order 100ms.

args : sequence

Arguments passed through to `subprocess.Popen`

kwargs : dict

keyword arguments to pass to `subprocess.Popen`

Returns **out** : int

0 on successful completion

Raises **TimeoutError** : If subprocess does not return in `timeout` seconds.

CalledProcessError : if command has non-zero exit status

Examples

```
>>> import spacepy.toolbox as tb
>>> tb.timeout_check_call(1, 'sleep 30', shell=True) #raises TimeoutError
```

spacepy.toolbox.TimeoutError

exception `spacepy.toolbox.`**`TimeoutError`**

Raised when a time-limited process times out

spacepy.toolbox.update

`spacepy.toolbox.`**`update`**(*all=True*, *QDomni=False*, *omni=False*, *omni2=False*, *leapsecs=False*, *PSDdata=False*)

Download and update local database for omni, leapsecs etc

Parameters **all** : boolean (optional)

if True, update OMNI2, Qin-Denton and leapsecs

omni : boolean (optional)

if True. update only omni (Qin-Denton)

omni2 : boolean (optional)

if True, update only original OMNI2

QDomni : boolean (optional)

if True, update OMNI2 and Qin-Denton

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)
```

Indices and tables

- genindex
- modindex
- search

Release 0.2.1

Doc generation date October 02, 2019

PYTHON MODULE INDEX