QuickStartGuide Documentation

Release 0.1

Josef Koller

November 18, 2010

CONTENTS

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

CHAPTER

ONE

INSTALLATION

Option 1) to install it in the standard loction (depending on your system):

```
python setup.py install
```

or:

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

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

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

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

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

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

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

for the `bash` shell you would:

    export SPACEPY=/a/different/dir

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

CHAPTER

TWO

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

CHAPTER

THREE

TIME AND COORDINATE TRANSFORMATIONS

Import the modules as:

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

3.1 Ticktock Class

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

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

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

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

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

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

FYI `t.UTC` is added automatically.

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

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

You can replace RTD with any from the list above.

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

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

3.2 Tickdelta Class

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

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

Then you can add by e.g.:

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

3.3 Coords Class

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

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

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

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

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

Unit conversion will be implemented in the future.

CHAPTER

FOUR

THE RADBELT MODULE

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

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

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

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

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

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

THE BORG MODULE

(NOT FULLY FUNCTIONAL YET) The borg module includes data assimilation functions as classes. It is automatically imported within `radbelt`.

Similar to the `radbelt` module, import and create a class instance:

```
>>> import spacepy.radbelt as sprb
>>> rb = sprb.RBmodel()
>>> rb.setup_ticks('2002-09-01T00:00:00', '2002-09-10T00:00:00', 0.25)
```

Before solving the diffusion equations, you need to add PSD data and then call the `assimilate` function:

```
>>> rb.add_PSD()
>>> rb.assimilate(method='enKF')
>>> rb.plot(values=rb.PSDa)
```

CHAPTER

SIX

OMNI MODULE

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

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

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

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

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

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

CHAPTER

SEVEN

THE IRBEMPY MODULE

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

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

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

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

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

Other function wrapped with the IRBEM library include:

- `find_Bmirror`
- `find_magequator`
- `corrd_trans`

CHAPTER

EIGHT

PYCDF - PYTHON ACCESS TO NASA CDF LIBRARY

pycdf provides a “pythonic” interface to the NASA CDF library (currently read-only). It requires that the base C library be properly installed. The module can then be imported, e.g.:

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

Extensive documentation is provided in epydoc format in docstrings.

To open and close a CDF file:

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

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

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

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

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

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

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

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

CHAPTER

NINE

EMPIRICALS MODULE

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

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

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

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

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

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

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

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

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

CHAPTER

TEN

SEAPY - SUPERPOSED EPOCH ANALYSIS IN PYTHON

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

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

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

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

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

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

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